

# ОПЕРАЦИОННЫЕ СИСТЕМЫ

*Внутренняя структура и принципы  
проектирования*



ВИЛЬЯМ СТОЛЛИНГС

 АКАДЕМИКА

 Pearson

*9-е издание*

# ОПЕРАЦИОННЫЕ СИСТЕМЫ

*Внутренняя  
структура и принципы  
проектирования*

*9-е издание*

# OPERATING SYSTEMS

*Internals and Design  
Principles*

*Ninth Edition*

William Stallings



# ОПЕРАЦИОННЫЕ СИСТЕМЫ

*Внутренняя  
структура и принципы  
проектирования*

*9-е издание*

Вильям Столлингс



Москва • Санкт-Петербург  
2020

ББК 32.973.26-018.2.75

Tlgm: @it\_boooks

С81

УДК 004.451

ООО “Диалектика”

Зав. редакцией С.Н. Тригуб

Перевод с английского И.В. Берштейна, канд. техн. наук И.В. Красикова

Под редакцией канд. техн. наук И.В. Красикова

По общим вопросам обращайтесь в издательство “Диалектика” по адресу:

info@dialektika.com, http://www.dialektika.com

Столлингс, Вильям.

С81 Операционные системы: внутренняя структура и принципы проектирования, 9-е изд. : Пер. с англ. — СПб. : ООО “Диалектика”, 2020. — 1264 с. : ил. — Парал. тит. англ.

ISBN 978-5-907203-08-2 (рус.)

ББК 32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотовырезывание и запись на магнитный носитель, если на это нет письменного разрешения издательства Pearson Education, Inc.

Copyright © 2020 by Dialektika Computer Publishing.

Authorized translation from the English language edition of *Operating Systems: Internals and Design Principles*, 9th Edition (ISBN 978-0-13-467095-9), published by Pearson Education, Inc., Copyright © 2018, 2015, 2012, 2009 by Pearson Education, Inc., Hoboken, New Jersey 07030.

All rights reserved. No part of this book may be reproduced in any form or by any electronic or mechanical means (including photocopying, recording, or information storage and retrieval) without permission in writing from the publisher.

*Научно-популярное издание*

**Вильям Столлингс**

**Операционные системы:**

**внутренняя структура и принципы проектирования**

**9-е издание**

Подписано в печать 14.02.2020. Формат 70×100/16

Гарнитура Times

Усл. печ. л. 101,9. Уч.-изд. л. 76,8

Тираж 300 экз. Заказ № 0000

Отпечатано в ОАО “Первая Образцовая типография”

Филиал “Чеховский Печатный Двор”

142300, Московская область, г. Чехов, ул. Полиграфистов, д. 1

Сайт: [www.chpd.ru](http://www.chpd.ru), E-mail: [sales@chpd.ru](mailto:sales@chpd.ru), тел. 8 (499) 270-73-59

ООО “Диалектика”, 195027, Санкт-Петербург, Магнитогорская ул., д. 30, лит. А, пом. 848

ISBN 978-5-907203-08-2 (рус.)

© ООО “Диалектика”, 2020,

перевод, оформление, макетирование

ISBN 978-0-13-467095-9 (англ.)

© 2018, 2015, 2012, 2009 by Pearson Education, Inc.,

Hoboken, New Jersey 07030, 2017

# ОГЛАВЛЕНИЕ

Tlgm: @it\_boooks

<b>Часть I. Основы</b>	37
Глава 1. Обзор компьютерной системы	39
Глава 2. Обзор операционных систем	83
<b>Часть II. Процессы</b>	153
Глава 3. Описание процессов и управление ими	155
Глава 4. Потоки	211
Глава 5. Параллельные вычисления: взаимоисключения и многозадачность	265
Глава 6. Параллельные вычисления: взаимоблокировка и голодание	341
<b>Часть III. Память</b>	397
Глава 7. Управление памятью	399
Глава 8. Виртуальная память	433
<b>Часть IV. Планирование</b>	497
Глава 9. Однопроцессорное планирование	499
Глава 10. Многопроцессорное планирование и планирование реального времени	539
<b>Часть V. Ввод-вывод и файлы</b>	591
Глава 11. Управление вводом-выводом и планирование дисковых операций	593
Глава 12. Управление файлами	645
<b>Часть VI. Дополнительные темы</b>	699
Глава 13. Встроенные операционные системы	701
Глава 14. Виртуальные машины	735
Глава 15. Безопасность операционных систем	771
Глава 16. Облачные операционные системы и операционные системы Интернета вещей	819
Глава 17. Сетевые протоколы	867
Глава 18. Распределенная обработка, вычисления “клиент/сервер” и кластеры	897
Глава 19. Управление распределенными процессами	933
Глава 20. Обзор вероятности и стохастических процессов	975
Глава 21. Анализ очередей	1001
<b>Приложение А. Вопросы параллельности</b>	1043
<b>Приложение Б. Проекты в области программирования и операционных систем</b>	1059
<b>Приложение В. Дополнительные вопросы параллельности</b>	1071
<b>Приложение Г. Объектно-ориентированное проектирование</b>	1083
<b>Приложение Д. Закон Амдала</b>	1097
<b>Приложение Е. Хеш-таблицы</b>	1099
<b>Приложение Ж. Время отклика</b>	1103
<b>Приложение З. Концепции теории массового обслуживания</b>	1107
<b>Приложение И. Сложность алгоритмов</b>	1115
<b>Приложение К. Дисковые устройства хранения</b>	1119
<b>Приложение Л. Криптографические алгоритмы</b>	1131
<b>Приложение М. Введение в программирование сокетов</b>	1143
<b>Приложение Н. Международный справочный алфавит</b>	1175
<b>Приложение О. Параллельная система программирования ВАСI</b>	1179
<b>Приложение П. Управление процедурами</b>	1193
<b>Приложение Р. eCos</b>	1199
<b>Глоссарий</b>	1217
<b>Сокращения</b>	1235
<b>Список литературы</b>	1236
<b>Предметный указатель</b>	1251

# СОДЕРЖАНИЕ

Об авторе	26
<b>Предисловие</b>	27
Новое в девятом издании	27
Цели	28
Примеры систем	28
Поддержка курса ACM/IEEE computer science curricula 2013	29
План книги	29
Материалы для преподавателей	31
Проекты и упражнения для студентов	32
OS/161	32
Моделирование	33
Анимации	33
Программные проекты	33
Онлайн-документация и видеопримечания для студентов	34
Благодарности	34
Ждем ваших отзывов!	36
<b>Часть I. Основы</b>	37
<b>Глава 1. Обзор компьютерной системы</b>	39
1.1. Основные элементы	40
1.2. Эволюция микропроцессоров	42
1.3. Выполнение команд	43
1.4. Прерывания	46
Прерывания и цикл команды	48
Обработка прерывания	50
Множественные прерывания	54
1.5. Иерархия памяти	57
1.6. Кеш	60
Обоснование	61
Принципы работы кеша	61
Внутреннее устройство кеша	64
1.7. Прямой доступ к памяти	65
1.8. Организация многопроцессорных и многоядерных систем	66
Симметричная многопроцессорность	67
Многоядерные компьютеры	69
1.9. Ключевые термины, контрольные вопросы и задачи	71
Ключевые термины	71
Контрольные вопросы	71
Задачи	72
Приложение 1.А. Характеристики производительности двухуровневой памяти	75
Локальность	75
Функционирование двухуровневой памяти	78
Производительность	78

<b>Глава 2. Обзор операционных систем</b>	83
2.1. Цели и функции операционных систем	85
Операционная система как интерфейс между пользователем и компьютером	86
Операционная система как диспетчер ресурсов	88
Простота развития операционной системы	89
2.2. Эволюция операционных систем	90
Последовательная обработка	90
Простые пакетные системы	91
Многозадачные пакетные системы	95
Системы, работающие в режиме разделения времени	98
2.3. Основные достижения	100
Процессы	101
Управление памятью	105
Защита информации и безопасность	108
Планирование и управление ресурсами	108
2.4. Разработки, ведущие к современным операционным системам	110
2.5. Отказоустойчивость	114
Фундаментальные концепции	114
Отказы	116
Механизмы операционных систем	117
2.6. Вопросы проектирования операционных систем для многопроцессорных и многоядерных систем	117
Операционные системы для SMP	117
Вопросы проектирования операционных систем для многоядерных систем	119
2.7. Обзор операционной системы Microsoft Windows	121
Основы	121
Архитектура	121
Модель “клиент/сервер”	125
Потоки и симметричная многопроцессорность	126
Объекты Windows	127
2.8. Традиционные системы UNIX	129
Историческая справка	129
Описание	130
2.9. Современные системы UNIX	132
System V Release 4 (SVR4)	133
BSD	134
Solaris 11	134
2.10. Linux	134
История	134
Модульная структура	136
Компоненты ядра	138
2.11. Android	141
Программная архитектура Android	142
Система времени выполнения Android	144
Системная архитектура Android	147
Операции	149
Управление электропитанием	149

2.12. Ключевые термины, контрольные вопросы и задачи	150
Ключевые термины	150
Контрольные вопросы	150
Задачи	151
<b>Часть II. Процессы</b>	153
<b>Глава 3. Описание процессов и управление ими</b>	155
3.1. Что такое процесс	157
Основы	157
Процессы и управляющие блоки процессов	158
3.2. Состояния процесса	159
Модель процесса с двумя состояниями	162
Создание и завершение процессов	163
Модель с пятью состояниями	166
Приостановленные процессы	170
3.3. Описание процессов	177
Управляющие структуры операционной системы	177
Структуры управления процессами	179
3.4. Управление процессами	188
Режимы выполнения	188
Создание процессов	189
Переключение процессов	190
3.5. Выполнение кода операционной системы	194
Ядро вне процессов	194
Выполнение в составе пользовательских процессов	195
Операционная система на основе процессов	197
3.6. Управление процессами в операционной системе UNIX SVR4	198
Состояния процессов	198
Описание процессов	200
Управление процессами	203
3.7. Резюме	204
3.8. Ключевые термины, контрольные вопросы и задачи	205
Ключевые термины	205
Контрольные вопросы	205
Задачи	206
<b>Глава 4. Потоки</b>	211
4.1. Процессы и потоки	213
Многопоточность	214
Функциональность потоков	218
4.2. Типы потоков	220
Потоки на пользовательском уровне и на уровне ядра	220
Другие схемы	226
4.3. Многоядерность и многопоточность	228
Производительность программного обеспечения в многоядерных системах	228
Пример приложения: игровые программы Valve	231

4.4. Управление процессами и потоками в Windows	233
Управление фоновыми задачами и жизненным циклом приложений	234
Процессы в Windows	236
Объекты процессов и потоков	237
Многопоточность	238
Состояния потоков	239
Поддержка подсистем операционной системы	240
4.5. Управление потоками и SMP в Solaris	241
Многопоточная архитектура	241
Мотивация	242
Структура процессов	242
Выполнение потоков	244
Прерывания в роли потоков	245
4.6. Управление процессами и потоками в Linux	246
Задания Linux	246
Потоки Linux	248
Пространства имен Linux	249
4.7. Управление процессами и потоками в Android	252
Приложения Android	252
Операции	253
Процессы и потоки	255
4.8. Mac OS X Grand Central Dispatch	256
4.9. Резюме	259
4.10. Ключевые термины, контрольные вопросы и задачи	259
Ключевые термины	259
Контрольные вопросы	259
Задачи	260
<b>Глава 5. Параллельные вычисления: взаимоисключения и многозадачность</b>	265
5.1. Взаимоисключения: программный подход	269
Алгоритм Деккера	269
Алгоритм Петерсона	275
5.2. Принципы параллельных вычислений	276
Простой пример	277
Состояние гонки	279
Участие операционной системы	279
Взаимодействие процессов	280
Требования к взаимным исключениям	285
5.3. Взаимоисключения: аппаратная поддержка	285
Отключение прерываний	285
Специальные машинные команды	286
5.4. Семафоры	289
Задача производителя/потребителя	296
Реализация семафоров	302
5.5. Мониторы	304
Мониторы с сигналами	305
Мониторы с оповещением и широковещанием	308

5.6. Передача сообщений	311
Синхронизация	312
Адресация	314
Формат сообщения	315
Принцип работы очереди	316
Взаимные исключения	316
5.7. Задача читателей/писателей	318
Приоритетное чтение	319
Приоритетная запись	319
5.8. Резюме	324
5.9. Ключевые термины, контрольные вопросы и задачи	325
Ключевые термины	325
Контрольные вопросы	325
Задачи	326
<b>Глава 6. Параллельные вычисления: взаимоблокировка и голодание</b>	341
6.1. Принципы взаимного блокирования	343
Повторно используемые ресурсы	347
Расходуемые ресурсы	349
Графы распределения ресурсов	349
Условия возникновения взаимоблокировок	351
6.2. Предотвращение взаимоблокировок	352
Взаимоисключения	352
Удержание и ожидание	353
Отсутствие перераспределения	353
Циклическое ожидание	353
6.3. Устранение взаимоблокировок	354
Запрещение запуска процесса	354
Запрет выделения ресурса	355
6.4. Обнаружение взаимоблокировок	360
Алгоритм обнаружения взаимоблокировки	360
Восстановление	361
6.5. Интегрированные стратегии разрешения взаимоблокировок	362
6.6. Задача об обедающих философах	363
Решение с использованием семафоров	364
Решение с использованием монитора	364
6.7. Механизмы параллельных вычислений в UNIX	367
Каналы	367
Сообщения	368
Совместно используемая память	368
Семафоры	368
Сигналы	369
6.8. Механизмы параллельных вычислений ядра Linux	370
Атомарные операции	371
Циклические блокировки	373
Семафоры	375
Барьеры	377

6.9. Примитивы синхронизации потоков Solaris	379
Блокировки взаимоисключений	380
Семафоры	381
Блокировки “читатели/писатель”	381
Условные переменные	382
6.10. Механизмы параллельных вычислений в Windows	382
Функции ожидания	382
Объекты диспетчера	383
Критические участки	384
Гибкие блокировки читателя/писателя и условные переменные	385
Синхронизация без участия блокировок	385
6.11. Межпроцессное взаимодействие в Android	386
6.12. Резюме	388
6.13. Ключевые термины, контрольные вопросы и задачи	388
Ключевые термины	388
Контрольные вопросы	389
Задачи	389
<b>Часть III. Память</b>	397
<b>Глава 7. Управление памятью</b>	399
7.1. Требования к управлению памятью	401
Перемещение	401
Защита	401
Совместное использование	403
Логическая организация	403
Физическая организация	403
7.2. Распределение памяти	404
Фиксированное распределение	404
Динамическое распределение	408
Система двойников	412
Перемещение	414
7.3. Страницчная организация памяти	416
7.4. Сегментация	419
7.5. Резюме	421
7.6. Ключевые термины, контрольные вопросы и задачи	422
Ключевые термины	422
Контрольные вопросы	422
Задачи	423
Приложение 7.А. Загрузка и связывание	426
Загрузка	426
Компоновка	430
<b>Глава 8. Виртуальная память</b>	433
8.1. Аппаратное обеспечение и управляющие структуры	436
Локальность и виртуальная память	437
Страницчная организация	439

Сегментация	451
Комбинация сегментации и страничной организации	453
Защита и совместное использование	454
8.2. Программное обеспечение операционной системы	455
Стратегия выборки	457
Стратегия размещения	457
Стратегия замещения	458
Управление резидентным множеством	465
Стратегия очистки	473
Управление загрузкой	473
8.3. Управление памятью в UNIX и Solaris	476
Страницчная система	476
Распределение памяти ядра	480
8.4. Управление памятью в Linux	481
Виртуальная память Linux	483
Распределение памяти ядра	485
8.5. Управление памятью в Windows	486
Карта виртуальных адресов Windows	486
Страницчная организация Windows	488
Свопинг в Windows	489
8.6. Управление памятью в Android	489
8.7. Резюме	490
8.8. Ключевые термины, контрольные вопросы и задачи	491
Ключевые термины	491
Контрольные вопросы	491
Задачи	492

## Часть IV. Планирование

<b>Глава 9. Однопроцессорное планирование</b>	499
9.1. Типы планирования процессора	501
Долгосрочное планирование	502
Среднесрочное планирование	503
Краткосрочное планирование	504
9.2. Алгоритмы планирования	504
Критерии краткосрочного планирования	504
Использование приоритетов	506
Альтернативные стратегии планирования	507
Сравнение производительности	521
Справедливое планирование	526
9.3. Традиционное планирование UNIX	528
9.4. Резюме	530
9.5. Ключевые термины, контрольные вопросы и задачи	532
Ключевые термины	532
Контрольные вопросы	532
Задачи	533

<b>Глава 10. Многопроцессорное планирование и планирование реального времени</b>	539
10.1. Многопроцессорное и многоядерное планирование	541
Зернистость	542
Вопросы проектирования	544
Планирование процессов	546
Планирование потоков	546
Планирование потоков в многоядерных системах	554
10.2. Планирование реального времени	555
Введение	555
Характеристики операционных систем реального времени	556
Планирование реального времени	560
Планирование с предельными сроками	562
Частотно-монотонное планирование	566
Инверсия приоритета	569
10.3. Планирование в Linux	572
Планирование реального времени	572
Обычное планирование	574
10.4. Планирование в UNIX SVR4	576
10.5. Планирование в UNIX FreeBSD	578
Классы приоритетов	578
Поддержка SMP и многоядерности	579
10.6. Планирование в Windows	581
Приоритеты процессов и потоков	582
Многопроцессорное планирование	584
10.7. Резюме	585
10.8. Ключевые термины, контрольные вопросы и задачи	585
Ключевые термины	585
Контрольные вопросы	586
Задачи	586
<b>Часть V. Ввод-вывод и файлы</b>	591
<b>Глава 11. Управление вводом-выводом и планирование дисковых операций</b>	593
11.1. Устройства ввода-вывода	595
11.2. Организация функций ввода-вывода	597
Эволюция функций ввода-вывода	598
Прямой доступ к памяти	599
11.3. Вопросы проектирования операционных систем	601
Цели проектирования	601
Логическая структура функций ввода-вывода	602
11.4. Буферизация операций ввода-вывода	604
Двойной буфер	607
Циклический буфер	607
Использование буферизации	607
11.5. Дисковое планирование	608
Параметры производительности диска	608
Стратегии дискового планирования	611

11.6. RAID	617
RAID 0	618
RAID 1	622
RAID 2	623
RAID 3	623
RAID 4	624
RAID 5	625
RAID 6	625
11.7. Дисковый кеш	626
Вопросы разработки	626
Вопросы производительности	628
11.8. Ввод-вывод в UNIX SVR4	630
Буфер кеша	630
Очередь символов	632
Небуферизованный ввод-вывод	632
Устройства UNIX	632
11.9. Ввод-вывод в Linux	633
Дисковое планирование	633
Страницный кеш Linux	637
11.10. Ввод-вывод в Windows	638
Основные средства ввода-вывода	638
Асинхронный и синхронный ввод-вывод	639
Программное обеспечение RAID	640
Теневые копии тома	640
Шифрование тома	640
11.11. Резюме	641
11.12. Ключевые термины, контрольные вопросы и задачи	642
Ключевые термины	642
Контрольные вопросы	642
Задачи	643
<b>Глава 12. Управление файлами</b>	645
12.1. Обзор	647
Файлы и файловые системы	647
Структура файла	648
Системы управления файлами	650
Функции управления файлами	652
12.2. Организация файлов и доступ к ним	654
Смешанный файл	654
Последовательный файл	656
Индексно-последовательный файл	657
Индексированный файл	658
Файл прямого доступа	658
12.3. В-деревья	659
12.4. Каталоги файлов	662
Содержимое	662
Структура	663
Именование	665

12.5. Совместное использование файлов	667
Права доступа	667
Одновременный доступ	668
12.6. Записи и блоки	668
12.7. Управление вторичной памятью	670
Размещение файлов	670
Управление свободным пространством	676
Тома	678
Надежность	679
12.8. Управление файлами в UNIX	679
Индексные узлы	680
Размещение файлов	682
Каталоги	683
Структура тома	684
12.9. Виртуальная файловая система Linux	684
Суперблок	686
Индексный узел	687
Запись каталога	687
Файл	687
Кеши	688
12.10. Файловая система Windows	688
Ключевые возможности NTFS	688
Том NTFS и файловая структура	689
Способность восстановления данных	692
12.11. Управление файлами в Android	693
Файловая система	693
SQLite	695
12.12. Резюме	695
12.13. Ключевые термины, контрольные вопросы и задачи	696
Ключевые термины	696
Контрольные вопросы	696
Задачи	697
<b>Часть VI. Дополнительные темы</b>	699
<b>Глава 13. Встроенные операционные системы</b>	701
13.1. Встроенные системы	703
Концепции встроенных систем	703
Прикладные и специализированные процессоры	705
Микропроцессоры	705
Микроконтроллеры	707
Глубоко встроенные системы	708
13.2. Характеристики встроенных операционных систем	709
Исходные и целевые среды	710
Подходы к разработке	712
Адаптация существующей коммерческой операционной системы	713
Специально разработанная встроенная операционная система	713

13.3. Встроенная система Linux	714
Характеристики встроенной системы Linux	714
Файловые системы встроенного Linux	716
Преимущества встроенных систем Linux	717
μLinux	718
Android	720
13.4. TinyOS	721
Беспроводные сети датчиков	722
Цели TinyOS	723
Компоненты TinyOS	724
Планировщик в TinyOS	727
Пример конфигурации	728
Интерфейс ресурсов TinyOS	730
13.5. Ключевые термины, контрольные вопросы и задачи	732
Ключевые термины	732
Контрольные вопросы	732
Задачи	733
<b>Глава 14. Виртуальные машины</b>	735
14.1. Концепции виртуальных машин	736
14.2. Гипервизоры	740
Назначение гипервизоров	740
Паравиртуализация	743
Аппаратно поддерживаемая виртуализация	744
Виртуальное устройство	744
14.3. Контейнерная виртуализация	745
Группы управления ядром	745
Концепции контейнеров	746
Файловая система контейнера	750
Микрослужбы	750
Docker	752
14.4. Вопросы виртуализации на уровне процессоров	753
14.5. Управление памятью	755
14.6. Управление вводом-выводом	757
14.7. VMware ESXi	760
14.8. Варианты Hyper-V и Xen от корпорации Microsoft	762
14.9. Java VM	764
14.10. Архитектура виртуальной машины Linux VServer	765
Архитектура	765
Планирование процессов	767
14.11. Резюме	769
14.12. Ключевые термины, контрольные вопросы и задачи	769
Ключевые термины	769
Контрольные вопросы	769
Задачи	770

<b>Глава 15. Безопасность операционных систем</b>	771
15.1. Злоумышленники и зловредные программы	773
Угрозы системного доступа	773
Контрмеры	775
15.2. Переполнение буфера	778
Атаки типа переполнения буфера	778
Защита времени компиляции	782
Защита времени выполнения	785
15.3. Управление доступом	787
Управление доступом к файловой системе	787
Стратегии управления доступом	790
15.4. Управление доступом в UNIX	797
Традиционное управление доступом к файлам в UNIX	797
Списки управления доступом в UNIX	799
15.5. Усиление защиты операционных систем	800
Установка операционной системы и применение обновлений	801
Удаление ненужных служб, приложений и протоколов	802
Конфигурирование пользователей, групп и аутентификации	802
Конфигурирование средств управления ресурсами	803
Установка дополнительных средств управления защитой	804
Тестирование защиты системы	805
15.6. Поддержание безопасности	805
Протоколирование	805
Резервное копирование и архивирование данных	806
15.7. Безопасность Windows	806
Схема управления доступом	807
Токен доступа	808
Дескрипторы безопасности	809
15.8. Резюме	813
15.9. Ключевые термины, контрольные вопросы и задачи	814
Ключевые термины	814
Контрольные вопросы	814
Задачи	814
<b>Глава 16. Облачные операционные системы и операционные системы Интернета вещей</b>	819
16.1. Облачные вычисления	821
Элементы облачных вычислений	821
Модели предоставления услуг облачных вычислений	823
Модели развертывания облака	824
Эталонная архитектура облачных вычислений	828
16.2. Облачные операционные системы	831
Инфраструктура как служба	832
Требования к облачной операционной системе	834
Общая архитектура облачной операционной системы	835
OpenStack	842

16.3. Интернет вещей	851
Вещи в Интернете вещей	851
Эволюция	852
Компоненты IoT-устройств	852
Интернет вещей в контексте облака	853
Границы	853
Туманные вычисления	853
Базовая сеть	855
Облачная сеть	855
16.4. Операционные системы для Интернета вещей	856
Устройства с ограниченными ресурсами	857
Требования к операционным системам для Интернета вещей	858
Архитектура операционной системы для Интернета вещей	860
Операционная система RIOT	862
16.5. Ключевые термины и контрольные вопросы	865
Ключевые термины	865
Контрольные вопросы	865
<b>Глава 17. Сетевые протоколы</b>	867
17.1. Потребность в архитектуре протоколов	869
17.2. Архитектура протоколов TCP/IP	872
Уровни протоколов TCP/IP	872
Протоколы TCP и UDP	873
Протоколы IP и IPv6	875
Принцип действия протоколов TCP/IP	875
Приложения протокола TCP/IP	879
17.3. Сокеты	879
Сокет	880
Вызовы интерфейса Socket	881
17.4. Организация сетей в Linux	884
Передача данных	885
Прием данных	885
17.5. Резюме	886
17.6. Ключевые термины, контрольные вопросы и задачи	887
Ключевые термины	887
Контрольные вопросы	887
Задачи	887
Приложение 17.А. Простой протокол передачи файлов	891
Введение в протокол TFTP	891
Пакеты TFTP	891
Краткий обзор передачи данных	893
Ошибки и задержки	894
Синтаксис, семантика и синхронизация	895

<b>Глава 18. Распределенная обработка, вычисления “клиент/сервер” и кластеры</b>	897
18.1. Вычисления “клиент/сервер”	899
Что такое вычисления “клиент/сервер”	899
Приложения “клиент/сервер”	901
Промежуточное программное обеспечение	909
18.2. Распределенный обмен сообщениями	912
Надежность и ненадежность	914
Блокировка и неблокирующее выполнение	915
18.3. Вызов удаленных процедур	915
Передача параметров	917
Представление параметров	917
Привязка к архитектуре “клиент/сервер”	917
Синхронность и асинхронность	918
Объектно-ориентированные механизмы	918
18.4. Кластеры	919
Конфигурации кластеров	920
Вопросы проектирования операционных систем	922
Архитектура кластерных вычислительных систем	924
Кластеры в сравнении с симметричной многопроцессорной обработкой	925
18.5. Кластерный сервер Windows	926
18.6. Кластеры Beowulf и Linux	928
Функциональные средства Beowulf	928
Программное обеспечение Beowulf	929
18.7. Резюме	930
18.8. Ключевые термины, контрольные вопросы и задачи	931
Ключевые термины	931
Контрольные вопросы	931
Задачи	931
<b>Глава 19. Управление распределенными процессами</b>	933
19.1. Перенос процессов	934
Побудительные причины	934
Механизмы переноса процессов	935
Согласования переноса процессов	939
Выселение	941
Вытесняющие переносы в сравнении с невытесняющими	942
19.2. Распределенные глобальные состояния	942
Глобальные состояния и распределенные моментальные снимки	942
Алгоритм распределенных моментальных снимков	945
19.3. Распределенное взаимное исключение	947
Принципы распределенного взаимного исключения	948
Упорядочение событий в распределенной системе	950
Распределенная очередь	953
Метод передачи эстафеты	957
19.4. Распределенная взаимоблокировка	958
Взаимоблокировка при распределении ресурсов	960
Взаимоблокировка при обмене сообщениями	968

19.5. Резюме	972
19.6. Ключевые термины, контрольные вопросы и задачи	972
Ключевые термины	972
Контрольные вопросы	972
Задачи	973
<b>Глава 20. Обзор вероятности и стохастических процессов</b>	975
20.1. Основы теории вероятности	976
Определения вероятности	976
Условная вероятность и независимость	979
Теорема Байеса	980
20.2. Случайные переменные	981
Функции распределения и плотности	982
Важные виды распределений	983
Множество случайных переменных	985
20.3. Элементарные понятия стохастических процессов	987
Статистика первого и второго порядка	988
Стационарные стохастические процессы	989
Спектральная плотность	990
Независимые приращения	991
Эргодичность	996
20.4. Задачи	997
<b>Глава 21. Анализ очередей</b>	1001
21.1. Простой пример поведения очередей	1003
21.2. Цель анализа очередей	1008
21.3. Модели очередей	1011
Одноканальная система массового обслуживания	1011
Многоканальная система массового обслуживания	1015
Основные соотношения из теории массового обслуживания	1016
Предположения	1017
21.4. Одноканальные системы массового обслуживания	1018
21.5. Многоканальные системы массового обслуживания	1022
21.6. Примеры	1022
Сервер базы данных	1022
Вычисление процентилей	1024
Сильно связанный мультипроцессор	1025
Задача построения многоканальной системы массового обслуживания	1027
21.7. Очереди с приоритетами	1029
21.8. Сети очередей	1031
Разделение и объединение потоков трафика	1031
Последовательные очереди	1031
Теорема Джексона	1032
Применение теоремы Джексона в сети с коммутацией пакетов	1033
21.9. Другие модели систем массового обслуживания	1035

21.10. Оценка параметров модели	1035
Выборка	1036
Ошибки выборки	1038
21.11. Задачи	1038
<b>Приложение А. Вопросы параллельности</b>	1043
A.1. Состояния гонки и семафоры	1044
Постановка задачи	1044
Первая попытка	1044
Вторая попытка	1046
Третья попытка	1047
Четвертая попытка	1049
Правильное решение	1050
A.2. Задача о парикмахерской	1052
Неполное решение задачи о парикмахерской	1052
Полное решение задачи о парикмахерской	1055
A.3. Задачи	1057
<b>Приложение Б. Проекты в области программирования и операционных систем</b>	1059
B1. Программный проект 1 — разработка оболочки	1060
Требования к проекту	1061
Представление проекта	1062
Требуемая документация	1063
B2. Программный проект 2 — диспетчер процессов HOST	1063
Диспетчер процессов с четырехуровневым приоритетом	1063
Ограничения ресурсов	1064
Выделение памяти	1065
Процессы	1066
Список диспетчеризации	1067
Требования к проекту	1068
Практические результаты	1069
Представление проекта	1069
<b>Приложение В. Дополнительные вопросы параллельности</b>	1071
B.1. Регистры процессора	1072
Регистры, доступные пользователю	1072
Управляющие регистры и регистры состояния	1073
B.2. Выполнение команд функций ввода-вывода	1074
B.3. Технологии ввода-вывода	1075
Программируемый ввод-вывод	1075
Ввод-вывод, управляемый прерываниями	1076
Прямой доступ к памяти	1077
B.4. Вопросы аппаратной производительности в многоядерных системах	1078
Увеличение степени параллелизма	1078
Энергопотребление	1080

<b>Приложение Г. Объектно-ориентированное проектирование</b>	1083
Г.1. Мотивация	1084
Г.2. Объектно-ориентированные концепции	1085
Структура объектов	1086
Классы объектов	1087
Включение	1090
Г.3. Преимущества объектно-ориентированного подхода	1090
Г.4. CORBA	1091
Г.5. Дополнительные материалы	1095
<b>Приложение Д. Закон Амдала</b>	1097
<b>Приложение Е. Хеш-таблицы</b>	1099
<b>Приложение Ж. Время отклика</b>	1103
<b>Приложение 3. Концепции теории массового обслуживания</b>	1107
3.1. Зачем нужна теория массового обслуживания	1107
3.2. Очередь в случае одного сервера	1109
3.3. Многоканальная очередь	1111
3.4. Пуассонова скорость поступления	1113
<b>Приложение И. Сложность алгоритмов</b>	1115
<b>Приложение К. Дисковые устройства хранения</b>	1119
K.1. Магнитные диски	1119
Организация данных и форматирование	1119
Физические характеристики	1122
K.2. Оптическая память	1126
CD-ROM	1126
CD с возможностью записи	1128
CD-R с возможностью перезаписи	1128
DVD	1129
Оптические диски высокой четкости	1129
<b>Приложение Л. Криптографические алгоритмы</b>	1131
L.1. Симметричное шифрование	1131
DES	1133
AES	1134
L.2. Шифрование с открытым ключом	1134
Алгоритм Ривеста–Шамира–Адлемана (RSA)	1137
L.3. Аутентификация сообщений и хеш-функции	1137
Аутентификация с использованием симметричного шифрования	1138
Аутентификация без шифрования	1138
Код аутентификации сообщения	1139
Функция одностороннего хеширования	1140
L.4. Безопасные хеш-функции	1142

<b>Приложение М. Введение в программирование сокетов</b>	1143
M.1. Сокеты, дескрипторы, порты и соединения	1145
M.2. Модель “клиент/сервер”	1146
Запуск программы с использованием сокетов на компьютере под управлением Windows, не подключенном к сети	1146
Запуск программы с использованием сокетов на компьютере под управлением Windows, подключенном к сети, когда и сервер, и клиент находятся на одной машине	1148
M.3. Работа с сокетами	1148
Создание сокета	1148
Адрес сокета	1148
Привязка к локальному порту	1149
Представление данных и порядок байтов	1150
Подключение сокета	1151
Функция <code>gethostbyname()</code>	1152
Прослушивание входящих соединений	1154
Прием соединения от клиента	1154
Отправка и получение сообщения через сокет	1155
Закрытие сокета	1156
Сообщение об ошибках	1157
Пример клиентской программы TCP/IP (инициация соединения)	1158
Пример серверной программы TCP/IP (пассивное ожидание соединения)	1159
M.4. Сокеты потоков и дейтаграмм	1161
Пример клиентской программы UDP (инициация соединения)	1162
Пример серверной программы UDP (пассивное ожидание соединения)	1164
M.5. Управление программой времени выполнения	1165
Неблокирующие вызовы сокетов	1165
Асинхронный ввод-вывод (ввод-вывод, управляемый сигналом)	1166
M.6. Удаленное выполнение консольного приложения Windows	1169
Локальный код	1169
Удаленный код	1172
<b>Приложение Н. Международный справочный алфавит</b>	1175
<b>Приложение О. Параллельная система программирования BACI</b>	1179
O.1. Введение	1180
O.2. BACI	1180
Обзор системы	1180
Параллельные конструкции BACI	1181
Как получить BACI	1183
O.3. Примеры программ BACI	1183
O.4. Проекты BACI	1188
Реализация примитивов синхронизации	1188
Семафоры, мониторы и их реализации	1188
O.5. Усовершенствования системы BACI	1190

<b>Приложение П. Управление процедурами</b>	1193
П.1. Реализация стека	1193
П.2. Вызов процедуры и возврат из нее	1194
П.3. Реентерабельные процедуры	1196
<b>Приложение Р. eCos</b>	1199
P.1. Настраиваемость	1200
P.2. Компоненты eCos	1202
Уровень аппаратных абстракций	1203
Ядро eCos	1204
Система ввода-вывода	1205
Стандартные библиотеки C	1208
P.3. Планировщик eCos	1209
Планировщик битовой карты	1209
Планировщик многоуровневой очереди	1209
P.4. Синхронизация потоков eCos	1211
Мьютексы	1211
Семафоры	1211
Условные переменные	1212
Флаги событий	1215
Почтовые ящики	1215
Циклические блокировки	1216
<b>Глоссарий</b>	1217
<b>Сокращения</b>	1235
<b>Список литературы</b>	1236
<b>Предметный указатель</b>	1251

— Посвящается Трише

## ОБ АВТОРЕ

Доктор **Вильям Столлингс** является автором 18 книг, а включая переиздания, — более 40 книг по компьютерной безопасности, компьютерным сетям и архитектуре компьютеров. Его перу принадлежат многочисленные публикации в журналах, включая такие издания, как *Proceedings of the IEEE*, *ACM Computing Reviews* и *Cryptologia*.

От Ассоциации академических авторов Вильям Столлингс 13 раз получал награду за лучший учебник года в области компьютерных наук.

За более чем 30 лет работы в этой области он был техническим сотрудником, техническим руководителем и исполняющим директором ряда высокотехнологичных фирм. Он спроектировал и реализовал системы протоколов на базе TCP/IP и OSI для различных компьютеров и операционных систем от микрокомпьютеров до мейнфреймов. Он консультировал правительственные учреждения, поставщиков компьютеров и программного обеспечения и крупных пользователей в области дизайна, выбора и использования сетевого программного обеспечения и продуктов.

Столлингс создал и поддерживает сайт для студентов *Computer Science Student Resource Site* по адресу ComputerScienceStudent.com. Этот сайт предоставляет документацию и ссылки на различные темы, представляющие общий интерес как для студентов в области компьютерных наук, так и для профессионалов. Он является членом редакционной коллегии *Cryptologia* — научного журнала, посвященного различным аспектам криптографии.

Столлингс имеет ученую степень доктора философии в области компьютерных наук в МТИ и степень бакалавра электротехники в Нотр-Дам.

# ПРЕДИСЛОВИЕ

## Новое в девятом издании

С момента публикации восьмого издания этой книги в области операционных систем наблюдаются непрерывные нововведения и улучшения. В этой, новой, редакции я попытался уделить внимание этим изменениям при сохранении всеобъемлющего характера охвата всей рассматриваемой области. Процесс пересмотра начался с того, что восьмое издание книги было всесторонне проанализировано рядом профессоров, преподающих этот предмет, и работающими в данной области профессионалами. В результате было уточнено множество мест книги, улучшены иллюстрации, а описания стали более строгими.

Помимо этих уточнений, для повышения дидактического уровня и удобочитаемости техническое содержание книги было кардинально обновлено, чтобы отразить все произошедшие в данной захватывающей области изменения. Самые заметные изменения книги включают следующее.

- **Обновление материала о Linux.** Материал о Linux был обновлен и расширен для отражения изменений, произошедших с ядром Linux после выхода восьмого издания.
- **Обновление материала об Android.** Материал об Android был обновлен и расширен для отражения изменений, произошедших с ядром Android после выхода восьмого издания.
- **Новый материал о виртуализации.** Глава о виртуальных машинах полностью переписана, чтобы обеспечить лучшую организацию материала, а также большее его соответствие последним разработкам в этой области. Кроме того, добавлен новый раздел по использованию контейнеров.
- **Новые облачные операционные системы.** Новинкой в этом издании является описание облачных операционных систем, включающее обзор облачных вычислений, обсуждение принципов и требований к облачным операционным системам, а также рассмотрение OpenStack, популярной облачной операционной системы с открытым исходным кодом.
- **Новые операционные системы для Интернета вещей.** Еще одной новинкой этого издания являются операционные системы для Интернета вещей (Internet of Things — IoT). Книга включает обзор IoT, обсуждение принципов и требований к операционной системе IoT и обсуждение RIOT, популярной операционной системы IoT с открытым исходным кодом.
- **Обновленные и расширенные встраиваемые операционные системы.** Эта глава существенно переработана и дополнена.
  - Раздел о встраиваемых системах расширен и теперь включает в себя обсуждение микроконтроллеров и глубоко встраиваемых систем.
  - Обзорный раздел о встраиваемых операционных системах расширен и обновлен.
  - Расширено рассмотрение встраиваемых систем Linux; добавлено обсуждение популярной встраиваемой системы Linux — μClinux.
- **Параллельные вычисления.** В руководство по проектам добавлены новые проекты, призванные помочь студентам лучше понять принципы параллелизма.

## ЦЕЛИ

Эта книга — о концепциях, структурах и механизмах операционных систем. Ее цель — максимально ясно и полно представить природу и характеристики современных операционных систем.

Это очень сложная задача по целому ряду причин. Во-первых, имеется огромный ассортимент и разнообразие компьютеров, для которых предназначены эти операционные системы. К ним относятся встраиваемые системы, смартфоны, однопользовательские рабочие станции и персональные компьютеры, разделяемые системы среднего размера, большие ЭВМ, суперкомпьютеры и специализированные машины, такие как системы реального времени. Разнообразие ограничивается не только емкостью и скоростью машин, но и областями их применения и требованиями системы сопровождения. Во-вторых, стремительные перемены, которые всегда были характерны для компьютерных систем, продолжаются и в настоящее время. Ряд ключевых областей в проектировании операционных систем имеют совсем недавнее происхождение, и исследования этих и других новых областей продолжается и сегодня.

Несмотря на это разнообразие и темпы изменений, некоторые фундаментальные концепции последовательно применяются во всех операционных системах. Конечно, применение этих концепций зависит от текущего состояния технологий и требований конкретного приложения. Цель этой книги — представить тщательное рассмотрение основ проектирования операционных систем и связать их с вопросами современного проектирования операционных систем и текущими направлениями в их разработке.

## ПРИМЕРЫ СИСТЕМ

Книга предназначена для знакомства читателя с принципами проектирования и аспектами реализации современных операционных систем. Соответственно, чисто концептуальное или теоретическое рассмотрение будет неадекватным. Для иллюстрации концепций и связи их с реальным проектированием и выбором методов и технологий, которые должны быть при этом сделаны, в качестве работающих практических примеров операционных систем выбраны следующие.

- **Windows.** Многозадачная операционная система для персональных компьютеров, рабочих станций, серверов и мобильных устройств. Эта операционная система включает в себя многие из последних достижений в области технологий операционных систем. Кроме того, Windows является одной из первых важных коммерческих операционных систем, которые основаны на принципах объектно-ориентированного проектирования. Эта книга охватывает технологии, используемые в версии Windows, известной как Windows 10.
- **Android.** Операционная система Android предназначена для встраиваемых устройств, в частности для мобильных телефонов. В книге приводится подробная информация о внутреннем устройстве Android и особое внимание уделяется уникальным требованиям встраиваемой среды.
- **UNIX.** Это многопользовательская операционная система, изначально предназначавшаяся для мини-компьютеров, но реализованная для широкого спектра машин — от мощных микрокомпьютеров до суперкомпьютеров. В качестве приме-

ров включены несколько видов UNIX. Одной из широко используемых систем является FreeBSD, которая включает в себя многие современные возможности. Еще одной широко используемой коммерческой версией UNIX является Solaris.

- **Linux.** Широко используемая версия UNIX с открытым исходным кодом.

Эти системы были выбраны с учетом их актуальности и представительности. Обсуждение перечисленных операционных систем разбросано по всему тексту, а не сконцентрировано в виде отдельной главы или приложения. Таким образом, в ходе обсуждения параллелизма описаны механизмы параллелизма каждого из образцов систем и рассматривается мотивация выбора того или иного индивидуального дизайна. При таком подходе концепции проектирования, рассматриваемые в каждой конкретной главе, оказываются подкрепленными примерами из реальной жизни. Для удобства все материалы по каждой из систем доступны также в виде онлайн-документации.

## **ПОДДЕРЖКА КУРСА ACM/IEEE COMPUTER SCIENCE CURRICULA 2013**

Данная книга предназначена как для академической, так и для профессиональной аудитории. В качестве учебника она предназначена для одно- или двухсеместрового курса по операционным системам для специальностей информатики, вычислительной техники и электротехники. Это издание предназначено для использования в качестве одного из учебников для учебного курса ACM/IEEE computer science curricula 2013 (CS2013). Рекомендации CS2013 включают операционные системы как одну из тем данного курса. CS2013 делит весь курс на три уровня: первый уровень — все темы, которые должны быть включены в учебную программу, второй уровень — темы, которые желательно включить в программу, и выбранные темы, для которых желательно обеспечить повышенные широту и глубину изложения. В области операционных систем CS2013 включает в себя две темы первого уровня, четыре — второго и шесть выбранных тем, каждая из которых имеет ряд подтем. Книга охватывает все разделы и подразделы, перечисленные во всех трех категориях CS2013.

В табл. 1 показан охват книгой тем по операционным системам курса CS2013. Полный список подтем в каждой теме можно найти в файле CS2013-OS.pdf на сайте [box.com/OS9e](http://box.com/OS9e).

## **ПЛАН КНИГИ**

Книга состоит из шести частей.

- I. Основы
- II. Процессы
- III. Память
- IV. Планирование
- V. Ввод-вывод и файлы
- VI. Дополнительные темы (встроенные ОС, виртуальные машины, безопасность ОС, операционные системы Интернета вещей и облачные операционные системы)

**ТАБЛИЦА 1. ТЕМЫ ПО ОПЕРАЦИОННЫМ СИСТЕМАМ КУРСА CS2013**

<b>Тема (уровень)</b>	<b>Материал книги по данной теме</b>
Обзор операционных систем (уровень 1)	Глава 2, “Обзор операционных систем”
Принципы операционных систем (уровень 1)	Глава 1, “Обзор компьютерной системы” Глава 2, “Обзор операционных систем”
Параллельность (уровень 2)	Глава 5, “Параллельные вычисления: взаимоисключения и многозадачность” Глава 6, “Параллельные вычисления: взаимоблокировка и голодание” Приложение А, “Вопросы параллельности” Глава 18, “Распределенная обработка, вычисления «клиент/сервер» и кластеры”
Планирование и диспетчеризация (уровень 2)	Глава 9, “Однопроцессорное планирование” Глава 10, “Многопроцессорное планирование и планирование реального времени”
Управление памятью (уровень 2)	Глава 7, “Управление памятью” Глава 8, “Виртуальная память”
Безопасность и защита (уровень 2)	Глава 15, “Безопасность операционных систем”
Виртуальные машины (избранные темы)	Глава 14, “Виртуальные машины”
Управление устройствами (избранные темы)	Глава 11, “Управление вводом-выводом и планирование дисковых операций”
Файловые системы (избранные темы)	Глава 12, “Управление файлами”
Системы реального времени и встроенные системы (избранные темы)	Глава 10, “Многопроцессорное планирование и планирование реального времени” Глава 13, “Встроенные операционные системы” Материалы по операционной системе Android по всей книге
Отказоустойчивость (избранные темы)	Раздел 2.5
Производительность системы (избранные темы)	Вопросы производительности, связанные с управлением памятью, планированием и другими областями, рассматриваемые по всей книге

Книга включает многочисленные рисунки и таблицы, облегчающие восприятие материала. Каждая глава содержит список ключевых слов, обзор вопросов и домашние задания. Книга также включает обширный словарь терминов, список часто используемых сокращений и библиографию. Кроме того, для преподавателей доступен банк тестов.

## МАТЕРИАЛЫ ДЛЯ ПРЕПОДАВАТЕЛЕЙ

Основная цель книги — быть эффективным средством обучения по этой фундаментальной, но все еще находящейся в непрерывном развитии теме. Эта цель находит свое отражение в структуре книги и вспомогательных материалах. Книга сопровождается следующими дополнительными материалами в помощь преподавателю.

- **Ответы и решения.** Ответы на вопросы и решения задач, предлагаемых в конце каждой главы.
- **Руководство по проектам.** Предлагаемые проекты для всех категорий проектов, перечисленных в этом предисловии.
- **Слайды PowerPoint.** Набор слайдов, охватывающих материал всех глав и пригодных для демонстрации на лекциях.
- **PDF-файлы.** Репродукции всех рисунков и таблиц в книге.
- **Банк тестов.** Множество вопросов ко всем главам с отдельными файлами ответов.
- **Видеопримечания по параллельным вычислениям.** Профессора вечно утверждают, что параллелизм — самая сложная для понимания студентами концепция в области операционных систем. Издание сопровождается рядом видеолекций, обсуждающих различные алгоритмы параллельных вычислений, рассматриваемых в книге.
- **Примерные программы обучения.** Текст содержит больше материала, чем может быть охвачено в пределах одного семестра. Соответственно, преподаватели обеспечиваются несколькими вариантами учебных программ, которые определяют, как использовать книгу в течение ограниченного времени. Эти примеры основаны на реальном опыте ряда профессоров, работавших с седьмым изданием книги.

Все эти материалы доступны в **Центре ресурсов для преподавателей** (Instructor Resource Center — IRC) этого учебника, доступ к которому можно получить через веб-сайт издателя [www.pearsonhighered.com/stallings](http://www.pearsonhighered.com/stallings) или по ссылке *Pearson Resources for Instructors* на сайте автора книги по адресу [WilliamStallings.com/OperatingSystems](http://WilliamStallings.com/OperatingSystems). Чтобы получить доступ к IRC, обратитесь к местному представителю Pearson через страницу [pearsonhighered.com/educator/relocator/requestSalesRep.page](http://pearsonhighered.com/educator/relocator/requestSalesRep.page) или позвоните в Pearson Faculty Services по номеру 1-800-526-0485.

**Сайт автора** [WilliamStallings.com/OperatingSystems](http://WilliamStallings.com/OperatingSystems) предлагает следующее:

- ссылки на сайты других курсов;
- подписка на список рассылки для преподавателей для обмена информацией, предложениями и вопросами друг с другом и с автором.

## ПРОЕКТЫ И УПРАЖНЕНИЯ ДЛЯ СТУДЕНТОВ

Для многих преподавателей важным компонентом курса по операционным системам является проект или ряд проектов, при работе над которыми студент получает практический опыт, закрепляющий знания концепций из текста книги. В онлайн-части текста предложены два основных программных проекта. Кроме того, во вспомогательных материалах для преподавателей, доступных через Pearson, имеется не только руководство о том, как распределять и структурировать различные проекты, но и набор руководств пользователя для различных типов проектов, а также конкретные задания, написанные специально для этой книги. Преподаватели могут давать студентам задания в следующих областях:

- **Проект OS/161:** описан ниже.
- **Имитационные проекты:** описаны ниже.
- **Проекты с семафорами:** призваны помочь учащимся понять концепции параллелизма, включая состояния гонки, голодание и взаимоблокировку.
- **Проекты ядра:** IRC включает полную поддержку преподавателя для двух различных наборов проектов программирования ядра Linux, а также набор проектов программирования ядра Android.
- **Программные проекты:** описаны ниже.
- **Исследовательские проекты:** серия исследовательских проектов, которые требуют от студентов провести поиск информации по конкретной теме в Интернете и написать отчет.
- **Задания по работе со статьями:** список статей, которые можно дать студентам для чтения и написания отчетов или докладов.
- **Письменные задания:** список письменных заданий для облегчения изучения материала.
- **Темы для дискуссий:** темы, которые можно использовать в классе, чате или на форуме для углубленного изучения некоторых тем и повышения уровня сотрудничества студентов.

Кроме того, представлена информация о пакете программного обеспечения BACI, который служит в качестве каркаса для изучения механизмов параллелизма.

Этот разнообразный набор проектов и другие упражнения для студентов позволяют преподавателю использовать книгу как один из компонентов обучения, адаптировав курс к конкретным потребностям преподавателя и студентов. Подробности — в приложении Б, “Проекты в области программирования и операционных систем”.

## OS/161

Это издание обеспечивает поддержку компонента активного обучения на основе OS/161. OS/161 — учебная операционная система, которая все шире и шире применяется в качестве платформы для обучения внутреннему устройству операционных сис-

тем. Она обеспечивает накопление студентами опыта работы с реальной операционной системой, не подавляя их сложностями полноценной операционной системы, такой как Linux. По сравнению с реальными операционными системами OS/161 невелика (около 20 тысяч строк кода и комментариев) и, следовательно, гораздо проще для понимания студентами.

IRC включает следующее.

1. Упакованный набор html-файлов, которые преподаватель может загрузить на сервер учебного курса для доступа студентов.
2. Начальное руководство для студентов, призванное помочь им начать работать с OS/161.
3. Комплект упражнений с использованием OS/161 для студентов.
4. Типовые решения для каждого упражнения, предназначенные для преподавателя.
5. Все эти материалы связаны перекрестными ссылками с соответствующими разделами книги, так что студент может читать материал учебника, а затем выполнять соответствующий проект OS/161.

## Моделирование

IRC обеспечивает поддержку набора семи проектов, связанных с **моделированием**, которые охватывают ключевые области проектирования операционных систем. Студент может использовать набор пакетов моделирования для анализа особенностей дизайна операционной системы. Имитаторы написаны на Java и могут выполняться как локальное Java-приложение или как онлайн-приложение через браузер. IRC включает конкретные задания, которые можно давать студентам, точно указывая им, что они должны сделать и какие результаты ожидаются.

## Программные проекты

Это издание обеспечивает также поддержку проектов по программированию. В книге описаны два крупных программных проекта — создание оболочки, или интерпретатора командной строки, а также построение диспетчера процессов. IRC предоставляет дополнительные сведения и пошаговые упражнения для разработки этих программ.

В качестве альтернативы преподаватель может предложить студентам более обширный ряд проектов, которые охватывают различный изложенный в книге материал. Студенты обеспечиваются подробными инструкциями по выполнению каждого из проектов. Кроме того, для студентов имеется ряд домашних заданий, которые затрагивают вопросы, относящиеся к каждому проекту.

Наконец, представленная на IRC документация по проектам включает ряд программных проектов, которые охватывают широкий спектр тем и которые могут быть реализованы на любом подходящем языке на любой платформе.

# ОНЛАЙН-ДОКУМЕНТАЦИЯ И ВИДЕОПРИМЕЧАНИЯ ДЛЯ СТУДЕНТОВ

В этом издании значительное количество оригинальных вспомогательных материалов для студентов сделано доступным онлайн в двух местах. Веб-сайт автора [WilliamStallings.com/OperatingSystems](http://WilliamStallings.com/OperatingSystems) (ищите на нем ссылку *Student Resources*) включает список ссылок по тематике книги, организованных по главам, а также список известных опечаток.

Купив этот учебник\*, читатель получит и двенадцатимесячный доступ к сайту сопровождения книги, который включает следующие материалы.

- **Онлайн-приложения.** Есть многочисленные интересные темы, которые расширяют материал в тексте книги, но включение которых в печатный текст не является оправданным. В общей сложности имеется 15 онлайн-приложений с дополнительными материалами для любознательных студентов.
- **Домашние задания и решения.** Чтобы помочь студентам в понимании материала, предоставляется отдельный набор домашних заданий с решениями.
- **Анимации.** Анимации обеспечивают мощный инструмент, облегчающий понимание сложных механизмов современных операционных систем. Для иллюстрации ключевых функций и алгоритмов в проектировании операционных систем используются в общей сложности 53 анимации — в главах 3, 5–9 и 11.
- **Видеопримечания.** Видеопримечания представляют собой видеолекции, специально предназначенные для пошагового разъяснения программных концепций, представленных в этом учебнике. Книга сопровождается рядом видеолекций с обсуждением различных алгоритмов параллелизма, описанных в книге.

Чтобы получить доступ к этому содержимому сайта, щелкните на ссылке *Premium Content* на сайте сопровождения.

## БЛАГОДАРНОСТИ

Я хотел бы поблагодарить следующих людей за вклад в данную книгу. Большую часть нового материала о Linux предоставил Рами Розен (Rami Rosen). Винет Чадха (Vineet Chadha) внес большой вклад в новую главу о виртуальных машинах. Дургадосс Раманатан (Durgadoss Ramanathan) предоставил новый материал об Android ART.

Уже многие годы (и издания) в книге используются материалы от сотен преподавателей и специалистов, которые не пожалели своего драгоценного времени и щедро поделились своим опытом. Здесь я перечисляю тех, чья помощь в особенности способствовала написанию настоящего издания книги.

Всю или большую часть рукописи этого издания читали и обсуждали следующие преподаватели: Джиянг Гуо (Jiang Guo) (California State University, Los Angeles), Эврипид Монтань (Euripides Montagne) (University of Central Florida), Кихонг Парк (Kihong Park) (Purdue University), Мухаммед Абдус Салам (Mohammad Abdus Salam) (Southern University and A&M College), Роберт Марморштейн (Robert Marmorstein) (Longwood

---

\* Речь идет об оригинальном издании книги. — Примеч. пер.

University), Кристофер Диаз (Christopher Diaz) (Seton Hill University) и Барбара Брэкен (Barbara Bracken) (Wilkes University).

Благодарю всех, кто представил подробные технические обзоры для одной или нескольких глав: Нишай Аникар (Nischay Anikar), Эдри Джовин (Adri Jovin), Рон Мюниц (Ron Munitz), Фатих Эйуп Нар (Fatih Euyup Nar), Этт Пелтомаки (Atte Peltomaki), Дургадосс Раманатан (Durgadoss Ramanathan), Карлос Виллавейджа (Carlos Villavieja), Вей Ванг (Wei Wang), Сербан Константинеску (Serban Constantinescu) и Чен Янг (Chen Yang).

Спасибо также всем тем, кто представил подробные обзоры различных систем. Материалы по Android предоставили Кристофер Мицински (Kristopher Micinski), Рон Мюниц (Ron Munitz), Этт Пелтомаки (Atte Peltomaki), Дургадосс Раманатан (Durgadoss Ramanathan), Маниш Шакья (Manish Shakya), Сэмюэль Симон (Samuel Simon), Вей Ванг (Wei Wang) и Чен Янг (Chen Yang). Материалы по Linux предоставили Тигран Айвазян (Tigran Aivazian), Кайван Биллимория (Kaiwan Billimoria), Питер Хьюви (Peter Huewe), Манмохан Манохаран (Manmohan Manoharan), Рами Розен (Rami Rosen), Неха Наик (Neha Naik) и Хуалинг Ю (Hualing Yu). Материалы по Windows предоставили Франиско Котрина (Francisco Cotrina), Сэм Хайдар (Sam Haidar), Кристофер Кулесци (Christopher Kulesci), Бенни Олссон (Benny Olsson) и Дэйв Проберт (Dave Probert). Материалы по RIOT предоставили Эмманюэль Баччелли (Emmanuel Baccelli) и Каспар Шляйзер (Kaspar Schleiser), а по OpenStack — Боб Каллавей (Bob Callaway). Материал по eCos предоставлен Ником Гарнеттом (Nick Garnett) из eCosCentric; а Филип Левис (Philip Levis), один из разработчиков TinyOS, предоставил материалы по TinyOS. Сид Юнг (Sid Young) помог материалами по визуализации контейнеров. Эндрю Петерсон (Andrew Peterson) из Университета в Торонто подготовил материалы по OS/161 для IRC. Джеймс Крейг Барли (James Craig Burley) разработал и записал видеопримечания.

Упражнения по моделированию подготовил Адам Критчли (Adam Critchley) (University of Texas), а Марк Спаркс (Matt Sparks) (University of Illinois) адаптировал набор задач по программированию для данной книги.

Лоури Браун (Lawrie Brown) из Australian Defence Force Academy подготовил материал по атакам с использованием переполнения буфера. Чинг-Куанг Шень (Ching-Kuang Shene) (Michigan Tech University) предоставил примеры, использованные в разделе о состоянии гонки. Трейси Камп (Tracy Camp) и Кейт Хеллман (Keith Hellman) (Colorado School of Mines) разработали новый набор домашних заданий. Кроме того, Фернандо Ариэль Гонт (Fernando Ariel Gont) разработал ряд новых домашних заданий.

Я хотел бы также поблагодарить Билла Байnuma (Bill Bynum) (College of William and Mary) и Трейси Камп (Tracy Camp) (Colorado School of Mines) за вклад в приложение О, “Параллельная система программирования BACI”; Стива Тейлора (Steve Taylor) (Worcester Polytechnic Institute) и профессора Тан Нгуена (Tan N. Nguyen) (George Mason University) — за вклад в программные проекты и руководство для преподавателя. Ян Грэхем (Ian G. Graham) (Griffith University) внес свой вклад в два программных проекта данной книги. Оскарс Рикстс (Oskars Rieksts) (Kutztown University) щедро позволил мне воспользоваться его лекциями, викторинами и проектами.

Наконец, я благодарю множество людей, ответственных за публикацию этой книги, которые, как всегда, сделали отличную работу. К ним относятся сотрудники Pearson, в частности мой редактор Трейси Джонсон (Tracy Johnson), ее помощник Кристи Алаура (Kristy Alaura), руководитель программы Кэрол Снайдер (Carole Snyder) и менеджер проекта Боб Энгельгардт (Bob Engelhardt). Спасибо также персоналу отдела маркетинга и продаж Pearson, без усилий которых эта книга не была бы у вас в руках.

## ЖДЕМ ВАШИХ ОТЗЫВОВ!

Вы, читатель этой книги, и есть главный ее критик. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересны любые ваши замечания в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам электронное письмо либо просто посетить наш веб-сайт и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится ли вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Отправляя письмо или сообщение, не забудьте указать название книги и ее авторов, а также свой обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию новых книг.

Наши электронные адреса:

E-mail: info@williamspublishing.com

WWW: <http://www.williamspublishing.com>

ЧАСТЬ I

---

# Основы



# Обзор компьютерной системы

В ЭТОЙ ГЛАВЕ...

## 1.1. Основные элементы

## 1.2. Эволюция микропроцессоров

## 1.3. Выполнение команд

## 1.4. Прерывания

Прерывания и цикл команды

Обработка прерывания

Множественные прерывания

## 1.5. Иерархия памяти

## 1.6. Кеш

Обоснование

Принципы работы кеша

Внутреннее устройство кеша

## 1.7. Прямой доступ к памяти

## 1.8. Организация многопроцессорных и многоядерных систем

Симметричная многопроцессорность

Многоядерные компьютеры

## 1.9. Ключевые термины, контрольные вопросы и задачи

Ключевые термины

Контрольные вопросы

Задачи

## Приложение 1.А. Характеристики производительности двухуровневой памяти

Локальность

Функционирование двухуровневой памяти

Производительность

## УЧЕБНЫЕ ЦЕЛИ

- Перечислить основные элементы компьютерной системы и их взаимодействия.
- Пояснить, какие действия предпринимает процессор для выполнения команды.
- Понимать концепцию прерываний и то, как и почему процессор использует прерывания.
- Перечислить и описать уровни типичной иерархии компьютерной памяти.
- Пояснить основные характеристики многопроцессорных систем и многоядерных компьютеров.
- Обсудить концепцию локальности и проанализировать производительность многоуровневой иерархии памяти.
- Понимать работу стека и его применение для поддержки вызовов процедур и возврата из них.

Операционная система обслуживает пользователей, предоставляя им набор служб и обращаясь для этого к ресурсам аппаратного обеспечения, в состав которых входит один или несколько процессоров. Кроме того, она управляет вторичной памятью и устройствами ввода-вывода. Поэтому, прежде чем приступить к исследованию операционных систем, важно получить некоторое представление о компьютерных системах, на которых они работают.

В данной главе представлен обзор аппаратного обеспечения компьютерных систем. Большинство вопросов освещено кратко, так как предполагается, что читатель знаком с предметом. Однако некоторые из них раскрыты более подробно, исходя из важности этих тем для дальнейшего усвоения материала. Дополнительные вопросы рассматриваются в приложении В, “Дополнительные вопросы параллельности”, а более подробное изложение можно найти в [242].

## 1.1. ОСНОВНЫЕ ЭЛЕМЕНТЫ

На верхнем уровне компьютер состоит из процессора, памяти и устройств ввода-вывода; при этом каждый компонент представлен одним или несколькими модулями. Чтобы компьютер мог выполнять свое основное предназначение, состоящее в выполнении программ, различные компоненты должны иметь возможность взаимодействовать между собой. Можно выделить четыре структурных компонента компьютера.

- **Процессор.** Осуществляет управление всеми действиями компьютера, а также выполняет функцию обработки данных. Если в системе есть только один процессор, он часто называется **центральным процессором** (*central processing unit — CPU*).
- **Основная память.** В ней хранятся данные и программы. Как правило, эта память является временной, т.е. при выключении компьютера ее содержимое теряется. Содержимое же дисковой памяти, напротив, сохраняется даже при выключении компьютерной системы. Часто ее называют *реальной*, *оперативной* или *первичной* памятью.

- Устройства ввода-вывода.** Служат для передачи данных между компьютером и внешним окружением, состоящим из различных периферийных устройств, в число которых входят вторичная память (например, диски), коммуникационное оборудование и терминалы.
- Системная шина.** Обеспечивает взаимодействие между процессорами, основной памятью и устройствами ввода-вывода.

На рис. 1.1 показаны компоненты верхнего уровня. Одной из функций процессора является обмен данными с памятью. Для этого он обычно использует два внутренних (по отношению к процессору) регистра: регистр адреса памяти (memory address register — MAR), куда заносится адрес ячейки памяти, в которой будет производиться операция чтения-записи, и регистр буфера памяти (memory buffer register — MBR), куда заносятся данные, предназначенные для записи в память, или те, которые были прочитаны из нее. Аналогично устройство ввода-вывода задается в регистре адреса ввода-вывода (I/O address register — I/O AR). Регистр буфера ввода-вывода (I/O buffer register — I/O BR) служит для обмена данными между устройством ввода-вывода и процессором.

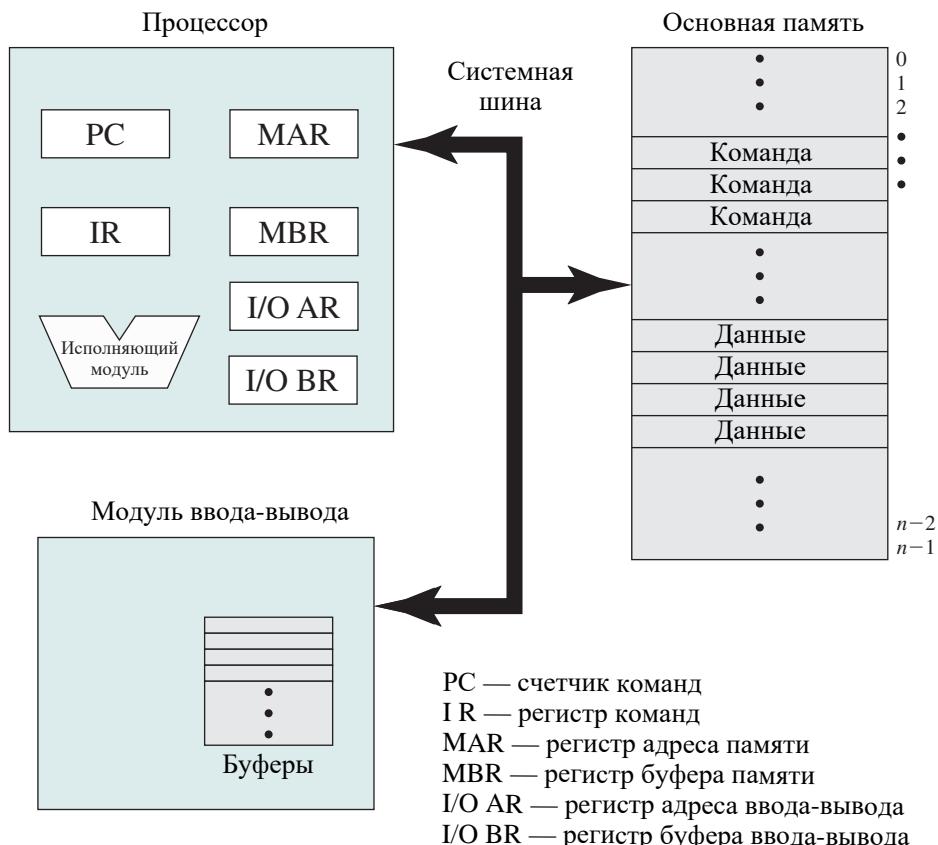


Рис. 1.1. Компоненты компьютера: общая структура

Модуль памяти состоит из множества последовательно пронумерованных ячеек. В каждую ячейку может быть записана последовательность битов, которая интерпретируется либо как команда, либо как данные. Модуль ввода-вывода служит для передачи данных от внешних устройств как в процессор и память, так и в обратном направлении. Для временного хранения данных до окончания их пересылки в нем есть свои внутренние буферы.

## 1.2. ЭВОЛЮЦИЯ МИКРОПРОЦЕССОРОВ

Революция, которая принесла нам настольные и переносные компьютеры, состояла в изобретении микропроцессора, который содержал процессор в одной микросхеме. Хотя первоначально такой процессор был гораздо более медленным, чем процессор на нескольких микросхемах, микропроцессоры постоянно развивались и в настоящее время для большинства вычислений оказываются гораздо более быстрыми благодаря физике, которая обеспечивает перемещение информации за субнаносекундные интервалы времени.

Но микропроцессоры стали не только быстрыми и доступными процессорами общего назначения; они теперь являются мультипроцессорами — в каждой микросхеме содержится несколько процессоров (называемых ядрами). Каждое такое ядро имеет несколько уровней кешей памяти большого объема, а несколько логических процессоров совместно используют исполнительные модули каждого ядра. По состоянию на 2010 год даже для ноутбука не является необычным наличие двух или четырех ядер, каждое с двумя аппаратными потоками, т.е. в общей сложности — четыре или восемь логических процессоров.

Хотя процессоры обеспечивают очень хорошую производительность для большинства видов вычислений, существует растущий спрос на численные вычисления. Графические процессоры (Graphical Processing Units — GPU) обеспечивают эффективные вычисления над массивами данных, используя возможность обработки множества данных одной командой (Single-Instruction Multiple Data — SIMD) — технологии, впервые появившейся в суперкомпьютерах. GPU больше не используются только лишь для вывода графики: они применяются и для числовой обработки общего назначения, например для моделирования физики для игр или вычислений в больших электронных таблицах. Одновременно процессоры получают все больше возможностей для работы с массивами данных — в архитектуру процессоров семейств x86 и AMD64 интегрируются все более и более мощные векторные модули.

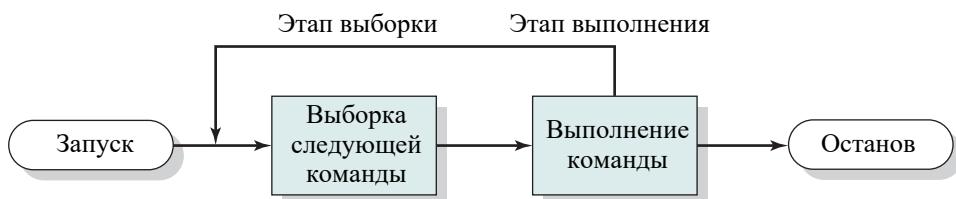
Процессоры и графические процессоры — не конец вычислительной истории современных персональных компьютеров. Имеются, например, процессоры для обработки цифровых сигналов (Digital Signal Processors — DSP), которые предназначены для работы с потоковыми сигналами, такими как аудио или видео. Они использовались для встраивания в устройства ввода-вывода, такие как модемы, но в настоящее время стали полноценными вычислительными устройствами, особенно в наладонниках. Другие специализированные вычислительные устройства (модули с фиксированными функциями) существуют с процессорами для поддержки других стандартных вычислений, например для кодирования/декодирования речи и видео (кодеки), или обеспечения поддержки шифрования и безопасности.

Чтобы удовлетворить требованиям к портативным устройствам, классический микропроцессор уступает место системам на кристалле (System on a Chip — SoC), в которых в одной микросхеме находятся не только процессор и кеш-память, но и многие другие компоненты системы, такие как DSP, GPU, устройства ввода-вывода (например, радио и кодеки), а также основная память.

## 1.3. ВЫПОЛНЕНИЕ КОМАНД

Программа, которую выполняет процессор, состоит из набора хранящихся в памяти команд. В простейшем виде обработка команд проходит в две стадии: процессор считывает (*выбирает*) из памяти, а затем выполняет очередную команду. Выполнение программы сводится к повторению процесса выборки команды и ее выполнения. Для выполнения одной команды может потребоваться несколько операций; их число определяется природой самой команды.

Набор действий, требующихся для реализации одной команды, называется ее *циклом*. На рис. 1.2 показан процесс обработки команд процессором в такой упрощенной схеме, включающей два этапа. Эти этапы называются *этапом выборки* и *этапом выполнения*. Прекращение работы программы происходит при выключении машины, в случае возникновения какой-либо фатальной (неисправимой) ошибки или если в программе имеется команда останова.



**Рис. 1.2.** Базовый цикл выполнения программы

В начале каждого цикла процессор выбирает из памяти команду. Обычно адрес ячейки, из которой нужно извлечь очередную команду, хранится в счетчике команд (PC). Если не указано иное, после извлечения каждой команды процессор увеличивает значение счетчика команд на единицу. Таким образом, команды выполняются в порядке возрастания номеров ячеек памяти, в которых они хранятся. Рассмотрим, например, упрощенный компьютер, в котором каждая команда занимает одно 16-битовое слово памяти. Предположим, что значение счетчика команд установлено равным 300. Это значит, что следующая команда, которую должен извлечь процессор, находится в 300-й ячейке. При успешном завершении цикла команды процессор перейдет к извлечению команд из ячеек 301, 302, 303 и т.д. Однако, как мы вскоре узнаем, эта последовательность может быть изменена.

Выбранные команды загружаются в регистр команд (IR). Команда состоит из последовательности битов, указывающих процессору, какие именно действия он должен выполнить. Процессор интерпретирует команду и выполняет требуемые действия. В общем случае все действия можно разбить на четыре категории.

- **Процессор — память.** Данные передаются из процессора в память или обратно.
- **Процессор — устройства ввода-вывода.** Данные из процессора поступают на периферийное устройство путем передачи их между процессором и устройством ввода-вывода. Возможен процесс и в обратном направлении.
- **Обработка данных.** Процессор может выполнять с данными различные арифметические или логические операции.

- Управление.** Команда может задавать изменение последовательности выполнения команд. Например, если процессор извлекает из ячейки 149 команду, которая указывает, что следующей по очереди должна быть выполнена команда из ячейки 182, то процессор устанавливает значение счетчика команд равным 182. Таким образом, на следующем этапе выборки команда извлекается не из ячейки 150, а из ячейки 182.

Для выполнения команды может потребоваться последовательность, состоящая из комбинации вышеперечисленных действий.

Рассмотрим простой пример гипотетической машины, характеристики которой приведены на рис. 1.3. В процессоре имеется единственный регистр данных, который называется аккумулятором (accumulator — AC). Команды и данные имеют длину 16 бит. В такой ситуации память удобно организовать в виде 16-битовых ячеек, в каждой из которых помещается одно слово. Формат команды предусматривает выделение 4 бит для кода операции. Таким образом, всего может быть  $2^4 = 16$  различных кодов операций (их можно представить одной шестнадцатеричной<sup>1</sup> цифрой), а адресовать можно до  $2^{12} = 4096$  (4 Кбайт) слов памяти (которые можно представить трехзначным шестнадцатеричным числом).

0	3 4	15
Код операции	Адрес	

а) Формат команды

0	1	15
S	Значение	

б) Формат целого числа

Счетчик команд (PC) = Адрес команды  
 Регистр команды (IR) = Выполняемая команда  
 Аккумулятор (AC) = Временная память

в) Внутренние регистры процессора

0001 — загрузить значение AC из памяти  
 0010 — сохранить значение AC в память  
 0101 — добавить к AC значение из памяти

г) Фрагмент списка операций

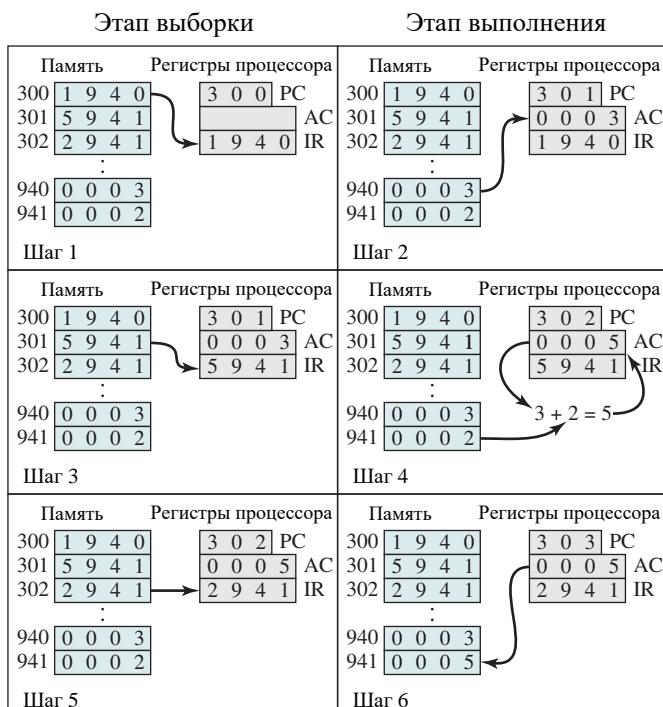
### Рис. 1.3. Характеристики гипотетической машины

На рис. 1.4, на котором показаны определенные ячейки памяти и регистры процессора, иллюстрируется выполнение фрагмента программы. В этом фрагменте слово, хранящееся в памяти по адресу 940, складывается со словом, хранящимся в памяти по адресу 941, а результат сложения заносится в ячейку 941.

<sup>1</sup> Основные сведения по системам счисления (десятичной, двоичной, шестнадцатеричной) можно найти в соответствующем документе на сайте Computer Science Student Resource Site по адресу ComputerScienceStudent.com.

Для выполнения этого действия потребуются три команды, каждая из которых включает свой этап выборки и этап выполнения.

1. Адрес первой команды, хранящейся в счетчике команд, — 300. Эта команда (она представлена шестнадцатеричным числом 1940) загружается в регистр команд (IR), а показание счетчика команд увеличивается на 1. Следует отметить, что в этом процессе участвуют регистры адреса памяти (MAR) и буферы памяти (MBR), однако для упрощения они не показаны.
2. Первые 4 бита (первая шестнадцатеричная цифра) в регистре команд указывают на то, что нужно загрузить значение в аккумулятор. Остальные 12 бит (три шестнадцатеричные цифры) указывают адрес 940.
3. Из ячейки 301 извлекается следующая команда (5941), после чего значение счетчика команд увеличивается на 1.
4. К содержимому аккумулятора прибавляется содержимое ячейки 941, и результат снова заносится в аккумулятор.
5. Из ячейки 302 извлекается следующая команда (2941), затем значение счетчика команд увеличивается на 1.
6. Содержимое аккумулятора заносится в ячейку 941.



**Рис. 1.4.** Пример выполнения программы (содержимое памяти и регистров представлено шестнадцатеричными числами)

Этот пример показывает, что для сложения содержимого ячеек 940 и 941 необходимы три цикла команды. При более сложном наборе команд циклов понадобилось бы меньше. Большинство современных процессоров содержат команды, в состав которых может входить несколько адресов. При этом во время цикла выполнения некоторых команд иногда выполняется несколько обращений к памяти. Вместо обращений к памяти в команде может быть задана операция ввода-вывода.

## 1.4. ПРЕРЫВАНИЯ

Почти во всех компьютерах предусмотрен механизм, с помощью которого различные модули (ввода-вывода, памяти) могут прервать нормальную последовательность работы процессора. Основные общепринятые классы прерываний перечислены в табл. 1.1.

**Таблица 1.1. Классы прерываний**

<b>Программные</b>	Генерируются в некоторых ситуациях, возникающих в результате выполнения команд. Такими ситуациями могут быть арифметическое переполнение, деление на нуль, попытка выполнить некорректную команду или обратиться к области памяти, доступ к которой пользователю запрещен
<b>Таймера</b>	Генерируется таймером процессора. Это прерывание позволяет операционной системе выполнять некоторые функции периодически, через заданные промежутки времени
<b>Ввода-вывода</b>	Генерируется контроллером ввода-вывода. Сигнализирует о нормальном завершении операции или о наличии ошибок
<b>Аппаратный сбой</b>	Генерируется при сбоях и аварийных ситуациях, как, например, падение напряжения в сети или ошибка контроля четности памяти

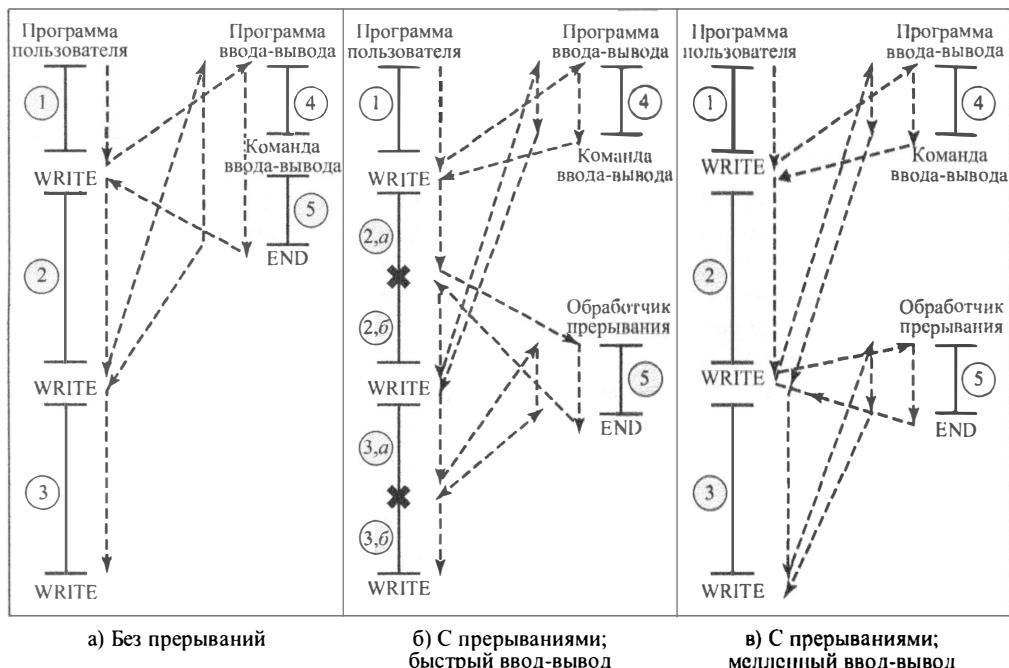
Прерывания предназначены в основном для повышения эффективности использования процессора. Например, большинство устройств ввода-вывода работают намного медленнее, чем процессор. Предположим, что процессор передает данные на принтер по схеме, показанной на рис. 1.2. После каждой операции процессор вынужден делать паузу и ждать, пока принтер примет данные. Длительность этой паузы может быть в тысячи или даже миллионы раз больше длительности цикла команды. Ясно, что подобное использование процессора является крайне неэффективным.

Чтобы привести конкретный пример, рассмотрим персональный компьютер, который работает с частотой 1 ГГц, что позволяет выполнять примерно  $10^9$  команд в секунду<sup>2</sup>. А типичный жесткий диск имеет скорость вращения 7200 об/мин, что дает время обращения половины дорожки около 4 мс — в 4 миллиона раз медленнее, чем работа процессора.

Это положение дел проиллюстрировано на рис. 1.5, а. Программа пользователя содержит ряд вызовов процедуры записи WRITE, в промежутках между которыми расположены другие команды. Сплошные вертикальные линии представляют отрезки кода программы. В отрезках 1, 2 и 3 находятся последовательности команд кода, в которых

<sup>2</sup> Обсуждение различных числовых приставок наподобие “гига” или “тера” можно найти в соответствующем документе на сайте Computer Science Student Resource Site по адресу ComputerScienceStudent.com.

не используется ввод-вывод. При вызове процедуры WRITE управление передается системной подпрограмме ввода-вывода, которая выполняет соответствующие операции.



**Рис. 1.5.** Ход выполнения программы без прерываний и с их использованием

Программа ввода-вывода состоит из трех частей.

- Последовательность команд, обозначенных на рисунке цифрой 4, которые служат для подготовки к собственно операциям ввода-вывода. В эту последовательность могут входить копирование выводимых данных в специальный буфер и подготовка набора параметров, необходимых для управления устройством.
- Собственно команда ввода-вывода. Если программа не использует прерываний, ей придется ждать, пока устройство ввода-вывода не выполнит требуемые операции (или периодически проверять его состояние путем опроса). При этом программе не остается ничего другого, как просто ждать, постоянно проверяя, завершилась ли операция ввода-вывода.
- Последовательность команд, обозначенных на рисунке цифрой 5, которые служат для завершения операции. Эта последовательность может содержать в себе установку флагов, свидетельствующих об успешном или неудачном завершении операции.

Пунктирная линия представляет путь выполнения, которому следует процессор, т.е. эта строка показывает последовательность, в которой выполняются команды. Таким образом, после первой встреченной команды WRITE выполнение пользовательской программы прерывается и процессор приступает к выполнению программы ввода-вывода. После завершения выполнения программы ввода-вывода возобновляется выполнение программы пользователя с команды, непосредственно следующей за командой WRITE.

Из-за того что для выполнения операции ввода-вывода может потребоваться сравнительно длительный промежуток времени, программа замедляет работу, ожидая завершения операции. Таким образом, там, где встречается вызов WRITE, производительность программы существенно уменьшается.

## Прерывания и цикл команды

Благодаря прерываниям во время выполнения операций ввода-вывода процессор может быть занят обработкой других команд. Рассмотрим ход процесса, показанный на рис. 1.5, б. Как и в предыдущем случае (без использования прерываний), вызвав процедуру WRITE, программа обращается к системе. При этом активизируется программа ввода-вывода, которая состоит из подготовительного кода и собственно команд ввода-вывода. После выполнения этих команд управление передается программе пользователя. Тем временем внешнее устройство занято приемом данных из памяти компьютера и их обработкой (например, если этим устройством является принтер, то под обработкой подразумевается распечатка). Ввод-вывод происходит одновременно с выполнением команд программы пользователя.

В тот момент, когда внешнее устройство освобождается и готово для дальнейшей работы, т.е. оно готово принять от процессора новую порцию данных, контроллер ввода-вывода этого устройства посыпает процессору сигнал *запроса прерывания* (interrupt request). В ответ процессор приостанавливает выполнение текущей программы, переключаясь на работу с программой, обслуживающей данное устройство ввода-вывода (эту программу называют обработчиком прерываний). Обслужив внешнее устройство, процессор снова возобновляет прерванную работу. На рис. 1.5, б места программы, в которых происходит прерывание, обозначены крестиком. Обратите внимание, что прерывание может произойти в любой точке основной программы, а не только в одной конкретной команде.

С точки зрения программы пользователя прерывания — это не что иное, как приостановка обычной последовательности исполнения. После завершения обработки прерывания работа возобновляется (рис. 1.6).

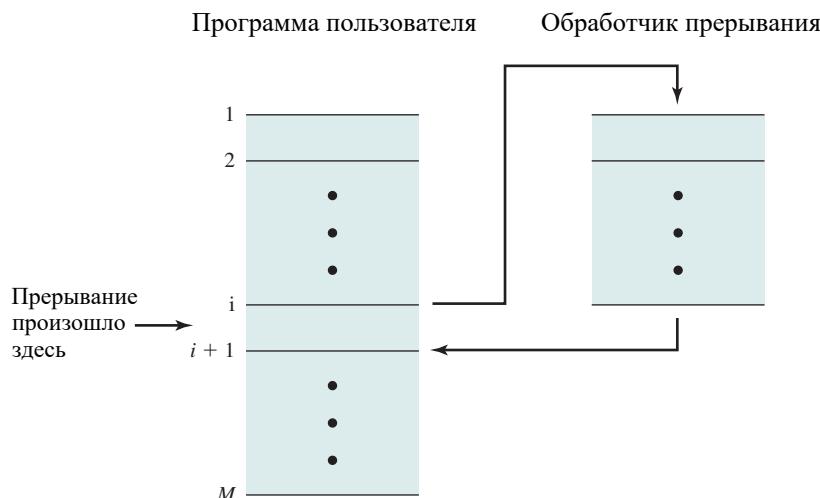
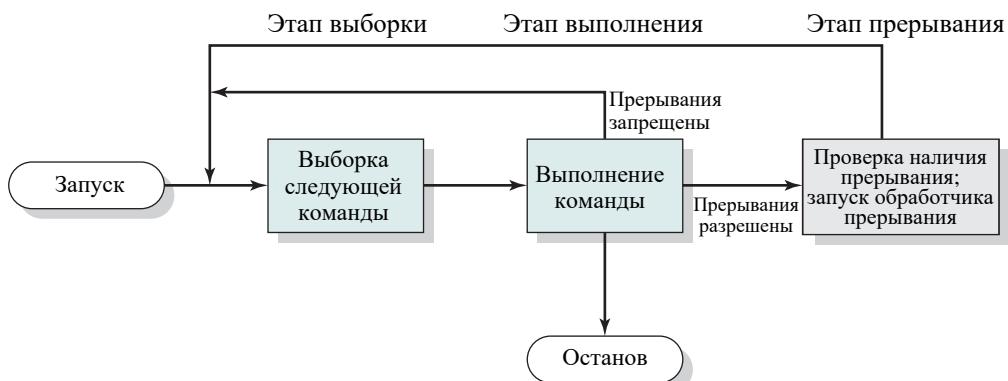


Рис. 1.6. Передача управления через прерывание

Таким образом, программа пользователя не должна содержать в себе какой-нибудь специальный код, чтобы приспособливаться к прерываниям. За приостановку программы пользователя и возобновление ее работы с того самого места, в котором она была прервана, отвечают процессор и операционная система.

Чтобы согласовать прерывание с программой, в цикл команды добавляется *этап прерывания* (рис. 1.7; сравните с рис. 1.2). На этапе прерывания процессор проверяет наличие сигналов прерываний, свидетельствующих о произошедших прерываниях. Если прерывания не было, процессор переходит к выборке следующей команды текущей программы. При поступлении прерывания процессор приостанавливает работу с текущей программой и выполняет *обработчик прерываний*. Обработчики прерываний обычно входят в состав операционной системы. Как правило, эти программы определяют природу прерывания и выполняют необходимые действия. Например, в используемом примере обработчик должен определить, какой из контроллеров ввода-вывода сгенерировал прерывание; кроме того, он может передавать управление программе, которая должна вывести данные на устройство ввода-вывода. Когда обработчик прерываний завершает свою работу, процессор возобновляет выполнение программы пользователя с того места, где она была прервана.

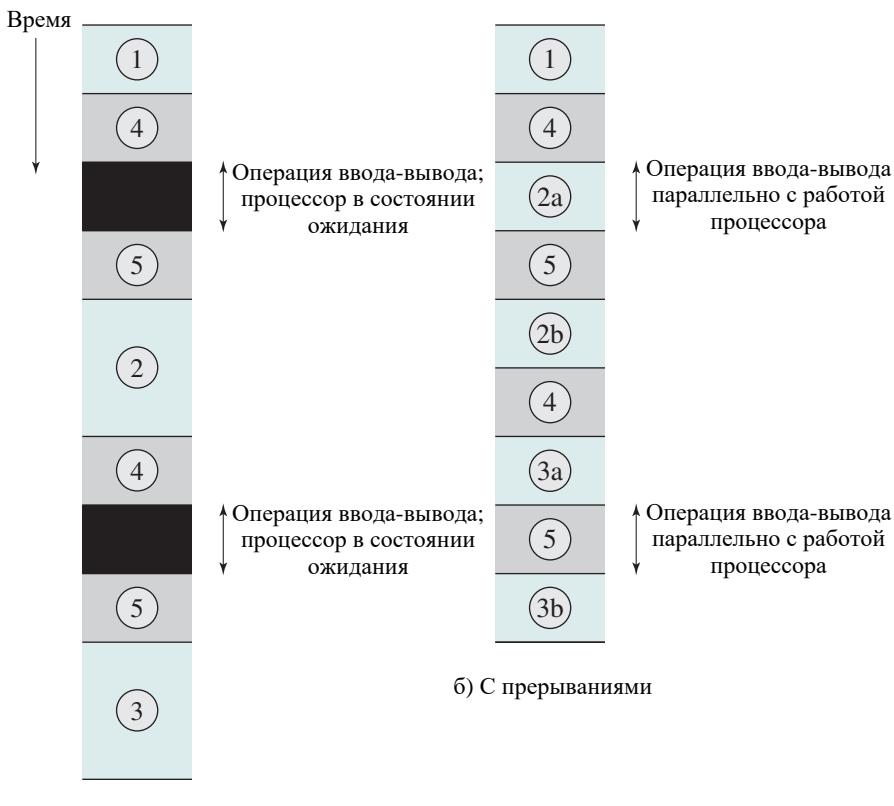


**Рис. 1.7.** Цикл команды с прерываниями

Ясно, что этот процесс включает в себя некоторые непроизводительные накладные расходы. Для определения природы прерывания и принятия решения о последующих действиях обработчик прерываний должен выполнить дополнительные команды. Тем не менее, ввиду того что для ожидания завершения операций ввода-вывода потребовался бы сравнительно большой отрезок времени, с помощью прерываний процессор можно использовать намного эффективнее.

Чтобы оценить выигрыш в эффективности, рассмотрим временную диаграмму (рис. 1.8), иллюстрирующую ход процессов, показанных на рис. 1.5, а и б. В ситуации, показанной на рис. 1.5, б и 1.8, предполагается, что для выполнения операций ввода-вывода требуется сравнительно короткое время, т.е. меньшее, чем время обработки команд, которые расположены в программе пользователя между операциями записи. Более типичным, особенно для таких медленных устройств, как принтер, является случай, когда операции ввода-вывода отнимают намного больше времени, чем требуется для выполнения последовательности команд пользователя. Такая ситуация показана на рис. 1.5, в.

В этом случае программа пользователя дойдет до следующего вызова WRITE еще до завершения операции ввода-вывода, порожденной предыдущим вызовом. В результате в этом месте программа пользователя будет приостановлена. После завершения обработки предыдущей операции ввода-вывода придет очередь обработать новое обращение к процедуре WRITE и будут запущены новые операции ввода-вывода.



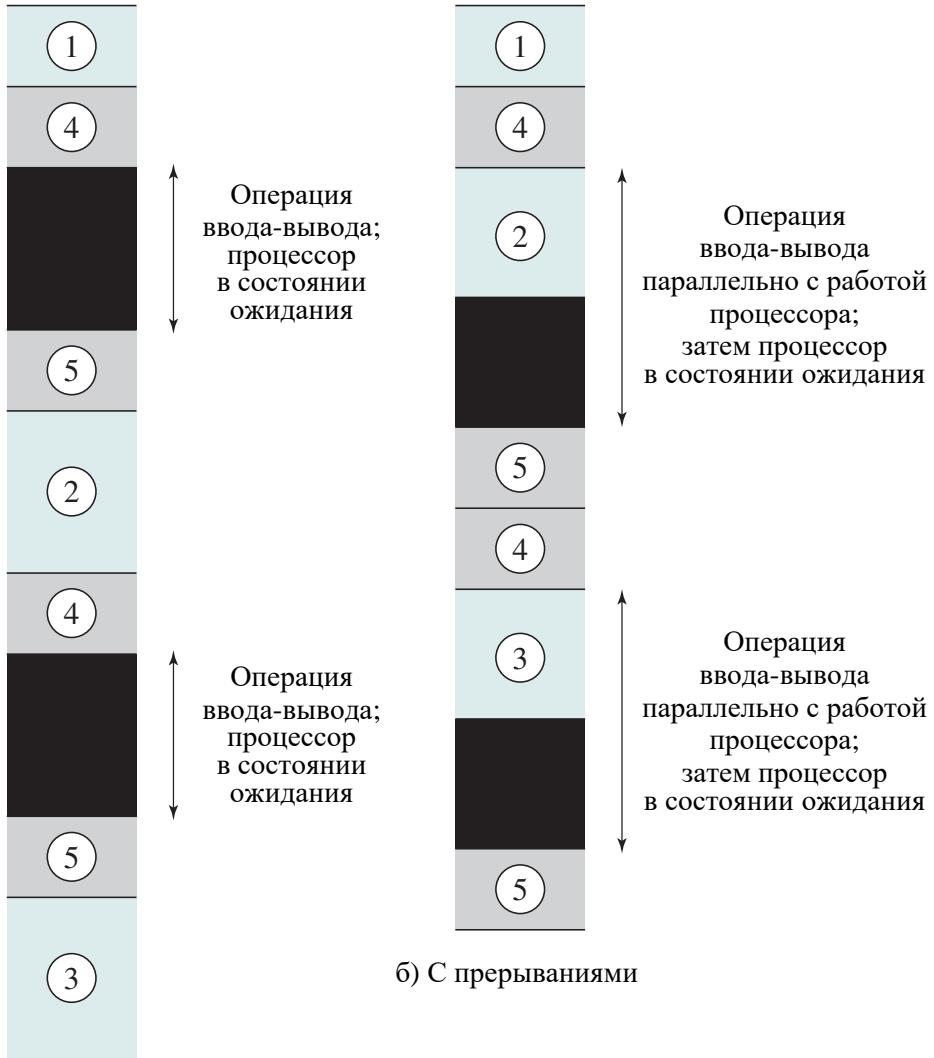
**Рис. 1.8.** Временная диаграмма программы: быстрый ввод-вывод

На рис. 1.9 представлена диаграмма выполнения программы в среде без прерываний и с прерываниями для описанного случая. Как видно, в такой ситуации выигрыш в эффективности все равно существует, так как часть времени, в течение которого выполняются операции ввода-вывода, перекрывается выполнением команд пользователя.

## Обработка прерывания

Прерывание вызывает ряд событий, которые происходят как в аппаратном, так и в программном обеспечении. На рис. 1.10 показана типичная последовательность этих событий.

Время

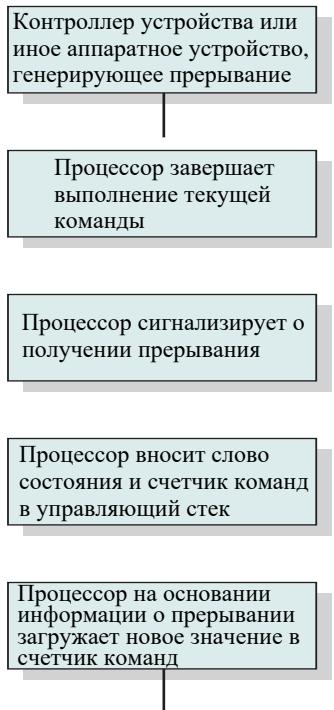


а) Без прерываний

б) С прерываниями

**Рис. 1.9.** Временная диаграмма программы: медленный ввод-вывод

## Аппаратное обеспечение



## Программное обеспечение

**Рис. 1.10.** Обработка простого прерывания

После завершения работы устройства ввода-вывода происходит следующая последовательность аппаратных событий.

1. Устройство посылает процессору сигнал прерывания.
2. Перед тем как ответить на прерывание, процессор должен завершить выполнение текущей команды (см. рис. 1.7).
3. Процессор производит проверку наличия прерывания, обнаруживает его и посыпает устройству, приславшему это прерывание, уведомляющий сигнал об успешном приеме. Этот сигнал позволяет устройству снять свой сигнал прерывания.
4. Теперь процессору нужно подготовиться к передаче управления программе обработчика прерываний. Сначала необходимо сохранить всю важную информацию, чтобы в дальнейшем можно было вернуться к тому месту текущей программы, где она была приостановлена. Минимальная требуемая информация — это слово состояния программы<sup>3</sup> (program status word — PSW) и адрес очередной выполняемой команды, который находится в счетчике команд. Эти данные заносятся в системный управ器яющий стек (см. приложение П, “Управление процедурами”).

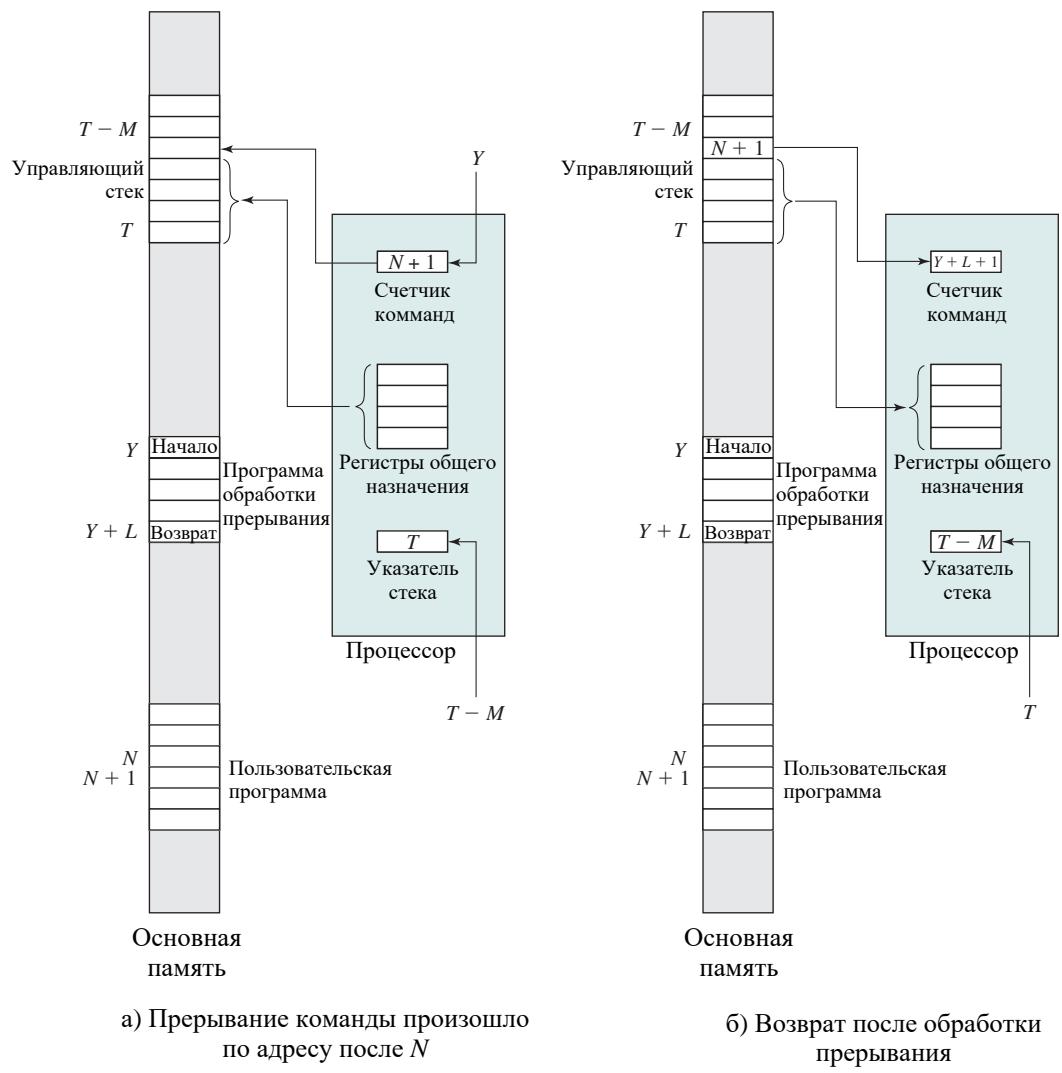
<sup>3</sup> Словосочетание “состояние программы” содержит информацию о текущем выполняемом процессе, включая информацию об использовании памяти, кодах состояния и иной информации о состоянии, такой как бит включения/отключения прерываний и бит режима ядра или пользовательского режима (см. более подробное описание в приложении В, “Дополнительные вопросы параллельности”).

- Далее в счетчик команд процессора загружается адрес входа программы обработки прерываний, которая отвечает за обработку данного прерывания. В зависимости от архитектуры компьютера и устройства операционной системы могут существовать и одна программа для обработки всех прерываний, и своя программа обработки для каждого устройства и каждого типа прерываний. Если для обработки прерываний имеется несколько программ, то процессор должен определить, к какой из них следует обратиться. Эта информация может содержаться в первоначальном сигнале прерывания; в противном случае для получения необходимой информации процессор может запросить устройство, которое сгенерировало прерывание, для получения нужной информации в ответе от него.

Как только в счетчик команд загружается новое значение, процессор переходит к следующему циклу команды, приступая к ее извлечению из памяти. Так как команда извлекается из ячейки, номер которой задается содержимым счетчика команд, управление переходит к программе обработки прерываний. Выполнение этой программы влечет за собой следующие операции.

- Содержимое счетчика команд и слово состояния прерываемой программы уже хранятся в системном стеке. Однако это еще не вся информация, имеющая отношение к состоянию выполняемой программы. Например, нужно сохранить содержимое регистров процессора, так как эти регистры могут понадобиться обработчику прерываний. Поэтому необходимо сохранить всю информацию о состоянии программы. Обычно обработчик прерываний начинает свою работу с записи в стек содержимого всех регистров. Другая информация, которая должна быть сохранена, обсуждается в главе 3, “Описание процессов и управление ими”. На рис. 1.11, *a* показан простой пример, в котором программа пользователя прерывается после выполнения команды из ячейки *N*. Содержимое всех регистров, а также адрес следующей команды (*N+1*), в сумме составляющие *M* слов, заносятся в управляющий стек. Указатель стека при этом обновляется, указывая на новую вершину стека. Обновляется и счетчик команд, указывая на начало программы обработки прерывания.
- Теперь обработчик прерываний начинает свою работу. В процесс обработки прерывания входит проверка информации состояния, имеющая отношение к операциям ввода-вывода или другим событиям, вызвавшим прерывание. Сюда может также входить пересылка устройствам ввода-вывода дополнительных инструкций или уведомляющих сообщений.
- После завершения обработки прерываний из стека извлекаются сохраненные ранее значения, которые вновь заносятся в регистры, возобновляя, таким образом, то состояние, в котором они пребывали до прерывания (см., например, рис. 1.11, *b*).
- Последний этап — восстановление из стека слова состояния программы и содержимого счетчика команд. В результате следующей будет выполняться команда прерванной программы.

Из-за того что прерывание не является подпрограммой, вызываемой из программы, для полного восстановления важно сохранить всю информацию состояния прерываемой программы. Однако прерывание может произойти в любой момент и в любом месте программы пользователя. Это событие непредсказуемо.

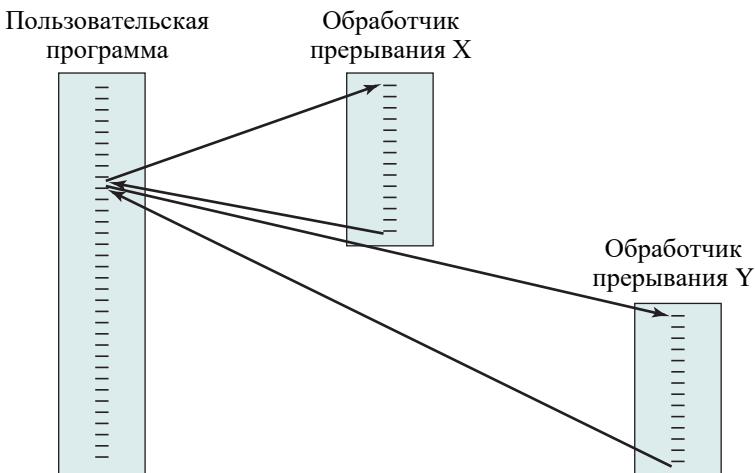


**Рис. 1.11.** Изменение памяти и регистров при обработке прерывания

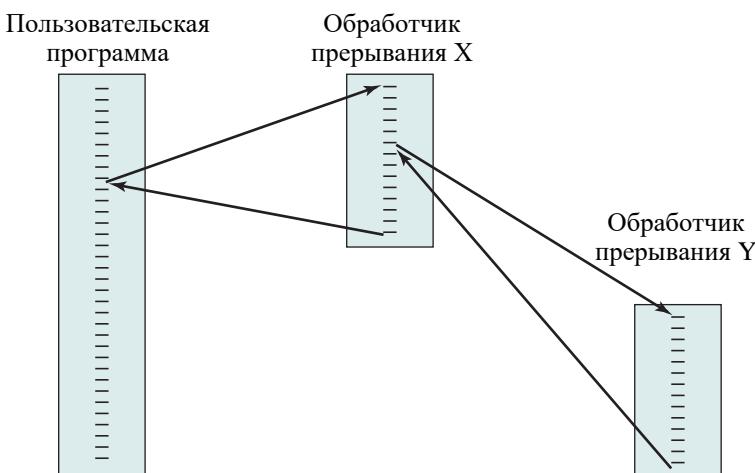
## Множественные прерывания

До сих пор нами рассматривался случай возникновения одного прерывания. Представим себе ситуацию, когда во время обработки прерывания может произойти одно или несколько прерываний. Например, программа получает данные по коммуникационной линии и одновременно распечатывает результат. Принтер будет генерировать прерывание при каждом завершении операции печати, а контроллер коммуникационной линии — при каждом поступлении новой порции данных. Эта порция может состоять из одного символа или из целого блока, в зависимости от установленного порядка обслуживания. В любом случае возможна ситуация, когда коммуникационное прерывание произойдет во время обработки прерывания принтера.

В такой ситуации возможны два подхода. Первый — это запретить новые прерывания до тех пор, пока обрабатывается предыдущее. *Запрет прерываний* означает, что процессор может и должен игнорировать любой новый сигнал прерывания. Если в это время происходит прерывание, оно обычно остается в состоянии ожидания, и до него дойдет очередь, когда процессору вновь можно будет обрабатывать прерывания. Таким образом, если во время работы программы пользователя происходит прерывание, на другие прерывания тут же накладывается запрет. После завершения работы программы обработки прерывания запрет снимается, и перед возвратом к выполнению прерванной программы процессор проверяет наличие других прерываний. Это простой подход, при котором прерывания обрабатываются в строго последовательном порядке (рис. 1.12, а).



а) Последовательная обработка прерываний



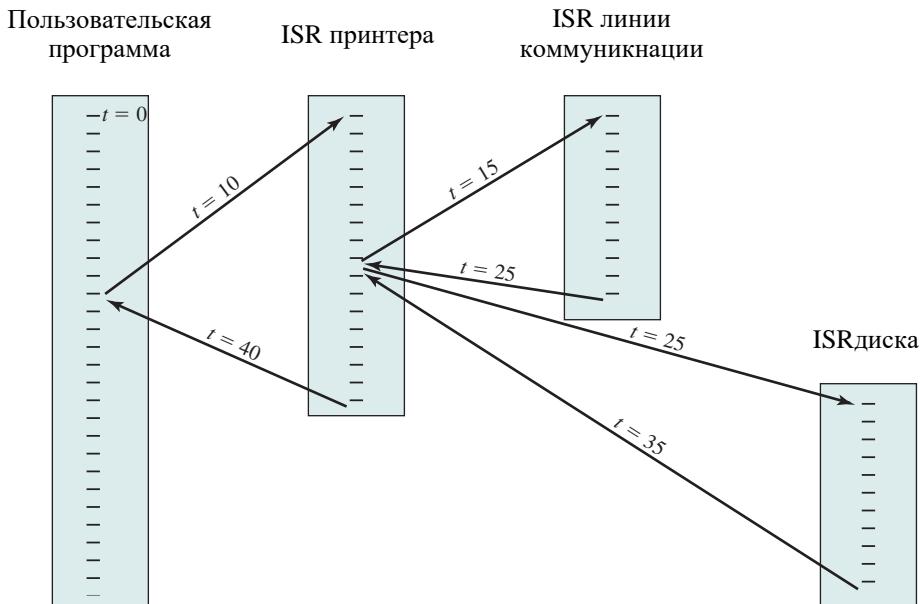
б) Вложенная обработка прерываний

**Рис. 1.12.** Передача управления при множественных прерываниях

Однако недостатком такого подхода является то, что не учитываются относительный приоритет прерываний и ситуации, в которых время является критическим параметром. Например, когда по коммуникационной линии приходит какая-то информация, может понадобиться быстро ее принять, чтобы освободить место для других входных данных. Если не обработать первый пакет входных данных перед получением второго пакета, данные могут потеряться вследствие загруженности и переполнения буфера устройства ввода-вывода.

При втором подходе учитывается приоритет прерывания, что позволяет приостановить обработку прерывания с более низким приоритетом в пользу прерывания с более высоким приоритетом (рис. 1.12, б). Как пример применения этого подхода рассмотрим систему с тремя устройствами ввода-вывода: принтером, диском и коммуникационной линией, которым присвоены приоритеты в возрастающей последовательности — 2, 4 и 5 соответственно.

На рис. 1.13 показана очередность обработки прерываний, поступивших от этих устройств. Программа пользователя запускается в момент времени  $t = 0$ . В момент  $t = 10$  происходит прерывание принтера. Информация о программе пользователя заносится в системный стек, и в действие вступает стандартная программа обслуживания прерывания (interrupt service routine — ISR). Во время ее работы в момент  $t = 15$  происходит коммуникационное прерывание. Из-за того что его приоритет выше, чем приоритет прерывания принтера, процессор приступает к его обработке. ISR принтера прерывается, информация о ее состоянии заносится в стек, а управление передается коммуникационной ISR. Далее, пока эта программа выполняется, происходит прерывание диска (в момент времени  $t = 20$ ). Так как его приоритет ниже, коммуникационная ISR продолжает свою работу, пока не закончит ее.



**Рис. 1.13.** Пример временной последовательности множественных прерываний

После выполнения ISR коммуникационной линии ( $t = 25$ ) восстанавливается предыдущее состояние процессора, т.е. выполнение ISR принтера. Однако, прежде чем успеет выполниться хоть одна команда этой программы, процессор приступает к обработке прерывания диска, которое обладает более высоким приоритетом, и управление передается ISR диска. И только после завершения этой программы ( $t = 35$ ) возобновляется работу ISR принтера. И наконец, после завершения обработки этого прерывания управление передается программе пользователя.

## 1.5. ИЕРАРХИЯ ПАМЯТИ

Проектные ограничения на конфигурацию памяти компьютера в основном определяются тремя параметрами: объемом, быстродействием, стоимостью.

Вопрос об объеме решить не так просто. Какой бы большой ни была память, все равно будут разработаны приложения, которым ее не хватит. В отношении быстродействия памяти все ясно: чем быстрее, тем лучше. Для достижения высшей производительности память должна иметь возможность быстро обмениваться данными с процессором. Но в реальной жизни едва ли не главным становится третий параметр. Стоимость памяти должна быть сравнима со стоимостью других компонентов.

Очевидно, можно найти некоторый компромисс между перечисленными характеристиками памяти. На любом этапе развития технологий производства запоминающих устройств выполняются следующие, достаточно устойчивые, соотношения.

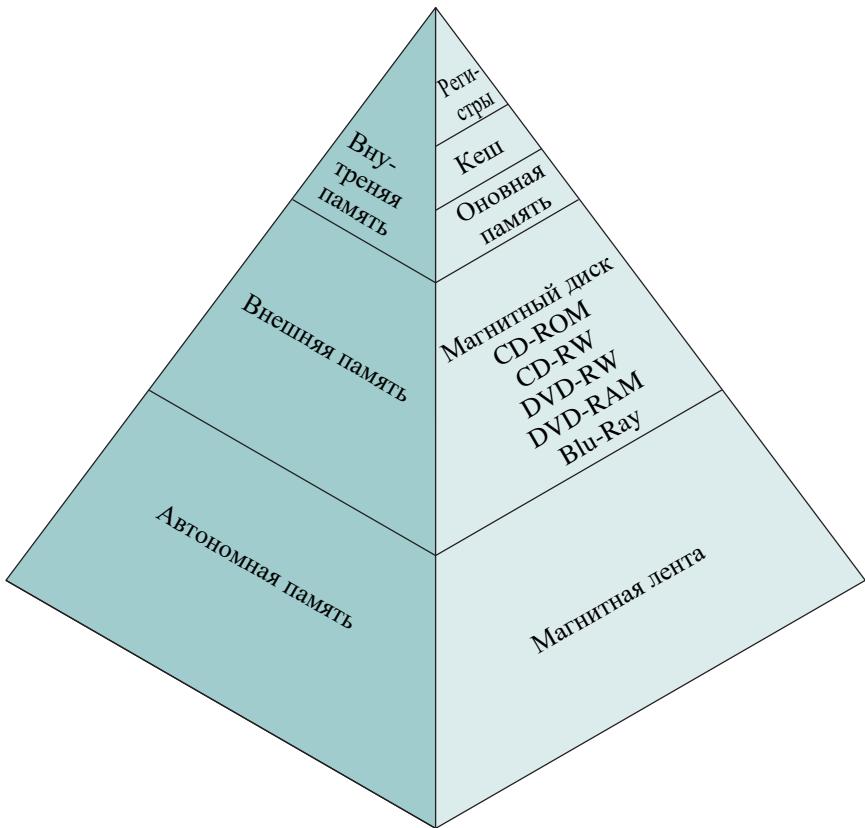
- Чем меньше время доступа, тем дороже каждый бит.
- Чем выше емкость, тем ниже стоимость бита.
- Чем выше емкость, тем больше время доступа.

Теперь ясна дилемма, стоящая перед конструктором. Вследствие возрастания требований к ресурсам конструктор стремится использовать те технологии, которые обеспечивают производство устройств с большой емкостью. Однако, чтобы удовлетворить потребности в высокой производительности, конструктор должен использовать дорогую память с меньшей емкостью (зато и с меньшим временем доступа).

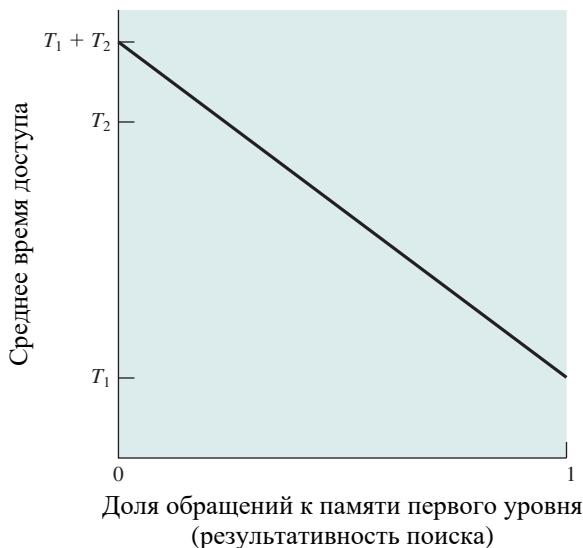
Чтобы найти выход из сложившейся ситуации, необходимо не опираться на отдельно взятый компонент памяти или технологию, а выстроить **иерархию запоминающих устройств**. На рис. 1.14 показана типичная иерархическая пирамида. При спуске к ее основанию происходит следующее.

1. Снижается стоимость бита.
2. Возрастает емкость.
3. Возрастает время доступа.
4. Снижается частота обращений процессора к памяти.

Таким образом, к более дорогим устройствам с меньшей емкостью и более высокой производительностью добавляются более емкие и дешевые, но менее производительные. При такой организации ключом к успеху является снижение частоты обращения к нижним уровням иерархии. Эта концепция будет рассмотрена более детально в следующих разделах данной главы, при обсуждении кеша и виртуальной памяти. А теперь рассмотрим конкретный пример.



**Рис. 1.14.** Иерархия запоминающих устройств



**Рис. 1.15.** Производительность простой двухуровневой системы

Предположим, процессор имеет доступ к памяти двух уровней. На первом уровне содержится 1000 байт, и он характеризуется временем доступа 0,1 мкс; второй уровень содержит 100000 байт со временем доступа 1 мкс. Будем считать, что к байтам, расположенным на первом уровне, процессор может обращаться непосредственно. Если же нужно получить доступ к байту, находящемуся на втором уровне, то этот байт сначала передается на первый уровень, и лишь потом процессор получает к нему доступ. Опустим для простоты вопрос о времени, которое требуется процессору для определения, на каком именно уровне находится слово. На рис. 1.15 показан общий вид кривой, описывающей данную ситуацию. На графике представлена зависимость среднего времени доступа к памяти второго уровня от **результативности поиска (hit ratio)**  $H$ . Символом  $H$  обозначено отношение числа находлений нужного слова в быстрой памяти (например, в кеше) к числу всех обращений к памяти,  $T_1$  — время доступа к первому уровню,  $T_2$  — время доступа ко второму уровню<sup>4</sup>. Видно, что при высокой частоте обращений к уровню 1 среднее время доступа намного ближе ко времени доступа к уровню 1, чем ко времени доступа к уровню 2.

Предположим, что в нашем примере 95% обращений к памяти приходится на кеш ( $H = 0,95$ ). Тогда среднее время доступа можно записать как

$$0,95 \times 0,1 \text{ мкс} + 0,05 \times (0,1 \text{ мкс} + 1 \text{ мкс}) = 0,095 + 0,055 = 0,15 \text{ мкс}$$

Получившийся результат довольно близок ко времени доступа к быстрой памяти. Таким образом, описанная стратегия работает, но лишь при соблюдении приведенных выше условий 1–4. Существует целый ряд разнообразных запоминающих устройств, созданных с применением различных технологий, для которых выполняются условия 1–3. К счастью, условие 4 в общем случае тоже справедливо.

Основой соблюдения условия 4 служит принцип, известный как **принцип локальности обращений** [60]. Адреса (как команд, так и данных), к которым во время выполнения программы обращается процессор, имеют тенденцию собираться в группы. Как правило, программы содержат в себе повторяющиеся циклы и подпрограммы. Как только наступает очередь цикла или подпрограммы, процессор обращается лишь к небольшому повторяющемуся набору команд. Работа с таблицами и массивами также предполагает доступ к сгруппированным данным. С течением времени одни используемые кластеры заменяются другими, но в течение небольших промежутков времени процессор работает преимущественно с фиксированными кластерами памяти.

Поэтому представляется возможным организовать данные в иерархической структуре так, чтобы частота обращений к каждому более низкому уровню была намного меньше, чем частота обращения к уровню, расположенному на ступень выше. Рассмотрим уже описанный пример с двумя уровнями памяти. Пусть на уровне 2 находятся все команды и данные программы. Обрабатываемые в данный момент кластеры можно поместить на уровень 1. При этом получится, что большинство обращений происходит к командам и данным, находящимся на уровне 1.

Этот принцип можно применять не только к памяти с двумя уровнями. Самую быструю, малоемкую и дорогую память образуют внутренние регистры процессора. Обычно в нем имеется несколько десятков таких регистров, хотя в некоторых процессорах могут быть сотни регистров. Спустимся на две ступени вниз, туда, где находится основ-

<sup>4</sup> Назовем *попаданием* такое обращение к памяти, при котором нужный байт находится в быстрой памяти. Соответственно *промахом* назовем ситуацию, когда его там нет.

ная память, являющаяся важным внутренним запоминающим устройством компьютера. Каждая ее ячейка характеризуется своим уникальным адресом, и при выполнении большинства машинных команд обращения происходят по одному или нескольким адресам. Расширением основной памяти служит кеш, который обладает меньшей емкостью, но является более быстрым. Кеш обычно не виден программисту или, что, по сути, то же, процессору. Это устройство выступает в роли промежуточного хранилища данных при их перемещении между основной памятью и регистрами процессора и позволяет повысить производительность.

Три только что описанных вида памяти являются обычно временными (при отключении питания находящаяся в них информация исчезает) и производятся на основе полупроводниковой технологии. Наличие этих уровней основано на том, что существуют различные по скорости и стоимости виды полупроводниковой памяти. Внешние устройства хранения большого объема данных являются постоянными, а наиболее часто встречающиеся из них — жесткий диск и такие съемные устройства, как переносные диски, ленты и оптические запоминающие устройства. Внешнюю, постоянную память называют также **вторичной или вспомогательной памятью**. Такие устройства используются для хранения файлов с программами и данными, и они доступны программисту чаще всего только на уровне файлов и записей, а не отдельных байтов или слов. Жесткий диск используется также как расширение основной памяти, известное под названием “виртуальная память” (см. главу 8, “Виртуальная память”).

В описываемую иерархическую структуру памяти программное обеспечение может добавлять и дополнительные уровни. Например, часть основной памяти может быть использована как буфер для временного хранения считываемых с диска данных. Повышение производительности с применением такой технологии, которую иногда называют **дисковым кешем** (подробнее она рассматривается в главе 11, “Управление вводом-выводом и планирование дисковых операций”), достигается двумя путями.

1. Данные на диск записываются в виде кластеров. Вместо передачи большого количества маленьких порций данных мы считываем несколько больших порций. Это повышает производительность диска и сводит к минимуму использование процессора.
2. Программа может обращаться к некоторым данным, предназначенным для записи на диск, до начала этой записи. В этом случае намного быстрее искать данные в кеше, чем на медленном диске.

Производительность памяти с несколькими уровнями рассматривается в приложении к данной главе.

## 1.6. Кеш

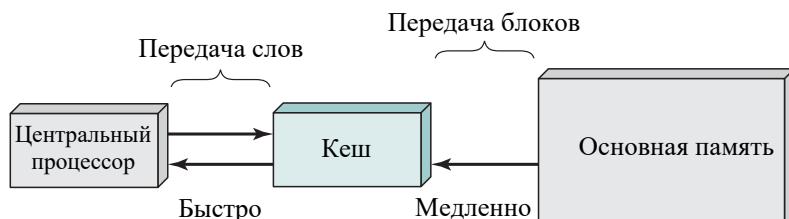
Хотя кеш и невидим для операционной системы, он взаимодействует с аппаратным обеспечением, связанным с памятью. Более того, многие из принципов, используемых в схемах виртуальной памяти (см. главу 8, “Виртуальная память”), применимы также к кеш-памяти.

## Обоснование

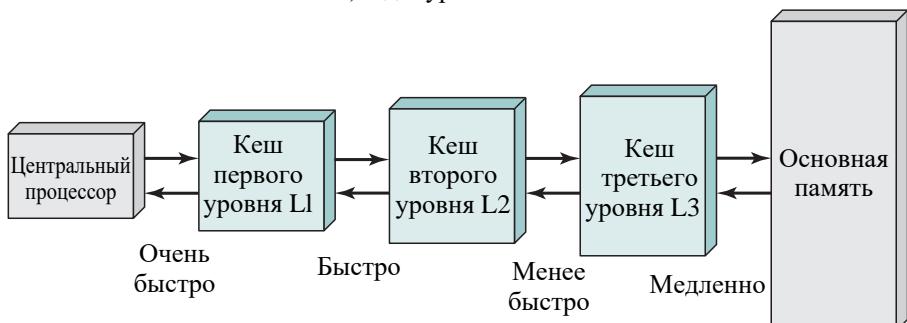
При выполнении каждого цикла команды процессор по крайней мере один раз обращается к памяти, чтобы произвести выборку команды. Часто это происходит повторно, причем возможны случаи нескольких повторных обращений, при которых извлекаются операнды и/или сохраняются результаты. Очевидно, что частота, с которой процессор выполняет команды, ограничена временем обращения к памяти. Это ограничение было существенной проблемой из-за постоянного несоответствия между скоростью процессора и скоростью доступа к основной памяти — в течение многих лет скорость процессора возрастала быстрее, чем скорость доступа к памяти. Постоянно нужно было искать компромисс между скоростью, стоимостью и емкостью. В идеале основная память должна была бы производиться по той же технологии, что и регистры процессора, чтобы время цикла памяти было сравнимо со временем цикла процессора. Однако эта стратегия приводит к слишком большой стоимости. Решением проблемы стало использование принципа локальности, в соответствии с которым между процессором и основной памятью помещается память с небольшой емкостью и быстрым временем доступа, а именно — кеш.

## Принципы работы кеша

Кеш предназначен для того, чтобы приблизить скорость доступа к памяти к максимально возможной и в то же время обеспечить большой объем памяти по цене более дешевых типов полупроводниковой памяти. Эта концепция представлена на рис. 16, а.



а) Одноуровневый кеш



б) Организация трехуровневого кеша

**Рис. 1.16.** Кеш и основная память

Наряду с относительно большой и более медленной основной памятью у нас есть кеш, обладающий меньшей емкостью, но и меньшим временем доступа. В кеше хранится копия фрагмента основной памяти. Когда процессор пытается прочесть байт или слово из памяти, выполняется проверка на наличие этого слова в кеше. Если оно там есть, этот байт или слово передается процессору. Если же его там нет, в кеш считывается блок основной памяти, состоящий из слов с определенными адресами, после чего требуемый байт или слово передается процессору. Вследствие локальности обращений при считывании в кеш блока данных, содержащего одно из требуемых слов, последующие обращения к данным с высокой вероятностью тоже будут выполняться к словам из этого блока.

На рис. 1.16,б изображен кеш, состоящий из нескольких уровней. Кеш L2 медленнее и обычно больше, чем кеш L1, а кеш L3 медленнее и обычно больше, чем кеш L2.

На рис. 1.17 показана структура основной памяти и кеша. Основная память состоит из  $2^n$  адресуемых слов, каждое из которых характеризуется своим уникальным  $n$ -битовым адресом. Для целей отображения предполагается, что вся память состоит из определенного количества блоков фиксированной длины, в каждый из которых входит  $K$  слов.

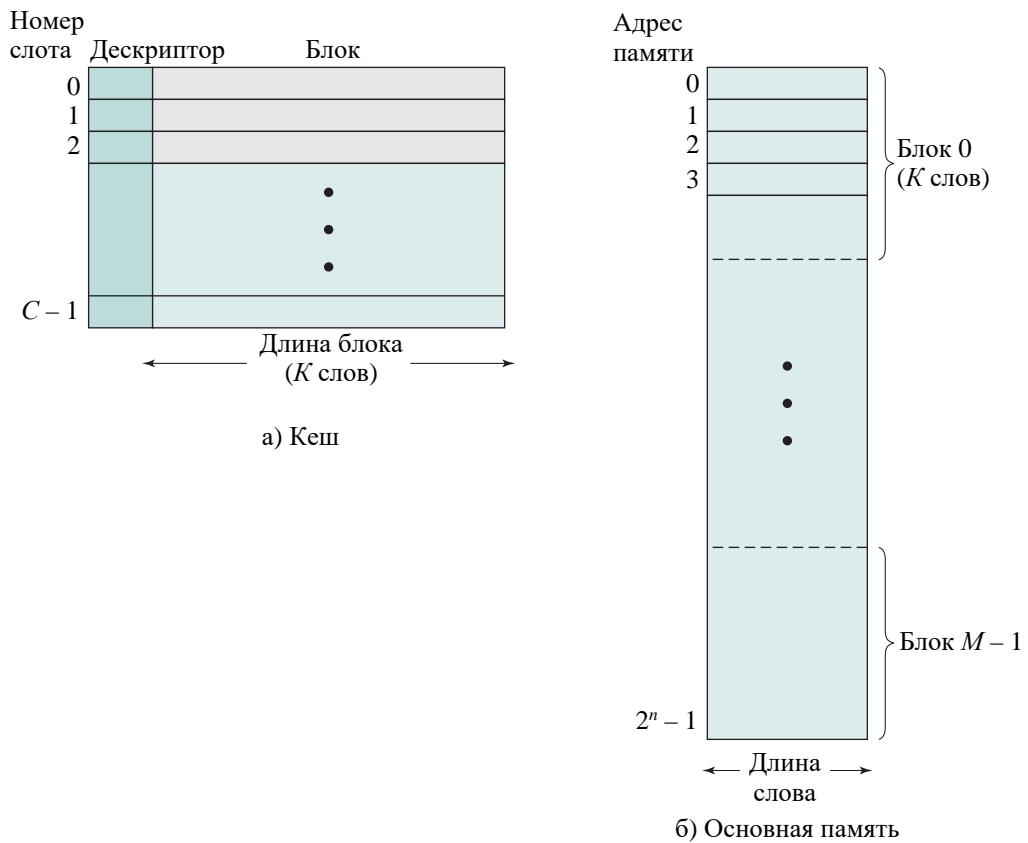


Рис. 1.17. Структура кеша и основной памяти

Таким образом, всего имеется  $M=2^n/K$  блоков. Кеш состоит из  $C$  слотов (именуемых также линиями) по  $K$  слов. При этом количество слотов намного меньше количества блоков ( $C \ll M$ )<sup>5</sup>. Некоторое подмножество блоков основной памяти хранится в слотах кеша. Если нужно прочесть из памяти слово из какого-то блока, которого нет в кеше, то этот блок передается в один из слотов кеша. Из-за того что блоков больше, чем слотов, нельзя закрепить за каждым блоком свой слот. Поэтому каждый слот должен содержать дескриптор, идентифицирующий хранящийся в нем блок. В роли дескриптора обычно выступает некоторое число, состоящее из старших битов адреса, и по нему происходит обращение ко всем адресам, которые начинаются этой последовательностью битов.

Рассмотрим простой пример, в котором адреса состоят из шести битов, а дескрипторы — из двух. Дескриптор 01 указывает на то, что в слоте находится блок, в который входят следующие адреса: 010000, 010001, 010010, 010011, 010100, 010101, 010110, 010111, 011000, 011001, 011010, 011011, 011100, 011101, 011110, 011111.

На рис. 1.18 показана блок-схема операции чтения слова из памяти.

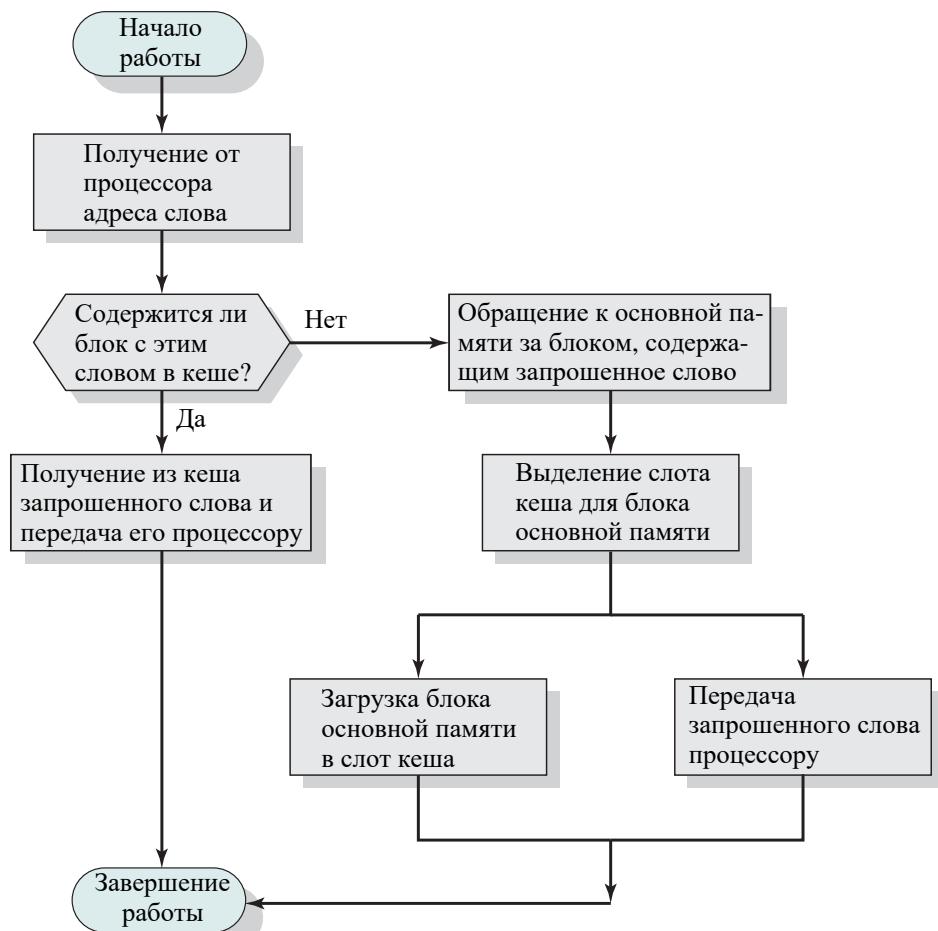


Рис. 1.18. Операция чтения из кеша

<sup>5</sup> Символ  $\ll$  означает “намного меньше, чем”, а символ  $\gg$  — “намного больше, чем”.

Процессор генерирует адрес слова, которое нужно прочесть. Если это слово хранится в кеше, оно передается процессору. В противном случае блок, содержащий это слово, загружается в кеш, и слово передается процессору.

## Внутреннее устройство кеша

В данной книге внутреннее устройство кеша подробно не рассматривается. В этом разделе кратко перечислены лишь основные его элементы. В дальнейшем читатель сможет убедиться, что при изучении устройства виртуальной памяти и дискового кеша мы имеем дело с похожими вопросами. Все их можно разбить на следующие категории:

- размер кеша;
- размер блока;
- функция отображения;
- алгоритм замещения;
- стратегия записи;
- количество уровней кеша.

С такой характеристикой, как **размер кеша**, мы уже знакомы. Оказывается, что даже сравнительно маленький кеш может оказывать значительное влияние на производительность компьютера. Другим важным параметром является **размер блока**, задающий величину порции данных, которая передается из основной памяти в кеш. Рассмотрим события, начиная с относительно небольшого размера блоков, а затем увеличивая его. При увеличении размера блока в кеш попадает больше полезных данных при каждой их передаче. В результате результативность поиска растет в соответствии с принципом локальности обращений: вероятность в ближайшем будущем обращения к данным, находящимся недалеко от слова, к которому было выполнено обращение, достаточно высока. Однако есть некое предельное значение, при превышении которого результативность поиска начинает уменьшаться. Это происходит тогда, когда вероятность использования вновь считанных данных становится меньше, чем вероятность повторного использования данных, которые необходимо удалить из кеша, чтобы освободить место для нового блока.

При считывании в кеш нового блока данных **функция отображения** определяет, какое место будет отведено для этого блока. Разрабатывая эту функцию, необходимо учитывать два фактора, накладывающих на нее определенные ограничения. Во-первых, при считывании блока, вероятно, он заменит другой блок в кеше. Хотелось бы сделать это таким образом, чтобы свести к минимуму вероятность того, что заменяемый блок понадобится в ближайшем будущем. Чем более гибкой является функция отображения, тем больше возможностей для разработки такого алгоритма замены, который позволил бы увеличить результативность поиска. Во-вторых, с увеличением гибкости функции отображения должны усложняться схемы, позволяющие определить наличие в кеше требуемой информации и обеспечить ее поиск.

При загрузке блоков в кеш в конце концов наступает момент, когда все слоты заполняются и новый блок нужно записывать на место, занятое каким-то другим блоком. Выбор этого блока осуществляется в соответствии с алгоритмом замещения, на который накладывает ограничения отображающая функция. При этом желательно было бы убрать именно тот блок, который, скорее всего, не понадобится в ближайшем будущем. Хотя

достоверно определить его невозможно, достаточно эффективной стратегией является замена блока, к которому дольше всего не было обращений. Такая стратегия называется алгоритмом давно неиспользованных блоков (*least-recently-used* — LRU). Для определения используемости блоков необходим соответствующий аппаратно реализованный механизм.

Если содержимое блока в кеше изменилось, его необходимо записать в основную память до замены другим блоком. Случаи, когда нужно выполнять операции записи, определяются **стратегией записи**. Одним из предельных случаев является стратегия, при которой запись производится при каждом обновлении блока. В другом случае запись производится только при замене данного блока новым. Такая стратегия сводит к минимуму количество операций записи в память, но при этом в блоках основной памяти содержится устаревшая информация, что может привести к ошибкам при многопроцессорной работе, а также при прямом доступе к памяти со стороны модулей ввода-вывода.

Наконец, в настоящее время распространены многоуровневые кеши, имеющие названия “L1” (ближайший к процессору кеш) и “L2”, а во многих случаях и “L3”. Обсуждение преимуществ производительности многоуровневых кешей выходит за рамки данной книги (см. обсуждение этого вопроса в [242]).

## 1.7. Прямой доступ к памяти

Возможны три метода выполнения операций ввода-вывода: программируемый ввод-вывод, ввод-вывод с использованием прерываний и прямой доступ к памяти (*direct memory access* — DMA). Перед тем как приступить к изучению DMA, бегло рассмотрим два других метода (подробности — в приложении B, “Дополнительные вопросы параллельности”).

Когда процессору при выполнении программы встречается команда, связанная с вводом-выводом, он выполняет ее, передавая соответствующие команды контроллеру ввода-вывода. При **программируемом вводе-выводе** это устройство выполняет требуемое действие, а затем устанавливает соответствующие биты в регистрах состояния ввода-вывода. Контроллер ввода-вывода больше не посылает процессору никаких сигналов, в том числе сигналов прерываний. Таким образом, после выполнения команды ввода-вывода процессор должен играть активную роль в выяснении, когда завершится команда ввода-вывода. Для этого он должен производить проверку состояния модуля ввода-вывода до тех пор, пока операция не завершится.

При использовании такого метода процессор должен длительное время ожидать, когда модуль ввода-вывода будет готов к приему или передаче данных. Процессор во время ожидания должен постоянно опрашивать состояние модуля ввода-вывода. В результате существенно снижается уровень производительности всей системы.

При альтернативном подходе, известном как **управляемый прерываниями ввод-вывод**, процессор может передать контроллеру команду ввода-вывода, а затем перейти к выполнению другой полезной работы. Затем, когда контроллер ввода-вывода снова будет готов обмениваться данными с процессором, он прервет процессор и потребует, чтобы его обслужили. Процессор передает ему новые данные, а затем возобновляет прерванную работу.

Хотя ввод-вывод, управляемый прерываниями, более эффективен, чем простой программируемый ввод-вывод, он все еще занимает много процессорного времени для пе-

редачи данных между памятью и контроллером ввода-вывода (при этом через процессор должны пройти все пересылаемые данные). Таким образом, обе описанные формы ввода-вывода обладают двумя неотъемлемыми недостатками:

1. скорость передачи данных при вводе-выводе ограничена скоростью, с которой процессор может проверять и обслуживать устройство;
2. процессор занят организацией передачи данных; при вводе-выводе для каждой передачи данных должна быть выполнена определенная последовательность команд.

Для перемещения больших объемов данных требуется более эффективный метод — **прямой доступ к памяти** (direct memory access — DMA). Функции DMA могут выполняться отдельным контроллером системной шины или могут быть встроены в контроллер ввода-вывода. В любом случае метод работает следующим образом. Когда процессору нужно прочитать или записать блок данных, он генерирует команду для модуля DMA, посыпая ему следующую информацию:

- указание, требуется ли выполнить чтение или запись;
- адрес устройства ввода-вывода;
- начальный адрес блока памяти, использующегося для чтения или записи;
- количество слов, которые должны быть прочитаны или записаны.

Делегировав полномочия по выполнению этих операций контроллеру DMA, процессор продолжает работу. Контроллер DMA слово за словом передает весь блок данных в память или из нее, минуя при этом процессор. После окончания передачи контроллер DMA посылает процессору сигнал прерывания. Таким образом, процессор участвует только в начале и в конце передачи.

Для передачи данных в память и из нее контроллеру DMA нужен контроль надшиной. Если в это время процессору также нужна шина, может возникнуть конфликтная ситуация, и процессор должен ждать окончания работы модуля DMA. Заметим, что в этом случае нельзя говорить о прерывании, так как процессор не сохраняет информацию о состоянии задачи и не переходит к выполнению других операций. Вместо этого он вынужден сделать паузу на время выполнения одного цикла шины. В результате это приведет к тому, что во время передачи данных с использованием прямого доступа к памяти замедляется выполнение процессором тех команд, для которых ему требуется шина. Тем не менее при передаче большого количества информации прямой доступ к памяти намного более эффективен, чем программируемый ввод-вывод или ввод-вывод, управляемый прерываниями.

## 1.8. ОРГАНИЗАЦИЯ МНОГОПРОЦЕССОРНЫХ И МНОГОЯДЕРНЫХ СИСТЕМ

Традиционно компьютер рассматривается как машина, предназначенная для выполнения последовательных действий. В большинстве языков программирования алгоритм задается в виде последовательных инструкций; при работе программы процессор выполняет машинные команды последовательно, одну за другой. Каждая команда представляется в виде последовательности операций (выборка команды, выборка операндов, выполнение операции, сохранение результатов).

Такая точка зрения на компьютер никогда не соответствовала действительности полностью. На уровне микроопераций одновременно генерируются несколько управляющих сигналов. Уже давно применяется конвейерная обработка команд, позволяющая выполнять одновременно по крайней мере операции выборки и выполнения. Оба приведенных примера являются образцами параллельного выполнения функций.

По мере развития компьютерных технологий и уменьшения стоимости аппаратного обеспечения разработчики компьютеров находили все больше и больше возможностей реализации параллелизма. Обычно это делалось для повышения производительности, а в некоторых случаях — для повышения надежности. В данной книге исследуются три наиболее популярных подхода к обеспечению параллелизма в многопроцессорных системах: симметричная многопроцессорность (*symmetric multiprocessor — SMP*), многоядерные компьютеры и кластеры. Симметричная многопроцессорная обработка и многоядерные компьютеры рассматриваются в этом разделе, а кластеры — в главе 16, “Облачные операционные системы и операционные системы Интернета вещей”.

## Симметричная многопроцессорность

*Определение.* SMP можно определить как изолированную компьютерную систему, обладающую следующими характеристиками.

1. Имеется не менее двух похожих процессоров со сравнимыми возможностями.
2. Эти процессоры используют одну и ту же основную память и устройства ввода-вывода и соединены шиной или иной схемой внутреннего соединения, так что время доступа к памяти оказывается примерно одинаковым для каждого процессора.
3. Все процессоры имеют общий доступ к устройствам ввода-вывода либо через одни и те же каналы, либо через различные каналы, которые обеспечивают пути к одному устройству.
4. Все процессоры могут выполнять одни и те же функции (отсюда и термин *симметричная*).
5. Система управляется интегрированной операционной системой, которая обеспечивает взаимодействие между процессорами и их программами на уровне задач, заданий, файлов и элементов данных.

Пункты 1–4 не должны вызывать непонимания. Пункт 5 иллюстрирует отличие от слабосвязанной многопроцессорной системы, такой как кластер. В последнем физической единице взаимодействия обычно является сообщение или полный файл. В SMP уровень взаимодействия могут представлять собой отдельные элементы данных, и может иметься высокая степень взаимодействия между процессами.

Организация SMP имеет ряд потенциальных преимуществ по сравнению с однопроцессорной организацией, включая следующие.

- **Производительность.** Если работа, которую должен выполнить компьютер, может быть организована таким образом, что некоторые ее части могут быть выполнены параллельно, то система с несколькими процессорами даст большую производительность, чем с одним процессором того же типа.
- **Надежность.** В симметричной многопроцессорной системе, в которой все процессоры могут выполнять одни и те же функции, отказ одного процессора не оста-

навливает машину. Вместо этого система может продолжать функционировать с пониженной производительностью.

- **Инкрементный рост.** Пользователь может повысить производительность системы путем добавления дополнительного процессора.
- **Масштабирование.** Поставщики могут предложить широкий спектр продуктов с различной ценой и производительностью, меняя количество процессоров, установленных в системе.

Важно отметить, что это потенциальные, а не гарантируемые преимущества. Операционная система должна предоставлять инструментарии и функции для эффективного использования параллельности в SMP-системе.

Привлекательной особенностью SMP является то, что существование нескольких процессоров является прозрачным для пользователя. Операционная система берет на себя планирование заданий отдельных процессоров и синхронизацию между процессорами.

*Организация.* На рис. 1.19 проиллюстрирована общая организация SMP. Имеется несколько процессоров, каждый из которых содержит собственный блок управления, арифметико-логическое устройство и регистры. Обычно каждый процессор имеет два выделенных уровня кеша, обозначенные как L1 и L2.

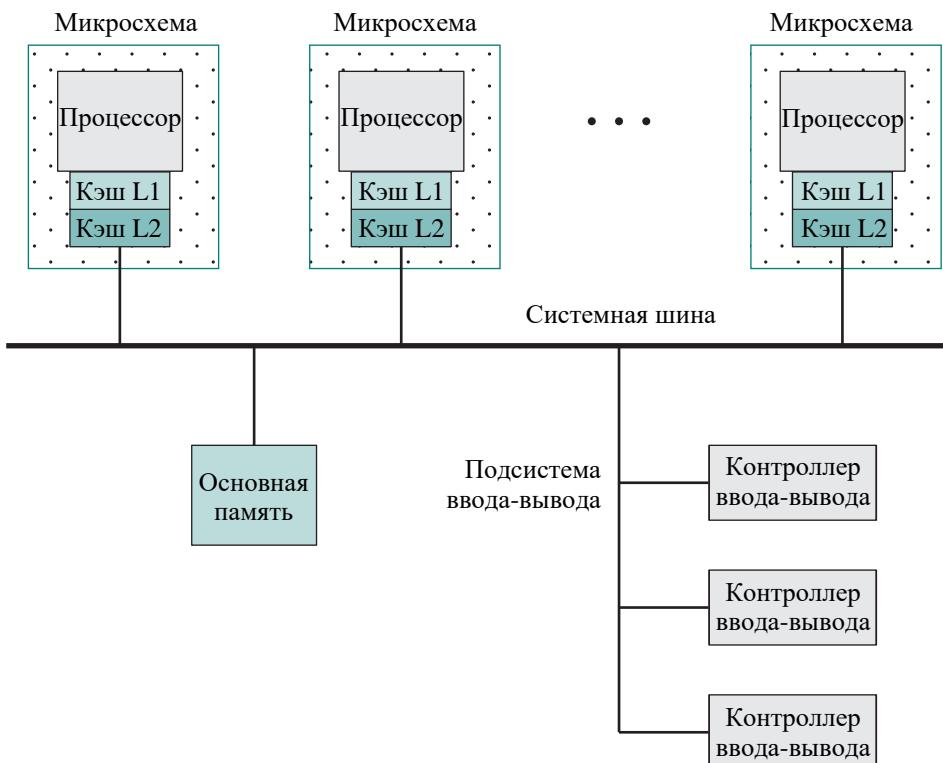


Рис. 1.19. Организация SMP

Как показано на рис. 1.19, каждый процессор и его кеши размещаются в отдельной микросхеме. Каждый процессор имеет доступ к общей основной памяти и устройствам ввода-вывода посредством некоторого механизма взаимосвязи; общая шина является совместно используемым объектом. Процессоры могут общаться друг с другом через память (сообщения и информация о состоянии остаются в общем адресном пространстве). Может также быть возможным непосредственный обмен процессоров сигналами. Память часто организована так, что возможны несколько одновременных доступов к разным блокам памяти.

В современных компьютерах процессоры, как правило, имеют по крайней мере один уровень кеш-памяти, принадлежащий только данному процессору. Такое использование кеша наводит на некоторые новые соображения относительно дизайна. Поскольку каждый локальный кеш содержит отражение части основной памяти, изменение слова в одном кеше может сделать недействительным слово в другом кеше. Для предотвращения этого другие процессоры должны быть предупреждены, что имело место обновление. Эта проблема известна как проблема согласованности кешей и обычно решается аппаратными средствами, а не операционной системой<sup>6</sup>.

## Многоядерные компьютеры

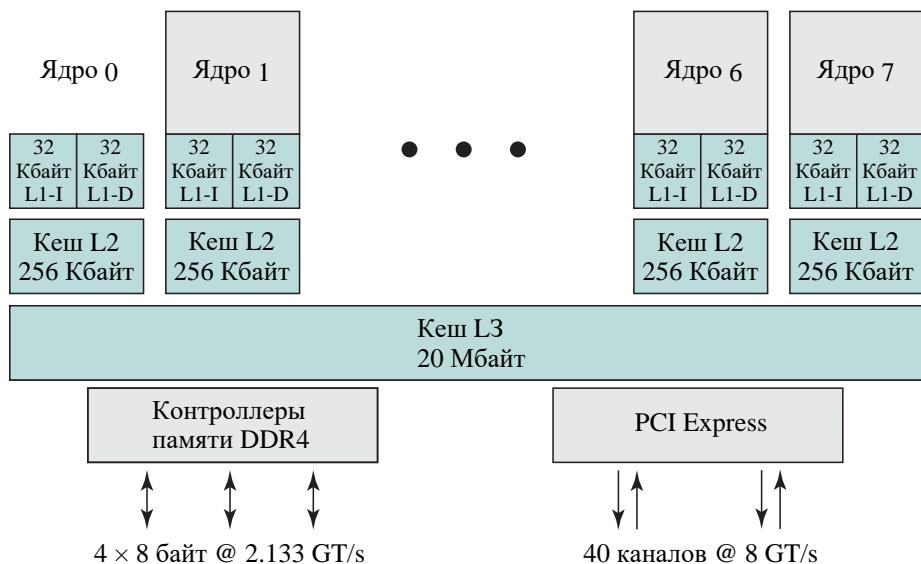
**Многоядерные** компьютеры, известные также как компьютеры с несколькими процессорами в одной микросхеме (*chip multiprocessor*), сочетают в себе несколько процессоров (так называемых ядер) в одном корпусе. Как правило, каждое ядро содержит все компоненты независимого процессора, такие как регистры, арифметико-логическое устройство (АЛУ), аппаратный конвейер и блок управления, а также кеш L1 команд и данных. Помимо нескольких ядер, современные многоядерные микросхемы содержат кеш L2, а в некоторых случаях — и кеш L3.

Мотивацию развития многоядерных компьютеров можно подытожить следующим образом. На протяжении десятилетий микропроцессорные системы испытывали устойчивое, практически экспоненциальное увеличение производительности. Это частично обусловлено тенденциями развития аппаратных средств, такими как увеличение тактовой частоты и возможность размещения кеш-памяти ближе к процессору из-за большей степени миниатюризации компонентов микрокомпьютера. Производительность росла также за счет повышенной сложности дизайна процессоров для использования параллелизма при выполнении команд и доступа к памяти. Вкратце, проектировщики столкнулись с практическими ограничениями возможностей достижения большей производительности с помощью более сложных процессоров и нашли, что наилучший способ повышения производительности за счет аппаратных возможностей состоит в установке нескольких процессоров и значительного объема кеш-памяти на одном чипе. Подробное обсуждение этого вопроса выходит за рамки данной книги, но подытоживается в приложении В, “Дополнительные вопросы параллельности”.

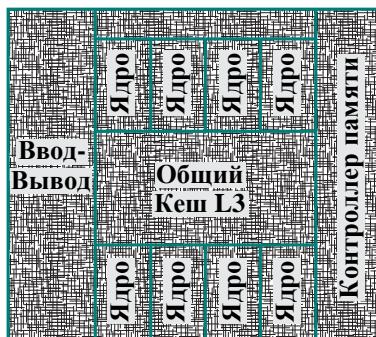
Примером многоядерной системы является Intel Core i7-5960X, который включает в себя восемь процессоров x86, каждый с выделенным кешем L2 и общим кешем L3 (рис. 1.20, а). Один из механизмов, использованных Intel, чтобы сделать кеши более эффективными, — предвыборка, при которой оборудование изучает шаблон обращений к памяти и пытается заполнить кеш данными, которые могут быть вскоре затребованы. На рис. 1.20, б показано физическое расположение компонентов процессора в микросхеме.

<sup>6</sup> Описание вариантов аппаратного решения согласованности кешей имеется в [242].

Core i7-5960X поддерживает две разновидности внешних коммуникаций с другими схемами. Контроллер памяти DDR4 обеспечивает двойную скорость передачи данных (double data rate — DDR) между основной памятью и схемой. Интерфейс поддерживает четыре канала шириной 8 байт для общей ширины шины 256 бит со скоростью передачи данных до 64 Гбайт/с. При наличии контроллера памяти в чипе устраняется необходимость вшине Front Side Bus. PCI Express представляет собой периферийную шину и обеспечивает высокоскоростную связь между подключенными к процессору схемами. PCI Express работает на скорости 8 GT/s (8 миллиардов обменов в секунду). При 40 битах на обмен скорость составляет до 40 Гбайт/с.



а) Блок-схема



б) Физическое размещение на плате

Рис. 1.20. Блок-схема Intel Core i7-5960X

## 1.9. КЛЮЧЕВЫЕ ТЕРМИНЫ, КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАЧИ

### Ключевые термины

Алгоритм замещения	Многопроцессорность	Системная шина
Блок	Многоядерность	Слот
Ввод-вывод	Основная память	Слот кеша
Ввод-вывод, управляемый прерываниями	Прерывание	Стек
Временная локальность	Программируемый ввод-вывод	Схема микропроцессора
Вспомогательная память	Пространственная локальность	Счетчик команд
Вторичная память	Процессор	Указатель стека
Иерархия памяти	Прямой доступ к памяти (DMA)	Центральный процессор
Кадр стека	Регистр	Цикл команды
Кеш-память	Регистр адреса	
Команда	Регистр данных	
Контроллер ввода-вывода	Регистр команд	
	Результативность поиска	

### Контрольные вопросы

- 1.1. Перечислите и кратко определите четыре основных элемента компьютера.
- 1.2. Определите две основные категории регистров процессора.
- 1.3. Каковы, в общих чертах, четыре различных действия, которые может указывать машинная команда?
- 1.4. Что такое прерывание?
- 1.5. Как обрабатываются множественные прерывания?
- 1.6. Какие характеристики отличают различные элементы иерархии памяти?
- 1.7. Что такое кеш-память?
- 1.8. В чем состоит разница между многопроцессорными и многоядерными системами?
- 1.9. В чем состоит разница между пространственной и временной локальностью?
- 1.10. Каковы в общем случае стратегии использования пространственной и временной локальности?

## Задачи

- 1.1. Предположим, что в гипотетической машине, изображенной на рис. 1.3, кроме указанных, имеются такие команды:

0011 — загрузить в аккумулятор данные, поступившие от устройства ввода-вывода;

0111 — вывести содержимое аккумулятора на устройство ввода-вывода.

При использовании этих команд устройство идентифицируется с помощью 12-битового адреса. Изобразите схему выполнения (по аналогии со схемой, представленной на рис. 1.4) следующей программы.

1. Загрузить аккумулятор данными из устройства номер 5.
2. Добавить к аккумулятору содержимое ячейки памяти 940.
3. Вывести содержимое аккумулятора на устройство номер 6.

Решите задачу при условии, что из устройства номер 5 поступит число 3, а в ячейке 940 находится число 2.

- 1.2. На рис. 1.4 выполнение программы разбито на шесть этапов. Расширьте это описание, добавив шаги с использованием регистров MAR и MBR.

- 1.3. Рассмотрим гипотетический 32-битовый микропроцессор, 32-битовые команды которого состоят из двух полей. В первом байте содержится код команды, а в остальной части команды — непосредственно операнд или его адрес.

- a. Какова максимально возможная емкость адресуемой памяти (в байтах)?
- b. Рассмотрите факторы, влияющие на скорость системы, если шина микропроцессора имеет 1) 32-битовую локальную адресную шину и 16-битовую локальную шину данных или 2) 16-битовую локальную адресную шину и 16-битовую локальную шину данных.
- c. Сколько битов требуется для счетчика команд и регистра команд?

- 1.4. Рассмотрим гипотетический микропроцессор, генерирующий 16-битовые адреса (предположим, например, что счетчик команд и адресные регистры имеют размер 16 бит) и обладающий 16-битовой шиной данных.

- a. Какое максимальное адресное пространство памяти может быть непосредственно доступно этому процессору, если он соединен с “16-битовой памятью”?
  - b. Какое максимальное адресное пространство может быть непосредственно доступно этому процессору, если он соединен с “8-битовой памятью”?
  - c. Какие особенности архитектуры позволяют этому микропроцессору получить доступ к отдельному “пространству ввода-вывода”?
  - d. Сколько портов ввода-вывода способен поддерживать этот микропроцессор, если в командах ввода и вывода задаются 8-битовые номера портов? Сколько портов ввода-вывода он может поддерживать с 16-битовыми портами?
- Поясните свой ответ.

- 1.5. Рассмотрим 32-битовый микропроцессор с 16-битовой внешней шиной данных, которая управляется синхронизатором с тактовой частотой 8 МГц. Пусть цикл шины этого микропроцессора по длительности равен четырем циклам синхронизатора. Какую максимальную скорость передачи данных может поддерживать этот процессор? Что будет лучше для повышения производительности: сменить его внешнюю шину данных на 32-битовую или удвоить частоту сигнала синхронизатора, поступающего на микропроцессор? Внесите свое предложение и обоснуйте его.  
Указание: определите количество байтов, которое может быть передано при каждом цикле шины.

- 1.6. Рассмотрим компьютерную систему, в которой содержится контроллер ввода-вывода, управляющий простейшим интерфейсом пользователя, представляющим собой телетайп “клавиатура/принтер”. В процессоре находятся следующие регистры, непосредственно связанные с системной шиной:

INPR — регистр входных данных, 8 бит;  
OUTR — регистр выходных данных, 8 бит;  
FGI — флаг входа, 1 бит;  
FGO — флаг выхода, 1 бит;  
IEN — регистр разрешения прерываний, 1 бит.

Входной поток данных, поступающий от клавиатуры, и выходной, выводимый на принтер, контролируются модулем ввода-вывода. Телетайп кодирует алфавитно-цифровые символы в 8-битовые слова и декодирует 8-битовые слова в алфавитно-цифровые символы. Флаг входа устанавливается при вводе 8-битового слова с телетайпа во входной регистр; флаг выхода устанавливается при выводе слова на принтер.

- Опишите, как процессор может осуществлять ввод-вывод с телетайпа, используя первые четыре перечисленных регистра.
- Опишите, как это можно сделать более эффективно, используя регистр IEN.

- 1.7. Практически во всех системах, в которые входят контроллеры DMA, доступ DMA к основной памяти выполняется с более высоким приоритетом, чем доступ процессора. Почему?

- 1.8. Контроллер DMA передает символы из внешнего устройства в основную память со скоростью 9600 бит в секунду. Процессор может выбирать команды со скоростью 1 млн команд в секунду. Насколько процессор замедлит свою работу из-за работы DMA?

- 1.9. Компьютер состоит из процессора и устройства ввода-вывода  $D$ , подсоединенного к основной памяти  $M$  через совместно используемую шину, которая используется как шина данных и имеет ширину, равную одному слову. Максимальная производительность процессора —  $10^6$  команд в секунду. Команда включает в себя в среднем пять машинных циклов, для трех из которых используется шина памяти. Операции чтения-записи в памяти включают в себя один машинный цикл. Предположим, что процессор все время выполняет программы в фоновом режиме, что требует 95% его производительности, а в самих программах не содержится ни одной команды ввода-вывода. Пусть длительность цикла процессора равна длительности цикла шины.

Теперь представим, что между  $M$  и  $D$  следует переслать очень большой блок данных.

- Оцените максимальную скорость передачи данных при выполнении операций ввода-вывода, которые могут пройти через  $D$ , при использовании программируемого ввода-вывода, если для операции передачи одного слова требуется выполнение двух команд.
- Оцените ту же скорость при передаче данных с использованием DMA.

#### 1.10. Рассмотрим следующий код.

```
for (i = 0; i < 20; i++)
    for (j = 0; j < 10; j++)
        a[i] = a[i] * j
```

- Приведите пример пространственной локальности этой последовательности.
- Приведите пример временной локальности этой последовательности.

#### 1.11. Обобщите уравнения (1.1) и (1.2) из приложения к данной главе для $n$ -уровневой иерархической структуры памяти.

#### 1.12. Рассмотрим основную память ( $m$ ) и кеш ( $c$ ), характеризующиеся следующими параметрами:

$$\begin{array}{ll} T_c = 100 \text{ нс} & C_c = 0.01 \text{ цент/бит} \\ T_m = 1200 \text{ нс} & C_m = 0.001 \text{ цент/бит} \end{array}$$

- Сколько стоит 1 Мбайт основной памяти?
- Сколько стоит 1 Мбайт основной памяти, выполненной по технологии кеша?
- Какова результативность поиска  $H$ , если эффективное время доступа на 10% больше, чем время доступа к кешу?

#### 1.13. В компьютере есть кеш, основная память и диск, выступающий в роли виртуальной памяти. Если запрашиваемое слово находится не в кеше, а в основной памяти, для его загрузки в кеш требуется 60 нс (сюда входит время, которое требуется для первоначальной проверки кеша). После этого происходит новый запрос. Если слова нет в оперативной памяти, чтобы получить его с диска, необходимо затратить 12 мс, а затем еще 60 нс, чтобы скопировать его в кеш; после этого происходит новый запрос. Результативность поиска в кеше равна 0.9, а результативность поиска в основной памяти — 0.6. Найти среднее время, которое требуется для получения доступа к слову в данной системе.

#### 1.14. Предположим, что при вызове процедур и возврате из них процессор использует стек. Можно ли в такой схеме обойтись без счетчика команд, используя вместо него вершину стека?

# ПРИЛОЖЕНИЕ 1.А. ХАРАКТЕРИСТИКИ ПРОИЗВОДИТЕЛЬНОСТИ ДВУХУРОВНЕВОЙ ПАМЯТИ

В данной главе упоминался кеш, который выступает в роли промежуточного буфера между процессором и основной памятью, что обеспечивает двухуровневую структуру памяти. Производительность работы памяти с такой архитектурой выше, чем у одноровневой памяти. Это повышение производительности достигается за счет свойства, известного как локальность (locality), которое и рассматривается в данном приложении.

Механизм кеширования основной памяти является составной частью компьютерной архитектуры. Он встроен в аппаратное обеспечение и обычно невидим операционной системе. Поэтому кеширование не рассматривается в данной книге. Однако есть еще два примера использования двухуровневой памяти, в которых также используется локальность и которые, по крайней мере частично, реализованы в операционной системе: виртуальная память и дисковый кеш (см. табл. 1.2). Эти темы являются предметом рассмотрения глав 8, “Виртуальная память”, и 11, “Управление вводом-выводом и планирование дисковых операций”, соответственно. Данное приложение поможет читателю познакомиться с некоторыми характеристиками производительности двухуровневой памяти, которые являются общими для всех трех подходов.

**Таблица 1.2. ХАРАКТЕРИСТИКИ ДВУХУРОВНЕВОЙ ПАМЯТИ**

	Кеш основной памяти	Виртуальная память (страничная организация)	Дисковый кеш
<b>Типичное соотношение времени доступа</b>	5:1	$10^6:1$	$10^6:1$
<b>Система управления памятью</b>	Специальное встроенное аппаратное обеспечение	Сочетание аппаратного и программного обеспечения	Системное программное обеспечение
<b>Типичный размер блока</b>	От 4 до 128 байт	От 64 до 4096 байт	От 64 до 4096 байт
<b>Доступ процессора ко второму уровню</b>	Прямой доступ	Косвенный доступ	Косвенный доступ

## ЛОКАЛЬНОСТЬ

Основой для повышения производительности двухуровневой памяти является принцип локальности, о котором шла речь в разделе 1.5. Этот принцип гласит, что последовательные обращения к памяти имеют тенденцию собираться в группы (кластеры). По истечении длительного периода времени один кластер сменяется другим, но на протяжении сравнительно небольших промежутков времени процессор преимущественно работает с адресами, входящими в один и тот же кластер памяти. Интуитивно принцип локальности имеет смысл. Рассмотрим следующую цепочку рассуждений.

1. Команды программы выполняются последовательно, за исключением тех случаев, когда встречаются команды ветвления или вызова (процедуры, функции и т.д.). Поэтому в большинстве случаев команды из памяти извлекаются последовательно, одна за другой в порядке их размещения.

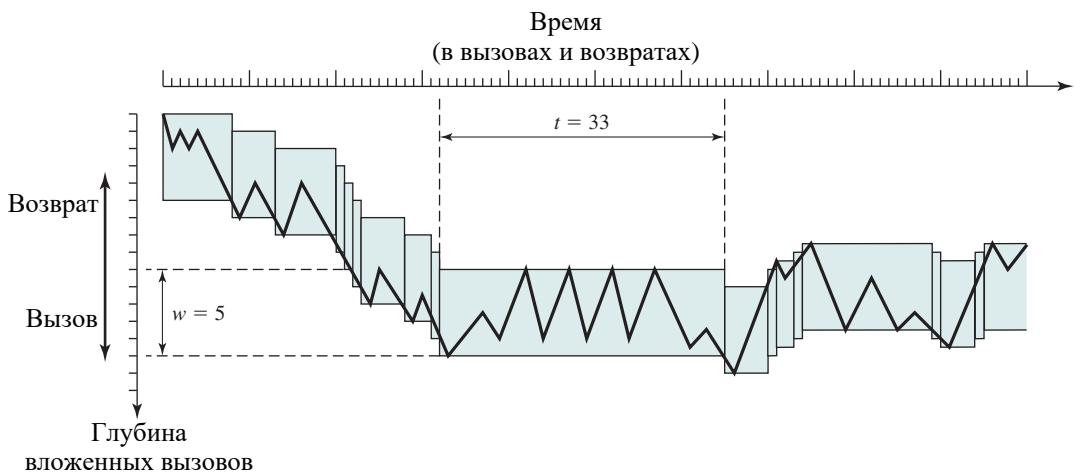
2. Длинная, не нарушающая прерываниями последовательность вызовов процедур, за которой идут соответствующие команды возврата, встречается довольно редко. Другими словами, глубина вложенного вызова процедур остается небольшой. Поэтому в течение короткого промежутка времени обращения, скорее всего, происходят к командам, которые находятся в небольшом числе процедур.
3. Большинство итерационных конструкций состоят из сравнительно небольшого числа многократно повторяющихся команд. Во время выполнения этих итераций вычисления сосредоточены на небольшом локальном участке программы.
4. Во многих программах значительная часть вычислений приходится на операции с такими структурами данных, как массивы и последовательности записей. В большинстве случаев данные, к которым происходит обращение, расположены близко друг к другу.

Приведенные рассуждения подтверждаются многими исследованиями. Для проверки утверждения 1 проводился разносторонний анализ поведения программ, составленных на языках высокого уровня. Результаты измерений частоты появления различных команд при выполнении программы, проведенные разными исследователями, представлены в табл. 1.3. Одно из самых ранних исследований поведения языка программирования проводилось Кнутом (Knuth) [135], рассматривавшим различные программы на языке FORTRAN, написанные студентами при выполнении практических заданий. Таненбаум (Tanenbaum) [253] опубликовал результаты измерений, проведенных более чем на 300 процедурах, которые использовались при разработке операционных систем и были написаны на языке, поддерживающем структурное программирование. Паттерсон (Patterson) и Секвин (Sequin) [185] провели анализ ряда измерений, выполненных над компиляторами, текстовыми редакторами, системами автоматизированного проектирования, программами сортировки данных и сравнения содержимого файлов (языки программирования С и Pascal). Хак (Huck) [112] проанализировал четыре программы, являющиеся типичными примерами программ для проведения научных вычислений. Среди них были, в частности, программы для выполнения быстрого Фурье-преобразования и для решения системы дифференциальных уравнений. Полученные результаты хорошо согласуются с утверждением, что ветвления и команды вызовов представляют собой лишь небольшую часть всех команд, выполняющихся во время работы программы. Таким образом, проведенные исследования подтверждают справедливость приведенного выше утверждения 1.

**Таблица 1.3. Относительная динамическая частота различных операций языков высокого уровня**

Исследование Язык Изучаемый материал	[112]		[135]		[185]		[253] SAL Системное ПО
	Pascal	Научное ПО	FORTRAN	Студенческие работы	Pascal	С	
	Системное ПО	Системное ПО	Системное ПО	Системное ПО	Системное ПО	Системное ПО	
Присваивание	74		67		45	38	42
Цикл	4		3		5	3	4
Вызов	1		3		15	12	12
IF	20		11		29	43	36
GOTO	2		9		—	3	—
Другие	—		7		6		6

Что касается утверждения 2, то его справедливость подтверждается исследованиями, результаты которых изложены в [184]. Это проиллюстрировано на рис. 1.21, где показано поведение программы в разрезе используемых в ней вызовов процедур и возврата из них. Каждый вызов представлен отрезком линии, идущим на одно деление вниз, а каждый возврат — отрезком, идущим на одно деление вверх. Весь график заключен в коридор шириной 5 делений. Этот коридор является подвижным, но сдвигается лишь в результате 6 последовательных команд вызова или возврата. Из графика видно, что программа во время своей работы может оставаться в стационарном коридоре на протяжении достаточно длительного периода времени. Изучение программ, написанных на C или Pascal, показало, что при расширении коридора до 8 делений он сдвигается меньше чем при 1% вызовов или возвратов [252].



**Рис. 1.21.** Пример поведения программы при вызове и возврате из процедур

В литературе различают пространственную и временную локальность. **Пространственная локальность** означает преимущественное обращение к некоторому количеству сгруппированных ячеек памяти. Это свойство отражает тенденцию процессора выполнять команды последовательно. Пространственная локальность свидетельствует также о склонности программы обращаться к данным, которые находятся в последовательных ячейках, как, например, при обработке данных, собранных в таблицу. **Временная локальность** отражает тенденцию процессора обращаться к ячейкам памяти, к которым недавно уже производился доступ. Например, при выполнении цикла итераций процессор многократно повторяет выполнение одного и того же набора команд.

Традиционно временная локальность используется путем сохранения недавно использованных команд и данных в кеш-памяти и использования иерархии кеша. Пространственная локальность обычно используется путем применения больших блоков кеша и включения механизма упреждающей выборки (выборка данных, использование которых ожидается) в логику управления кешем. В последнее время наблюдается значительная исследовательская работа по уточнению этих методов для достижения большей производительности, но базовые стратегии остаются теми же.

## ФУНКЦИОНИРОВАНИЕ ДВУХУРОВНЕВОЙ ПАМЯТИ

Свойство локальности может быть использовано при разработке схемы двухуровневой памяти. Память верхнего уровня (M1) имеет меньшую емкость, она быстрее, и каждый ее бит дороже по сравнению с памятью нижнего уровня (M2). M1 используется для временного хранения определенной части содержимого более емкого уровня M2. При каждом обращении к памяти сначала предпринимается попытка найти нужные данные в M1. Если она завершается успехом, происходит быстрый доступ. В противном случае из M2 в M1 копируется нужный блок ячеек памяти, а затем снова осуществляется доступ к M1. В соответствии со свойством локальности к ячейкам памяти вновь перенесенного блока произойдет еще ряд обращений, что приведет к общему ускорению работы программы.

Чтобы выразить среднее время доступа, нужно учитывать не только скорость работы обоих уровней памяти, но и вероятность обнаружения требуемых данных в M1. В результате получим

$$T_s = H \times T_1 + (1 - H) \times (T_1 + T_2) = T_1 + (1 - H) \times T_2, \quad (1.1)$$

где

$T_s$  — среднее (системное) время доступа,

$T_1$  — время доступа к M1 (например, кешу, дисковому кешу),

$T_2$  — время доступа к M2 (например, основной памяти, диску),

$H$  — результативность поиска (доля ссылок на данные в M1).

На рис. 1.15 показано среднее время доступа как функция результативности поиска. При высокой результативности среднее время доступа намного ближе ко времени доступа к M1, чем ко времени доступа к M2.

## ПРОИЗВОДИТЕЛЬНОСТЬ

Рассмотрим некоторые параметры, характеризующие механизм двухуровневой памяти. Сначала рассмотрим стоимость, которая выражается следующим образом:

$$C_s = \frac{C_1 S_1 + C_2 S_2}{S_1 + S_2}, \quad (1.2)$$

где

$C_s$  — средняя стоимость одного бита двухуровневой памяти,

$C_1$  — средняя стоимость одного бита памяти верхнего уровня M1,

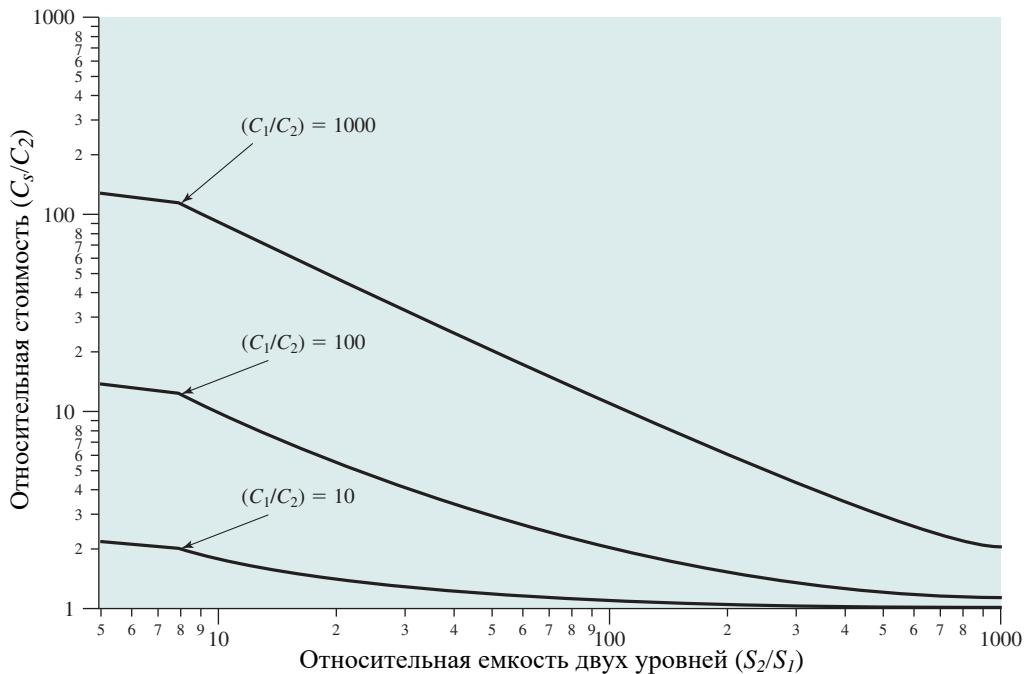
$C_2$  — средняя стоимость одного бита памяти нижнего уровня M2,

$S_1$  — емкость M1,

$S_2$  — емкость M2.

Желательно добиться соотношения  $C_s \approx C_2$ . При условии  $C_1 \gg C_2$  для этого требуется, чтобы выполнялось условие  $S_1 \ll S_2$ . Получающаяся зависимость представлена на рис. 1.22<sup>7</sup>.

<sup>7</sup> Обратите внимание, что для обеих осей выбрана логарифмическая шкала. Обзор основных сведений по логарифмическим шкалам приведен в сборнике документов для повторения математики, который находится на сайте Computer Science Student Resource Site по адресу ComputerScienceStudent.com.



**Рис. 1.22.** Зависимость средней стоимости двухуровневой памяти от относительной емкости уровней

Теперь рассмотрим время доступа. Для высокой производительности двухуровневой памяти необходимо, чтобы выполнялось условие  $T_s \approx T_1$ . Поскольку обычно  $T_1 \ll T_2$ , нужно, чтобы результативность поиска была близка к 1.

Таким образом, мы хотим, чтобы уровень M1 обладал малой емкостью (что позволило бы снизить стоимость), но был достаточно большим для того, чтобы повысить результативность поиска и как следствие производительность. Можно ли так подобрать размер M1, чтобы в определенной степени он удовлетворял обоим требованиям? Этот вопрос можно разбить на несколько подвопросов.

- Какая результативность поиска удовлетворяет требованиям производительности?
- Какая емкость M1 даст гарантию достижения требуемой результативности поиска?
- Будет ли эта емкость иметь приемлемую стоимость?

Чтобы ответить на эти вопросы, рассмотрим величину  $T_1/T_s$ , которая называется *эффективностью доступа*. Она является мерой того, насколько среднее время доступа  $T_s$  отличается от времени доступа  $T_1$  к M1. Из уравнения (1.1) находим

$$\frac{T_1}{T_s} = \frac{1}{1 + (1 - H) \frac{T_2}{T_1}} \quad (1.3)$$

На рис. 1.23 представлен график зависимости  $T_1/T_s$  от результативности поиска  $H$  при разных значениях параметра  $T_2/T_1$ . Чтобы удовлетворить требованиям эффективности, похоже, величина результативности поиска должна находиться в пределах от 0,8 до 0,9.

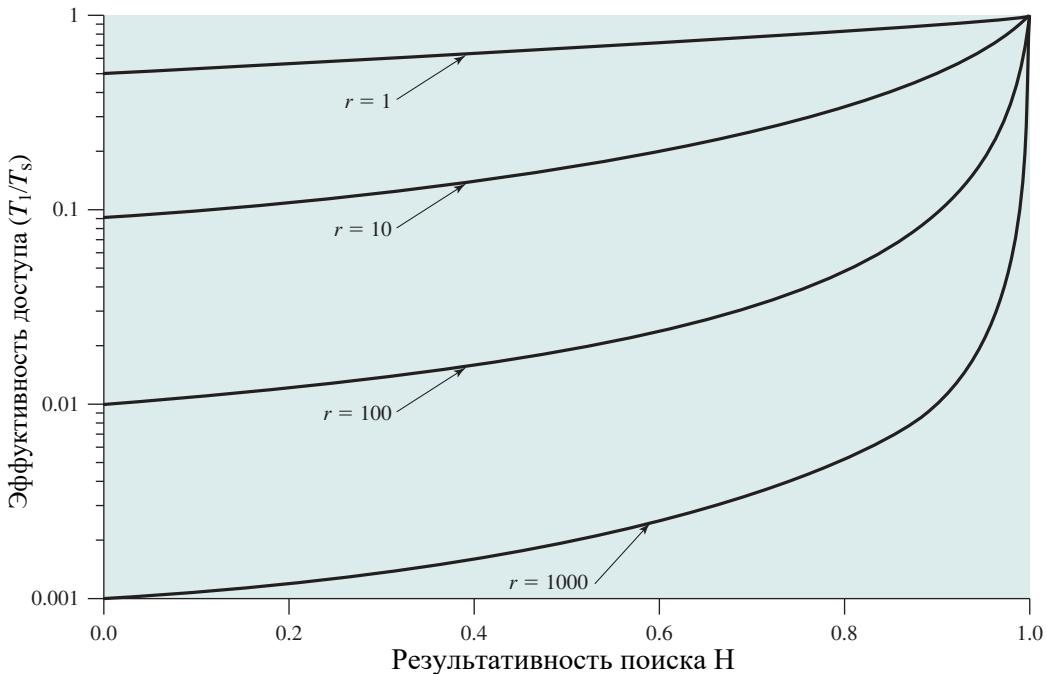
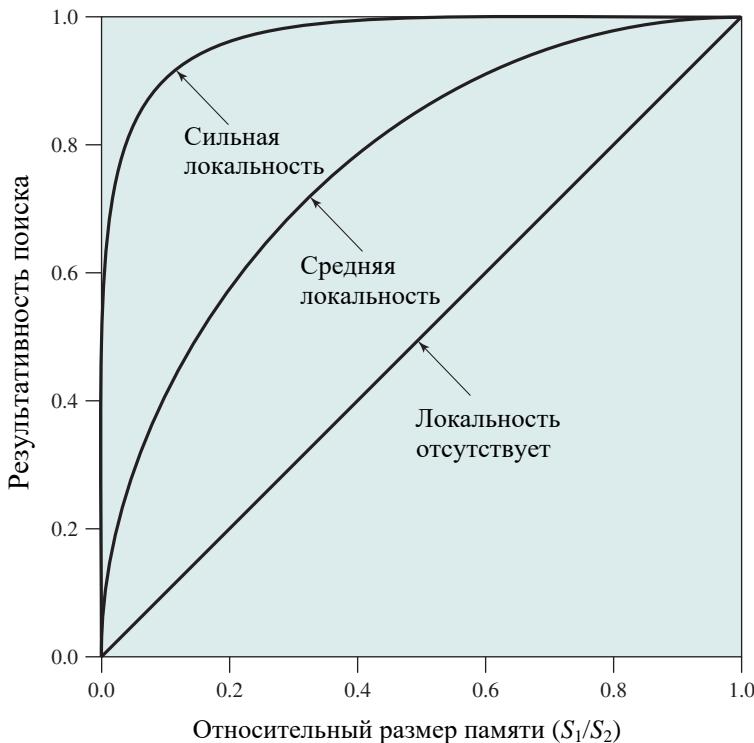


Рис. 1.23. Эффективность доступа в зависимости от результативности поиска

Теперь вопрос об относительной емкости памяти можно сформулировать более точно. Можно ли при условии  $S_1 \ll S_2$  добиться, чтобы результативность поиска достигала значения 0,8 или превышала его? Это зависит от нескольких факторов, в число которых входят вид используемого программного обеспечения и детали устройства двухуровневой памяти. Основное влияние, конечно, оказывает степень локальности. На рис. 1.24 показано, какое влияние оказывает локальность на результативность поиска. Очевидно, что если емкость уровня M1 равна емкости уровня M2, то результативность поиска будет 1.0, поскольку все содержимое M2 находится в M1. Теперь предположим, что нет никакой локальности, т.е. все обращения происходят в случайном порядке. В этом случае результативность поиска линейно зависит от относительного размера памяти. Например, если объем M1 равен половине объема M2, то в любой момент времени на уровне M1 находится ровно половина всех данных уровня M2, и результативность поиска равна 0,5. Однако на практике проявляется эффект локальности обращений. На графике показано влияние локальности средней и сильной степени.

Таким образом, сильная локальность позволяет достичь высокой результативности поиска даже при сравнительно небольших объемах памяти верхнего уровня. Например, многочисленные исследования подтверждают, что при сравнительно небольшом размере кеша результативность поиска превышает 0,75 независимо от размера основной памяти ([1], [195], [246] и [234]). В то время как типичный размер основной памяти в наши дни составляет гигабайты, вполне достаточным является кеш, емкость которого лежит в пределах от 1 тысячи до 128 тысяч слов. При рассмотрении виртуальной памяти и дискового кеша можно сослаться на другие исследования, подтверждающие справедливость такого же утверждения, а именно — благодаря локальности относительно малый размер M1 обеспечивает высокую результативность поиска.



**Рис. 1.24.** Результативность поиска в зависимости от относительного размера памяти

Теперь можно перейти к последнему из перечисленных ранее вопросов: удовлетворяет ли относительный размер двух уровней памяти требованиям стоимости? Ответ очевиден: да. Если для повышения производительности достаточно добавить верхний уровень сравнительно небольшой емкости, то средняя стоимость обоих уровней в расчете на один бит будет лишь немного превосходить стоимость бита более дешевой памяти второго уровня. Обратите внимание, что проведенный анализ при наличии кеша L2 (и тем более при наличии кешей L2 и L3) становится гораздо более сложным. Подробное обсуждение приведено в [188] и [101].



# ГЛАВА 2

---

## Обзор операционных систем

В ЭТОЙ ГЛАВЕ...

### 2.1. Цели и функции операционных систем

- Операционная система как интерфейс между пользователем и компьютером
- Операционная система как диспетчер ресурсов
- Простота развития операционной системы

### 2.2. Эволюция операционных систем

- Последовательная обработка
- Простые пакетные системы
- Многозадачные пакетные системы
- Системы, работающие в режиме разделения времени

### 2.3. Основные достижения

- Процессы
- Управление памятью
- Защита информации и безопасность
- Планирование и управление ресурсами

### 2.4. Разработки, ведущие к современным операционным системам

### 2.5. Отказоустойчивость

- Фундаментальные концепции
- Отказы
- Механизмы операционных систем

### 2.6. Вопросы проектирования операционных систем

#### для многопроцессорных и многоядерных систем

- Операционные системы для SMP
- Вопросы проектирования операционных систем для многоядерных систем
  - Параллелизм в приложениях
  - Виртуальная машина

## 2.7. Обзор операционной системы Microsoft Windows

Основы

Архитектура

Организация операционной системы

Процессы пользовательского режима

Модель “клиент/сервер”

Потоки и симметричная многопроцессорность

Объекты Windows

## 2.8. Традиционные системы UNIX

Историческая справка

Описание

## 2.9. Современные системы UNIX

System V Release 4 (SVR4)

BSD

Solaris 11

## 2.10. Linux

История

Модульная структура

Компоненты ядра

## 2.11. Android

Программная архитектура Android

Приложения

Каркас приложений

Системные библиотеки

Ядро Linux

Система времени выполнения Android

Виртуальная машина Dalvik

Формат Dex

Концепции Android Runtime

Преимущества и недостатки

Системная архитектура Android

Операции

Управление электропитанием

## 2.12. Ключевые термины, контрольные вопросы и задачи

Ключевые термины

Контрольные вопросы

Задачи

## УЧЕБНЫЕ ЦЕЛИ

- Понимать ключевые высокоуровневые функции операционных систем.
- Обсудить эволюцию операционных систем от ранних простых пакетных систем до современных сложных операционных систем.
- Дать краткие пояснения каждого из основных достижений в области изучения операционных систем.
- Обсудить ключевые области, сыгравшие особую роль в развитии современных операционных систем.
- Определить и обсудить виртуальные машины и виртуализацию.
- Понимать вопросы проектирования операционных систем, связанные с многопроцессорными и многоядерными системами.
- Понимать базовую структуру Windows.
- Описать основные элементы традиционных Unix-систем.
- Пояснить новые функциональные возможности современных Unix-систем.
- Обсудить операционную систему Linux и ее взаимосвязь с Unix.

Мы начинаем изучение операционных систем с краткого обзора истории развития операционных систем. Эта история интересна как сама по себе, так и как обзор принципов построения операционных систем. В первом разделе изучаются цели и функции операционных систем. Затем рассматривается эволюция операционных систем от примитивных пакетных систем до сложных многозадачных многопользовательских систем. Оставшаяся часть главы представляет собой обзор истории и характеристик двух операционных систем, которые служат в этой книге в качестве примеров.

## 2.1. ЦЕЛИ И ФУНКЦИИ ОПЕРАЦИОННЫХ СИСТЕМ

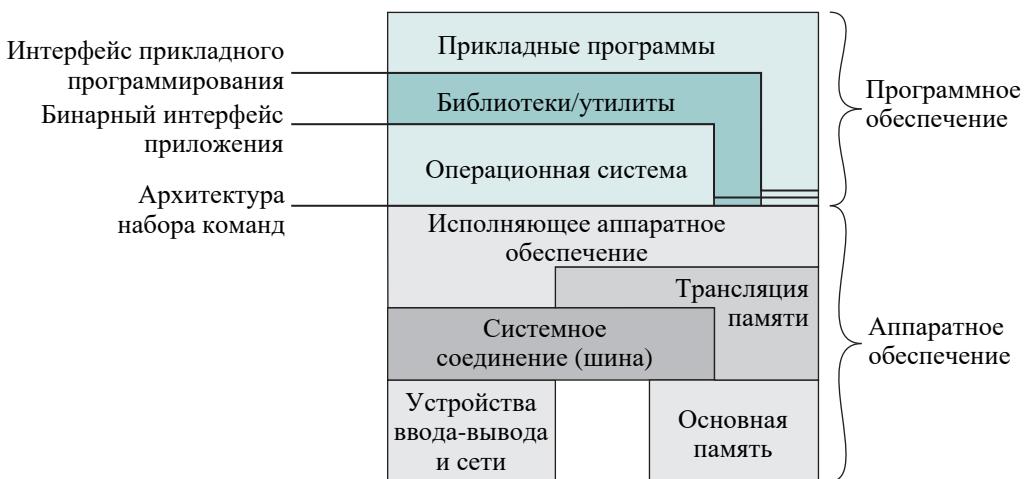
Операционная система — это программа, контролирующая выполнение прикладных программ и исполняющая роль интерфейса между приложениями и аппаратным обеспечением компьютера. Ее предназначение можно разделить на три основные составляющие.

- **Удобство.** Операционная система делает использование компьютера простым и удобным.
- **Эффективность.** Операционная система позволяет эффективно использовать ресурсы компьютерной системы.
- **Возможность развития.** Операционная система должна быть организована так, чтобы допускать эффективную разработку, тестирование и внедрение новых приложений и системных функций, причем это не должно мешать нормальному функционированию вычислительной системы.

Рассмотрим поочередно все три аспекта работы операционных систем.

## Операционная система как интерфейс между пользователем и компьютером

На рис. 2.1 представлена “слоистая” структура программного и аппаратного обеспечения, использующегося для предоставления конечному пользователю возможности работы с приложениями. Конечный пользователь обычно не интересуется деталями устройства аппаратного обеспечения компьютера. Компьютер видится ему как набор приложений. Приложение можно написать на каком-то из языков программирования; эту задачу выполняют прикладные программисты. Если бы кто-то задумал разработать реализованную в виде набора машинных команд программу, которая полностью отвечает за управление аппаратным обеспечением компьютера, то это оказалось бы слишком сложной задачей. Чтобы ее упростить, имеется набор системных программ, некоторые из которых называются утилитами или библиотечными программами. С их помощью реализуются часто использующиеся функции, которые помогают при создании пользовательских программ, работе с файлами и управлении устройствами ввода-вывода. Программист использует эти средства при разработке собственных программ, а приложения во время выполнения обращаются к утилитам для выполнения определенных функций. Наиболее важной из системных программ является операционная система, которая скрывает от программиста детали аппаратного обеспечения и предоставляет ему удобный интерфейс для использования системы. Операционная система выступает в роли посредника, облегчая программисту и программным приложениям доступ к различным службам и возможностям.



**Рис. 2.1.** Структура программного и аппаратного обеспечения компьютера

Приведем краткий список услуг, предоставляемых типичными операционными системами.

- **Разработка программ.** Содействуя программисту при разработке программ, операционная система предоставляет ему разнообразные инструменты и службы, например редакторы или отладчики. Обычно эти службы реализованы в виде программ-утилит, которые поддерживаются операционной системой, хотя и не входят в ее ядро. Такие программы называются инструментами разработки прикладных программ.

- **Выполнение программ.** Для выполнения программы требуется совершить ряд действий. Следует загрузить в основную память команды и данные, инициализировать устройства ввода-вывода и файлы, а также подготовить другие ресурсы. Операционная система выполняет всю эту рутинную работу вместо пользователя.
  - **Доступ к устройствам ввода-вывода.** Для управления работой каждого устройства ввода-вывода нужен свой особый набор команд или контрольных сигналов. Операционная система предоставляет пользователю единообразный интерфейс, который скрывает все эти детали и обеспечивает программисту доступ к устройствам ввода-вывода с помощью простых команд чтения и записи.
  - **Контролируемый доступ к файлам.** При работе с файлами управление со стороны операционной системы предполагает не только глубокое понимание природы устройств ввода-вывода (дисковода, лентопротяжного устройства), но и знание структур данных, записанных в файлах. Многопользовательские операционные системы, кроме того, могут обеспечивать работу механизмов защиты при обращении к файлам.
  - **Системный доступ.** Операционная система управляет доступом к совместно используемой или общедоступной вычислительной системе в целом, а также к отдельным системным ресурсам. Она должна обеспечивать защиту ресурсов и данных от несанкционированного использования, а также разрешать конфликтные ситуации.
  - **Обнаружение ошибок и их обработка.** При работе компьютерной системы могут происходить разнообразные сбои. К их числу относятся внутренние и внешние ошибки, возникшие в аппаратном обеспечении (например, ошибки памяти, отказ или сбой устройств). Возможны и различные программные ошибки (такие, как арифметическое переполнение, деление на нуль, попытка обратиться к ячейке памяти, доступ к которой запрещен, или невозможность выполнения запроса приложения). В каждом из этих случаев операционная система должна выполнить действия, минимизирующие влияние ошибки на работу приложения. Реакция операционной системы на ошибку может быть различной — от простого сообщения об ошибке до аварийного останова программы, вызвавшей ее.
  - **Учет использования ресурсов.** Хорошая операционная система должна иметь средства учета использования различных ресурсов и отображения параметров производительности. Эта информация крайне важна в любой системе, особенно в связи с необходимостью дальнейших улучшений и настройки вычислительной системы для повышения ее производительности. В многопользовательских системах эта информация может применяться для выставления счетов.
- На рис. 2.1 также показаны три ключевых интерфейса типичной вычислительной системы.
- **Структура системы команд (instruction set architecture — ISA).** Определяет набор команд машинного языка, которые может выполнять компьютер. Этот интерфейс является границей между аппаратным и программным обеспечением. Обратите внимание, что и прикладные программы, и утилиты могут получить непосредственный доступ к ISA. Для этих программ доступно подмножество команд (пользовательская ISA). Операционная система имеет доступ к дополнительным командам машинного языка, которые относятся к управлению ресурсами системы (системная ISA).

- **Бинарный интерфейс приложения** (application binary interface — ABI). ABI определяет стандарт бинарной переносимости между программами. ABI определяет интерфейс системных вызовов операционной системы и аппаратных ресурсов и служб, доступных в системе через пользовательскую ISA.
- **Интерфейс прикладного программирования** (application programming interface — API). API обеспечивает программе доступ к аппаратным ресурсам и службам, доступным в системе через пользовательскую ISA с библиотечными вызовами на языке высокого уровня. Обычно любые системные вызовы выполняются через библиотеки. Применение API обеспечивает легкую переносимость прикладного программного обеспечения на другие системы, поддерживающие тот же API, путем перекомпиляции.

## Операционная система как диспетчер ресурсов

Операционная система ответственна за управление использованием ресурсов компьютера, таких как ввод-вывод, основная и вторичная память и время работы процессора. Обычно мы представляем себе управляющий механизм как нечто внешнее по отношению к тому, чем он управляет, или по крайней мере как нечто отличающееся от управляемой системы или являющееся ее отдельной частью. (Например, система отопления жилых помещений управляет термостатом, который реализован в виде отдельного узла и отличается от аппаратуры выделения и распределения тепла.) С операционной системой дело обстоит по-другому, так как этот управляющий механизм является необычным в двух отношениях.

- Функции операционной системы работают точно так же, как и все остальное программное обеспечение, т.е. они реализованы в виде отдельных программ или набора программ, исполняющихся процессором.
- Операционная система часто передает управление другим процессам и должна ожидать, когда процессор снова позволит ей выполнять свои обязанности.

Как и любая другая программа, операционная система состоит из команд, выполняемых процессором. Во время работы операционная система указывает процессору, как использовать другие системные ресурсы и как распределять время при исполнении других программ. Но для того, чтобы реализовать действия, предписываемые операционной системой, процессор должен приостановить работу с ней и перейти к выполнению других программ. Таким образом, операционная система уступает управление процессору, чтобы он смог выполнить некоторую “полезную” работу, а затем возобновляет контроль ровно настолько, чтобы подготовить процессор к следующей части работы. Прочитав данную главу, читатель должен получить отчетливое представление о механизмах, принимающих участие в этих процессах.

На рис. 2.2 показаны основные ресурсы, которыми управляет операционная система. Часть операционной системы находится в основной памяти. В эту часть входит **ядро** (kernel, nucleus), содержащее основную часть наиболее часто используемых функций; там же находятся и некоторые другие компоненты операционной системы, использующиеся в данный момент времени. Остальная используемая часть основной памяти содержит другие программы и данные пользователя. Вскоре читатель сможет убедиться, что размещение этих данных в основной памяти управляется совместно операционной системой и аппаратной частью процессора, предназначеннной для управления памятью.

Операционная система принимает решение, когда исполняющаяся программа может использовать нужные ей устройства ввода-вывода, и управляет доступом к файлам и их использованием. Процессор сам по себе также является ресурсом, поэтому операционная система должна определить, сколько времени он должен уделить исполнению той или иной пользовательской программы.



**Рис. 2.2.** Операционная система как диспетчер ресурсов

## Простота развития операционной системы

Большинство операционных систем постоянно развиваются. Происходит это в силу следующих причин.

- **Обновление и возникновение новых видов аппаратного обеспечения.** Например, ранние версии операционных систем UNIX и Macintosh OS не использовали механизмы страничной организации памяти, потому что они работали на машинах, не обеспеченных соответствующими аппаратными средствами<sup>1</sup>. Более поздние версии операционных систем были доработаны таким образом, чтобы они могли использовать новые аппаратные возможности. Точно так же на устройство операционных систем повлияло использование графических терминалов и терминалов, работающих в страничном режиме, вместо алфавитно-цифровых терминалов с построчной разверткой. Такой терминал позволяет пользователю работать

<sup>1</sup> Краткое рассмотрение страничной организации памяти приведено в последующих разделах данной главы; более подробно этот материал изложен в главе 7, "Управление памятью".

одновременно с несколькими приложениями в различных окнах экрана. Такая возможность требует более сложной поддержки со стороны операционной системы.

- **Новые сервисы.** Для удовлетворения требований пользователей или нужд системных администраторов операционные системы предоставляют новые возможности. Например, если станет трудно поддерживать высокую производительность при работе с имеющимся на определенный момент инструментарием пользователя, в операционную систему могут быть добавлены новые инструменты для контроля и оценки производительности.
- **Исправления.** В каждой операционной системе есть ошибки. Время от времени они обнаруживаются и исправляются. Конечно, в исправление могут вкрасться новые ошибки.

Необходимость регулярных изменений операционных систем накладывает определенные требования к их устройству. Очевидно, что эти системы должны иметь модульную конструкцию с четко определенным взаимодействием модулей; очень важную роль играет хорошая и полная документированность. Для больших программ, которыми на сегодняшний день являются типичные операционные системы, недостаточно выполнить то, что называется непосредственной модуляризацией [64]; нужно сделать нечто большее, чем простая разбивка целой программы на отдельные подпрограммы. В данной главе мы вернемся к этому вопросу.

## 2.2. ЭВОЛЮЦИЯ ОПЕРАЦИОННЫХ СИСТЕМ

Пытаясь понять основные требования, предъявляемые к операционным системам, а также значение основных возможностей современных операционных систем, полезно проследить за их эволюцией, происходившей на протяжении многих лет.

### Последовательная обработка

В первых компьютерах, в период от конца 1940-х до середины 1950-х годов, программы непосредственно взаимодействовали с аппаратным обеспечением машины; операционных систем в то время еще не было. Эти компьютеры управлялись с пульта управления, состоящего из сигнальных ламп, тумблеров, некоторого устройства для ввода данных и принтера. Программы в машинных кодах загружались через устройство ввода данных (например, устройство ввода с перфокарт). Если из-за ошибки происходил останов программы, о возникновении сбойной ситуации свидетельствовали аварийные сигнальные лампы. Чтобы определить причину ошибки, программист должен был проверить состояние регистров процессора и основной памяти. Если программа успешно завершала свою работу, ее выходные данные распечатывались на принтере. Эти ранние системы имели две основные проблемы.

- **Расписание работы.** На большинстве машин нужно было предварительно заказать машинное время, записавшись в специальный график. Обычно пользователь мог заказать время, кратное некоторому периоду, например получасу. Тогда, записавшись на 1 час, он мог закончить работу за 45 минут, что приводило к простою компьютера. С другой стороны, если пользователь не укладывался в отведенное время, он должен был прекращать работу, прежде чем задача завершит выполнение.

- **Время подготовки к работе.** Для запуска каждой программы, называемой **заданием** (job), нужно было загрузить в память компилятор и саму программу, обычно составленную на языке высокого уровня (исходный текст), сохранить скомпилированную программу (объектный код), а затем загрузить и скомпоновать объектный код с библиотечными функциями. Для каждого из этих этапов могли понадобиться установка и съем магнитных лент или загрузка колоды перфокарт. При возникновении фатальной ошибки беспомощному пользователю не оставалось ничего другого, как начинать весь подготовительный процесс заново. Таким образом, значительное время затрачивалось лишь на то, чтобы подготовить программу к собственно исполнению.

Такой режим работы можно назвать *последовательной обработкой* (serial processing). Это название отражает тот факт, что пользовательские программы выполнялись на компьютере последовательно. Со временем в попытке повысить эффективность последовательной обработки были разработаны различные системные инструменты. К ним относятся библиотеки функций, редакторы связей, загрузчики, отладчики и драйверы ввода-вывода, существующие в виде программного обеспечения, общедоступного для всех пользователей.

## Простые пакетные системы

Ранние машины были очень дорогими, поэтому было важно использовать их как можно эффективнее. Просто, происходившие из-за несогласованности расписания, а также время, затраченное на подготовку задачи, — все это обходилось слишком дорого; эти непроизводительные затраты были непозволительной роскошью.

Чтобы повысить эффективность работы, была предложена концепция пакетной операционной системы. Похоже, первые пакетные операционные системы (и вообще первые операционные системы какого бы то ни было типа) были разработаны в средине 1950-х годов в компании General Motors для машин IBM 701 [266]. Впоследствии эта концепция была усовершенствована и внедрена рядом пользователей на IBM 704. В начале 1960-х годов некоторые поставщики разработали пакетные операционные системы для своих компьютеров. Одной из заметных систем того времени является IBSYS фирмы IBM, разработанная для компьютеров 7090/7094 и оказавшая значительное влияние на другие системы.

Центральная идея, лежащая в основе простых пакетных схем обработки, состоит в использовании программы, известной под названием **монитор** (monitor). Используя операционную систему такого типа, пользователь не имеет непосредственного доступа к машине. Вместо этого он передает свое задание на перфокартах или магнитной ленте оператору компьютера, который собирает разные задания в пакеты и помещает их в устройство ввода данных. Так они передаются монитору. Каждая программа составлена таким образом, что при завершении ее работы управление переходит к монитору, который автоматически загружает следующую программу.

Чтобы понять работу этой схемы, рассмотрим ее с точки зрения монитора и процессора.

- **Работа схемы с точки зрения монитора.** Монитор управляет последовательностью событий. Чтобы это было возможно, большая его часть должна всегда находиться в основной памяти и быть готовой к работе (рис. 2.3). Этую часть монитора

называют **резидентным монитором**. Оставшуюся часть составляют утилиты и общие функции, которые загружаются в виде подпрограмм, вызываемых программой пользователя в начале выполнения каждого задания (если они для него требуются). Монитор считывает с устройства ввода данных (обычно это устройство ввода с перфокарт или магнитная лента) по одному заданию. При этом текущее задание размещается в области памяти, предназначенной для программ пользователя, и ему передается управление. По завершении задания оно возвращает управление монитору, который сразу же начинает считывать следующее задание. Результат исполнения каждого задания направляется на устройство вывода, например на принтер для передачи пользователю.

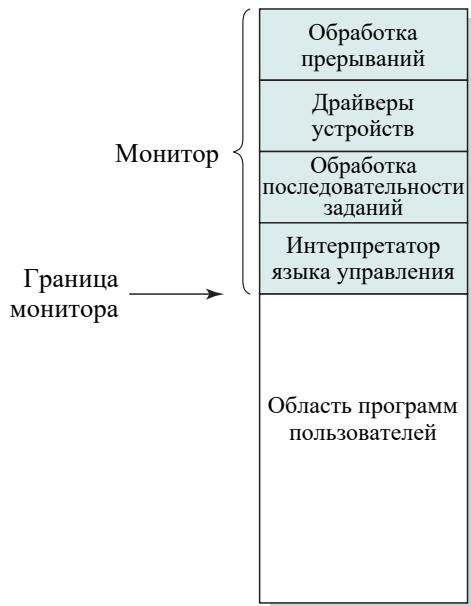


Рис. 2.3. Схема размещения резидентного монитора в памяти

- **Работа схемы с точки зрения процессора.** В некоторый момент времени процессор выполняет команды, которые находятся в той части основной памяти, которую занимает монитор. Это приводит к тому, что в другую область памяти считывается новое задание. После того как задание полностью считано, монитор отдает процессору команду перехода, по которой он должен начать исполнение программы пользователя. Процессор переходит к обработке программы пользователя и выполняет ее команды до тех пор, пока не дойдет до конца или пока не возникнет сбойная ситуация. В любом из этих двух случаев следующей командой, которую процессор извлечет из памяти, будет команда монитора. Таким образом, фраза “контроль передается заданию” означает, что процессор перешел к извлечению и выполнению команд программы пользователя. Фраза же “контроль возвращается монитору” означает, что теперь процессор извлекает из памяти и выполняет команды монитора.

Как видите, монитор выполняет функцию планирования: задания в пакетах выстраиваются в очередь и выполняются без простоев настолько быстро, насколько это возможно. Кроме того, монитор помогает в подготовке программы к исполнению. В каждое задание включаются простые команды языка управления заданиями (job control language — JCL). Это специальный тип языка программирования, используемый для того, чтобы отдавать команды монитору. Рассмотрим простой пример, в котором нужно принять на обработку программу пользователя, составленную на языке FORTRAN, и дополнительные данные, используемые этой программой. Все команды языка FORTRAN и данные находятся на отдельных перфокартах или в отдельных записях на магнитной ленте. Каждое задание, кроме операторов языка FORTRAN и данных, содержит команды управления заданием, каждая из которых начинается знаком \$. Формат задания в целом выглядит следующим образом.

\$JOB	
\$FTN	
•	}
•	
•	
\$LOAD	}
\$RUN	
•	
•	
\$END	}
•	

Команды FORTRAN

Данные

Чтобы выполнить это задание, монитор читает строку \$FTN и загружает с запоминающего устройства большой емкости (обычно это лента) компилятор соответствующего языка. Компилятор преобразует программу пользователя в объектный код, который записывается в память или на запоминающее устройство. Если этот код заносится в память, то операция называется “компиляция, загрузка и запуск”. Если же он записывается на магнитную ленту, то нужна дополнительная команда \$LOAD. Монитор, к которому вернулось управление после компиляции, читает эту команду и обращается к загрузчику, который загружает объектную программу в память (на место компилятора) и передает ей управление. В таком режиме различные подсистемы совместно используют один и тот же участок основной памяти, хотя в каждый момент времени работает только одна из этих подсистем.

Во время выполнения программы пользователя по каждой команде ввода считывается только одна строка данных. Команда ввода программы пользователя обращается к подпрограмме ввода, которая является составной частью операционной системы. Подпрограмма ввода проверяет, не произошло ли случайное считывание строки языка JCL. Если это произошло, управление передается монитору. По завершении задания пользователя монитор проверяет строки задания, пока не дойдет до строки с командой на языке управления, что защищает систему от программ, в которых оказалось слишком много или слишком мало строк с данными.

Таким образом, монитор, или пакетная операционная система, — это обычная компьютерная программа. Ее работа основана на способности процессора выбирать команды из различных областей основной памяти; при этом происходит передача и возврат управления. Желательно также использование и других возможностей аппаратного обеспечения.

- **Защита памяти.** Во время работы программы пользователя не должна вносить изменения в область памяти, в которой находится монитор. Если же такая попытка предпринята, аппаратное обеспечение процессора должно обнаружить ошибку и передать управление монитору. Затем монитор снимет задачу с выполнения, распечатает сообщение об ошибке и загрузит следующее задание.
- **Таймер.** Таймер используется для того, чтобы предотвратить ситуацию, когда одна задача захватывает безраздельный контроль над системой. Таймер выставляется в начале каждого задания. По истечении определенного промежутка времени программа пользователя останавливается и управление передается монитору.
- **Привилегированные команды.** Некоторые команды машинного уровня имеют повышенные привилегии и могут выполняться только монитором. Если процессор натолкнется на такую команду во время исполнения программы пользователя, возникнет ошибка, при которой управление будет передано монитору. В число привилегированных команд входят команды ввода-вывода; это значит, что все устройства ввода-вывода контролируются монитором. Это, например, предотвращает случайное чтение программой пользователя команд управления, относящихся к следующему заданию. Если программе пользователя нужно произвести ввод-вывод, она должна запросить для выполнения этих операций монитор.
- **Прерывания.** В первых моделях компьютеров этой возможности не было. Она придает операционной системе большую гибкость при передаче управления программе пользователя и его возобновлении.

Вопросы защиты памяти и привилегированных команд приводят к концепции режимов работы. Программа пользователя выполняется в **пользовательском режиме**, в котором определенные области памяти защищены от использования пользователем и некоторые команды не могут быть выполнены. Монитор работает в системном режиме, позже названном **режимом ядра**, в котором могут выполняться привилегированные команды и могут быть доступными защищенные области памяти.

Конечно, операционную систему можно разработать и без учета описанных выше возможностей. Однако поставщики компьютеров скоро поняли, что это приведет к хаосу, поэтому даже сравнительно простые пакетные операционные системы использовали эти возможности аппаратного обеспечения.

При работе пакетных операционных систем машинное время распределялось между исполнением программы пользователя и монитора. При этом в жертву приносились два вида ресурсов: монитор занимал некоторую часть оперативной памяти, им же потреблялось некоторое машинное время. И то, и другое приводило к непроизводительным издержкам. Несмотря на это простые пакетные системы существенно повышали эффективность использования компьютера.

## Многозадачные пакетные системы

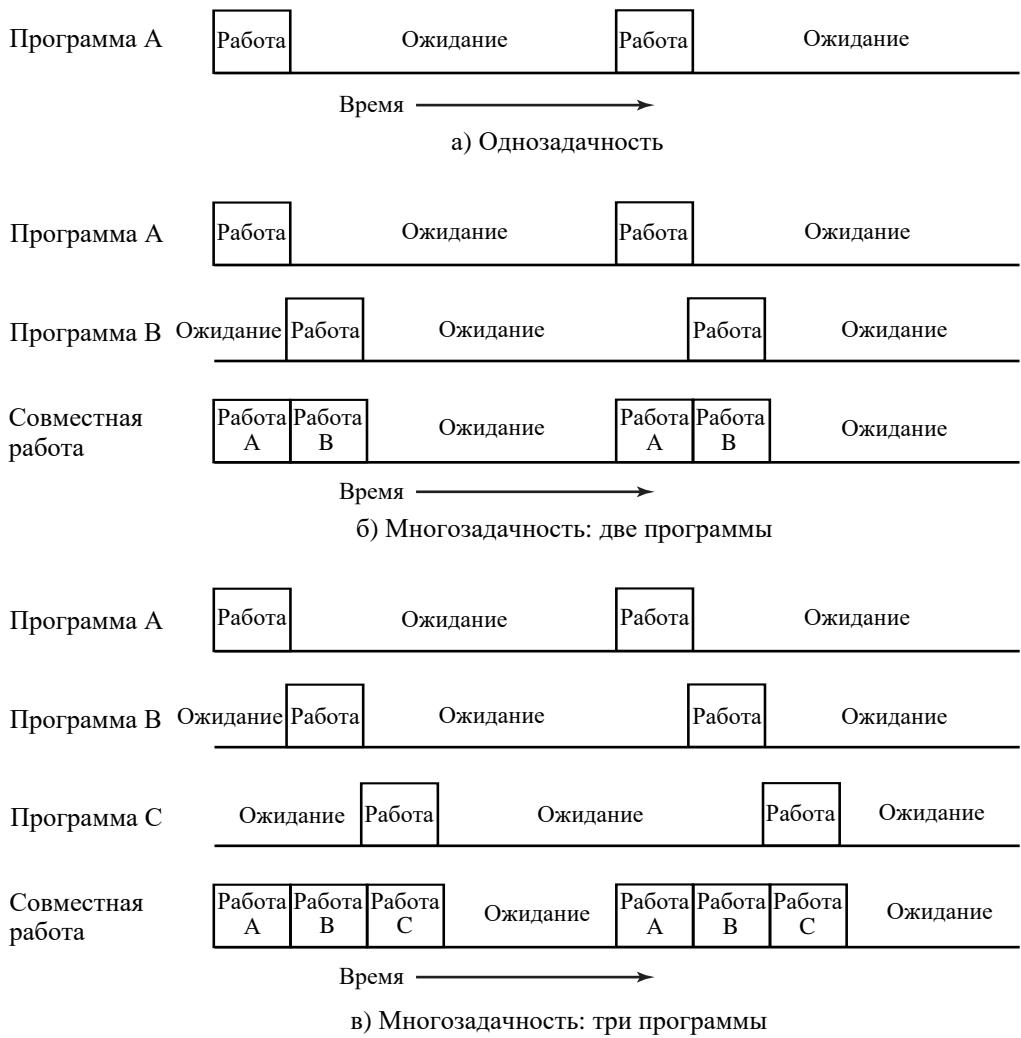
Процессору часто приходилось простаивать даже при автоматическом выполнении заданий под управлением простой пакетной операционной системы. Проблема заключается в том, что устройства ввода-вывода работают намного медленнее, чем процессор. На рис. 2.4 представлены соответствующие расчеты, выполненные для программы, которая обрабатывает файл с записями, причем для обработки одной записи требуется в среднем 100 машинных команд. В этом примере 96% всего времени процессор ждет, пока устройства ввода-вывода закончат передачу данных в файл и из него. На рис. 2.5, а показана такая ситуация для одной программы. Некоторое время процессор выполняет команды; затем, дойдя до команды ввода-вывода, он должен подождать, пока она закончится. Только после этого процессор сможет продолжить работу.

Чтение одной записи из файла	$15 \mu s$
Выполнение 100 машинных команд	$1 \mu s$
Внесение одной записи в файл	$15 \mu s$
Всего	$31 \mu s$
Степень использования процессора	$= \frac{1}{31} = 0.032 = 3.2\%$

Рис. 2.4. Пример использования системы

Эффективность использования процессора можно повысить. Мы знаем, что памяти должно хватить, чтобы разместить в ней операционную систему (резидентный монитор) и программу пользователя. Предположим, что в памяти достаточно места для операционной системы и двух программ пользователя. Теперь, когда одно из заданий ждет завершения операций ввода-вывода, процессор может переключиться на другое задание, для которого в данный момент ввод-вывод, скорее всего, не требуется (рис. 2.5, б). Более того, если памяти достаточно для размещения большего количества программ, то процессор может выполнять их параллельно, переключаясь с одной на другую (рис. 2.5, в). Такой режим известен как **многозадачность (multiprogramming)** и является основной особенностью современных операционных систем.

Приведем простой пример, иллюстрирующий преимущества многозадачности. Рассмотрим компьютер, имеющий 250 Мбайт доступной памяти (не используемой операционной системой), диск, терминал и принтер. На обработку одновременно приняты три программы, JOB1, JOB2 и JOB3, атрибуты которых перечислены в табл. 2.1. Предположим, что для выполнения заданий JOB2 и JOB3 использование процессора минимально и задание JOB3 постоянно обращается к диску и принтеру. В простой среде с пакетной обработкой эти задания выполняются последовательно. Для завершения JOB1 требуется 5 мин. Задание JOB2 должно ждать, пока пройдут эти 5 мин, после чего оно выполняется в течение 15 мин. По истечении 20 мин начинает работу задание JOB3; его выполнение заканчивается через 30 мин после того, как оно было принято на обработку. Средний процент использования ресурсов, производительность и время отклика показаны в столбце табл. 2.2, соответствующем однозадачности, а на рис. 2.6, а показано использование различных устройств в этом режиме. Очевидно, что эффективность использования всех ресурсов, усредненная по всему периоду времени (30 мин), является крайне низкой.

**Рис. 2.5.** Пример многозадачности**ТАБЛИЦА 2.1. Свойства трех программ-примеров**

	<b>JOB1</b>	<b>JOB2</b>	<b>JOB3</b>
<b>Тип задания</b>	Интенсивные вычисления	Интенсивный ввод-вывод	Интенсивный ввод-вывод
<b>Продолжительность</b>	5 мин	15 мин	10 мин
<b>Требуемая память</b>	50 Мбайт	100 Мбайт	75 Мбайт
<b>Требуется ли диск</b>	Нет	Нет	Да
<b>Требуется ли терминал</b>	Нет	Да	Нет
<b>Требуется ли принтер</b>	Нет	Нет	Да

Таблица 2.2. Влияние многозадачности на использование ресурсов

	Однозадачность	Многозадачность
Использование процессора	20%	40%
Использование памяти	30%	67%
Использование диска	33%	67%
Использование принтера	33%	67%
Затраченное время	30 мин	15 мин
Производительность	6 заданий/ч	12 заданий/ч
Среднее время отклика	18 мин	10 мин

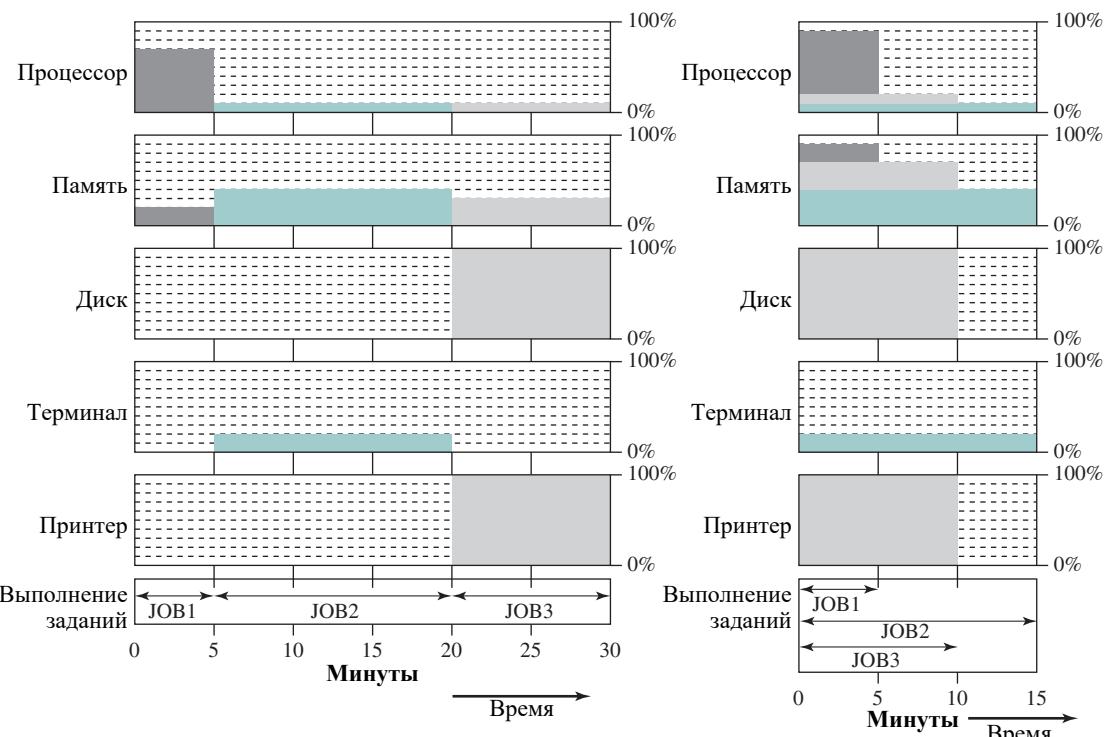


Рис. 2.6. Гистограммы использования ресурсов

Теперь предположим, что задания выполняются одновременно под управлением многозадачной операционной системы. Из-за того что они используют разные ресурсы, их совместное выполнение на компьютере длится минимальное время (при условии, что заданиям JOB2 и JOB3 предоставляется достаточно процессорного времени для поддержки осуществляемого ими ввода и вывода в активном состоянии). Для выполнения задания JOB1 потребуется 5 мин, но к этому времени задание JOB2 будет выполнено на

треть, а задание JOB3 — на половину. Все три задания будут выполнены через 15 мин. Если посмотреть на столбец табл. 2.2, соответствующий многозадачному режиму, его преимущества станут очевидными, как и из гистограммы, приведенной на рис. 2.6, б.

Работа многозадачной пакетной системы, как и работа простой пакетной системы, базируется на некоторых аппаратных возможностях компьютера. Наиболее значительными дополнениями, полезными для многозадачности, являются аппаратное обеспечение, поддерживающее прерывания ввода-вывода, и прямой доступ к памяти. Используя эти возможности, процессор может генерировать команду ввода-вывода для одного задания и переходить к другому на то время, пока контроллером устройства выполняется ввод-вывод. После завершения операции ввода-вывода процессор прерывается и управление передается программе обработки прерываний из состава операционной системы. Затем операционная система передает управление другому заданию.

Многозадачные операционные системы сложнее однозадачных систем последовательной обработки заданий. Для того чтобы можно было обрабатывать несколько заданий одновременно, они должны находиться в основной памяти, а для этого требуется некоторая система **управления памятью**. Кроме того, если к работе готовы несколько заданий, процессор должен решить, какое из них следует запустить, для чего необходим некоторый алгоритм планирования. Эти концепции обсуждаются ниже в данной главе.

## Системы, работающие в режиме разделения времени

Использование многозадачности в **пакетной обработке** может привести к существенному повышению эффективности. Однако для многих заданий желательно обеспечить такой режим, в котором пользователь мог бы непосредственно взаимодействовать с компьютером. В самом деле, во многих случаях интерактивный режим является обязательным условием работы.

Сегодня требование интерактивных вычислений может быть удовлетворено путем использования отдельного персонального компьютера или рабочей станции. Этот вариант был недоступен в 1960-е годы, когда большинство компьютеров были большими и дорогостоящими. Тогда и был разработан режим разделения времени.

Многозадачность позволяет процессору одновременно обрабатывать не только несколько заданий в пакетном режиме, но и несколько интерактивных заданий. Такую организацию называют **разделением времени**, потому что процессорное время распределяется между различными пользователями. В системе разделения времени несколько пользователей одновременно получают доступ к системе с помощью терминалов, а операционная система чередует исполнение программ каждого пользователя через малые промежутки времени. Таким образом, если нужно одновременно обслужить *n* пользователей, каждому из них предоставляется в среднем лишь  $1/n$  часть рабочего времени компьютера, не считая затрат на работу операционной системы. Однако, принимая во внимание относительно медленную реакцию человека, время отклика на компьютере с хорошо настроенной системой будет сравнимо со временем реакции пользователя.

Как пакетная обработка, так и разделение времени используют многозадачность. Основные различия этих двух режимов перечислены в табл. 2.3.

Одной из первых операционных систем разделения времени была система CTSS (Compatible Time-Sharing System) [52], разработанная в Массачусетском технологическом институте группой, известной как Project MAC (Machine-Aided Cognition или Multiple-Access Computers). Система была разработана в 1961 году для IBM 709, а затем перенесена на IBM 7094.

**ТАБЛИЦА 2.3. ПАКЕТНАЯ МНОГОЗАДАЧНОСТЬ И РАЗДЕЛЕНИЕ ВРЕМЕНИ**

	<b>Пакетная многозадачность</b>	<b>Разделение времени</b>
<b>Основная цель</b>	Максимальное использование процессора	Уменьшение времени отклика
<b>Источник указаний операционной системе</b>	Команды языка управления заданиями, помещаемые в задание	Команды, вводимые с терминала

По сравнению с более поздними системами, CTSS была довольно примитивна. Она работала на машине с основной памятью емкостью 32 000 36-битовых слов, из которых 5 000 слов занимал монитор. Когда управление должно было быть передано очередному интерактивному пользователю, его программа и данные загружались в остальные 27 000 слов основной памяти. Программа всегда загружалась так, что ее начало находилось в ячейке номер 5000, что упрощало управление как монитором, так и памятью. Приблизительно через каждые 0,2 с системный таймер генерировал прерывание. При каждом прерывании таймера управление передавалось операционной системе, и процессор мог перейти в распоряжение другого пользователя. Такая технология получила название **квантование времени** (time slicing). Таким образом, данные текущего пользователя через регулярные интервалы времени выгружались, а вместо них загружались другие. Перед считыванием программы и данных нового пользователя программа и данные предыдущего пользователя записывались на диск для сохранения до дальнейшего выполнения. Впоследствии, когда очередь этого пользователя наступит снова, код и данные его программы будут восстановлены в основной памяти.

Чтобы уменьшить обмен с диском, содержимое памяти, занимаемое данным пользователем, записывается на него лишь в том случае, если для загрузки новой программы не хватает места. Этот принцип проиллюстрирован на рис. 2.7. Предположим, что всего работает четыре пользователя, которым нужны следующие объемы памяти в словах.

- JOB1: 15 000
- JOB2: 20 000
- JOB3: 5 000
- JOB4: 10 000

Сначала монитор загружает задание JOB1 и передает ему управление (рис. 2.7, а). Затем он принимает решение передать управление заданию JOB2. Из-за того что JOB2 занимает больше памяти, чем JOB1, сначала нужно сохранить данные JOB1, а затем можно загружать JOB2 (рис. 2.7, б). Затем для обработки загружается JOB3. Но поскольку это задание меньше, чем JOB2, часть задания JOB2 может оставаться в основной памяти, сокращая время записи на диск (рис. 2.7, в). Позже монитор вновь передает управление заданию JOB1. Чтобы загрузить его в память, на диск необходимо записать еще часть задания JOB2 (рис. 2.7, г). При загрузке задания JOB4 в памяти остается часть заданий JOB1 и JOB2 (рис. 2.7, д). Если теперь будет активизировано задание JOB1 или JOB2, то потребуется лишь частичная его загрузка. Следующим в этом примере запускается задание JOB2. Для этого требуется, чтобы JOB4 и оставшаяся в памяти часть JOB1 были записаны на диск, а в память была считана недостающая часть JOB2 (рис. 2.7, е).

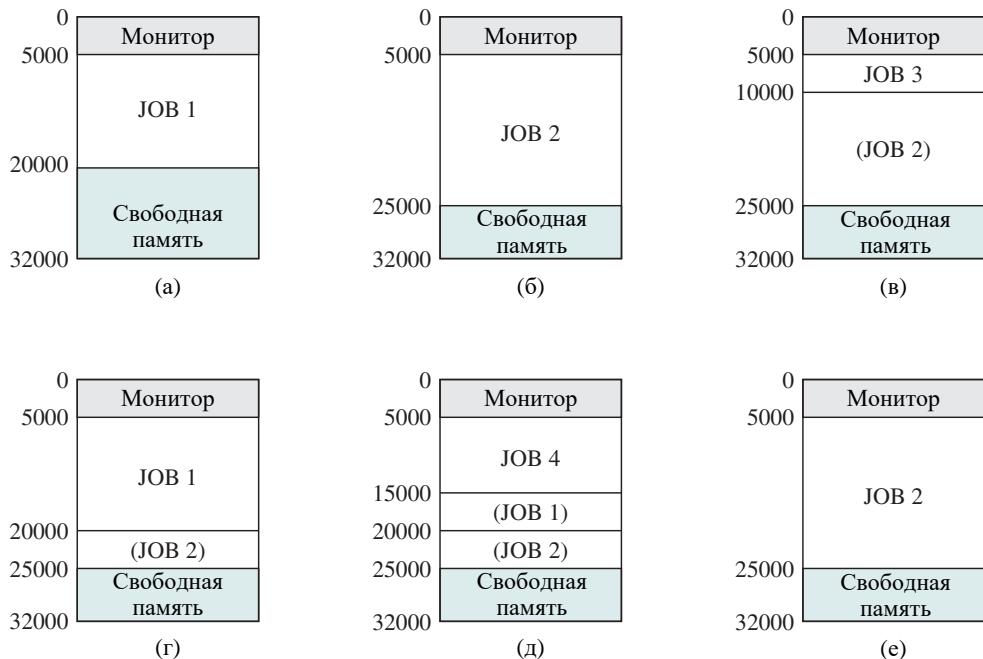


Рис. 2.7. Работа CTSS

Подход CTSS по сравнению с современными принципами разделения времени является примитивным, но он был вполне эффективным. Его простота позволяла использовать монитор минимальных размеров. Из-за того что задание всегда загружалось в одно и то же место в памяти, не было необходимости применять во время загрузки методы перемещения (которые рассматриваются позже). Необходим был лишь метод записи, позволяющий уменьшить активность диска. Работая на машине 7094, CTSS могла обслуживать до 32 пользователей.

С появлением разделения времени и многозадачности перед создателями операционных систем появилось много новых проблем. Если в памяти находится несколько заданий, их нужно защищать друг от друга, иначе одно задание может, например, изменить данные другого задания. Если в интерактивном режиме работают несколько пользователей, то файловая система должна быть защищена, чтобы к каждому конкретному файлу доступ был только у определенных пользователей. Нужно разрешать конфликтные ситуации, возникающие при работе с различными ресурсами, например с принтером и устройствами хранения данных. Ниже в книге будут рассмотрены эти и другие возможные проблемы, а также пути их решения.

## 2.3. ОСНОВНЫЕ ДОСТИЖЕНИЯ

Операционные системы относят к числу самых сложных программ. Это отражает стремление их разработчиков сделать системы такими, чтобы они удовлетворяли требованиям удобства и эффективности и при этом не утратили способности к развитию. Согласно [64] в процессе развития операционных систем были проведены исследования в четырех основных направлениях.

- Процессы
- Управление памятью
- Защита информации и безопасность
- Планирование и управление ресурсами

Каждое из этих направлений можно охарактеризовать набором абстрактных принципов, разработанных для решения сложных практических задач. В основном развитие современных операционных систем также происходит по перечисленным выше направлениям. Краткий их обзор, приведенный в этом разделе, поможет читателю получить представление о содержании большей части настоящей книги.

## Процессы

Одной из основополагающих концепций проектирования операционных систем является концепция *процессов*. Этот термин впервые был применен в 1960-х годах разработчиками операционной системы Multics [56]. Процесс — несколько более общий термин, чем *задание* (job). Есть много определений термина *процесс*, в том числе:

- выполняющаяся программа;
- экземпляр программы, выполняющейся на компьютере;
- объект, который можно идентифицировать и выполнять на процессоре;
- единица активности, которую можно охарактеризовать единой цепочкой последовательных действий, текущим состоянием и связанным с ней набором системных ресурсов.

Данная концепция должна становиться более понятной по мере освоения материала.

В процессе развития компьютерных систем при решении проблем, связанных с распределением времени и синхронизацией, вклад в развитие концепции процесса был сделан в трех основных направлениях: многозадачные пакетные операции, разделение времени и транзакции в реальном времени. Как мы уже могли убедиться, многозадачный режим дает возможность процессору и устройствам ввода-вывода работать одновременно, повышая тем самым эффективность использования компьютерной системы. При этом ключевой механизм состоит в следующем: в ответ на сигналы, свидетельствующие о завершении транзакций ввода-вывода, процессор переключается с одной программы на другую, находящуюся в основной памяти.

Другим направлением развития являются системы разделения времени общего назначения. Основная цель их разработки — удовлетворение потребностей каждого пользователя при обеспечении их одновременной работы (для снижения стоимости). В этих системах используется тот факт, что пользователь реагирует на события намного медленнее, чем компьютер. К примеру, если потребность пользователя во времени процессора для обработки его действий в среднем составляет 2 с в течение 1 мин, то в одной и той же системе, не мешая друг другу, могут работать до 30 пользователей. Конечно же, в таких расчетах нужно учитывать время, которое требуется для работы самой операционной системы.

Еще одним важным направлением развития являются системы обработки транзакций в реальном времени. При работе таких систем некоторое число пользователей отправля-

ют запросы в базу данных или вносят в нее изменения. Пример — система бронирования авиабилетов. Основное различие между системой обработки транзакций и системой разделения времени состоит в том, что в первой из них выполняются одно-два приложения, в то время как пользователи системы с разделением времени могут заниматься разработкой программ, запускать их и пользоваться многими различными приложениями. В обоих случаях ключевым фактором является время отклика системы.

Прерывание было важным инструментом, который стал доступен системным программистам еще на ранних стадиях развития многозадачных и многопользовательских интерактивных систем. Выполнение любого задания может быть прервано при наступлении определенного события, например завершения ввода-вывода. При этом процессор должен сохранить определенную информацию (такую, как содержимое счетчика команд, общих и системных регистров) и переключиться на выполнение программы обработки прерываний, которая выясняет природу прерывания, обрабатывает его, а затем возобновляет выполнение одного из заданий.

Дизайн системного программного обеспечения, координирующего подобные процессы, оказался очень сложным. При одновременной обработке многих заданий, каждое из которых включает в себя длинную последовательность действий, становится невозможно проанализировать все возможные комбинации последовательностей событий. Ввиду отсутствия систематических средств обеспечения координации и взаимодействия разных видов деятельности систем программисты обратились к специальным методам, основанным на представлении о той среде, работу которой должна контролировать операционная система. При этом они подвергались риску допустить трудноуловимые ошибки, которые проявляются только в очень редких случаях, при выполнении определенных последовательностей событий. Такие ошибки трудно обнаружить, потому что их нужно отличить от ошибок в приложениях и ошибок, возникающих при сбоях аппаратного обеспечения. Еще одной особенностью, затрудняющей определение причины этих ошибок (когда они обнаружены), является то, что воспроизвести точные условия, в которых эти ошибки проявляются, крайне трудно. Ниже перечислены основные причины подобных ошибок [64].

- **Неправильная синхронизация.** Часто случается так, что программа должна приостановить свою работу и ожидать наступления какого-то события в системе. Например, программа, которая инициировала операцию ввода-вывода, не сможет продолжать работу, пока в буфере не будут доступны необходимые ей данные. В этом случае требуется передача сигнала от какой-то другой программы. Недостаточная надежность сигнального механизма может привести к тому, что сигнал будет потерян или будет получено два таких сигнала.
- **Сбой взаимного исключения.** Часто один и тот же совместно используемый ресурс одновременно пытаются использовать несколько пользователей или несколько программ. Например, два пользователя могут попытаться одновременно редактировать один и тот же файл. Если эти обращения не контролируются должным образом, возможно возникновение ошибок. Для корректной работы требуется некоторый механизм взаимного исключения, позволяющий в каждый момент времени выполнять обновление файла только одной программе. Правильность реализации такого взаимного исключения при всех возможных последовательностях событий крайне трудно проверить.

- **Недетерминированное поведение программы.** Результат работы каждой программы обычно должен зависеть только от ее ввода и не должен зависеть от работы других программ, выполняющихся в этой же системе. Однако в условиях совместного использования памяти и процессора программы могут влиять на работу друг друга, переписывая общие области памяти непредсказуемым образом. При этом результат работы программ может зависеть от порядка, в котором они выполняются.
- **Взаимоблокировки.** Возможны ситуации, в которых две или более программ “зависают”, ожидая действий друг друга. Например, двум программам может понадобиться, чтобы устройства ввода-вывода выполнили некоторую операцию (например, копирование с диска на магнитную ленту). Одна из этих программ осуществляет управление одним из устройств, а другая — другим. Каждая из них ждет, пока другая программа освободит нужный ресурс. Выйти из такой тупиковой ситуации может помочь система распределения ресурсов.

Для решения перечисленных проблем нужен систематический метод, основанный на слежении за различными выполняющимися процессором программами и на управлении ими. В основе такого метода лежит концепция процесса. Мысленно процесс можно разделить на три компонента.

1. Выполняющаяся программа.
2. Данные, необходимые для ее работы (переменные, рабочее пространство, буферы и т.д.).
3. Контекст выполнения программы.

Последний элемент является очень важным. **Контекст выполнения** (execution context), или **состояние процесса** (process state), включает в себя всю информацию, нужную операционной системе для управления процессом и процессору — для его выполнения. Данные, характеризующие это состояние, включают в себя содержимое различных регистров процессора, таких как счетчик команд и регистры данных. Сюда же входит информация, использующаяся операционной системой, такая как приоритет процесса и сведения о том, находится ли данный процесс в состоянии ожидания какого-то события, связанного с вводом-выводом.

На рис. 2.8 показан пример реализации процессов. Два процесса, А и В, находятся в различных областях основной памяти. Другими словами, каждому процессу отведен блок памяти, в котором содержится код программы, данные и информация о состоянии процесса. Каждый процесс заносится в список процессов, который создается и поддерживается операционной системой. Часть этого списка, соответствующая определенному процессу, содержит указатель на размещение этого процесса в памяти. Кроме того, сюда же частично или полностью может входить и информация о состоянии процесса. Остальные данные могут храниться в другом месте, возможно, в самом процессе (как показано на рис. 2.8). В регистре индекса процесса содержится индекс выполняющегося в текущий момент времени процесса, идентифицирующий его в списке процессов. Содержимое счетчика команд указывает на очередную инструкцию, которую нужно выполнить. Базовый и граничный регистры задают область памяти, занимаемую процессом. В базовый регистр заносится адрес начальной ячейки этой области, а в граничный — ее размер (в байтах или словах). Содержимое счетчика команд и всех ссылок на данные

отсчитывается от значения базового регистра; по своей величине эти ссылки не могут превосходить значение граничного регистра (что защищает процессы от воздействия один на другой).

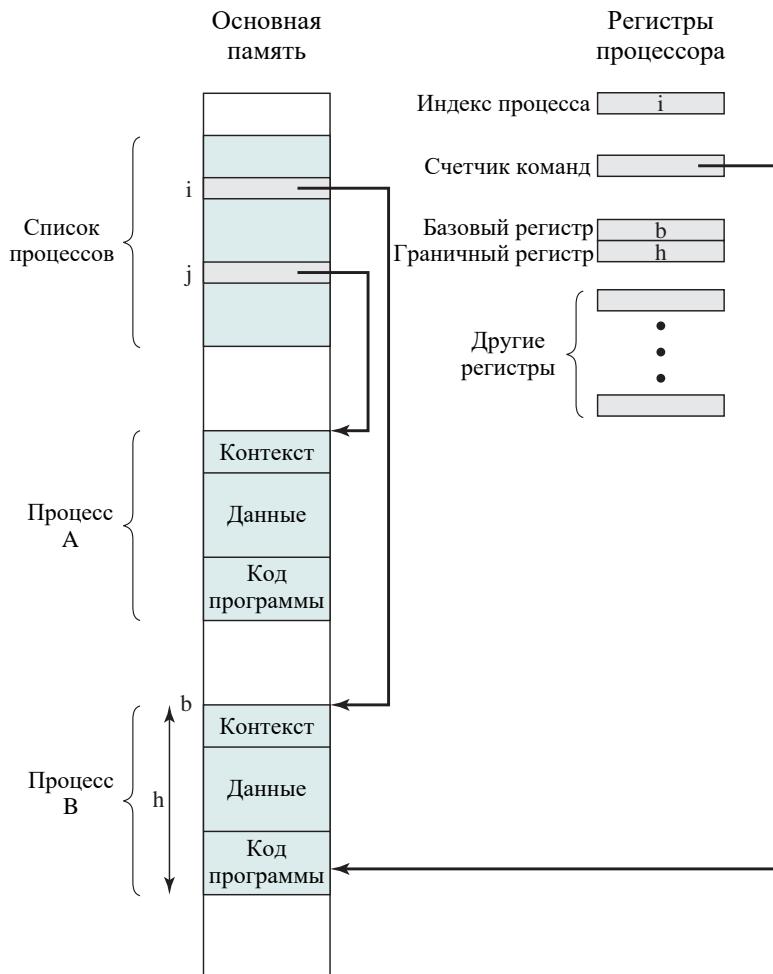


Рис. 2.8. Типичная реализация процессов

Регистр индекса процесса, изображенный на рис. 2.8, указывает, что выполняется процесс В. До этого выполнялся процесс А, но он временно прерван. Содержимое всех регистров в момент прекращения этого процесса записано в виде данных о состоянии процесса. Впоследствии операционная система сможет вернуться к выполнению процесса А; при этом будет сохранен контекст выполнения процесса В и восстановлен контекст выполнения процесса А. Когда в счетчик команд загружается значение, указывающее на область кода программы процесса А, автоматически возобновляется выполнение этого процесса.

Таким образом, процесс реализуется в виде структуры данных. Он может выполнятьсь или находиться в состоянии ожидания. Состояние процесса в каждый момент врем-

ни заносится в специально отведенную область данных. Использование этой структуры позволяет развивать мощные методы координации и взаимодействия процессов. В рамках операционной системы на основе данных о состоянии процесса путем их расширения и добавления в них дополнительной информации о процессе можно разрабатывать новые возможности операционных систем (например, приоритеты процессов). При чтении книги нам встретится множество примеров использования описанной структуры в решении задач, возникающих при разработке многозадачных и многопользовательских операционных систем.

Последний момент, с которым мы вкратце познакомим вас здесь, — концепция потока (thread). По сути, единый процесс, который получает определенные ресурсы, может быть разбит на несколько параллельно работающих потоков, которые совместно выполняют работу процесса. Это привносит новый уровень параллельности, управляемый аппаратным и программным обеспечением.

## Управление памятью

Лучше всего потребности пользователя удовлетворяются вычислительной средой, поддерживающей модульное программирование и гибкое использование данных. Нужно обеспечить эффективный и систематичный контроль над размещением данных в запоминающем устройстве со стороны управляющих программ операционной системы. Исходя из сформулированных требований, операционная система должна выполнять такие функции.

- Изоляция процессов.** Операционная система должна следить за тем, чтобы ни один из независимых процессов не смог изменить содержимое памяти, отведенное другому процессу, и наоборот.
- Автоматическое размещение и управление.** Программы должны динамически размещаться в памяти в соответствии с определенными требованиями. Распределение памяти должно быть прозрачным для программиста. Таким образом, программист будет избавлен от необходимости следить за ограничениями, связанными с конечностью памяти, а операционная система повышает эффективность работы вычислительной системы, выделяя заданиям только тот объем памяти, который им необходим.
- Поддержка модульного программирования.** Программист должен иметь возможность определять модули программы, а также динамически их создавать, уничтожать и изменять их размер.
- Защита и контроль доступа.** При совместном использовании памяти на каждом ее иерархическом уровне есть вероятность, что одна программа обратится к пространству памяти другой программы. Такая возможность может понадобиться, если она заложена в принцип работы данного приложения. С другой стороны, это угроза целостности программ и самой операционной системы. Операционная система должна следить за тем, каким образом различные пользователи могут осуществлять доступ к различным областям памяти.
- Долгосрочное хранение.** Многим приложениям требуется средства, с помощью которых можно было бы хранить информацию в течение длительного периода времени после выключения компьютера.

Обычно операционные системы выполняют эти требования с помощью средств виртуальной памяти и файловой системы. Файловая система обеспечивает долгосрочное хранение информации, помещаемой в именованные объекты, которые называются файлами. Файл — это удобная для программиста концепция, доступ к которой и защита которой осуществляются операционной системой.

**Виртуальная память** — это функциональная возможность, позволяющая программистам рассматривать память с логической точки зрения, не заботясь о наличии физической памяти достаточного объема. Принципы работы с виртуальной памятью были разработаны, чтобы задания нескольких пользователей, выполняясь параллельно, могли одновременно присутствовать в основной памяти. При такой организации процессов нет задержки между их выполнением: как только один из процессов заносится на вспомогательное запоминающее устройство, считывается следующий процесс. Из-за различий в количестве памяти, требуемемся для разных процессов, при переключении процессора с одного процесса на другой трудно компактно разместить их в основной памяти. Поэтому были разработаны системы со страничной организацией памяти, при которой процесс разбивается на блоки фиксированного размера, которые называются страницами. Обращение программы к слову памяти происходит по **виртуальному адресу** (*virtual address*), который состоит из номера страницы и смещения относительно ее начала. Страницы одного и того же процесса могут быть разбросаны по всей основной памяти. Системы со страничной организацией обеспечивают динамическое отображение виртуального адреса, использующегося программой, и **реальным** (*real address*), или **физическим**, адресом основной памяти.

Следующим логическим шагом развития в этом направлении (при наличии аппаратного обеспечения, позволяющего выполнять динамическое отображение) было исключение требования, чтобы все страницы процесса одновременно находились в основной памяти; достаточно, чтобы все они хранились на диске. Во время выполнения процесса только некоторые его страницы находятся в основной памяти. Если программа обращается к странице, которая там отсутствует, аппаратное обеспечение, управляющее памятью, обнаруживает это и организует загрузку недостающих страниц. Такая схема называется **виртуальной памятью**; она проиллюстрирована на рис. 2.9.

Аппаратное обеспечение процессора вместе с операционной системой предоставляют пользователю “**виртуальный процессор**”, который имеет доступ к виртуальной памяти. Это хранилище может быть организовано в виде линейного адресного пространства или в виде коллекции сегментов, представляющих собой смежные блоки переменной длины. При каждом из этих способов организации можно обращаться к ячейкам виртуальной памяти, в которых содержатся программа и ее данные, с помощью команд языка программирования. Чтобы изолировать процессы один от другого, каждому из них можно выделить свою область памяти, не пересекающуюся с областью памяти другого процесса. Общее использование памяти можно организовать, частично перекрывая части двух пространств виртуальной памяти. Файлы хранятся на долговременном запоминающем устройстве. Чтобы с ними могли работать программы, файлы или их фрагменты могут копироваться в виртуальную память.

На рис. 2.10 поясняется концепция адресации в схеме виртуальной памяти. Хранилище состоит из основной памяти, открытой для прямого доступа (осуществляемого с помощью машинных команд), а также более медленной вспомогательной памяти, доступ к которой осуществляется косвенно, путем загрузки блоков в основную память. Между процессором и памятью находятся аппаратные средства (модули управления памятью) для трансляции адресов.

A.1			
	A.0	A.2	
	A.5		
B.0	B.1	B.2	B.3
		A.7	
	A.9		
		A.8	
	B.5	B.6	

Основная память

Основная память состоит из кадров (блоков, или фреймов) фиксированного размера, равного размеру страницы. Для работы программы некоторые (или все) ее страницы должны находиться в основной памяти

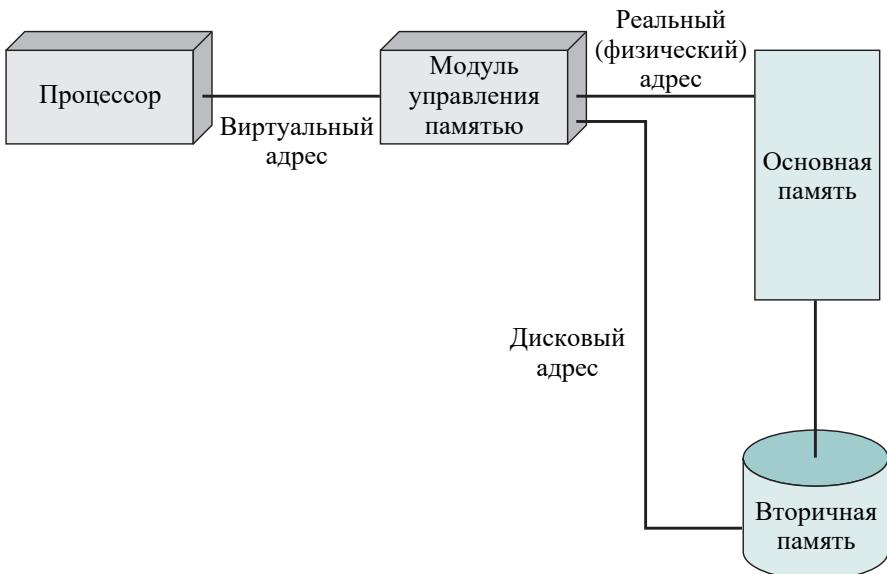


Диск

Вторичная память (диск) может хранить большое количество страниц фиксированного размера. Пользовательская программа состоит из некоторого количества страниц. Страницы всех программ и операционной системы хранятся на диске, как и файлы

Рис. 2.9. Концепция виртуальной памяти

Программы обращаются к ячейкам памяти с использованием виртуальных адресов, транслируемых в ходе обращения в реальные адреса основной памяти. Если происходит обращение к виртуальному адресу, который не загружен в основную память, то один из блоков реальной памяти меняется местами с нужным блоком, который находится во вспомогательной памяти. Во время этого обмена процесс, который обратился к данному адресу, должен быть приостановлен. Задача разработки такого механизма преобразования адресов, который не требовал бы больших дополнительных ресурсов, и такой стратегии размещения данных в хранилище, которая сводила бы к минимуму перемещение данных между различными уровнями памяти, возлагается на разработчика операционной системы.



**Рис. 2.10.** Адресация виртуальной памяти

## Защита информации и безопасность

С ростом популярности систем разделения времени — а впоследствии с появлением компьютерных сетей — возникла проблема защиты информации. В зависимости от обстоятельств природа угрозы, нависшей над определенной организацией, может быть самой разнообразной. Однако в компьютеры и операционные системы могут быть встроены некоторые инструменты общего назначения, поддерживающие различные механизмы защиты и обеспечивающие безопасность. Если говорить в общих чертах, мы сталкиваемся с проблемой контроля над доступом к компьютерным системам и хранящейся в них информации.

Большую часть задач по обеспечению безопасности и защиты информации можно условно разбить на четыре категории.

1. **Доступ.** Связан с защитой системы от постороннего вмешательства.
2. **Конфиденциальность.** Гарантирует невозможность чтения данных неавторизованным пользователем.
3. **Целостность данных.** Защита данных от неавторизованного изменения.
4. **Аутентификация.** Обеспечение надлежащей верификации пользователей и корректности сообщений или данных.

## Планирование и управление ресурсами

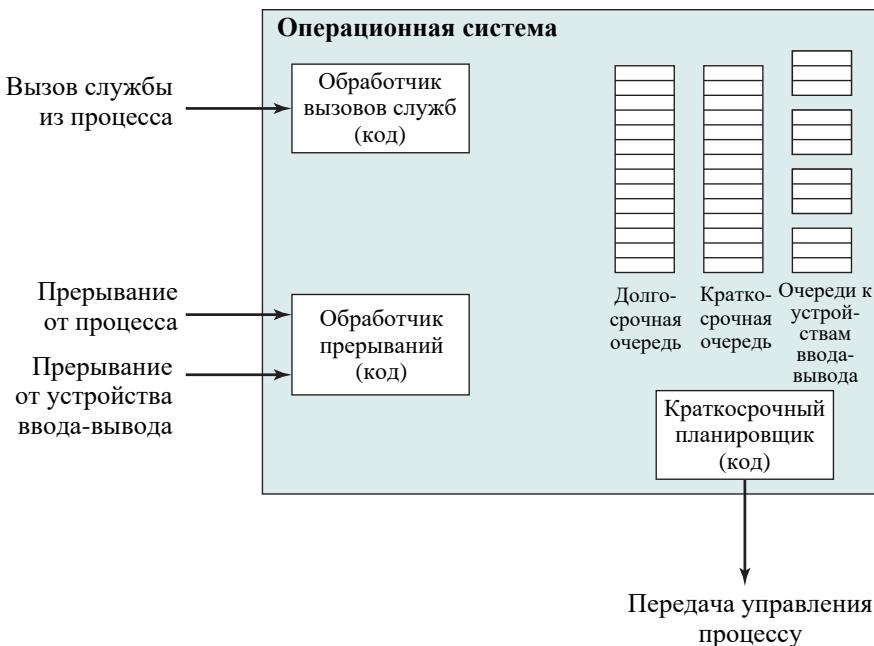
Одной из важных задач операционной системы является управление имеющимися в ее распоряжении ресурсами (основной памятью, устройствами ввода-вывода, процессором), а также планирование их использования между разными активными процессами. При разработке стратегии распределения ресурсов необходимо принимать во внимание следующие факторы.

1. **Равноправность.** Обычно желательно, чтобы всем процессам, претендующим на какой-то определенный ресурс, предоставлялся к нему одинаковый доступ. В особенности это касается заданий, принадлежащих к одному и тому же классу, т.е. заданий с аналогичными требованиями к ресурсам.
2. **Дифференциация отклика.** С другой стороны, может понадобиться, чтобы операционная система по-разному относилась к заданиям различных классов, имеющим различные запросы. Нужно попытаться сделать так, чтобы операционная система выполняла распределение ресурсов в соответствии с целым набором требований. Операционная система должна действовать динамически, в зависимости от обстоятельств. Например, если какой-то процесс ожидает доступа к устройству ввода-вывода, операционная система может спланировать выполнение этого процесса так, чтобы как можно скорее освободить устройство для дальнейшего использования другими процессами.
3. **Эффективность.** Операционная система должна повышать пропускную способность системы, сводить к минимуму время ее отклика и, если она работает в системе разделения времени, обслуживать максимально возможное количество пользователей. Эти требования несколько противоречат друг другу; насущной проблемой исследования операционных систем является поиск нужного соотношения в каждой конкретной ситуации.

Задача управления ресурсами и их распределения типична для исследований в области операционных систем; здесь могут применяться математические результаты, полученные в этой области. Кроме того, важно измерять активность системы, что позволяет следить за ее производительностью и вносить корректизы в ее работу.

На рис. 2.11 показаны основные элементы операционной системы, участвующие в планировании процессов и распределении ресурсов в многозадачной среде. Операционная система поддерживает несколько очередей, каждая из которых является просто списком процессов, ожидающих своей очереди на использование какого-то ресурса. В краткосрочную очередь заносятся процессы, которые (или по крайней мере основные части которых) находятся в основной памяти и готовы к выполнению. Выбор очередного процесса осуществляется краткосрочным планировщиком, или диспетчером. Общая стратегия состоит в том, чтобы каждому находящемуся в очереди процессу давать доступ по очереди; такой метод называют **циклическим** (round-robin). Кроме того, процессам можно присваивать различный приоритет.

В долгосрочной очереди находится список новых процессов, ожидающих возможности использовать процессор. Операционная система добавляет их в систему, перенося из долгосрочной очереди в краткосрочную. В этот момент процессу необходимо выделить определенную часть основной памяти. Таким образом, операционная система должна следить за тем, чтобы не перегрузить память или процессор, добавляя в систему слишком много процессов. К одному и тому же устройству ввода-вывода могут обращаться несколько процессов, поэтому для каждого устройства создается своя очередь (запрос на использование одного и того же устройства ввода-вывода может исходить от разных процессов, и все процессы, ожидающие доступ к данному устройству, выстраиваются в очередь этого устройства). И здесь операционная система должна решать, какому процессу предоставить освободившееся устройство ввода-вывода в первую очередь.



**Рис. 2.11.** Ключевые элементы многозадачной операционной системы

Во время прерывания управление переходит к обработчику прерываний, который является частью операционной системы. В силу своей функциональности процесс может обратиться к некоторой службе операционной системы, например к драйверу устройства ввода-вывода. При этом происходит вызов обработчика обращений к службам, который становится точкой входа в операционную систему. Независимо от того, произошло ли прерывание или обращение к службе, после его обработки планировщик выберет из краткосрочной очереди процесс для выполнения.

Далее в этом разделе приводится чисто функциональное описание; эти модули в различных операционных системах имеют разные особенности и устройство. Большая часть научно-исследовательских усилий в области операционных систем была направлена на выбор алгоритмов и структур данных для этой функциональности, обеспечивающих равноправность, дифференциацию отклика и эффективность.

## 2.4. РАЗРАБОТКИ, ВЕДУЩИЕ К СОВРЕМЕННЫМ ОПЕРАЦИОННЫМ СИСТЕМАМ

Год за годом происходит эволюция структуры и возможностей операционных систем. В последнее время в состав новых операционных систем и новых версий уже существующих операционных систем вошли некоторые структурные элементы, которые внесли большие изменения в природу этих систем. Современные операционные системы отвечают требованиям постоянно развивающегося аппаратного и программного обеспечения и новым угрозам для систем безопасности. Они способны управлять работой многопроцессорных систем, работающих быстрее обычных машин, высокоскоростных сетевых

приспособлений и разнообразных запоминающих устройств, число которых постоянно увеличивается. Из приложений, оказавших влияние на устройство операционных систем, следует отметить мультимедийные приложения, средства доступа к Интернету, а также модель “клиент/сервер”. Что касается безопасности, то в результате возможностей доступа к компьютерам через Интернет значительно возросли потенциальные угрозы их безопасности, а все более сложные атаки (вирусов, червей и взломщиков) оказали глубокое влияние на дизайн операционных систем.

Неуклонный рост требований к операционным системам приводит не только к усовершенствованию их архитектуры, но и к возникновению новых способов их организации. В экспериментальных и коммерческих операционных системах опробуются самые разнообразные подходы и структурные элементы, большинство из которых можно объединить в следующие категории.

- Архитектура микроядра
- Многопоточность
- Симметричная многопроцессорность
- Распределенные операционные системы
- Объектно-ориентированный дизайн

Отличительной особенностью большинства операционных систем до сегодняшнего дня является большое **монолитное ядро**. Ядро операционной системы обеспечивает большинство ее возможностей, включая планирование, работу с файловой системой, сетевые функции, работу драйверов различных устройств, управление памятью и многие другие. Обычно монолитное ядро реализуется как единый процесс, все элементы которого используют одно и то же адресное пространство. В **архитектуре микроядра** ядру отводится лишь несколько самых важных функций, в число которых входят управление адресным пространством, обеспечение взаимодействия между процессами (*interprocess communication — IPC*) и основное планирование. Работу других сервисов операционной системы обеспечивают процессы, которые иногда называют **серверами**. Эти процессы запускаются в пользовательском режиме, и микроядро работает с ними так же, как и с другими приложениями. Такой подход позволяет разделить задачу разработки операционной системы на разработку ядра и разработку сервера. Серверы можно настраивать для требований конкретных приложений или среды. Выделение в структуре системы микроядра упрощает реализацию системы, обеспечивает ее гибкость, а также хорошо вписывается в распределенную среду. Фактически микроядро взаимодействует с локальным и удаленным серверами по одной и той же схеме, что упрощает построение распределенных систем.

**Многопоточность (multithreading)** — это технология, при которой процесс, выполняющий приложение, разделяется на несколько одновременно выполняемых потоков. Ниже приведены основные различия между потоком и процессом.

- **Поток (thread).** Диспетчеризуемая единица работы, включающая контекст процессора (в который входит содержимое счетчика команд и указателя вершины стека), а также собственную область стека (для организации вызова подпрограмм). Команды потока выполняются последовательно; поток может быть прерван при переключении процессора на обработку другого потока.

- **Процесс.** Набор из одного или нескольких потоков, а также связанных с этими потоками системных ресурсов (таких, как область памяти, в которую входят код и данные, открытые файлы, различные устройства). Эта концепция очень близка концепции выполняющейся программы. Разбивая приложение на несколько потоков, программист получает все преимущества модульности приложения и возможность управления связанными с приложением временными событиями.

Многопоточность оказывается весьма полезной для приложений, выполняющих несколько независимых заданий, которые не требуют последовательного выполнения. В качестве примера такого приложения можно привести сервер базы данных, который одновременно принимает и обрабатывает несколько запросов клиентов. Если в пределах одного и того же процесса работают несколько потоков, то при переключении между различными потоками непроизводительный расход ресурсов процессора меньше, чем при переключении между разными процессами. Кроме того, потоки полезны при описанном в последующих главах структурировании процессов, которые являются частью ядра операционной системы.

**Симметричная многопроцессорность** (symmetric multiprocessing — SMP) — термин, относящийся к архитектуре аппаратного обеспечения компьютера (описанной в главе 1, “Обзор компьютерной системы”), а также к поведению операционной системы, соответствующему этой архитектурной особенности. Операционная система симметричной многопроцессорности распределяет процессы и потоки по процессорам. SMP имеет ряд потенциальных преимуществ по сравнению с однопроцессорными системами, включая следующие.

- **Производительность.** Если задание, которое должен выполнить компьютер, можно организовать так, что какие-то части этого задания будут выполняться параллельно, это приведет к повышению производительности по сравнению с однопроцессорной системой с процессором того же типа. Сформулированное выше положение проиллюстрировано на рис. 2.12. В многозадачном режиме в один и тот же момент времени может выполняться только один процесс, тогда как остальные процессы вынуждены ожидать своей очереди. В многопроцессорной системе могут выполняться одновременно несколько процессов, причем каждый из них будет работать на отдельном процессоре.
- **Надежность.** При симметричной многопроцессорной обработке отказ одного из процессоров не приведет к остановке машины, потому что все процессоры могут выполнять одни и те же функции. После такого сбоя система продолжит свою работу, хотя производительность ее несколько снизится.
- **Наращивание.** Добавляя в систему дополнительные процессоры, пользователь может повысить ее производительность.
- **Масштабируемость.** Производители могут предлагать свои продукты в различных по цене и производительности конфигурациях, предназначенных для работы с разным количеством процессоров.

Важно отметить, что перечисленные выше преимущества являются скорее потенциальными, чем гарантированными. Чтобы надлежащим образом реализовать потенциал, заключенный в многопроцессорных вычислительных системах, операционная система должна предоставлять адекватный набор инструментов и возможностей.

Время →



а) Чередование (много задач, один процессор)



б) Чередование и перекрытие (много задач, два процессора)

 Заблокирован  Выполняется

**Рис. 2.12. Многозадачность многопроцессорность**

Часто можно встретить совместное обсуждение многопоточности и многопроцессорности, однако эти два понятия являются независимыми. Многопоточность — полезная концепция для структурирования процессов приложений и ядра даже на машине с одним процессором. С другой стороны, многопроцессорная система может обладать преимуществами по сравнению с однопроцессорной, даже если процессы не разделены на несколько потоков, потому что в такой системе можно запустить несколько процессов одновременно. Однако обе эти возможности хорошо согласуются между собой, а их совместное использование может дать заметный эффект.

Привлекательной особенностью многопроцессорных систем является то, что наличие нескольких процессоров прозрачно для пользователя — за распределение потоков между процессорами и за синхронизацию разных процессов отвечает операционная система. В этой книге рассматриваются механизмы планирования и синхронизации, которые используются, чтобы все процессы и процессоры были видны пользователю в виде единой системы. Другая задача более высокого уровня — представление в виде единой системы кластера из нескольких отдельных компьютеров. В этом случае мы имеем дело с набором компьютеров, каждый из которых обладает собственной основной и вторичной памятью и своими модулями ввода-вывода. **Распределенная операционная система** создает иллюзию единого пространства основной и вторичной памяти, а также еди-

ной файловой системы. Хотя популярность кластеров неуклонно возрастает и на рынке появляется все больше и больше кластерных продуктов, современные распределенные операционные системы по-прежнему отстают в развитии от одно- и многопроцессорных систем. С подобными системами вы познакомитесь ниже в данной книге.

Одним из последних новшеств в устройстве операционных систем стало использование объектно-ориентированных технологий. **Объектно-ориентированный дизайн** помогает навести порядок в процессе добавления к основному небольшому ядру дополнительных модулей. На уровне операционной системы основанная на объектах структура позволяет программистам настраивать операционную систему, не нарушая ее целостности. Кроме того, этот подход облегчает разработку распределенных инструментов и полноценных распределенных операционных систем.

## 2.5. Отказоустойчивость

Отказоустойчивость означает способность системы или компонента к продолжению нормальной работы, несмотря на наличие ошибок аппаратного или программного обеспечения. Обычно отказоустойчивость предполагает определенную степень избыточности. Отказоустойчивость предназначена для повышения степени надежности системы. Как правило, увеличение отказоустойчивости (и соответственно, повышение надежности) имеет определенную стоимость, либо финансовую, либо выражющуюся в падении производительности (либо и то, и другое одновременно). Таким образом, определение желаемой степени отказоустойчивости должно учитывать, что именно является критическим ресурсом.

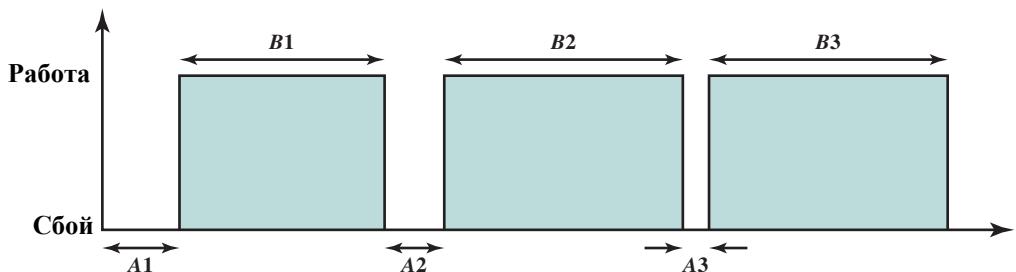
### Фундаментальные концепции

Имеются три основных показателя качества функционирования системы, связанных с отказоустойчивостью: надежность, среднее время наработки на отказ и доступность. Эти концепции разработаны с особым акцентом на аппаратные сбои, но в целом применяются к сбоям как аппаратного, так и программного обеспечения.

**Надежность** (*reliability*) системы определяется как вероятность ее беспроблемной работы до времени  $t$  при условии ее корректной работы в момент времени  $t=0$ . Для операционных систем и компьютеров термин *беспроблемная работа* означает правильное выполнение набора программ и защиту данных от случайного изменения. **Среднее время наработки на отказ** (*mean time to failure* — MTTF) определяется как

$$\text{MTTF} = \int_0^x R(t)dt$$

**Среднее время восстановления** (*mean time to repair* — MTTR) представляет собой среднее время, необходимое для ремонта или замены неисправного элемента. На рис. 2.13 показана связь между MTTF и MTTR.



$$MTTF = \frac{B1 + B2 + B3}{3} \quad MTTR = \frac{A1 + A2 + A3}{3}$$

**Рис. 2.13.** Оперативное состояние системы

Доступность (availability) системы или службы определяется как доля времени, когда система доступна для обслуживания запросов пользователей. Доступность, по сути, является вероятностью того, что система при заданных условиях корректно функционирует в данный момент времени. Время, в течение которого система недоступна, называется **простоем** (downtime); время, в течение которого она доступна, — **временем безотказной работы** (uptime). Доступность системы может быть выражена следующим образом:

$$\text{Доступность} = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}}$$

В табл. 2.4 показаны некоторые распространенные уровни доступности и соответствующие времена простоев в течение года.

**Таблица 2.4. Классы доступности**

Класс	Доступность	Ежегодный простой
Непрерывная работа	1,0	0
Отказоустойчивый	0,99999	5 мин
Восстанавливаемый	0,9999	53 мин
Высокодоступный	0,999	8,3 ч
Обычная доступность	0,99–0,995	44–87 ч

Зачастую среднее время наработки на отказ MTTF является лучшим показателем, чем доступность. Небольшое время простоев в сочетании с краткими периодами безотказной работы могут привести к высокому показателю доступности, но пользователи не смогут воспользоваться всеми услугами системы, если время ее безотказной работы меньше времени, необходимого для завершения действий службы.

## Отказы

Словарь стандартов IEEE определяет **отказ** (fault) как ошибочное состояние аппаратного или программного обеспечения в результате сбоя некоторого компонента, ошибки оператора, физической помехи от окружающей среды, ошибки проектирования, программирования или структуры данных. Стандарт также определяет, что отказ проявляется как 1) дефект аппаратного устройства или компонента (например, короткое замыкание или обрыв соединения) или 2) неправильное действие, процесс или данные в компьютерной программе.

Можно сгруппировать отказы в следующие категории.

- **Постоянные.** Такой сбой, после того как происходит, присутствует в системе постоянно. Неисправность приводит к некорректной работе системы до тех пор, пока неисправный компонент не будет заменен или отремонтирован. Примерами могут служить поломка считывающих головок диска, ошибка программного обеспечения или сгоревшая сетевая плата.
- **Временные.** Сбой, который не сохраняется все время эксплуатации системы. Такие сбои можно разделить на следующие группы.
  - **Преходящие.** Такой сбой однократен. Например, сбой при передаче в некотором бите из-за импульсного шума в линии или изменение бита памяти из-за внешнего излучения.
  - **Периодические.** Сбои, происходящие в разные, непредсказуемые моменты времени. Примером такого периодического сбоя могут быть ошибки, вызванные неплотным соединением, приводящим к кратковременным потерям связи.

В общем случае отказоустойчивость системы обеспечивается путем внесения в нее избыточности. К методам резервирования относятся следующие.

- **Пространственная (физическая) избыточность.** Физическая избыточность предполагает использование нескольких компонентов, которые одновременно выполняют одну и ту же функцию или настроены так, что один компонент доступен в качестве резервной копии, включающейся в случае сбоя другого компонента. Примерами могут служить использование нескольких параллельных схем или резервный сервер доменных имен в Интернете.
- **Временная избыточность.** Временная избыточность включает повторение выполнения функции или операции при обнаружении ошибки. Этот подход эффективен для временных ошибок, но непригоден для постоянных сбоев. Примером является ретрансляция блока данных при обнаружении ошибки передачи, как это делают протоколы управления линиями передачи данных.
- **Информационная избыточность.** Данная избыточность обеспечивает отказоустойчивость путем репликации или кодирования данных таким образом, чтобы ошибки в отдельных битах могли быть обнаружены и исправлены. Примером являются схемы с коррекцией ошибок, используемые в системах памяти, а также методы коррекции ошибок в RAID-дисках, о чем будет рассказано позже.

## Механизмы операционных систем

Ряд методов поддержки отказоустойчивости могут быть включены в программное обеспечение операционных систем. Ряд примеров будет встречаться вам на протяжении всей книги. В следующем списке приведены некоторые из примеров.

- **Изоляция процессов:** как упоминалось ранее в этой главе, процессы обычно изолированы один от другого с точки зрения основной памяти, доступа к файлам и потока выполнения. Структура, предоставляемая операционной системой для управления процессами, обеспечивает определенный уровень защиты от сбояного процесса других процессов.
- **Управление параллелизмом:** в главах 5, “Параллельные вычисления: взаимоисключения и многозадачность”, и 6, “Параллельные вычисления: взаимоблокировка и голодание”, будут обсуждаться некоторые трудности и ошибки, которые могут возникнуть при взаимодействии процессов или обмене информацией между ними. В этих главах будут также рассмотрены методы, используемые для обеспечения корректной работы и восстановления после сбоев, таких как, например, взаимоблокировка.
- **Виртуальные машины:** виртуальные машины, которые будут рассмотрены в главе 14, “Виртуальные машины”, обеспечивают более высокую степень изоляции приложений, а следовательно, и изоляцию сбоев. Виртуальные машины могут также использоваться для обеспечения избыточности, когда одна виртуальная машина выступает в качестве резервной копии для другой.
- **Точки восстановления и откаты:** точка восстановления представляет собой копию состояния приложения, сохраненную в некотором устройстве хранения, защищенном от рассматриваемых сбоев. Откат перезапускает выполнение из ранее сохраненной точки восстановления. При возникновении сбоя состояние приложения откатывается до предыдущей точки восстановления и перезапускается. Этот метод может использоваться для восстановления как после временных, так и после постоянных аппаратных сбоев и определенных типов сбоев программного обеспечения. Системы управления базами данных и транзакциями обычно обладают такими возможностями, встроенными в сами системы.

Можно было бы обсудить гораздо более широкий спектр методов, но полное рассмотрение отказоустойчивости операционных систем выходит за рамки данной книги.

## 2.6. ВОПРОСЫ ПРОЕКТИРОВАНИЯ ОПЕРАЦИОННЫХ СИСТЕМ ДЛЯ МНОГОПРОЦЕССОРНЫХ И МНОГОЯДЕРНЫХ СИСТЕМ

### Операционные системы для SMP

В системе SMP ядро может выполняться на любом процессоре, и обычно каждый процессор самостоятельно планирует свою работу из пула доступных процессов или потоков. Ядро может быть построено как многопроцессное или многопоточное, позволяя

параллельно выполняться нескольким частям ядра. SMP-подход усложняет операционную систему. Проектировщик операционной системы должен решать сложные вопросы совместного использования ресурсов (например, различных структур данных), координации действий (например, доступа к устройствам) нескольких частей операционной системы в одно и то же время. Для решения этих вопросов требуется применение самых современных технологий.

Операционная система, предназначенная для симметричной многопроцессорной системы, управляет процессорами и другими ресурсами компьютера таким образом, чтобы с точки зрения пользователя многопроцессорная система выглядела так же, как и многозадачная однопроцессорная система. Пользователь может создавать приложения с использованием нескольких процессов или нескольких потоков в процессах, не заботясь о том, какое количество процессоров будет доступно — один или несколько. Таким образом, многопроцессорная операционная система должна выполнять все функции многозадачной системы, а также обладать дополнительными возможностями по распределению вычислений среди процессоров. В число особенностей архитектуры такой операционной системы входят следующие.

- **Одновременные параллельные процессы или потоки.** Чтобы несколько различных процессов могли одновременно выполнять один и тот же код ядра, он должен быть реenterабельным. При выполнении несколькими процессорами одного и того же кода ядра (или разных его частей) необходима организация управления таблицами и управляющими структурами ядра, чтобы избежать взаимоблокировок или неправильного выполнения операции.
- **Планирование.** Планирование может выполняться на любом из процессоров, поэтому необходимо предусмотреть механизм, позволяющий избежать конфликтов повреждения данных. При использовании многопоточности на уровне ядра несколько потоков одного и того же процесса могут выполняться на разных процессорах. Планирование в многопроцессорных системах рассматривается в главе 10, “Многопроцессорное планирование и планирование реального времени”.
- **Синхронизация.** В ситуации, когда несколько активных процессов имеют возможность доступа к совместным адресным пространствам или ресурсам ввода-вывода, необходимо позаботиться об их эффективной синхронизации. Синхронизация — это средство, обеспечивающее реализацию взаимоисключений и упорядочение событий. Общепринятым механизмом синхронизации в многопроцессорных операционных системах являются блокировки, описанные в главе 5, “Параллельные вычисления: взаимоисключения и многозадачность”.
- **Управление памятью.** Система управления памятью в многопроцессорной системе должна быть способна разрешать все проблемы, возникающие в однопроцессорных машинах. Кроме того, операционная система должна уметь использовать возможности, предоставляемые аппаратным обеспечением, для достижения наивысшей производительности. Механизмы страничной организации памяти разных процессоров должны быть скординированы, чтобы обеспечить согласованность работы в ситуации, когда несколько процессоров используют одну и ту же страницу или один и тот же сегмент и принимают решение по вопросу замещения страниц.

- **Надежность и отказоустойчивость.** При отказе одного из процессоров операционная система должна обеспечить продолжение корректной работы системы. Планировщик операционной системы (как и другие ее части) должен получить информацию о потере одного из процессоров и соответствующим образом перестроить свои управляющие таблицы.

Поскольку при описании архитектуры многопроцессорной операционной системы, как правило, рассматриваются те же вопросы (с добавлением некоторых других), что и при описании устройства однопроцессорной операционной системы, мы не будем останавливаться на многопроцессорных операционных системах отдельно. Вместо этого по ходу изложения материала книги будем обращаться к вопросам, являющимся специфичными для многопроцессорных систем.

## Вопросы проектирования операционных систем для многоядерных систем

Вопросы проектирования для многоядерных систем включают в себя все вопросы, которые рассматриваются в данном разделе для SMP-систем. Однако возникают и дополнительные проблемы. Одним из вопросов является масштаб потенциального параллелизма. В настоящее время производители многоядерных процессоров предлагают системы с десятью и более ядрами на одном кристалле. С каждым новым поколением технологии процессоров количество ядер, как и количество общего и выделенного кеша памяти, постоянно увеличивается, так что сейчас мы вступаем в эпоху многоядерных систем.

При проектировании многоядерных систем необходимо эффективно использовать всю мощь многоядерной обработки данных и грамотно управлять значительными ресурсами процессоров. Центральной проблемой является обеспечение соответствия параллельной природы многоядерных систем требованиям к производительности приложений. Потенциал параллелизма в современной многоядерной системе существует на трех уровнях. Во-первых, имеется аппаратный параллелизм в рамках каждого ядра процессора, известный как параллелизм уровня команд, который может использоваться программистами приложений и компиляторами (но может и остаться неиспользованным). Во-вторых, существует потенциал для многозадачности и многопоточного выполнения программ в рамках каждого процессора. Наконец, потенциально единое приложение может выполняться параллельными процессами или потоками на нескольких ядрах. Без мощной и эффективной поддержки только что упомянутых последних двух типов параллелизма операционной системой аппаратные ресурсы не будут использоваться эффективно.

По сути, с появлением многоядерных технологий разработчики операционных систем постоянно решают проблемы, как повысить степень использования параллелизма. В настоящее время активно изучаются различные подходы, которые будут использованы в грядущем поколении операционных систем. В этом разделе мы представим две общие стратегии и рассмотрим некоторые их детали в последующих главах.

### Параллелизм в приложениях

Большинство приложений в принципе могут быть подразделены на несколько задач, которые могут выполняться параллельно, причем эти задачи реализуются как несколько процессов, возможно, каждый из них — с несколькими потоками. Трудность заключает-

ся в том, что разработчик должен решить, как разделить работу приложения на независимо выполняемые задачи. То есть разработчик должен решить, какие части могут (или должны) быть выполнены асинхронно, или параллельно. В основном процесс разработки параллельных программ поддерживается компилятором и возможностями языка программирования. Операционная система может поддерживать этот процесс как минимум путем эффективного распределения ресурсов среди параллельных задач, определенных разработчиком.

Одним из наиболее эффективных инициатив по поддержке разработчиков является технология Grand Central Dispatch (GCD), реализованная в последних версиях операционных систем iOS и Mac OS X на базе UNIX. GCD представляет собой технологию поддержки многоядерных процессоров. Она не помогает разработчику решить, как разделить задачу или приложение на отдельные параллельные части. Но после того как разработчик определил, что именно можно выделить в отдельную задачу, технология GCD облегчает выполнение этой задачи.

По сути, GCD представляет собой механизм пула потоков, в котором операционная система отображает задания на потоки, представляющие доступную степень параллелизма (плюс потоки, блокируемые вводом-выводом). Windows (начиная с версии 2000) также использует механизм пула потоков; подобные пулы потоков активно используются в серверных приложениях многие годы. Новинкой в GCD является расширение для языков программирования, которое позволяет использовать безымянные функции (именуемые блоками) в качестве способа указания задач. Следовательно, GCD не является крупным эволюционным шагом. Тем не менее это новый и ценный инструмент для использования доступных параллелизма в многоядерных системах.

Один из лозунгов Apple для GCD — “острова сериализации в море параллелизма”. Он относится к реальной практике, добавляя больший параллелизм в выполнение обычных настольных приложений. Эти острова изолируют разработчиков от сложных проблем одновременного доступа к данным, взаимоблокировок и прочих ловушек многопоточности. Разработчикам предлагается просто определить функции в их приложениях, которые желательно выполнять вне основного потока, даже если они состоят из нескольких последовательных или частично взаимозависимых задач. GCD позволяет легко разделить модуль работы на части при сохранении существующего порядка и зависимостей между подзадачами. В последующих главах мы рассмотрим некоторые подробности работы с GCD.

## Виртуальная машина

Этот альтернативный подход заключается в признании, что при постоянно растущем количестве ядер на чипе попытка программировать отдельные ядра для поддержки нескольких приложений может оказаться просто неверным использованием ресурсов [117]. Если вместо этого позволить одному или нескольким ядрам быть выделенными для конкретного процесса, а затем предоставить процессору самостоятельно направлять свои усилия на выполнение этого процесса, то таким образом можно избежать множества накладных расходов, связанных с переключениями и планированием задач. Многоядерная операционная система может выступать в качестве гипервизора, который принимает решения высокого уровня о выделении ядер приложениям, но, кроме этого, мало заботится о распределении ресурсов.

Обоснование такого подхода заключается в следующем. На раннем этапе развития вычислительной техники одна программа выполнялась на одном процессоре. При мно-

гозадачности у каждого приложения возникает иллюзия, что оно работает на выделенном для него процессоре. Многозадачность основана на концепции процесса, который представляет собой абстракцию среды выполнения. Для управления процессами операционная система требует защищенного пространства, с которым не могут взаимодействовать пользователи и программы. С этой целью была разработана возможность работы в режиме ядра и в пользовательском режиме. По сути, режим ядра и пользовательский режим работы разделяют процессор на два. Однако эти виртуальные процессоры начинают борьбу за внимание реального процессора. Накладные расходы переключения между всеми этими процессорами начинают расти до точки, когда начинает страдать отклик системы, в особенности при наличии нескольких ядер. Но в многоядерных системах можно подумать об отбрасывании различий между режимами пользователя и ядра. При таком подходе операционная система действует в большей степени как гипервизор. Сами программы берут на себя многие из функций управления ресурсами. Операционная система назначает приложению процессор и некоторую память, и программа, используя сгенерированные компилятором метаданные, сама знает, как использовать эти ресурсы.

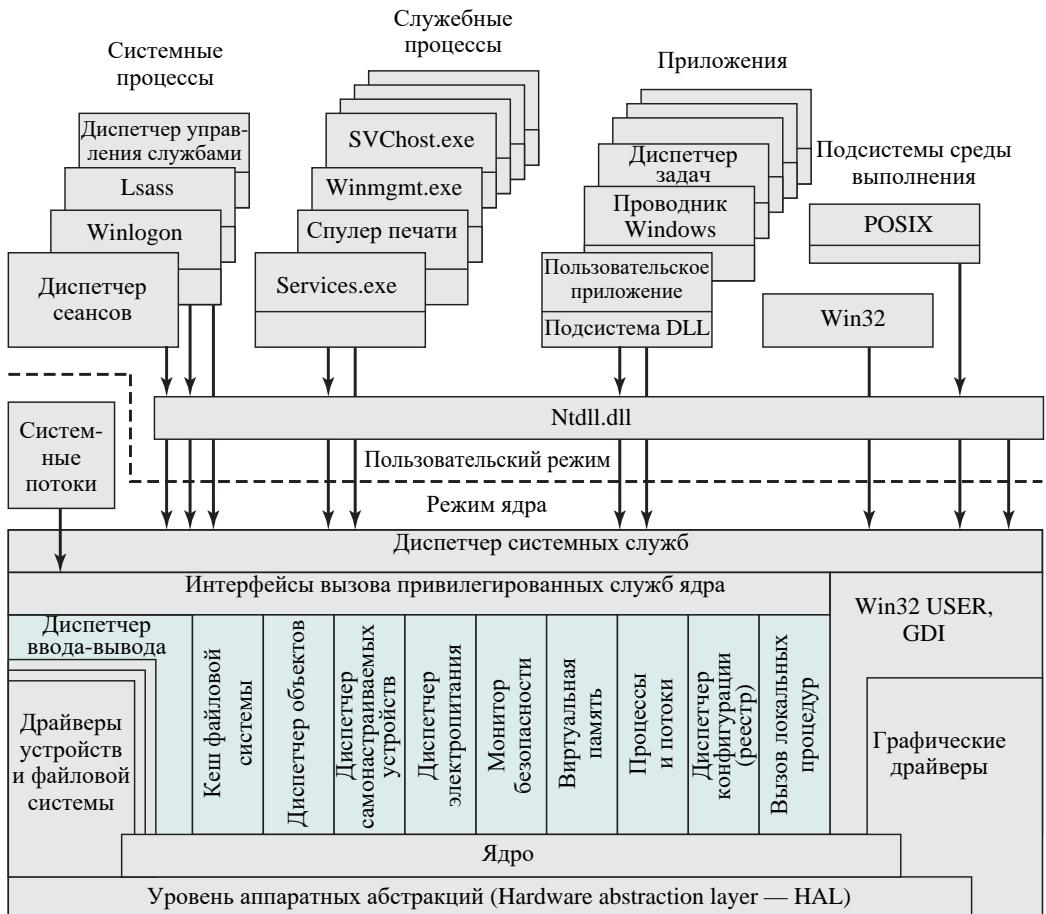
## 2.7. ОБЗОР ОПЕРАЦИОННОЙ СИСТЕМЫ MICROSOFT WINDOWS

### Основы

Впервые Microsoft использовала имя “Windows” в 1985 году для операционной среды, расширяющей возможности примитивной операционной системы MS-DOS, которая успешно использовалась на ранних персональных компьютерах. Такая комбинация Windows/MS-DOS в конечном итоге была заменена новой версией Windows, известной как Windows NT, впервые выпущенной в 1993 году и предназначеннной для ноутбуков и настольных систем. Хотя основная внутренняя архитектура остается примерно той же, что и в Windows NT, сама операционная система продолжает развиваться и дополняться новыми функциями и возможностями. Последняя версия на момент написания этой книги — Windows 10. Windows 10 включает в себя возможности предыдущей операционной системы для настольных и портативных компьютеров Windows 8.1, а также версий Windows, предназначенных для мобильных устройств для Интернета вещей (Internet of Things — IoT). Windows 10 также включает в себя программное обеспечение Xbox One. В результате единая унифицированная операционная система Windows 10 поддерживает настольные компьютеры, ноутбуки, смартфоны, планшеты и Xbox One.

### Архитектура

На рис. 2.14 представлена общая структура Windows. Как и почти все операционные системы, Windows отделяет программное обеспечение, ориентированное на прикладные программы, от ядра операционной системы. К последним относятся исполняющая система, микроядро, драйверы устройств и уровень аппаратных абстракций (hardware abstraction layer — HAL), которые работают в режиме ядра. Программы, выполняющиеся в этом режиме, имеют доступ к системным данным и к аппаратному обеспечению. Остальные программы, работающие в пользовательском режиме, имеют ограниченный доступ к системным данным.



Lsass — сервер проверки подлинности локальной системы безопасности

Выделенная область показывает исполнительную систему

POSIX — интерфейс переносимых операционных систем

GDI — интерфейс графического устройства

DLL — динамически подключаемая библиотека

**Рис. 2.14.** Архитектура Windows [212]

### Организация операционной системы

Windows присуще четкое разделение на модули. Каждая функция системы управляется только одним компонентом операционной системы. Остальные ее части и все приложения обращаются к этой функции через стандартный интерфейс. Доступ к основным системным данным можно получить только через определенные функции. В принципе любой модуль можно удалить, обновить или заменить, не переписывая всю систему или стандартный интерфейс прикладного программирования (application program interface — API).

Далее перечислены компоненты Windows, работающие в режиме ядра.

- **Исполнительная система.** Содержит основные службы операционной системы, такие как управление памятью, процессами и потоками, безопасность, ввод-вывод и межпроцессное взаимодействие.
- **Ядро.** Управляет работой процессоров. Ядро управляет планированием потоков, переключением процессов, обработкой исключений и прерываний, а также много-процессорной синхронизацией. В отличие от остальной части исполняющей системы и уровня пользователя, код самого ядра не выполняется потоками.
- **Уровень аппаратных абстракций.** Выполняет отображение обобщенных команд и ответов аппаратного обеспечения на уникальные команды и ответы аппаратного обеспечения конкретной платформы. Изолирует операционную систему от различных аппаратных платформ. HAL делает системную шину, контроллер памяти прямого доступа (DMA), контроллер прерываний, системные таймеры и контроллер памяти разных компьютеров выглядящими одинаково для компонентов исполнительной системы и ядра. Он также обеспечивает необходимую для SMP поддержку (о чем будет сказано ниже).
- **Драйверы устройств.** Динамические библиотеки, расширяющие функциональность исполнительной системы. К ним относятся драйверы устройств аппаратного обеспечения, которые транслируют пользовательские вызовы функций ввода-вывода в запросы к конкретным аппаратным устройствам, и программные компоненты реализации файловых систем, сетевых протоколов и других системных расширений, которые должны выполняться в режиме ядра.
- **Окна и графическая системы.** Реализует функции графического интерфейса пользователя, такие как работа с окнами, управление интерфейсом пользователя и вывод на экран.

Исполнительная система Windows включает компоненты для определенных системных функций и предоставления работающим в пользовательском режиме программам соответствующего API. Ниже приведено краткое описание каждого из модулей исполнительной системы.

- **Диспетчер ввода-вывода.** Поддерживает каркас, с помощью которого устройства ввода-вывода доступны для приложений, и отвечает за координацию работы драйверов устройств, выполняющих дальнейшую обработку. Диспетчер ввода-вывода реализует все API ввода-вывода Windows и (с помощью диспетчера объектов) следит за безопасностью и именованием устройств, сетевых протоколов и файловых систем. Система ввода-вывода Windows рассматривается в главе 11, “Управление вводом-выводом и планирование дисковых операций”.
- **Диспетчер кеша.** Повышает производительность файлового ввода-вывода путем хранения в основной памяти тех данных с диска, к которым недавно производилось обращение. Кроме того, обеспечивает отложенную запись на диск, некоторое время храня в памяти обновления дисковых файлов.
- **Диспетчер объектов.** Создает и удаляет объекты и абстрактные типы данных исполнительной системы Windows, а также управляет ими. Эти объекты и абстрактные типы данных используются для представления таких ресурсов, как процессы, потоки и объекты синхронизации. Диспетчер объектов обеспечивает выполнение стандартных правил поддержки объектов, именования и безопасности. Кроме того, этот диспетчер создает дескрипторы объектов, в которых содержится информация

о правах доступа и указатель на объект. Объекты операционной системы Windows обсуждаются немного позже.

- **Диспетчер самонастраиваемых устройств.** Определяет, какие драйверы требуются для поддержки определенного устройства, и загружает их.
- **Диспетчер электропитания.** Координирует энергопитание различных устройств и может быть настроен для снижения энергопотребления путем отключения бездействующих устройств, погружения процессора в состояние сна и даже для записи памяти на диск и выключения питания всей системы.
- **Монитор безопасности.** Обеспечивает выполнение правил прав доступа и аудита. Объектно-ориентированная модель операционной системы Windows позволяет сформировать согласованный и единообразный взгляд на безопасность фундаментальных составляющих исполнительной системы. Так, для авторизации доступа и аудита всех защищенных объектов, включая файлы, процессы, адресные пространства и устройства ввода-вывода, операционная система Windows использует одни и те же служебные программы. Безопасность Windows обсуждается в главе 15, “Безопасность операционных систем”.
- **Диспетчер виртуальной памяти.** Управляет виртуальными адресами, физической памятью и файлами подкачки на диске. Контролирует аппаратное обеспечение управления памятью и структурами данных, которые отображают виртуальные адреса адресного пространства процессов на физические страницы памяти компьютера. Управление виртуальной памятью в операционной системе Windows описано в главе 8, “Виртуальная память”.
- **Диспетчер процессов и потоков.** Создает и удаляет объекты, а также следит за процессами и потоками. Управление процессами и потоками в операционной системе Windows рассматривается в главе 4, “Потоки”.
- **Диспетчер конфигурации.** Отвечает за реализацию и управление системным реестром, который является единым хранилищем настроек и параметров как для всей системы, так и для каждого пользователя.
- **Расширенный вызов локальных процедур.** Это средство (advanced local procedure call — ALPC) обеспечивает эффективный механизм межпроцессного вызова процедур для обмена информацией между локальными процессами, реализующими службы и подсистемы. ALPC схоже с вызовом удаленных процедур (remote procedure call — RPC), используемым при распределенных вычислениях.

### *Процессы пользовательского режима*

Windows поддерживает четыре основных типа процессов пользовательского режима.

1. **Специальные системные процессы.** К таким процессам относятся служебные программы, которые не вошли в операционную систему Windows, например процесс входа в систему, система аутентификации и диспетчер сессий.
2. **Служебные процессы.** Очередь заданий принтера, запись событий, пользовательские компоненты для взаимодействия с драйверами устройств, различные сетевые службы и многое другое. Службы используются как Microsoft, так и сторонними разработчиками для расширения функциональности системы, поскольку являются единственным средством выполнения фоновой пользовательской активности в системе Windows.

3. **Подсистемы среды.** Предоставляют приложениям пользователя службы Windows, обеспечивая, таким образом, среду операционной системы. Поддерживаются такие подсистемы, как Win32 и POSIX. В каждую подсистему среды входят динамически подключаемые библиотеки, преобразующие вызовы приложений пользователя в вызовы ALPC и/или вызовы Windows.
4. **Пользовательские приложения.** Выполнимые файлы (EXE) и динамически подключаемые библиотеки (DLL), обеспечивающие пользователям возможность применения системы. В общем случае выполнимые файлы и динамически подключаемые библиотеки ориентированы на подсистему конкретной среды, хотя некоторые из программ, предоставляемых в качестве части операционной системы, используют естественные системные интерфейсы (NT API). Имеется также поддержка выполнения 32-разрядных программ в 64-разрядных системах.

Операционная система Windows поддерживает приложения, написанные для разных операционных систем. Эта поддержка обеспечивается с помощью общего набора компонентов ядра, лежащих в основе подсистем операционных сред. Реализация каждой подсистемы среды включает отдельный процесс, который содержит общие структуры данных, привилегии и дескрипторы исполняемых объектов, необходимые для реализации конкретной среды. Такой процесс запускается диспетчером сеансов при первом запуске приложения соответствующего типа. Процесс подсистемы работает от имени пользователя системы, так что исполнительная система защищает его адресное пространство от процессов других пользователей.

Подсистема среды обеспечивает пользовательский интерфейс — графический или командной строки, который определяет внешний вид операционной системы для пользователя. Кроме того, каждая подсистема предоставляет API для этой конкретной среды. Это означает, что приложения, созданные для конкретной операционной среды, должны всего лишь быть перекомпилированы для запуска на Windows. Поскольку интерфейс операционной системы для всех приложений такой же, как и у систем, для которых они были написаны; исходный код не требует внесения изменений.

## Модель “клиент/сервер”

Структура исполнительной системы, защищенных подсистем и приложений выполнена в соответствии с вычислительной моделью “клиент/сервер” — общепринятой моделью распределенных вычислений, которая обсуждается в части VI, “Дополнительные темы”. Эта же архитектура может использоваться внутренне в одной системе, что мы и видим на примере Windows.

Естественный NT API представляет собой набор служб на базе ядра, которые обеспечивают основные абстракции, используемые в системе, такие как процессы, потоки, виртуальная память, ввода-вывода и коммуникации. Windows предоставляет гораздо больший набор служб, используя модель “клиент/сервер” для реализации функциональности пользовательских процессов. Как подсистемы среды, так и службы пользовательского режима Windows реализованы как процессы, которые общаются с клиентами с использованием RPC. Каждый серверный процесс ожидает от клиента запрос к одной из его служб (например, к службе памяти, службе создания процессов или сетевых служб). Клиент, который может быть прикладной программой или другой серверной программой, запрашивает службу путем отправки сообщения. Сообщение направляется через

исполнительную систему соответствующему серверу. Сервер выполняет запрашиваемую операцию и возвращает информацию о состоянии или результаты посредством другого сообщения, отправляемого через исполнительную систему обратно клиенту.

К преимуществам модели “клиент/сервер” можно отнести следующие.

- **Упрощение исполнительной системы.** Можно разработать ряд API, не имеющих конфликтов или дублирования по отношению к исполнительной системе. Новые API могут быть легко добавлены в систему.
- **Повышение надежности.** Каждый новый сервер запускается вне ядра, в виде отдельного процесса, которому отводится своя область памяти, защищенная от других серверов. Сбой в работе одного из серверов не приводит к аварийному отказу или повреждению остальной части операционной системы.
- **Приложениям с помощью RPC предоставляются однотипные средства обмена информацией с исполнительной системой без потери гибкости.** Процесс передачи сообщения скрыт от клиента функциями-заглушками, которые представляют собой небольшие фрагменты кода вокруг вызовов RPC. При вызове приложением API заглушка пакует переданные при вызове параметры и передает их в виде сообщения процессу сервера, реализующему этот вызов.
- **Модель является базой для распределенных вычислений.** Обычно распределенные вычисления используют модель “клиент/сервер” с реализацией удаленных вызовов процедур посредством распределенных модулей клиентов и серверов, а также путем обмена сообщениями между клиентами и серверами. В операционной системе Windows локальный сервер может передавать сообщение от локального приложения-клиента для обработки на удаленном сервере. Клиентам нет нужды знать о том, как обрабатываются их запросы — локально или удаленно. Способ обработки может изменяться динамически в зависимости от загруженности системы и от изменений конфигурации.

## ПОТОКИ И СИММЕТРИЧНАЯ МНОГОПРОЦЕССОРНОСТЬ

Возможности поддержки потоков и поддержки симметричной многопроцессорности, о которых мы говорили в разделе 2.4, — две важные характеристики операционной системы Windows. Ниже перечислены основные возможности поддержки потоков и SMP в операционной системе Windows [212].

- Служебные программы операционной системы могут выполняться на любом из свободных процессоров; различные программы могут выполняться одновременно на разных процессорах.
- Операционная система Windows поддерживает выполнение одного процесса, разделенного на несколько потоков. Эти потоки могут выполняться одновременно на нескольких процессорах.
- Серверные процессы могут использовать несколько потоков для одновременной обработки запросов, поступающих от разных клиентов.
- Операционная система Windows предоставляет механизмы совместного использования данных и ресурсов различными процессами, а также гибкие возможности обмена информацией между процессами.

## Объекты Windows

Хотя ядро операционной системы Windows написано на языке программирования С, его дизайн в значительной мере следует концепциям объектно-ориентированного проектирования. Этот подход способствует совместному использованию ресурсов и данных различными процессами, а также защите ресурсов от несанкционированного доступа. Среди ключевых объектно-ориентированных концепций, использованных в операционной системе Windows, следует упомянуть следующие.

- **Инкапсуляция.** Объект состоит из одного или нескольких элементов данных (*атрибутов*) и одной или нескольких процедур, выполняемых над этими данными (*методы, сервисы*). Единственный способ получить доступ к данным объекта — запросить один из его методов (сервисов). Таким образом, данные объекта легко защитить от несанкционированного или некорректного использования (например, от попытки выполнить невыполнимый фрагмент данных).
- **Классы объектов и экземпляры.** Класс объекта представляет собой шаблон, в котором перечислены его атрибуты и сервисы, а также определены некоторые его характеристики. При необходимости операционная система может создавать экземпляры объектов класса. Например, имеется класс одиночных процессов, объектом которого является текущий процесс. Такой подход упрощает создание объектов и управление ими.
- **Наследование.** Хотя реализация этого механизма требует ручного кодирования, исполнительная система использует его для расширения возможностей объектов путем добавления новых возможностей. Каждый класс исполнительной системы основан на базовом классе, который определяет виртуальные методы для поддержки создания, именования, обеспечения безопасности и удаления объектов. Объекты диспетчеров представляют собой объекты исполнительной системы, которые наследуют свойства объекта события, так что они могут использовать обычные методы синхронизации. Другие типы объектов, такие как класс устройства, позволяют классам конкретных устройств наследовать базовый класс и добавлять к нему данные и методы.
- **Полиморфизм.** Внутренне для управления объектами любого типа операционная система Windows использует общий набор функций API — в этом и заключается ее полиморфизм. Однако Windows не является полностью полиморфной, потому что в ее состав входит множество API для конкретных типов объектов.

Читатель, не знакомый с объектно-ориентированными концепциями, должен обратиться к приложению Г, “Объектно-ориентированное проектирование”.

Не все сущности операционной системы Windows являются объектами. Объекты используются в тех случаях, когда данные открыты для доступа в пользовательском режиме, а также при совместном использовании данных и ограничении доступа. Среди представляемых объектами сущностей — файлы, процессы, потоки, семафоры, таймеры и графические окна. Windows создает все типы объектов и управляет ими одним и тем же способом — с помощью диспетчера объектов. Этот диспетчер отвечает за создание и уничтожение объектов, нужных для работы приложений, а также за предоставление доступа к сервисам и данным объектов.

Каждый объект исполнительной системы (иногда эти объекты называются объектами ядра, чтобы отличать их от объектов пользовательского уровня, не имеющих отношения к исполнительной системе) находится в области памяти, выделяемой ядром, доступ к которой имеют только компоненты ядра. Некоторые элементы структуры данных присущи объектам всех типов (например, имена объектов, параметры безопасности, счетчик использований). С другой стороны, каждый отдельный тип объектов имеет свои специфические элементы (например, приоритет потока объекта). Поскольку структуры данных таких объектов являются частью адресного пространства процесса, доступного только ядру, приложение не может ни размещать в памяти эти структуры данных, ни непосредственно считывать или записывать в них информацию. Вместо этого приложения работают с объектами опосредованно, через набор функций для работы с объектами, которые поддерживаются исполнительной системой. Когда создается объект для какого-нибудь приложения, последнему возвращается дескриптор созданного объекта. По сути, дескриптор является индексом записи в таблице исполнительной системы для данного процесса, которая содержит указатель на этот объект. Этот дескриптор объекта может использоваться любым потоком этого процесса при вызове функций Win32, работающих с объектами, или может быть дублирован в другой процесс.

С объектами может быть связана информация о безопасности, представленная в виде дескриптора безопасности (Security Descriptor — SD). Эта информация используется для ограничения доступа к объекту на основе содержимого объекта, описывающего определенного пользователя. Например, процессом может быть создан объект, являющийся именованным семафором, открывать и использовать который будет позволено лишь некоторым пользователям. В дескрипторе безопасности этого семафора могут быть перечислены пользователи, которым разрешен (или запрещен) к нему доступ, а также тип разрешенного доступа (для чтения, записи, изменения и т.д.).

В операционной системе Windows объекты могут быть именованными или неименованными. Если при работе процесса создается неименованный объект, то диспетчер объектов возвращает дескриптор этого объекта. Впоследствии обратиться к этому объекту можно будет только через его дескриптор. Дескрипторы могут наследоваться дочерними процессами или дублироваться для передачи между процессами. У именованного объекта, кроме того, есть имя, с помощью которого другие процессы могут получить его дескриптор. Например, если нужно, чтобы процесс A выполнялся синхронно с процессом B, в нем можно создать объект-событие, а затем передать его имя процессу B, в котором это событие будет использовано для синхронизации. Однако если нужно синхронизовать два потока одного и того же процесса A, то в нем можно создать неименованный объект-событие, потому что другие процессы не должны ничего о нем знать.

Имеется две категории объектов, используемых операционной системой Windows для синхронизации.

- **Объекты диспетчера.** Подмножество объектов исполнительной системы, которые могут ожидаться потоками для диспетчеризации и синхронизации операций системы на уровне потоков. Эти объекты описаны в главе 6, “Параллельные вычисления: взаимоблокировка и голодание”.
- **Объекты управления.** Объекты этого типа используются компонентами ядра для управления операциями процессора в областях, не управляемых обычным планированием потоков. Объекты управления ядра перечислены в табл. 2.5.

**Таблица 2.5. Объекты управления ядра Windows**

<b>Асинхронный вызов процедуры</b>	Используется для прерывания выполнения определенного потока и вызова процедуры в указанном режиме процессора
Отложенный вызов процедуры	Используется для откладывания обработки прерывания во избежание задержки аппаратных прерываний. Используется также для реализации таймеров и межпроцессорных сообщений
Прерывание	Используется для связи источника прерывания с программой обслуживания прерывания посредством записи из таблицы диспетчирования прерываний (Interrupt Dispatch Table — IDT). Такая таблица, используемая для диспетчеризации прерываний, имеется у каждого процессора
Процесс	Представляет собой виртуальное адресное пространство и управляющую информацию, которые необходимы для выполнения набора потоков. В процессе содержится указатель на карту адресов, список готовых к выполнению потоков, список всех потоков процесса, совокупное время выполнения всех потоков процесса, а также базовый приоритет
Поток	Представляет объекты потоков, включая приоритет планирования и квантования, а также информацию о том, на каких процессорах может выполняться поток
Профиль	Используется в качестве меры при распределении времени выполнения в пределах блока кода. Профиль может быть определен как для кода пользователя, так и для системного кода

Операционная система Windows не является в полной мере объектно-ориентированной. Она реализована не на объектно-ориентированном языке программирования. Структуры данных, содержащиеся в компоненте исполнительной системы, не представлены в виде объектов. Тем не менее Windows иллюстрирует мощь объектно-ориентированной технологии и ее использование при проектировании операционных систем.

## 2.8. ТРАДИЦИОННЫЕ СИСТЕМЫ UNIX

### Историческая справка

Изначально операционная система UNIX была разработана компанией Bell Labs и запущена в эксплуатацию в 1970 году на системе PDP-7. В результате разработки системы UNIX в компании Bell Labs, а впоследствии — и в других местах, появились различные версии этой операционной системы. Первой значительной вехой стал перенос системы UNIX с PDP-7 на PDP-11. Это послужило первым указанием на тот факт, что система UNIX может быть использована в качестве операционной системы на всех компьютерах. Вторым важным этапом развития этой системы стало то, что она была переписана на языке программирования С. Для того времени это было неслыханно. Считалось, что такая сложная программа, какой является операционная система, для которой важным параметром является время ее работы, должна быть написана только на языке ассемблера.

Причины такого мнения включают в себя следующее.

- Память (как основная, так и вторичная) была в то время малого размера и дорогой по сегодняшним меркам, поэтому было крайне важно эффективное ее использование, включавшее различные методы перекрытия в памяти различных сегментов кода и данных и применение самомодифицирующегося кода.
- Несмотря на доступность компиляторов с 1950-х годов, в компьютерной промышленности было распространено (не изжитое и поныне) скептическое отношение к качеству автоматически генерированного кода. При малых доступных ресурсах небольшой эффективный (как с точки зрения времени выполнения, так и с точки зрения требуемой памяти) код имеет важное значение.
- В то время и процессор, и шина были относительно медленными, так что экономия тактов процессора могла привести к существенному различию времени выполнения.

Реализация на языке С продемонстрировала преимущество языка программирования высокого уровня если не для всех, то для подавляющего большинства фрагментов системного кода. В настоящее время почти все реализации операционной системы UNIX написаны на С.

Ранние версии UNIX были очень популярны в компании Bell Labs. В 1974 году система UNIX была впервые описана в техническом журнале [206], что вызвало к ней большой интерес. Лицензии на UNIX были предоставлены коммерческим организациям и университетам. Версия 6 этой системы, появившаяся в 1976 году, стала первой широко используемой за пределами Bell Labs версией. Следующая версия, версия 7, выпущенная в 1978 году, стала прототипом большинства современных систем UNIX. Наиболее важные системы, не являющиеся продуктами фирмы AT&T, были разработаны в Калифорнийском университете в Беркли и получили название UNIX BSD; они эксплуатировались на машинах PDP и VAX. Фирма AT&T доработала и усовершенствовала эти системы. В 1982 году компания Bell Labs скомбинировала несколько вариантов системы UNIX фирмы AT&T в единую систему, которая появилась в продаже под названием “UNIX System III”. Впоследствии к этой операционной системе было добавлено несколько новых возможностей, в результате чего появилась система UNIX System V.

## Описание

Классическую архитектуру системы UNIX можно изобразить с помощью трех уровней: аппаратное обеспечение, ядро и пользовательский уровень. Операционную систему часто называют системным ядром или просто ядром, чтобы подчеркнуть ее изолированность от пользователя и приложений. Именно эта часть системы UNIX будет представлять для нас интерес в данной книге. UNIX также снабжается различными пользовательскими сервисами и интерфейсами, которые рассматриваются как часть этой системы. Их можно сгруппировать в оболочку, которая поддерживает системные вызовы от приложений, другое интерфейсное программное обеспечение и компоненты компилятора С (компилятор, ассемблер, загрузчик). Внешний по отношению к этой части системы уровень состоит из приложений пользователя и интерфейса компилятора С.

Рис. 2.15 дает более полное представление о системе. Программы пользователя могут вызывать сервисы операционной системы как непосредственно, так и через библиотечные программы.



Рис. 2.15. Традиционная архитектура UNIX

Интерфейс системных вызовов является границей между пользовательским уровнем и уровнем ядра и позволяет программам более высокого уровня получить доступ к определенным функциям ядра. На другом конце находятся нижние уровни операционной системы, которые содержат примитивные программы, непосредственно взаимодействующие с аппаратным обеспечением. Между этими уровнями находятся системные компоненты; их можно разделить на две основные части, одна из которых относится к управлению процессами, а другая — к вводу-выводу. Подсистема управления процессами отвечает за управление памятью, распределение ресурсов между процессами, планирование и диспетчеризацию, синхронизацию и взаимодействие разных процессов. Файловая система выполняет обмен данными между памятью и внешними устройствами либо в виде потоков символов, либо в виде блоков с использованием различных драйверов устройств. Поблочная передача данных осуществляется с участием дискового кеша — системного буфера в основной памяти, являющегося промежуточным звеном между адресным пространством пользователя и внешним устройством.

В этом подразделе рассказывается о тех системах UNIX, которые можно назвать *традиционными*; в [261] этот термин используется, когда речь идет о System V Release 3 (SVR3), 4.3BSD и более ранних версиях. Ниже приведены общие положения, касающиеся традиционных систем UNIX. Они предназначены для работы на однопроцессорных системах и не обладают достаточной возможностью по защите своих структур данных

от одновременного доступа при работе на нескольких процессорах. Их ядра не слишком разносторонни; они поддерживают один тип файловой системы, стратегию распределения ресурсов между процессами и формат выполнимых файлов. Ядро традиционной системы UNIX не является наращиваемым, в нем мало возможностей повторного использования кода. Все это приводило к тому, что при добавлении в очередных версиях UNIX новых возможностей приходилось в больших количествах писать новый код. В результате ядро оказывалось громоздким и немодульным.

## 2.9. СОВРЕМЕННЫЕ СИСТЕМЫ UNIX

В процессе развития операционной системы UNIX появилось много ее реализаций, и каждая из них обладала своими полезными возможностями. Впоследствии возникла необходимость создать реализацию, в которой были бы унифицированы многие важные нововведения, добавлены возможности других современных операционных систем и которая обладала бы модульной архитектурой. Архитектура типичного современного ядра системы UNIX изображена на рис. 2.16.

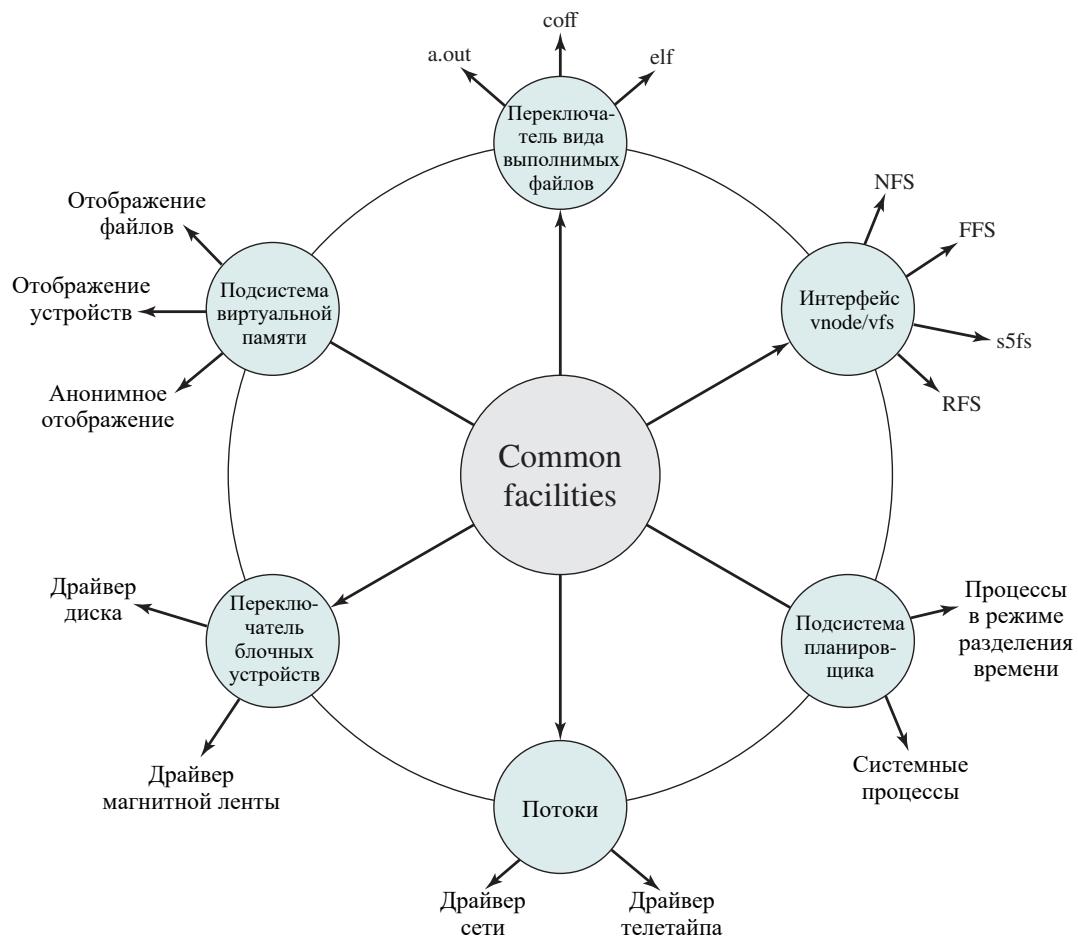


Рис. 2.16. Ядро современной системы UNIX

В этой архитектуре имеется небольшое ядро, которое может работать с различными модулями, предоставляемыми различным процессам операционной системы необходимые функции и сервисы. Каждый внешний круг рисунка соответствует различным функциям и интерфейсу, которые можно реализовать самыми различными способами.

А теперь перейдем к рассмотрению некоторых примеров современных систем UNIX (рис. 2.17).

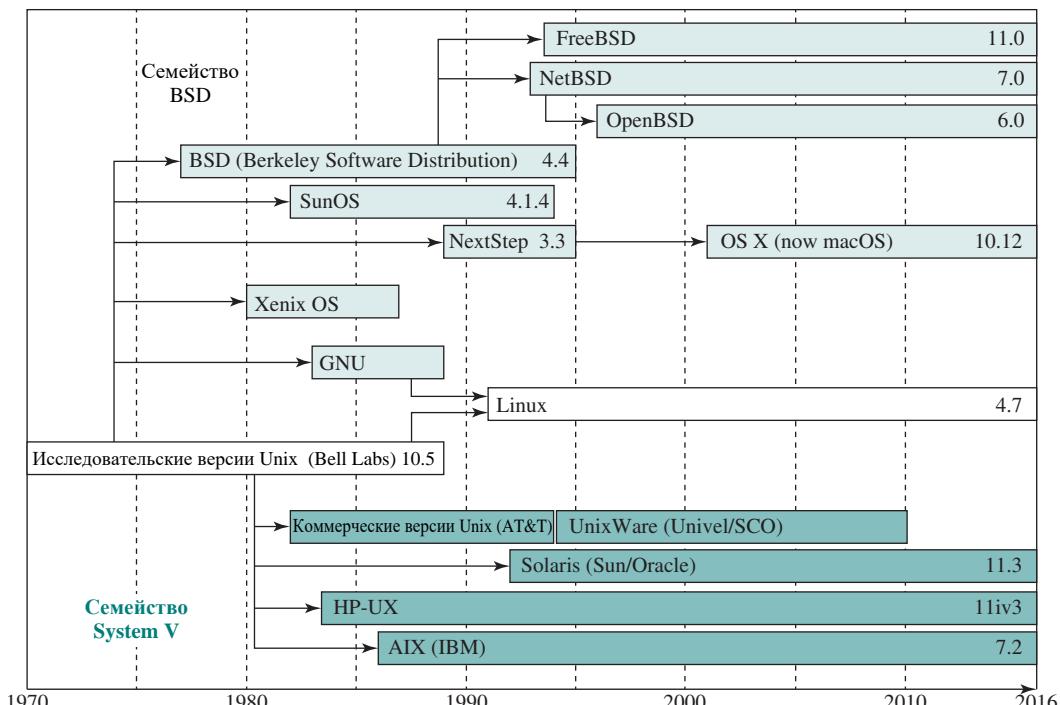


Рис. 2.17. Дерево семейства UNIX

## System V Release 4 (SVR4)

Версия SVR4, разработанная совместно компаниями AT&T и Sun Microsystems, сочетает в себе особенности версий SVR3, 4.3 BSD, Microsoft Xenix System V и SunOS. Ядро System V было почти полностью переписано, в результате чего появилась очищенная от всего лишнего, хотя и сложная, реализация. Среди новых возможностей этой версии следует отметить поддержку обработки данных в реальном времени, наличие классов планирования процессов, динамически распределяемые структуры данных, управление виртуальной памятью, наличие виртуальной файловой системы и ядра с вытеснением.

При создании системы SVR4 объединились усилия как коммерческих, так и академических разработчиков; разработка системы велась, чтобы обеспечить унифицированную платформу для коммерческих реализаций операционной системы UNIX. Эта цель была достигнута, а SVR4 на данный момент, по-видимому, является важнейшей версией UNIX. В ней удачно (с точки зрения конкурентоспособности) сочетаются наиболее важные возможности, реализованные во всех предыдущих системах UNIX. Система SVR4 может работать на компьютерах самых разнообразных типов, начиная с машин, в которых установлены 32-разрядные процессоры, и заканчивая суперкомпьютерами.

## BSD

Важную роль в развитии теории устройства операционных систем сыграла серия версий Berkeley Software Distribution (BSD) системы UNIX, разработанных в Калифорнийском университете. Серия 4.xBSD широко используется в академических организациях; она послужила основой для создания многих коммерческих продуктов UNIX. Можно сказать, что именно благодаря BSD операционная система UNIX приобрела свою популярность, а многие улучшения этой операционной системы впервые появились в версиях BSD.

Последней версией этой серии, выпущенной в Беркли, является система 4.4BSD. Эта версия является основным обновлением версии 4.3BSD, в которую вошли новая система управления виртуальной памятью, ядро с измененной структурой, а также длинный список улучшений других возможностей.

Существует несколько широко распространенных версий BSD с открытым исходным кодом. FreeBSD — популярная операционная система для интернет-серверов и брандмауэров; используется в ряде встраиваемых систем. NetBSD доступна для многих платформ, включая крупные серверные системы, настольные системы и портативные устройства, и часто используется во встраиваемых системах. OpenBSD является операционной системой с открытым исходным кодом, в которой уделяется особое внимание вопросам безопасности.

Последняя версия Macintosh OS, изначально известная как OS X, а ныне именуемая “MacOS”, основана на FreeBSD 5.0 и микроядре Mach 3.0.

## Solaris 11

Система Solaris — это версия операционной системы UNIX, разработанная фирмой Oracle на основе SVR4. На время написания книги последней вышедшей версией Solaris была версия 11. Solaris обладает всеми возможностями системы SVR4, а также некоторыми дополнительными, такими как полная вытесняемость, наличие многопоточного ядра, полнофункциональная поддержка SMP и объектно-ориентированный интерфейс файловых систем. Solaris — одна из наиболее широко применяемых и пользующаяся коммерческим успехом реализаций операционной системы UNIX.

## 2.10. LINUX

### История

Система Linux возникла как вариант операционной системы UNIX, предназначенный для персональных компьютеров с архитектурой IBM PC (Intel 80386). Первоначальная версия была написана Линусом Торвальдсом (Linus Torvalds), финским студентом, изучавшим теорию вычислительных машин. В 1991 году Торвальдс представил в Интернете первую версию системы Linux. С тех пор множество людей, сотрудничая посредством Интернета, развивают Linux под общим руководством ее создателя. Благодаря тому что система Linux является бесплатной и можно беспрепятственно получить ее исходный код, она стала первой альтернативой для рабочих станций UNIX, предлагавшихся фирмами Sun Microsystems и IBM. На сегодняшний день Linux является полнофункциональной системой семейства UNIX, способной работать почти на всех платформах.

Залогом успеха Linux является то, что она бесплатно распространяется при поддержке Фонда бесплатно распространяемых программ (Free Software Foundation — FSF). Целью этой организации является создание надежного аппаратно-независимого программного обеспечения, которое было бы бесплатным, обладало высоким качеством и пользовалось широкой популярностью среди пользователей. Проект GNU<sup>2</sup> фонда предоставляет инструменты для разработки программного обеспечения под эгидой общедоступной лицензии GNU (GNU Public License — GPL). Таким образом, система Linux в таком виде, в котором она существует сегодня, является продуктом, появившимся в результате усилий Торвальдса, а затем и многих других его единомышленников во всем мире, и распространяющимся в рамках проекта GNU.

Linux используется не только многими отдельными программистами; она проникла и в корпоративную среду. В основном это произошло благодаря высокому качеству ядра операционной системы Linux, а не из-за того, что эта система является бесплатной. В эту популярную версию внесли свой вклад многие талантливые программисты, в результате чего появился впечатляющий технический продукт. К достоинствам Linux можно отнести то, что она является модульной и легко настраиваемой. Благодаря этому можно достичь высокой производительности ее работы на самых разнообразных аппаратных платформах. К тому же, получая в свое распоряжение исходный код, производители программного обеспечения могут улучшать качество приложений и служебных программ, с тем чтобы они удовлетворяли конкретным требованиям их пользователей. Имеются также коммерческие компании, такие как Red Hat и Canonical, которые обеспечивают высокопрофессиональную и надежную поддержку своих дистрибутивов Linux. В этой книге подробности внутреннего устройства ядра Linux излагаются на основе ядра Linux 4.7, выпущенного в 2016 году.

Большая часть успеха операционной системы Linux связана с используемой ею моделью развития. Разработчики пользуются единым списком рассылки под названием “LKML” (Linux Kernel Mailing List — список рассылки ядра Linux). Кроме того, имеется множество других списков рассылки, каждый из которых посвящен той или иной подсистеме ядра Linux (список рассылки netdev для сети, linux-pci — для подсистемы PCI, linux-acpi — для подсистемы ACPI и др.). Обновления, отправляемые в эти списки рассылки, должны соответствовать строгим правилам (главным образом — соглашениям о кодировании ядра Linux) и изучаются разработчиками со всего мира, подписанными на эти списки рассылки. Любой пользователь может отправлять свои обновления в эти списки рассылки. Статистика (например, время от времени публикуемая на сайте lwn.net) показывает, что многие обновления предлагаются разработчиками из известных коммерческих компаний, таких как Intel, Red Hat, Google, Samsung и др. Кроме того, многие разработчики являются сотрудниками коммерческих компаний (как Дэвид Миллер (David Miller), поддерживающий сетевые функции и работающий в компании Red Hat). Такие обновления изучаются и обсуждаются в списке рассылки, после чего в них вносятся исправления, и цикл обсуждения начинается заново. В конце концов принимается решение, следует ли принять или отклонить эти исправления. Каждый руководитель подсистемы время от времени отправляет запрос о размещении исправлений его части в основном ядре, который обрабатывается Линусом Торвальдсом. Сам Линус

<sup>2</sup> GNU — рекурсивная аббревиатура для GNU's Not Unix (GNU — не Unix). Проект GNU представляет собой бесплатный набор программных пакетов и инструментов для разработки UNIX-подобной операционной системы.

выпускает новую версию ядра примерно каждые 7–10 недель, причем каждый такой выпуск имеет около 5–8 предварительных версий-кандидатов.

Интересно попытаться понять, почему другие операционные системы с открытым кодом, такие как различные версии BSD или OpenSolaris, не имеют таких успеха и популярности, которыми обладает Linux. Тому может быть много причин; конечно, открытость модели развития Linux способствовала популярности и успеху этой операционной системы. Но эта тема выходит за рамки нашей книги.

## Модульная структура

Ядра большинства версий операционной системы UNIX являются монолитными. Напомним, что монолитное ядро — это ядро, которое включает в себя почти все возможности операционной системы в виде одного большого блока кода, который запускается как единый процесс в едином адресном пространстве. Все функциональные компоненты такого ядра имеют доступ ко всем его внутренним структурам данных и ко всем программам. При внесении изменений в любую из частей типичной монолитной операционной системы все ее модули и подпрограммы необходимо повторно компоновать и переустанавливать, а перед тем как изменения вступят в силу, систему нужно будет перезагрузить. В результате усложняется внесение любых модификаций, таких как добавление драйвера нового устройства или новых функций файловой системы. Особенно остро эта проблема встает в Linux, глобальную разработку которой выполняют объединенные на добровольных началах группы независимых программистов.

Хотя Linux не использует подход микроядра, она достигает многих потенциальных преимуществ такого подхода посредством своей модульной архитектуры. Операционная система Linux организована в виде набора модулей, ряд которых может автоматически загружаться и выгружаться по требованию. Эти относительно независимые блоки называются **загружаемыми модулями** (*loadable modules*) [92]. По сути, модуль представляет собой объектный файл, код которого может быть связан и отключен от ядра во время выполнения. Как правило, модуль реализует некоторые специальные функции, например файловую систему, драйвер устройства или некоторые другие функции верхнего уровня ядра. Модуль не должен выполняться как собственный процесс или поток, хотя при необходимости может создавать потоки ядра для различных целей. Вместо этого модуль выполняется в режиме ядра от имени текущего процесса.

Таким образом, хотя Linux может считаться монолитной операционной системой, ее модульная структура позволяет преодолевать многие трудности разработки и развития ядра. Загружаемые модули Linux имеют две важные характеристики.

- Динамическое подключение.** Модуль ядра может быть загружен и связан с ядром тогда, когда ядро уже находится в памяти и выполняется. Модуль также в любое время может быть отсоединен от ядра и удален из памяти.
- Стековая организация.** Модули организованы в виде определенной иерархической структуры. Отдельные модули могут выполнять роль библиотек при обращении к ним модулей более высоких уровней в рамках этой структуры; они сами также могут обращаться к модулям на более низких уровнях.

Динамическое подключение [83] облегчает настройку системы и экономит память, которую занимает ядро. В операционной системе Linux программа пользователя или сам пользователь может явно загружать или выгружать модули с помощью команд `insmod`,

`modprobe` и `rmmod`. Само ядро управляет работой отдельных функций и по мере необходимости загружает нужные модули или выгружает те, нужда в которых уже отпала. Кроме того, стековая организация позволяет задавать зависимости модулей, что дает два основных преимущества.

1. Код, являющийся общим для набора однотипных модулей (например, драйверы похожих аппаратных устройств), можно поместить в один модуль, что позволяет сократить количество дублирований.
2. Ядро может проверить наличие в памяти нужных модулей, воздерживаясь от выгрузки модуля, который нужен для работы других, зависимых от него, и загружая вместе с новым требуемым модулем все необходимые дополнительные модули.

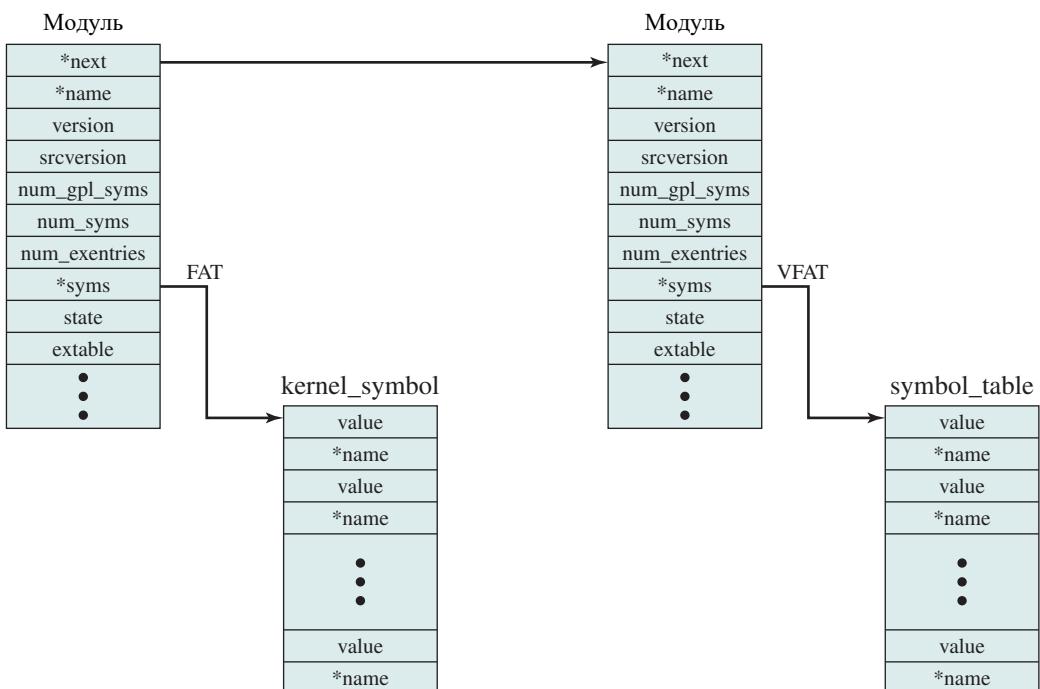


Рис. 2.18. Пример списка модулей ядра операционной системы Linux

На рис. 2.18 показан пример, иллюстрирующий структуры, которые используются операционной системой Linux для управления модулями. На рисунке приведен список модулей ядра после загрузки модулей FAT и VFAT. Каждый модуль задается двумя таблицами: таблицей модулей и таблицей символов (`kernel_symbol`). В таблицу модулей входят перечисленные ниже элементы.

- **\*name:** имя модуля.
- **refcnt:** счетчик модуля. Этот счетчик увеличивается, когда начинается операция, использующая функции модуля, и уменьшается по ее завершении.
- **num\_syms:** количество экспортруемых символов.
- **\*syms:** указатель на таблицу символов данного модуля.

В таблице символов перечисляются символы, определяемые данным модулем и используемые где-то в ином месте.

## Компоненты ядра

На рис. 2.19, взятом из [176], показаны основные компоненты типичной реализации ядра Linux. На рисунке показаны несколько процессов, запущенных поверх ядра. Каждая рамка указывает отдельный процесс, в то время как каждая волнистая линия со стрелкой в рамке представляет поток выполнения. Само ядро состоит из набора взаимодействующих компонентов со стрелками, указывающими основные взаимодействия этих компонентов. Базовые аппаратные средства также изображены как набор компонентов со стрелками, указывающими, какие компоненты ядра используют или контролируют те или иные аппаратные компоненты. Понятно, что все компоненты ядра выполняются процессором. Для простоты эти отношения не показаны.

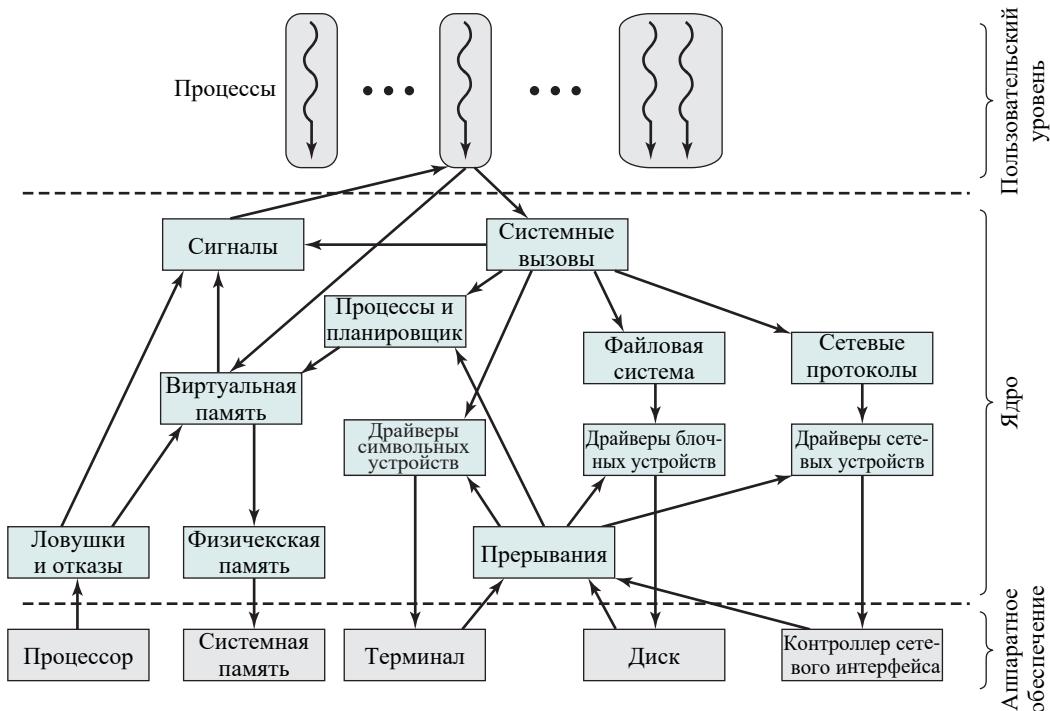


Рис. 2.19. Компоненты ядра Linux

Вкратце основными компонентами ядра являются следующие.

- **Сигналы.** Ядро использует сигналы для обращения к процессу. Например, сигналы используются для уведомления процесса о некоторых ошибках, таких как деление на нуль. В табл. 2.6 показано несколько примеров сигналов.
- **Системные вызовы.** Системный вызов является средством, с помощью которого процесс запрашивает определенную службу ядра. Существует несколько сотен системных вызовов, которые можно грубо сгруппировать в такие категории: файловая система, процессы, планирование, межпроцессное взаимодействие, сокеты (сети) и пр. В табл. 2.7 показано несколько примеров в каждой категории.

**Таблица 2.6. Некоторые сигналы Linux**


---

SIGHUP	Закрытие терминала
SIGQUIT	Сигнал выхода от терминала
SIGTRAP	Ловушка трассировки
SIGBUS	Неверное обращение к физической памяти
SIGKILL	Безусловное завершение
SIGSEGV	Нарушение обращения к памяти
SIGPIPT	Запись в разорванное соединение
SIGTERM	Сигнал завершения
SIGCHLD	Дочерний процесс завершен или остановлен
SIGCONT	Продолжить выполнение ранее остановленного процесса
SIGTSTP	Сигнал остановки от терминала
SIGTTOU	Попытка записи в терминал фоновым процессом
SIGXCPU	Процесс превысил лимит процессорного времени
SIGVTALRM	Истечение “виртуального таймера”
SIGWINCH	Изменение размера окна управляющего терминала
SIGPWR	Сбой электропитания
SIGRTMIN	Первый сигнал реального времени
SIGRTMAX	Последний сигнал реального времени

---

**Таблица 2.7. Некоторые системные вызовы Linux**


---

Файловая система	
close	Закрытие файлового дескриптора
link	Создание нового имени для файла
open	Открытие (с возможным созданием) файла или устройства
read	Чтение из файлового дескриптора
write	Запись в файловый дескриптор
	Процессы
execve	Выполнение программы
exit	Завершение вызванного процесса
getpid	Получение идентификатора процесса
setuid	Установка идентификатора пользователя текущего процесса
ptrace	Предоставляет родительскому процессу средства наблюдения и управления выполнением другого процесса для изучения и изменения его образа в памяти и регистров

---

<b>Планирование</b>	
<code>sched_getparam</code>	Установка параметров планирования для стратегии планирования процесса с определенным идентификатором
<code>sched_get_priority_max</code>	Возвращает значение максимального приоритета, которое может быть использовано алгоритмом планирования, определяемым стратегией
<code>sched_setscheduler</code>	Устанавливает как стратегию планирования (например, FIFO), так и связанные с ней параметры для данного идентификатора процесса
<code>sched_rr_get_interval</code>	Выполняет запись в структуру <code>timespec</code> , на которую указывает параметр, значения кванта времени для данного идентификатора процесса
<code>sched_yield</code>	Позволяет процессу отказаться от получения процессорного времени без блокировки. Этот процесс перемещается в конец очереди для соответствующего статического приоритета, и выполнение переходит к другому процессу
<b>Межпроцессное взаимодействие</b>	
<code>msgrecv</code>	Выделение структуры буфера сообщения для получения сообщения. Системный вызов читает сообщение из очереди, определенной <code>msqid</code> , в созданный буфер сообщения
<code>semctl</code>	Выполнение управляющей операции, определенной <code>cmd</code> , над семафором <code>semid</code>
<code>semop</code>	Выполнение операции над выбранными членами набора семафоров <code>semid</code>
<code>shmat</code>	Подключение сегмента разделяемой памяти, идентифицируемого с помощью <code>semid</code> , к сегменту данных вызывающего процесса
<code>shmctl</code>	Разрешить пользователю получать информацию о сегментах общей памяти; задать владельца, группу и разрешения для сегментов разделяемой памяти; уничтожить сегмент
<b>Сеть</b>	
<code>bind</code>	Назначение локального IP-адреса и порта сокету. Возвращает 0 при успехе, -1 при ошибке
<code>connect</code>	Устанавливает соединение между данным сокетом и удаленным сокетом, связанным с <code>sockaddr</code>
<code>gethostname</code>	Возвращает имя локального хоста
<code>send</code>	Отсылает байты, содержащиеся в буфере, через определенный сокет
<code>setsockopt</code>	Настраивает работу сокета
	Разное
<code>fsync</code>	Копирует все находящиеся в основной памяти части файла на диск и ожидает, пока устройство сообщит, что все части находятся в постоянном хранилище
<code>time</code>	Возвращает количество секунд, прошедших с 1 января 1970 года
<code>vhangup</code>	Имитирует "зависание" на текущем терминале. Позволяет другим пользователям получить "чистый" терминал во время входа

- **Процессы и планировщик.** Создает, управляет и планирует процессы.
- **Виртуальная память.** Выделяет виртуальную память для процессов и управляет ею.
- **Файловая система.** Предоставляет глобальное иерархическое пространство имен для файлов, каталогов и других объектов, связанных с файлами и функциями файловой системы.
- **Сетевые протоколы.** Поддержка пользовательского интерфейса сокетов для набора протоколов TCP/IP.
- **Драйверы символьных устройств.** Управление устройствами, которые требуют от ядра отправки или получения данных по одному байту, например терминалами, принтерами или модемами.
- **Драйверы блочных устройств.** Управление устройствами, которые читают и записывают данные блоками, как, например, различные виды вторичной памяти (магнитные диски, CD-ROM и т.п.).
- **Драйверы сетевых устройств.** Управление картами сетевых интерфейсов и коммуникационными портами, которые подключаются к сетевым устройствам, таким как мосты или роутеры.
- **Ловушки и отказы.** Обработка генерируемых процессором прерываний, как, например, при сбое памяти.
- **Физическая память.** Управляет пулом кадров страниц и выделяет страницы для виртуальной памяти.
- **Прерывания.** Обработка прерываний от периферийных устройств.

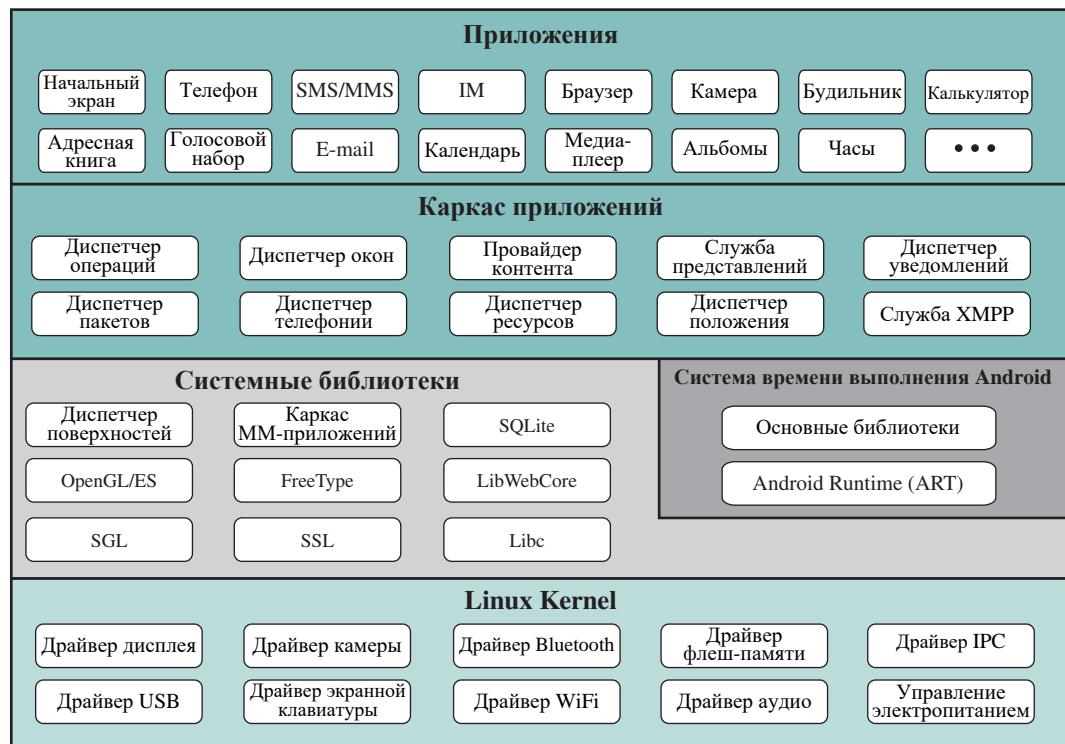
## 2.11. ANDROID

Операционная система Android основана на операционной системе Linux и первоначально предназначалась для мобильных телефонов. Это наиболее популярная (с большим отрывом) операционная система для мобильных устройств: телефоны с Android обгоняют iPhone фирмы Apple примерно как 4 к 1 [175]. Но это только один из элементов растущего влияния Android. Эта операционная система все чаще оказывается установленной на практически любом устройстве с компьютерным чипом, за исключением серверов и ПК. Android — широко используемая операционная система для Интернета вещей.

Первоначальная разработка операционной системы Android велась фирмой Android, Inc., которая была куплена компанией Google в 2005 году. Первая коммерческая версия Android 1.0 была выпущена в 2008 году. На момент выпуска русскоязычного издания данной книги последней версией является версия Android 10 (Pixel). Android имеет активное сообщество разработчиков и энтузиастов, которые используют исходные тексты проекта Android Open Source Project (AOSP) для разработки и распространения собственных измененных версий этой операционной системы. Открытый исходный код Android был ключом к ее успеху.

## Программная архитектура Android

Android представляет собой стек программ, который включает модифицированную версию ядра Linux, промежуточное программное обеспечение и ключевые приложения. На рис. 2.20 архитектура программного обеспечения Android показана немного более детально. Таким образом, Android следует рассматривать как полный программный стек, а не только как операционную систему.



Реализация:

Приложения, каркас приложений: Java

Системные библиотеки, система времени выполнения Android: C, C++

Ядро Linux: C

**Рис. 2.20.** Программная архитектура Android

### Приложения

Все приложения, с которыми непосредственно взаимодействует пользователь, являются частью уровня приложений. Сюда входит базовый набор приложений общего назначения, таких как клиент электронной почты, SMS-программа, календарь, геолокационные карты, браузер, книга контактов и другие приложения, стандартные для любого мобильного устройства. Приложения обычно реализуются на языке Java. Ключевая цель

архитектуры Android с открытым исходным кодом — облегчить для разработчиков реализацию новых приложений для конкретных устройств и для требований конкретных конечных пользователей. Java позволяет разработчикам освободиться от привязки к аппаратному обеспечению, а также задействовать возможности языка Java более высокого уровня, такие как предопределенные классы. На рис. 2.20 показаны примеры базовых приложений для платформы Android.

## Каркас приложений

Уровень каркаса приложений (Application Framework) предоставляет высокоуровневые “строительные блоки” посредством стандартизованных API, которые программист может использовать для создания новых приложений. Архитектура разработана таким образом, чтобы максимально упростить повторное использование компонентов. Некоторые ключевые компоненты каркаса приложений перечислены ниже.

- **Диспетчер активности.** Управляет жизненным циклом приложений. Отвечает за запуск, приостановку и возобновление выполнения различных приложений.
- **Диспетчер окон.** Абстрагирование языком Java базового диспетчера поверхности, который обрабатывает взаимодействие буферов кадров и низкоуровневый вывод изображений, тогда как диспетчер окон обеспечивает более высокий уровень, дающий приложениям возможность объявлять свои клиентские области и использовать различные функциональные возможности, такие, например, как строка состояния.
- **Диспетчер пакетов.** Устанавливает и удаляет приложения.
- **Диспетчер телефонии.** Обеспечивает взаимодействие с телефоном, службами SMS и MMS.
- **Провайдер контента.** Эти функции инкапсулируют данные приложений, которые должны распределяться между приложениями (например, такие как информация о контактах).
- **Диспетчер ресурсов.** Управляет ресурсами приложений, такими как локализованные строки или рисунки.
- **Служба представлений.** Предоставляет примитивы графического интерфейса пользователя, такие как кнопки, списки, выбор даты и другие элементы управления, а также события пользовательского интерфейса (например, касания и жесты).
- **Диспетчер положения.** Позволяет разработчикам использовать местоположение клиента в своих программах, в том числе GPS, идентификаторы базовых станций или локальные базы WiFi.
- **Диспетчер уведомлений.** Управляет событиями, такими как поступающие сообщения и назначения.
- **XMPP.** Предоставляет стандартизованную систему функций для сообщений между приложениями.

## Системные библиотеки

Уровень ниже Application Framework состоит из двух частей: системных библиотек и системы времени выполнения Android. Системная библиотека представляет собой набор

полезных системных функций, написанных на C или C++ и используется различными компонентами Android. Они вызываются из приложений через интерфейс Java. Эти функциональные возможности предоставляются разработчикам Android Application Framework. Ключевые системные библиотеки включают следующее.

- **Диспетчер поверхности.** Android использует составной оконный менеджер, похожий на Vista или Compiz, но гораздо проще. Вместо рисования непосредственно в экранном буфере ваши команды рисования создают закадровый растровый рисунок, который затем объединяется с другими растровыми изображениями и формирует видимое пользователю содержимое экрана. Это позволяет системе создавать различные интересные эффекты, такие как полупрозрачные окна и необычные переходы.
- **OpenGL.** OpenGL (Open Graphics Library) представляет собой многоязычный, многоплатформенный API для создания двух- и трехмерной компьютерной графики. OpenGL/ES (OpenGL для встраиваемых систем) является подмножеством OpenGL, разработанным для встраиваемых систем.
- **Каркас мультимедийных приложений.** Каркас мультимедийных приложений поддерживает запись и проигрывание видео во многих форматах, включая AAC, AVC (H.264), H.263, MP3 и MPEG-4.
- **База данных SQL.** Android включает облегченную СУБД SQLite для хранения постоянных данных. SQLite обсуждается в следующем разделе.
- **Механизм браузера.** Для быстрого отображения HTML-содержимого Android использует библиотеку WebKit, которая, по сути, представляет собой ту же библиотеку, что и используемая в Safari и iPhone. Она также использовалась браузером Google Chrome, пока Google не переключился на Blink.
- **Bionic LibC.** Это урезанная версия стандартной системной библиотеки C, настроенная для встраиваемых устройств на базе Linux. Интерфейс является стандартным интерфейсом Java (Java Native Interface — JNI).

## Ядро Linux

Ядро операционной системы Android подобно, но не идентично, ядру стандартного дистрибутива Linux. Следует отметить одно важное изменение — в ядре Android отсутствуют драйверы, не требующиеся в мобильных средах, что делает ядро меньшим по размеру. Кроме того, Android расширяет ядро Linux функциями, которые предназначены для работы в мобильной среде и в общем случае бесполезны на настольном компьютере или ноутбуке.

Android основан на ядре Linux в случае таких фундаментальных системных служб, как безопасность, управление памятью, управление процессами, сетевой стек и модель драйверов. Ядро также действует в качестве уровня абстракции между оборудованием и остальной частью стека программ и позволяет Android использовать широкий спектр драйверов, которые поддерживает Linux.

## Система времени выполнения Android

Большинство операционных систем, используемых на мобильных устройствах, такие как iOS и Windows, используют программное обеспечение, которое компилируется непосредственно в двоичный код для конкретной аппаратной платформы. В отличие от

этого подхода большая часть программного обеспечения Android отображается в байткод, который затем превращается в машинные команды уже на самом устройстве. Более ранние версии Android используют схему, известную как Dalvik. Однако Dalvik имеет ряд ограничений с точки зрения масштабирования для систем с большей памятью и многоядерных архитектур, поэтому последние версии Android основаны на схеме, известной как система времени выполнения Android (Android runtime — ART). ART полностью совместим с существующим форматом байткода Dalvik, именуемым “dex” (Dalvik Executable), так что разработчикам приложений не нужно изменять их схему кодирования для работы под управлением ART. Сначала мы рассмотрим Dalvik, а затем — ART.

### **Виртуальная машина Dalvik**

Виртуальная машина Dalvik (Dalvik VM — DVM) выполняет файлы в dex-формате, который оптимизирован для эффективного хранения и выполнения. Виртуальная машина может запускать на выполнение классы, скомпилированные компилятором Java, которые были преобразованы в формат команд данной платформы с помощью прилагаемого инструмента “dx”. Виртуальная машина работает поверх ядра Linux, которое она использует для получения базовой функциональности (наподобие работы с потоками или управления памятью). Базовая библиотека классов Dalvik предназначена для представления знакомой среды разработки для тех, кто работает с Java Standard Edition, но ориентированной на нужды небольших мобильных устройств.

Каждое приложение Android работает как отдельный процесс, с собственным экземпляром виртуальной машины Dalvik. Dalvik разработана таким образом, чтобы устройство могло эффективно запускать несколько виртуальных машин.

### **Формат Dex**

DVM запускает приложения и код, написанный на Java. Стандартный компилятор Java превращает исходный код (представленный в виде текстовых файлов) в байткод. Байткод, в свою очередь, компилируется в файл .dex, который DVM может читать и использовать. По сути, файлы классов преобразуются в файлы .dex (подобны файлу .jar при использовании стандартной виртуальной машины Java), которые затем читаются и выполняются DVM. Повторяющиеся данные, используемые в файлах классов, включены в файл .dex только один раз, что экономит пространство и приводит к меньшим накладным расходам. Выполнимые файлы могут быть изменены и оптимизированы при инсталляции приложения для мобильных устройств.

### **Концепции Android Runtime**

ART представляет собой текущую среду выполнения приложений, используемую Android и введенную начиная с версии Android 4.4 (KitKat). Первоначально Android был разработан для устройств с одним ядром (и с минимальной аппаратной поддержкой многопоточности) и малым количеством памяти, для которых Dalvik представлялся подходящей средой выполнения приложений. Однако в последнее время устройства, использующие Android, имеют многоядерные процессоры и большее количество памяти (при относительно низкой стоимости), что заставило Google переосмыслить проект среды выполнения, чтобы предоставить разработчикам и пользователям более богатые возможности по использованию имеющегося оборудования.

И для Dalvik, и для ART все Android-приложения, написанные на Java, компилируются в байткод dex. Dalvik использует dex-формат байткода для переносимости, так

что этот код должен быть преобразован (скомпилирован) в машинный код, который будет работать на процессоре. Среда выполнения Dalvik выполняет такое преобразование dex-байткода в машинный код при выполнении приложения, и этот процесс был назван оперативной компиляцией (just-in-time — JIT). Поскольку JIT компилирует только часть кода, он имеет меньший объем и использует меньше физического пространства на устройстве. (Полностью хранятся только dex-файлы — в отличие от фактического машинного кода.) Dalvik идентифицирует часто выполняемые части кода и кеширует их, так что при последующих вызовах эти части выполняются быстрее. Страницы физической памяти, которые хранят кешированный код, не заменяются и не выгружаются, что несколько нагружает систему. Но даже с этими оптимизациями Dalvik приходится выполнять JIT-компиляцию всякий раз, когда приложение запускается, что требует значительного количества ресурсов процессора. Обратите внимание, что процессор используется не только для фактического запуска приложения, но и для преобразования байткода dex в машинный код, что также не способствует эффективности. Такое использование процессора являлось также причиной плохой работы графического интерфейса пользователя при запуске некоторых “тяжелых” приложений.

Для преодоления некоторых из этих проблем и более эффективного использования имеющегося оборудования Android представил систему ART. ART также выполняет байткод dex, но вместо компиляции байткода во время выполнения ART компилирует байткод в машинный во время установки приложения. Этот прием называется досрочной (ahead-of-time — АОТ) компиляцией. ART использует инструмент dex2oat для компиляции во время установки приложения. Результат работы этого инструмента представляет собой файл, который выполняется при запуске приложения.

На рис. 2.21 показан жизненный цикл APK — пакета приложения, который поступает к пользователю от разработчика. Цикл начинается с исходного кода, который компилируется в формат dex и вместе с кодом поддержки и ресурсами формирует APK. На устройстве пользователя полученный APK распаковывается. Ресурсы и машинный код обычно устанавливаются непосредственно в каталог приложения. Однако dex-код требует дополнительной обработки, как в случае Dalvik, так и в случае ART. В Dalvik к dex-файлу применяется инструмент под названием dexopt, который создает оптимизированную версию dex-кода (odex), иногда называемую ускоренным dex. Цель заключается в том, чтобы обеспечить более быстрое выполнение dex-кода интерпретатором. В ART инструмент dex2oat выполняет такую же оптимизацию, как и dexopt. Кроме того, он компилирует dex-код и создает машинный код целевого устройства. Результатом работы dex2oat является выполнимый файл в формате ELF, который запускается непосредственно, без интерпретатора.

## Преимущества и недостатки

Преимущества использования ART включают следующее.

- Снижение времени запуска приложений из-за применения машинного кода.
- Удлинение срока жизни аккумуляторов из-за ненужности JIT-компиляции.
- Уменьшение количества памяти, необходимой для запуска приложения (так как не требуется память для хранения кеша JIT). Кроме того, отсутствие невыгружаемого кеша кода JIT обеспечивает большую гибкость использования оперативной памяти при ее небольшом количестве.
- В ART имеется ряд оптимизаций сборки мусора и отладки.

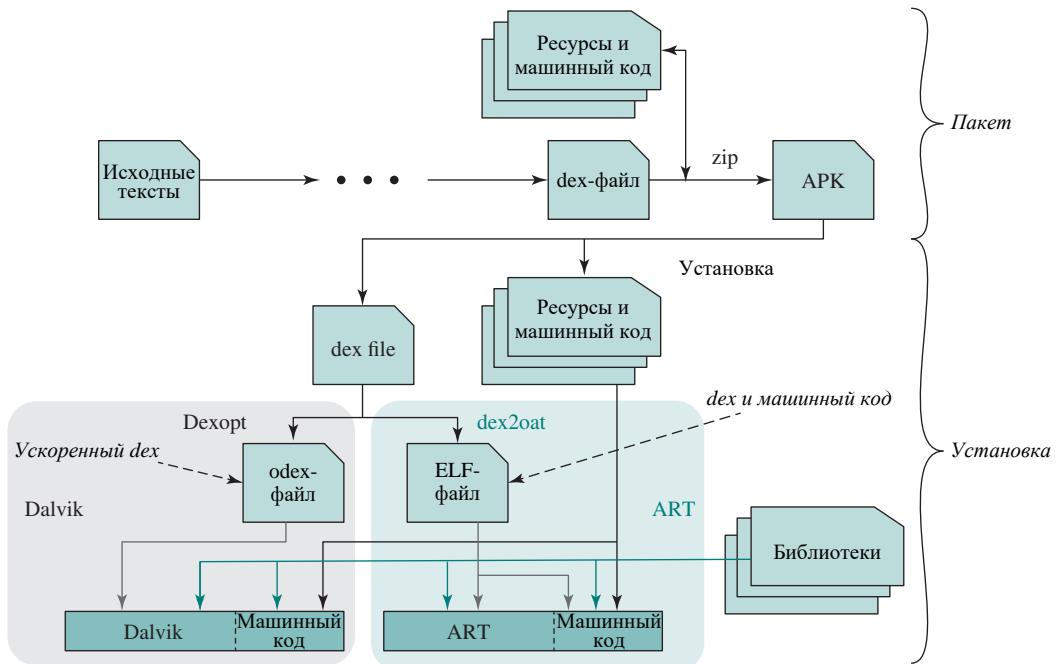


Рис. 2.21. Жизненный цикл APK

У ART имеются и потенциальные недостатки.

- Поскольку преобразование байткода в машинный код выполняется во время установки, установка приложения отнимает больше времени. Для разработчиков Android, которые вынуждены многократно загружать приложение во время тестирования, это время может стать заметным.
- При первой загрузке свежего приложения или при первой загрузке после сброса до фабричного состояния все приложения, установленные на устройстве, компилируются в машинный код с помощью инструмента `dex2oat`. Таким образом, при первой загрузке достижение начального экрана по сравнению с первой загрузкой Dalvik может потребовать значительно больше времени.
- Сгенерированный таким образом машинный код хранится во внутренней памяти, что может потребовать значительного ее количества.

## Системная архитектура Android

Полезно проиллюстрировать Android с точки зрения разработчика приложений, как показано на рис. 2.22. Эта системная архитектура является упрощенной абстракцией архитектуры программного обеспечения, показанной на рис. 2.20. Рассматриваемая с этой точки зрения, операционная система Android состоит из следующих уровней.

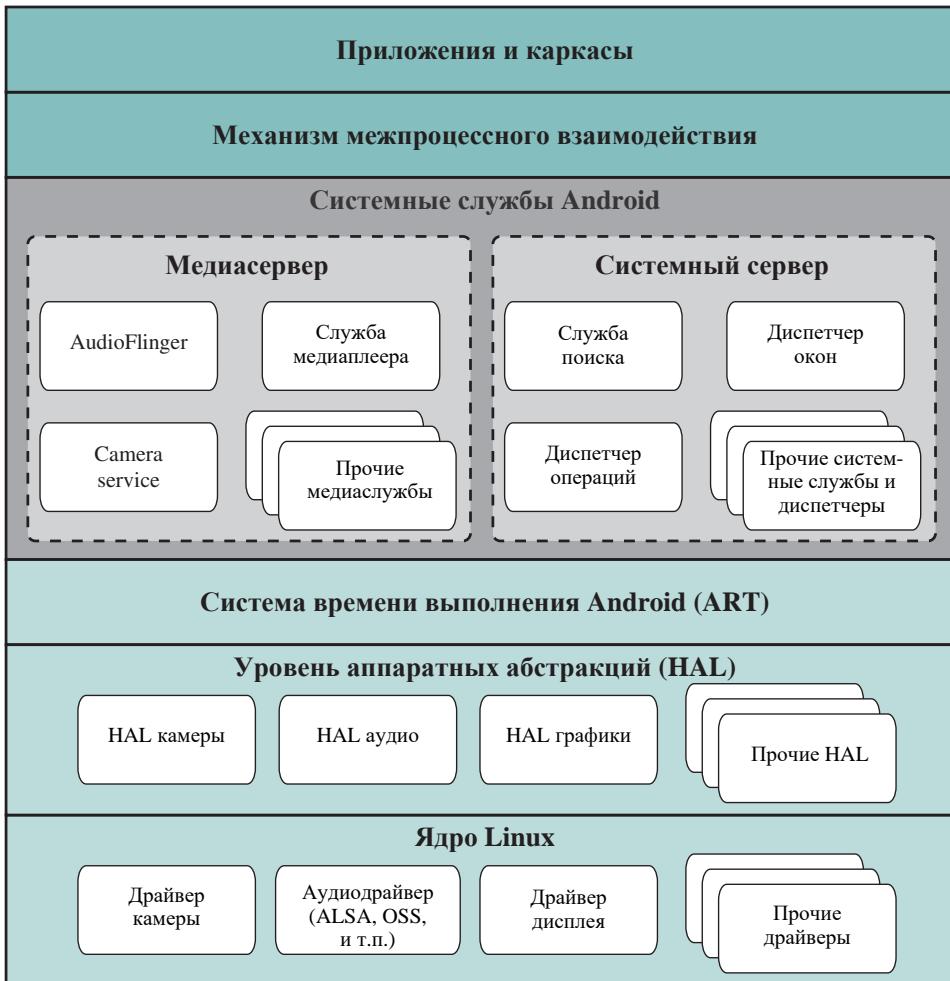


Рис. 2.22. Системная архитектура Android

- **Приложения и каркасы.** Разработчики приложений прежде всего связаны с этим уровнем и API, которые обеспечивают доступ к службам нижних уровней.
- **Механизм межпроцессного взаимодействия.** Этот механизм позволяет пересекать границы процессов и вызывать код системных служб Android. В основном это позволяет API высокого уровня взаимодействовать с системными службами Android.
- **Системные службы Android.** Большинство функциональных возможностей, предоставляемых через API каркаса приложений, вызывают системные службы, которые, в свою очередь, обращаются к аппаратному обеспечению и функциям ядра. Службы могут рассматриваться как организованные в виде двух групп: медиаслужбы, работающие с записью и воспроизведением медиа, и системные службы, работающие с функциональными возможностями системного уровня, такими как электропитание, работа с местоположением или уведомления.

- **Уровень аппаратных абстракций.** Уровень аппаратных абстракций (Hardware Abstraction Layer — HAL) обеспечивает стандартный интерфейс к драйверам устройств уровня ядра, так что коду верхнего уровня не нужно ничего знать о деталях реализации конкретных драйверов и аппаратного обеспечения. HAL практически не отличается от используемого в стандартных дистрибутивах Linux. Этот уровень используется для абстрагирования возможностей конкретных устройств (которые поддерживаются оборудованием и предоставляются ядром) из пользовательского пространства. Пространство пользователя может быть службами или приложениями Android. HAL предназначен для того, чтобы поддерживать пространство пользователя согласованным по отношению к различным устройствам. Кроме того, производители оборудования могут делать собственные усовершенствования и добавлять их в уровень HAL, не затрагивая пространство пользователя. Примером этого является HwC (Hardware Composer), который является реализацией HAL от производителя, способной работать с возможностями визуализации аппаратного оборудования. Диспетчер поверхности в состоянии одинаково работать с различными реализациями HwC от разных производителей.
- **Ядро Linux.** Ядро Linux настроено для удовлетворения специфических потребностей мобильной среды.

## Операции

Операция (*activity*<sup>3</sup>) является единственным компонентом визуального пользовательского интерфейса, включая такие объекты, как меню, пиктограммы, переключатели и т.п. Каждый экран приложения является расширением класса операции. Операция использует представления (*view*) для формирования графических пользовательских интерфейсов, которые отображают информацию и реагируют на действия пользователя. Мы будем обсуждать операции в главе 4, “Потоки”.

## Управление электропитанием

Android добавляет в ядро Linux две функции для повышения способности осуществлять управление электропитанием: будильники (*alarm*) и блокировки сна (*wakelock*).

Будильники реализованы в ядре Linux и являются видимыми для разработчика приложений через диспетчер *AlarmManager* в ядре библиотеки времени выполнения. С помощью *AlarmManager* приложение может запросить службу пробуждения в определенное время. Будильники реализованы в ядре так, что могут срабатывать, даже если система находится в режиме сна. Это позволяет системе переходить в спящий режим, экономя электроэнергию, даже при наличии процесса, который требует, чтобы система была активна.

Блокировки сна предотвращают переход системы Android в режим сна. Приложения могут использовать одну из перечисленных далее блокировок.

- **Full\_Wake\_Lock.** Процессор включен, экран и клавиатура подсвечены.
- **Partial\_Wake\_Lock.** Процессор включен, экран и клавиатура отключены.

---

<sup>3</sup> Перевод этого термина в русскоязычной литературе окончательно не устоялся; используются такие термины, как “активность”, “деятельность”, “действие”. — Примеч. пер.

- **Screen\_Dim\_Wake\_Lock.** Процессор включен, экран приглушен, клавиатура яркая.
- **Screen\_Bright\_Wake\_Lock.** Процессор включен, экран подсвечен, клавиатура отключена.

Всякий раз, когда приложению требуется, чтобы одно из управляемых периферийных устройств оставалось включенным, через API запрашивается соответствующая блокировка. Если блокировок для устройства нет, оно выключается для экономии батареи.

Эти объекты ядра сделаны видимыми для приложений в пользовательском пространстве посредством файлов `/sys/power/wavelock`. Файлы `wake_lock` и `wake_unlock` могут использоваться для включения и выключения блокировок путем записи в соответствующий файл.

## 2.12. КЛЮЧЕВЫЕ ТЕРМИНЫ, КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАЧИ

### Ключевые термины

Виртуальная машина	Монолитное ядро	Реальный адрес
Виртуальная память	Надежность	Режим ядра
Виртуальный адрес	Объектно-ориентированное проектирование	Резидентный монитор
Время безотказной работы	Операционная система	Сбой
Время простостоя	Пакетная обработка	Симметричная многопроцессорность
Загружаемые модули	Пакетная система	Система с разделением времени
Задание	Планирование	Состояние
Задача	Пользовательский режим	Состояние процесса
Квант времени	Последовательная обработка	Среднее время восстановления
Контекст выполнения	Поток	Среднее время работы на отказ
Микроядро	Прерывание	Управление памятью
Многозадачность	Привилегированная команда	Физический адрес
Многопоточность	Процесс	Циклическое планирование
Многозадачная пакетная система	Разделение времени	Ядро
Монитор	Распределенная операционная система	Язык управления заданиями

### Контрольные вопросы

- 2.1. Каковы цели проектирования операционных систем?
- 2.2. Что такое ядро операционной системы?
- 2.3. Что такое многозадачность?
- 2.4. Что такое процесс?
- 2.5. Как операционная система использует контекст выполнения процесса?

- 2.6. Перечислите и кратко поясните функции операционной системы по управлению памятью.
- 2.7. Поясните отличие реальных адресов от виртуальных.
- 2.8. Опишите методику циклического планирования.
- 2.9. Поясните различие между монолитным ядром и микроядром.
- 2.10. Что такое многопоточность?
- 2.11. Перечислите ключевые проблемы проектирования операционных систем с симметричной многопроцессорностью.

## Задачи

2.1. Предположим, у нас есть многозадачный компьютер, в котором каждое задание имеет идентичные характеристики. В течение цикла вычисления одного задания  $T$  половину времени занимает ввод-вывод, а вторую половину — работа процессора. Для выполнения каждого задания требуется  $N$  циклов. Допустим, что для планирования используется простой алгоритм циклического обслуживания и что ввод-вывод может выполняться одновременно с работой процессора. Определите значения следующих величин.

- Реальное время, затрачиваемое на выполнение задания.
- Среднее количество заданий, которые выполняются в течение одного цикла  $T$ .
- Доля времени, в течение которого процессор активен (не находится в режиме ожидания).

Вычислите эти значения для одного, двух и четырех одновременно выполняющихся заданий, считая, что время цикла  $T$  распределяется одним из следующих способов.

- a. В течение первой половины периода выполняется ввод-вывод, а в течение второй — работа процессора.
  - b. В течение первой и четвертой четвертей выполняется ввод-вывод, а в течение второй и третьей — работа процессора.
- 2.2. Перегруженной операциями ввода-вывода называется такая программа, которая, будучи запущенной сама по себе, тратит больше времени на ввод-вывод, чем на работу процессора. Программой, преимущественно использующей процессор, называется программа, в которой соотношение затрат времени изменяется в пользу процессора. Предположим, что в алгоритме краткосрочного планирования предпочтение отдается тем программам, которые в течение недавнего времени использовали процессор меньше других. Объясните, почему в таком алгоритме отдается предпочтение программам, перегруженным операциями ввода-вывода, а не преимущественно использующим процессор.
- 2.3. Сравните стратегии планирования, которые могли бы использоваться для оптимизации системы разделения времени и многозадачной пакетной системы.
- 2.4. В чем назначение системных вызовов и как они соотносятся с операционной системой и с концепцией работы в режиме ядра и режиме пользователя?

2.5. Одним из основных модулей ядра операционной системы OS/390 для мейнфреймов IBM является System Resource Manager (SRM). Этот модуль распределяет ресурсы между адресными пространствами (процессами). Именно этот модуль делает операционную систему OS/390 одной из самых интеллектуальных. Никакие другие операционные системы для мейнфреймов, а тем более другие виды операционных систем, не могут выполнять функции, аналогичные тем, которые выполняет модуль SRM. В концепцию ресурсов входят процессор, реальная память и каналы ввода-вывода. SRM собирает статистику относительно использования процессора, каналов и различных ключевых структур данных; на основе анализа собранной статистики обеспечивается оптимальная производительность системы. Может производиться дополнительная настройка модуля для различных целей, в соответствии с которыми модуль динамически изменяет конфигурацию и характеристики производительности выполнения заданий. Модуль SRM, в свою очередь, составляет отчеты, на основании которых подготовленный оператор может улучшить производительность и изменить настройку системы с целью улучшения обслуживания клиентов.

В этой задаче идет речь об одном из видов деятельности модуля SRM. Реальная память подразделяется на блоки одинакового размера, которые называются кадрами. Компьютер может содержать многие тысячи кадров, в каждом из которых может находиться блок виртуальной памяти, называющийся страницей. Управление к модулю SRM переходит с частотой примерно 20 раз в секунду; при этом происходит проверка каждой из страниц памяти. Если данная страница не запрашивалась и не изменилась, показания счетчика увеличиваются на 1. Время от времени модуль SRM усредняет эти числа и определяет среднее время, в течение которого система не использует данную страницу кадра памяти. Для чего можно использовать эти данные и что для этого должен предпринять модуль SRM?

2.6. К многопроцессорной системе с восемью процессорами подключено 20 накопителей на магнитной ленте. Имеется большое количество заданий, переданных системе, и каждое из них для завершения требует не более четырех накопителей. Предположим, что каждое задание в течение длительного периода после запуска требует только трех накопителей, а четвертый накопитель требуется только в течение короткого времени ближе к концу работы. Предположим также, что имеется бесконечное количество таких заданий.

- a. Предположим, что планировщик операционной системы не будет запускать задание, пока не будут доступны четыре накопителя. Когда задание запускается, ему сразу же выделяются четыре накопителя, которые не освобождаются до тех пор, пока задание не будет выполнено. Каково максимальное количество заданий, которые могут одновременно находиться в работе? Каково максимальное и минимальное количество ленточных накопителей, которые могут простоять в результате этой стратегии?
- б. Предложите альтернативную стратегию для повышения эффективности использования накопителей при гарантированном отсутствии взаимоблокировок в системе. Каково максимальное количество заданий, которые могут выполняться одновременно? Каковы границы количества простояющих накопителей?

ЧАСТЬ ||

---

# ПРОЦЕССЫ



# ОПИСАНИЕ ПРОЦЕССОВ И УПРАВЛЕНИЕ ИМИ

**В ЭТОЙ ГЛАВЕ...**

## **3.1. Что такое процесс**

Основы

Процессы и управляющие блоки процессов

## **3.2. Состояния процесса**

Модель процесса с двумя состояниями

Создание и завершение процессов

Создание процессов

Завершение процессов

Модель с пятью состояниями

Приостановленные процессы

Необходимость сволинга

Другие использования приостановки

## **3.3. Описание процессов**

Управляющие структуры операционной системы

Структуры управления процессами

Местоположение процесса

Атрибуты процессов

Роль управляющего блока процесса

## **3.4. Управление процессами**

Режимы выполнения

Создание процессов

Переключение процессов

Когда нужно переключать процессы

Переключение режимов

Изменение состояния процесса

## **3.5. Выполнение кода операционной системы**

Ядро вне процессов

Выполнение в составе пользовательских процессов

Операционная система на основе процессов

## **3.6. Управление процессами в операционной системе UNIX SVR4**

Состояния процессов

Описание процессов

Управление процессами

## **3.7. Резюме**

## **3.8. Ключевые термины, контрольные вопросы и задачи**

Ключевые термины

Контрольные вопросы

Задачи

## Учебные цели

- Определить термин “процесс” и пояснить отношения между процессами и управляющими блоками процессов.
- Объяснить понятие “состояние процесса” и обсудить переходы между состояниями, которые проходят процессы.
- Перечислить и описать цели структур данных и их элементов, используемые операционными системами для управления процессами.
- Оценить требования к управлению процессами со стороны операционной системы.
- Понимать вопросы, связанные с выполнением кода операционной системы.
- Описать схему управления процессами в UNIX SVR4.

Все многозадачные операционные системы — от однопользовательских систем, таких как Windows для конечных пользователей, до больших ЭВМ, таких как мейнфреймы IBM с операционной системой z/OS, которая может поддерживать тысячи пользователей, — строятся вокруг концепции процесса. Большинство требований, которым должны соответствовать операционные системы, могут быть выражены с использованием понятия процесса.

- Операционная система должна чередовать выполнение нескольких процессов, чтобы повысить степень использования процессора при обеспечении разумного времени отклика.
- Операционная система должна распределять ресурсы между процессами в соответствии с заданной стратегией (т.е. предоставляя определенным функциям или приложениям более высокий приоритет), избегая в то же время взаимоблокировок.<sup>1</sup>
- От операционной системы могут потребоваться поддержка обмена информацией между процессами, а также обеспечение возможности создания процессов пользователями. Обе эти возможности могут помочь в структурировании приложений.

Подробное изучение операционных систем мы начнем со знакомства с представлением процессов и управлением ими. Затем мы рассмотрим состояния процессов, характеризующие их поведение; после перейдем к структурам данных, которые нужны операционной системе для управления процессами. К ним относятся структуры данных для представления состояния каждого процесса и структуры данных, регистрирующих другие характеристики процессов, необходимые операционной системе для достижения ее целей. Далее мы рассмотрим способы использования операционной системой этих структур данных для управления выполнением процессов. И наконец, будет рассмотрено управление процессами в операционной системе UNIX SVR4. Более современные примеры управления процессами будут рассмотрены в главе 4, “Потоки”.

<sup>1</sup> Взаимоблокировки рассматриваются в главе 6, “Параллельные вычисления: взаимоблокировка и голодание”. В качестве простейшего примера взаимоблокировка возникает тогда, когда каждый из двух процессов захватывает ресурсы, которые нужны другому процессу. Каждый из этих двух процессов может бесконечно долго ждать освобождения нужных ему ресурсов.

В этой главе время от времени упоминается виртуальная память. При описании процессов в большинстве случаев концепцию виртуальной памяти можно игнорировать, но при рассмотрении некоторых специализированных вопросов уместно обратить на нее особое внимание. Подробно виртуальная память обсуждается только в главе 8, “Виртуальная память”, но ее краткий обзор имеется в главе 2, “Обзор операционных систем”.

## 3.1. ЧТО ТАКОЕ ПРОЦЕСС

### ОСНОВЫ

Перед тем как определить термин *процесс*, следует подытожить некоторые введенные в главах 1, “Обзор компьютерной системы”, и 2, “Обзор операционных систем”, концепции.

1. Компьютерная платформа состоит из набора аппаратных ресурсов, таких как процессоры, основная память, модули ввода-вывода, таймеры, дисковые драйверы и т.д.
2. Компьютерные приложения разработаны для выполнения тех или иных заданий. Обычно они получают входные данные из внешнего мира, некоторым образом их обрабатывают и генерируют выходные данные.
3. Создание приложений для конкретной аппаратной платформы не эффективно. Вот основные причины этого.
  - Для одной и той же платформы могут быть разработаны разнообразные приложения. Поэтому имеет смысл разработка общих процедур для доступа к ресурсам компьютера.
  - Сам по себе процессор обеспечивает лишь ограниченную поддержку многозадачности. Для реальной многозадачности требуется программное обеспечение для управления совместным использованием процессора и других ресурсов несколькими приложениями в одно и то же время.
  - Когда несколько приложений активны в одно и то же время, необходимы защита данных и использование ввода-вывода и других ресурсов каждого приложения от других приложений.
4. Операционная система предназначается для обеспечения удобного, функционально богатого, безопасного и единообразного интерфейса для использования приложениями. Операционная система представляет собой уровень программного обеспечения между приложениями и аппаратным обеспечением компьютера (см. рис. 2.1), который поддерживает приложения и утилиты.
5. Мы можем рассматривать операционную систему как предоставляющую единое абстрактное представление ресурсов, которые могут быть запрошены приложениями и к которым они могут получить доступ. Эти ресурсы включают основную память, сетевые интерфейсы, файловые системы и т.д. После того как операционная система создала абстракции ресурсов для использования приложениями, она должна также управлять их использованием. Например, операционная система может разрешить совместное использование ресурсов и обеспечить их защиту.

Теперь, когда у нас есть концепции приложений, системного программного обеспечения и ресурсов, мы в состоянии рассмотреть, как операционная система может так систематически управлять выполнением приложений, что

- ресурсы оказываются доступными множеству приложений;
- физический процессор переключается между множеством приложений, так что все они выглядят выполняющимися одновременно;
- процессор и устройства ввода-вывода могут использоваться эффективно.

Этот подход, принятый во всех современных операционных системах, полагается на модель, в которой выполнение приложения соответствует существованию одного или нескольких процессов.

## Процессы и управляющие блоки процессов

Вспомним из главы 2, “Обзор операционных систем”, что мы предложили несколько определений термина *процесс*, включая следующие:

- выполняемая программа;
- экземпляр программы, выполняющейся на компьютере;
- сущность, которая может быть назначена процессору и выполнена на нем;
- единица активности, характеризуемая выполнением последовательности команд, текущим состоянием и связанным с ней множеством системных ресурсов.

Мы также можем рассматривать процесс как сущность, которая состоит из нескольких элементов. Двумя основными элементами процесса являются **программный код** (который может совместно использоваться другими процессами, которые выполняют ту же самую программу) и **набор данных**, связанный с этим кодом. Предположим, что процессор начинает выполнять этот программный код, и мы говорим об этой выполняемой сущности как о процессе. В любой конкретный момент *времени выполнения программы* этот процесс может быть уникально охарактеризован рядом элементов, включая следующие.

- **Идентификатор.** Уникальный идентификатор, связанный с этим процессом, чтобы отличать его от всех прочих процессов.
- **Состояние.** Если процесс выполняется в настоящее время, он находится в **состоянии выполнения**.
- **Приоритет.** Уровень приоритета по отношению к другим процессам.
- **Программный счетчик.** Адрес очередной выполняемой команды программы.
- **Указатели памяти.** Включают указатели на программный код и данные, связанные с этим процессом, а также на любые блоки памяти, совместно используемые с другими процессами.
- **Данные контекста.** Это данные, присутствующие в регистрах процессора во время выполнения процесса.
- **Информация о состоянии ввода-вывода.** Включает в себя внешние запросы ввода-вывода, устройства ввода-вывода, назначенные процессу, список файлов, используемых процессом, и т.д.
- **Учетная информация.** Может включать количество процессорного времени и времени работы, учетные записи и т.д.

Информация из представленного списка хранится в структуре данных, которая обычно называется **управляющим блоком процесса** (process control block, рис. 3.1), создается и управляет операционной системой. Важной информацией об управляющем блоке процесса является то, что он содержит достаточную информацию, чтобы можно было прервать выполняющийся процесс и позднее возобновить его выполнение, как если бы прерывание выполнения не произошло. Управляющий блок процесса является ключевым инструментом, который позволяет операционной системе обеспечивать поддержку нескольких процессов и многопроцессорную работу. Когда процесс прерывается, текущие значения программного счетчика и регистров процессора (данных контекста) сохраняются в соответствующих полях управляющего блока процесса, и состояние процесса изменяется на некоторое другое значение, такое как *заблокирован* (blocked) или *готов* (ready) (описаны позже). После этого операционная система может перевести в состояние выполнения некоторый другой процесс. Программный счетчик и данные контекста для этого процесса загружаются в регистры процессора, после чего начинается выполнение этого процесса.

Таким образом, можно сказать, что процесс состоит из кода программы и связанных с ним данных, плюс управляющий блок процесса. На однопроцессорном компьютере в любой момент времени выполняется не более одного процесса, и этот процесс находится в состоянии *выполнения*.

Идентификатор
Состояние
Приоритет
Счетчик команд
Указатели памяти
Данные контекста
Информация о состоянии ввода-вывода
Учетная информация
•
•
•

**Рис. 3.1.** Упрощенный управляющий блок процесса

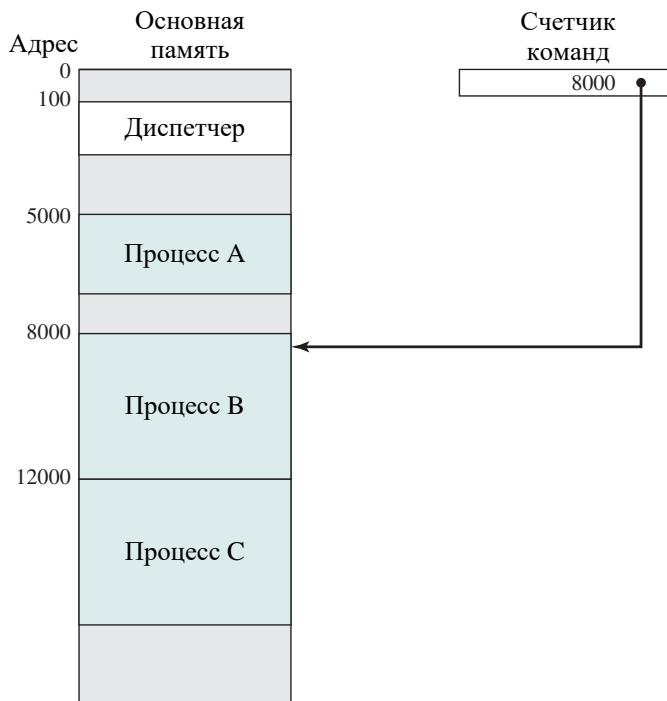
## 3.2. СОСТОЯНИЯ ПРОЦЕССА

Как уже говорилось, для каждой программы, которая должна быть выполнена, создается свой процесс, или задание. С точки зрения процесса его работа состоит в выполнении определенного набора команд; последовательность выполнения этих команд задается значениями, которые заносятся в регистр программного счетчика. Через некоторое время программный счетчик может указывать на код других программ, которые являются частями других процессов. Однако с точки зрения отдельной программы ее выполнение предусматривает последовательное выполнение ее команд.

Поведение процесса можно охарактеризовать, последовательно перечислив последовательность выполненных в ходе его работы команд. Такой перечень выполненных команд процесса называется его **следом** (trace). Поведение процессора можно охарактеризовать, показав, как чередуются следы различных процессов.

Рассмотрим очень простой пример. На рис. 3.2 показано размещение в памяти трех процессов. Чтобы упростить обсуждение, предположим, что виртуальная память не используется; таким образом, все три процесса представлены программами, которые полностью загружены в основную память. Кроме этих программ, в памяти находится небольшая программа-диспетчер, выполняющая переключение с одного процесса на другой. На рис. 3.3 показаны следы трех рассматриваемых процессов на ранних стадиях

их выполнения. Представлены первые 12 выполненных команд в процессах А и С; в процессе В выполнено четыре команды, и мы считаем, что они включают в себя операцию ввода-вывода, завершения которой должен ожидать процесс.



**Рис. 3.2.** Состояние системы в момент выполнения 13-го командного цикла

5000	8000	12000
5001	8001	12001
5002	8002	12002
5003	8003	12003
5004		12004
5005		12005
5006		12006
5007		12007
5008		12008
5009		12009
5010		12010
5011		12011

а) След процесса А

б) След процесса В

в) След процесса С

5000 Начальный адрес программы процесса А

8000 Начальный адрес программы процесса В

12000 Начальный адрес программы процесса С

**Рис. 3.3.** Следы процессов, показанных на рис. 3.2

Теперь рассмотрим эти следы с точки зрения процессора. На рис. 3.4 показаны чередующиеся следы, получившиеся в результате выполнения первых 52 командных циклов (для удобства они пронумерованы). Заштрихованные области на рисунке представляют код, выполняемый диспетчером. Диспетчер выполняет одну и ту же последовательность команд, так как от него требуется одна и та же функциональность. Будем считать, что операционная система позволяет непрерывно выполнять не более шести командных циклов одного и того же процесса, после чего процесс прерывается — это предотвращает монопольное использование всего процессорного времени одним из процессов. Из рис. 3.4 видно, что после первых шести команд процесса А следует перерыв, в течение которого выполняется некоторый код диспетчера, состоящий из шести команд, после чего управление передается процессу В.<sup>2</sup> Выполнив четыре команды, процесс В запрашивает операцию ввода-вывода и должен ожидать ее завершения. Поэтому процессор прекращает выполнять процесс В и с помощью диспетчера переходит к выполнению процесса С.

1	5000	27	12004
2	5001	28	12005
3	5002	-----Тайм-аут	
4	5003	29	100
5	5004	30	101
6	5005	31	102
-----Тайм-аут		32	103
7	100	33	104
8	101	34	105
9	102	35	5006
10	]103	36	5007
11	]104	37	5008
12	105	38	5009
13	8000	39	5010
14	8001	40	5011
15	8002	-----Тайм-аут	
16	8003	41	100
-----Ввод-Вывод		42	101
17	100	43	102
18	101	44	103
19	102	45	104
20	103	46	105
21	104	47	12006
22	105	48	12007
23	12000	49	12008
24	12001	50	12009
25	12002	51	12010
26	12003	52	12011
-----Тайм-аут			

100 — начальный адрес программы-диспетчера

Заштрихованные области — выполнение команд диспетчера. В первом столбце указаны номера командных циклов, во втором — адреса выполняемых команд

**Рис. 3.4.** Составной след процессов, изображенных на рис. 3.2

<sup>2</sup> Указанное количество команд, выполняемых при работе процессов и диспетчера, намного меньше, чем в действительности; в этом учебном примере такое неправдоподобно маленькое число используется для упрощения рассмотрения.

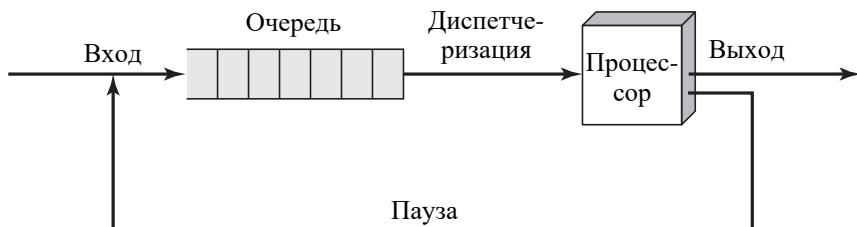
После очередного перерыва процессор возобновляет выполнение процесса А. По истечении отведенного этому процессу времени процесс В все еще ожидает завершения операции ввода-вывода, поэтому диспетчер снова передает управление процессу С.

## Модель процесса с двумя состояниями

Основной задачей операционной системы является управление выполнением процессов; эту задачу входит определение схемы чередования процессов и выделения им ресурсов. Первый шаг, который следует предпринять при составлении программы, предназначенной для управления процессами, состоит в описании ожидаемого поведения процессов.

Простейшую модель можно построить, исходя из того, что в любой момент времени процесс либо выполняется, либо не выполняется. Таким образом, процесс может быть в одном из двух состояний: выполняющийся или не выполняющийся (рис. 3.5, а). После создания нового процесса операционная система создает его управляющий блок и вводит его в систему в состоянии не выполняющегося. Созданный процесс, о существовании которого известно операционной системе, ждет, пока он сможет быть запущен. Время от времени выполняющиеся процессы будут прерываться, и та часть операционной системы, которая выполняет функции диспетчера, будет выбирать для выполнения другой процесс. Выполняющийся перед этим процесс перейдет из состояния выполняющегося в состояние не выполняющегося, а в состояние выполняющегося перейдет один из ожидающих процессов.

Анализируя эту простую модель, можно сделать некоторые выводы относительно элементов дизайна операционной системы. Необходим способ, с помощью которого будет представлен каждый процесс, чтобы операционная система могла за ним следить.



б) Диаграмма использования очереди

**Рис. 3.5. Модель процесса с двумя состояниями**

С каждым процессом нужно связать определенную информацию, в которую будет входить его текущее состояние и размещение в памяти. Не выполняющиеся процессы следует организовать в какую-то очередь, где они ожидали бы своего выполнения. Один из возможных вариантов предложен на рис. 3.5, б. Здесь имеется единая очередь, ее элементами являются указатели на управляющие блоки процессов. Можно предложить и другую схему, в которой очередь состоит из связанного списка блоков данных, где каждый блок представляет отдельный процесс; позже мы вернемся к исследованию этой реализации.

Поведение диспетчера можно описать следующим образом. Процесс, работа которого прервана, переходит в очередь процессов, ожидающих выполнения. Если же процесс завершен, он выводится из системы. В любом случае для выполнения диспетчер выбирает из очереди следующий процесс.

## Создание и завершение процессов

Прежде чем предпринять попытку улучшить нашу простую модель с двумя состояниями, будет полезно обсудить создание и завершение процессов, ведь время жизни процесса ограничивается моментами его создания и завершения вне зависимости от модели поведения.

### Создание процессов

Когда операционная система собирается добавить новый процесс к тем, которые уже состоят на учете, она создает структуры данных, использующиеся при управлении этим процессом, и размещает его адресное пространство в основной памяти. Эти структуры данных будут описаны в разделе 3.3. С помощью указанных действий и создается новый процесс.

К созданию процесса могут привести четыре события, перечисленные в табл. 3.1.

**Таблица 3.1. Причины создания процессов**

<b>Новое пакетное задание</b>	В операционную систему для обработки поступает управляющий поток пакетных заданий (обычно с ленты или с диска). Готовясь принять на обработку новое задание, операционная система считывает очередную последовательность команд управления заданиями
<b>Вход в систему в интерактивном режиме</b>	В систему с терминала входит новый пользователь
<b>Создание операционной системой процесса, необходимого для работы каких-либо служб</b>	Операционная система может создать процесс для выполнения некоторой функции, которая требуется для программы пользователя. При этом пользователь не должен ждать, пока закончится ее выполнение (как в примере, в котором создавался процесс управления печатью)
<b>Порождение одного процесса другим</b>	С целью структуризации программы или использования возможностей параллельных вычислений программа может создавать другие процессы

В среде пакетной обработки процесс создается в ответ на поступление задания; в интерактивной среде процесс создается при попытке нового пользователя войти в систему. В обоих случаях ответственность за создание нового процесса лежит на операционной системе. Кроме того, операционная система может создавать процесс по требованию приложения. Например, если пользователь отправляет запрос на распечатку файла, операционная система может создать процесс, управляющий печатью. Затем процесс, производивший запрос, может продолжить свою работу, независимо от того, сколько времени понадобится для печати.

Традиционно операционная система создает все процессы незаметно для пользователя или приложения; такой способ принят во многих современных операционных системах. Однако иногда требуется, чтобы один процесс мог послужить причиной создания другого процесса. Например, процесс приложения может сгенерировать другой процесс, который будет получать данные от первого процесса и приводить их к виду, удобному для дальнейшего анализа. Новый процесс будет работать параллельно с приложением и время от времени активизироваться для получения ставшими доступными новых данных. Такая организация может быть очень полезна для структурирования приложений. В качестве другого примера можно привести ситуацию, в которой процесс-сервер (например, сервер печати или файловый сервер) может генерировать новый процесс для каждого обрабатываемого им запроса. Создание операционной системой процесса по явному запросу другого процесса называется **порождением процесса** (process spawning).

Когда один процесс порождает другой, то порождающий процесс называется **родительским**, или предком (parent), а порождаемый процесс — **дочерним**, или потомком (child). Обычно “родственные” процессы обмениваются между собой информацией и взаимодействуют друг с другом. Организация такого взаимодействия является достаточно трудной задачей для программиста (эта тема рассматривается в главе 5, “Параллельные вычисления: взаимоисключения и многозадачность”).

### **Завершение процессов**

В табл. 3.2 перечислены типичные причины завершения процессов. В любой компьютерной системе должны быть средства, позволяющие определить, закончилось выполнение процесса или нет. Пакетное задание должно включать в себя команду типа Halt (останов) или какой-то явный вызов службы операционной системы, приводящий к завершению процесса. В первом случае генерируется прерывание для извещения операционной системы о завершении процесса. Например, в системе с разделением времени процесс пользователя должен быть завершен, когда пользователь выходит из системы или выключает терминал. На персональном компьютере или рабочей станции пользователь может выйти из приложения (например, закрыть программу обработки текста или электронную таблицу). Все эти действия в конечном счете приведут к тому, что будет вызвана служба операционной системы, завершающая процесс.

Кроме того, к завершению процессов могут привести и другие ошибки или условия отказа. В табл. 3.2 перечислены некоторые из наиболее часто возникающих условий.<sup>3</sup>

<sup>3</sup> Синхордитальная операционная система в некоторых случаях может позволять пользователю восстановить работу процесса, в котором возникли условия отказа, не прекращая его. Например, если пользователь запросил доступ к файлу, к которому у него нет доступа, операционная система может просто сообщить пользователю, что ему отказано в доступе, предоставив процессу возможность продолжать свою работу.

Наконец, в некоторых операционных системах процесс может быть завершен процессом, который его породил, а также при завершении самого родительского процесса.

**Таблица 3.2. Причины завершения процессов**

<b>Обычное завершение</b>	Процесс вызывает службу операционной системы, чтобы сообщить, что он завершил свою работу
<b>Превышение лимита отведенного программе времени</b>	Общее время выполнения процесса превышает заданное предельное значение. Это время может измеряться несколькими способами. Одним из них является учет полного времени, затраченного на выполнение ("по настенным часам"); при выполнении интерактивного процесса время можно отсчитывать с момента последнего ввода данных пользователем
<b>Недостаточный объем памяти</b>	Для работы процесса требуется больше памяти, чем имеется в системе
<b>Выход за пределы отведенной области памяти</b>	Процесс пытается получить доступ к ячейке памяти, к которой у него нет прав доступа
<b>Ошибка защиты</b>	Процесс пытается использовать недоступный для него ресурс или файл или пытается сделать это недопустимым образом, например производит попытку записи в файл, открытый только для чтения
<b>Арифметическая ошибка</b>	Процесс пытается выполнить запрещенную арифметическую операцию, например деление на нуль, или пытается использовать число, превышающее возможности аппаратного обеспечения
<b>Излишнее ожидание</b>	Процесс ждет наступления определенного события дольше, чем задано в параметрах системы
<b>Ошибка ввода-вывода</b>	Во время ввода или вывода происходит ошибка. Например, не удается найти нужный файл или выполнить чтение или запись за максимально возможное количество попыток (когда, например, на магнитном носителе попался дефектный участок) или производится попытка выполнить недопустимую операцию (например, чтение с печатающего устройства)
<b>Неверная команда</b>	Процесс пытается выполнить несуществующую команду (часто это бывает, если процесс переходит в область данных и пытается интерпретировать их как команду)
<b>Команда с недоступными привилегиями</b>	Процесс пытается использовать команду, зарезервированную для операционной системы
<b>Неправильное использование данных</b>	Часть данных принадлежит не к тому типу или не инициализирована
<b>Вмешательство оператора или операционной системы</b>	По какой-либо причине операционная система может завершить процесс (например, в случае взаимоблокировки)
<b>Завершение родительского процесса</b>	При завершении родительского процесса операционная система может автоматически прекращать все его дочерние процессы
<b>Запрос со стороны родительского процесса</b>	Обычно родительский процесс имеет право прекращать любой из своих дочерних процессов

## Модель с пятью состояниями

Если бы все процессы всегда были готовы к выполнению, то очередь на рис. 3.4, б могла бы работать вполне эффективно. Такая очередь работает по принципу обработки в порядке поступления, а процессор обслуживает имеющиеся в наличии процессы **циклическим** (круговым<sup>4</sup>, round-robin) методом (каждому процессу в очереди отводится определенный промежуток времени, по истечении которого процесс возвращается обратно в очередь, если он не был блокирован). Однако даже в таком простом примере, который описан выше, подобная реализация не является адекватной: одни из не выполняющихся процессов готовы к выполнению, в то время как другие являются заблокированными и ждут окончания операции ввода-вывода. Таким образом, при наличии только одной очереди диспетчер не может просто выбрать для выполнения первый процесс из очереди. Перед этим он должен просмотреть весь список, отыскивая незаблокированный процесс, который находится в очереди дальше других.

Естественнее было бы разделить все не выполняющиеся процессы на два типа: готовые к выполнению и заблокированные. Такая схема показана на рис. 3.6.

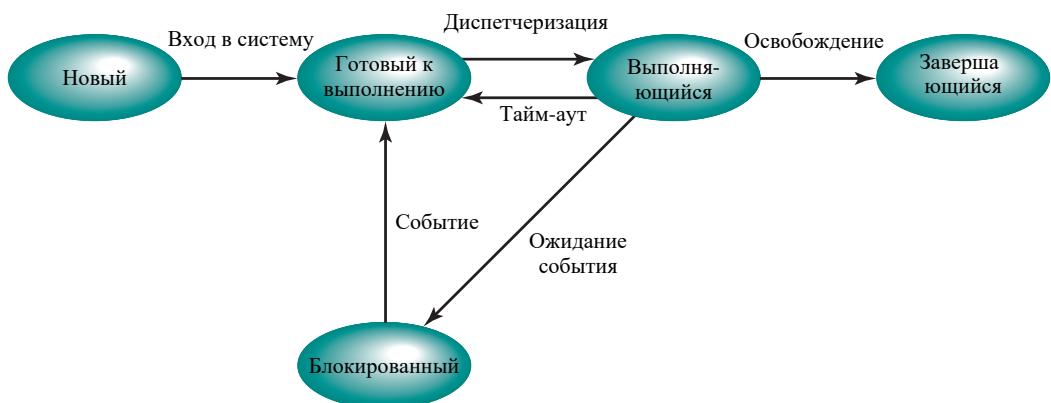


Рис. 3.6. Модель с пятью состояниями

Здесь добавлены еще два состояния, которые окажутся полезными в дальнейшем. Опишем каждое из пяти состояний процессов, представленных на диаграмме.

- Выполняющийся.** Процесс, который выполняется в текущий момент времени. В настоящей главе предполагается, что на компьютере установлен только один процессор, поэтому в этом состоянии может находиться только один процесс.
- Готовый к выполнению.** Процесс, который может быть запущен, как только для этого представится возможность.
- Блокированный/Ожидающий<sup>5</sup>.** Процесс, который не может выполняться до тех пор, пока не произойдет некоторое событие, например завершение операции ввода-вывода.

<sup>4</sup> Иногда используется термин “карусельный метод”. — Примеч. пер.

<sup>5</sup> *Ожидающий* часто используется как альтернативное название состояния *Блокированный*. В общем случае мы будем использовать термин *Блокированный*, но эти термины взаимозаменяемы.

4. **Новый.** Только что созданный процесс, который еще не помещен операционной системой в пул выполнимых процессов. Обычно это новый процесс, который еще не загружен в основную память, хотя управляющий блок процесса уже создан.
5. **Завершающийся.** Процесс, удаленный операционной системой из пула выполнимых процессов из-за завершения его работы или аварийно прерванный по какой-либо иной причине.

Состояния “новый” и “завершающийся” представляют собой полезные конструкции для управления процессами. Первое из них соответствует процессу, который был только что определен. Например, если новый пользователь пытается войти в систему с разделением времени или в систему поступает новое пакетное задание, операционная система может определить новый процесс в два этапа. Во-первых, она выполняет всю необходимую рутинную работу: процессу присваивается идентификатор и формируются все необходимые для управления процессом таблицы. В этот период времени процесс находится в состоянии нового. Это означает, что операционная система выполнила необходимые для создания процесса действия, но еще не приготовилась к его запуску. Например, операционная система может иметь ограничения по количеству одновременно выполняющихся процессов. Такие ограничения устанавливаются, например, чтобы не снижать производительность системы (или не переполнять основную память). Таблицы управления новыми процессами с необходимой операционной системе информацией содержатся в основной памяти, однако самих процессов там еще нет, т.е. код программы, которую нужно выполнить, не загружен в память, и данным, относящимся к этой программе, не выделено пространство. Отвечающая такому процессу программа остается во вторичной памяти (обычно это диск).<sup>6</sup>

Выход процесса из системы также происходит в два этапа. Во-первых, процесс переходит в состояние завершающегося при достижении точки естественного завершения, а также когда он останавливается из-за возникновения неустранимой ошибки или когда его останавливает другой процесс, обладающий необходимыми для этого полномочиями. После этого момента процесс больше не может выполняться. Операционная система временно сохраняет таблицы и другую информацию, связанную с этим заданием, так что вспомогательные программы могут получить все необходимые сведения о завершившемся процессе (например, эти данные могут понадобиться программе, ведущей учет использования процессорного времени и других ресурсов). После того как эти программы извлекают всю необходимую информацию, операционной системе больше не нужно хранить данные, связанные с процессом, и он полностью удаляется из системы.

На рис. 3.6 показаны типы событий, соответствующие каждому из возможных переходов из одного состояния в другое. Возможны следующие переходы.

- **Нулевое состояние → Новый.** Для выполнения программы создается новый процесс. Это событие может быть вызвано одной из причин, перечисленных в табл. 3.1.

---

<sup>6</sup> В рассматриваемой модели концепция виртуальной памяти во внимание не принимается. В системах, поддерживающих виртуальную память, при переходе процесса из состояния нового в состояние готового к выполнению код, и данные программы загружаются в виртуальную память. Краткое обсуждение виртуальной памяти можно найти в главе 2, “Обзор операционных систем”, а более подробное — в главе 8, “Виртуальная память”.

- **Новый → Готовый.** Операционная система переводит процесс из состояния нового в состояние готового к выполнению, когда она будет готова к обработке дополнительных процессов. В большинстве систем устанавливается ограничение на количество существующих процессов или на объем выделяемой для процессов виртуальной памяти. Таким образом предотвращается снижение производительности, которое может произойти, если будет загружено слишком много активных процессов.
- **Готовый → Выполняющийся.** Когда наступает момент выбора нового процесса для запуска, операционная система выбирает один из готовых для выполнения процессов. Принцип этого выбора обсуждается в части 4 книги.
- **Выполняющийся → Завершающийся.** Если процесс сигнализирует об окончании своей работы или происходит его аварийное завершение, операционная система прекращает его выполнение. (См. табл. 3.2.)
- **Выполняющийся → Готовый.** Этот переход чаще всего происходит из-за того, что процесс выполняется в течение максимального промежутка времени, отведенного для непрерывной работы одного процесса. Подобная стратегия планирования используется практически во всех многозадачных операционных системах. Такой переход возможен и по некоторым другим причинам, зависящим от конкретной операционной системы (эти причины могут быть не реализованы в той или иной конкретной операционной системе). Например, если операционная система назначает разным процессам различные приоритеты, то может случиться так, что процесс будет выгружен из-за появления процесса с более высоким приоритетом. Предположим, что выполняется процесс А, имеющий определенный приоритет, а процесс В, приоритет которого выше, блокирован. Когда операционная система обнаружит, что произошло событие, ожидаемое процессом В, она переведет этот процесс в состояние готовности, в результате чего процесс А может быть прерван и управление перейдет к процессу В. Говорят, что операционная система **вытесняет** (preempt) процесс А.<sup>7</sup> Наконец, процесс сам по себе может отказаться от использования процессора. Примером может служить некоторый фоновый процесс, который периодически выполняет некоторые функции учета или обслуживания.
- **Выполняющийся → Блокированный.** Процесс переводится в заблокированное состояние, если для продолжения работы требуется наступление некоторого события, которое он вынужден ожидать. Посыпаемый операционной системе запрос обычно имеет вид вызова какой-нибудь системной службы, т.е. вызова процедуры, являющейся частью кода операционной системы. Процесс может запросить ресурс (например, файл или совместно используемую часть виртуальной памяти), который окажется временно недоступным, и потребуется подождать его освобождения. Кроме того, возможна ситуация, в которой для продолжения процесса требуется выполнить некоторое действие, например операцию ввода-вывода. Если процессы обмениваются информацией друг с другом, один из них может быть блокирован в состоянии ожидания ввода или сообщения от другого процесса.

<sup>7</sup> Вообще говоря, термин **вытеснение** означает, что процесс лишается некоторого ресурса до того, как он завершил свою работу с ним. В данном случае ресурсом является процессор — выполняющийся процесс мог бы продолжаться, но его выполнение прерывается из-за другого процесса.

- **Блокированный → Готовый.** Заблокированный процесс переходит в состояние готовности к выполнению в тот момент, когда происходит ожидаемое им событие.
- **Готовый → Завершающийся.** Чтобы не усложнять картину, этот переход на диаграмме состояний не показан. В некоторых системах родительский процесс может в любой момент прервать выполнение дочернего процесса. Кроме того, дочерние процессы могут прекратиться при завершении родительского процесса.
- **Блокированный → Завершающийся.** (См. комментарии к предыдущему пункту.)

Возвратимся к нашему простому примеру. На рис. 3.7 показаны переходы каждого из трех рассматриваемых процессов в различные состояния. На рис. 3.8, а предложен один из способов реализации порядка очередности. Имеется две очереди: очередь готовых к выполнению процессов и очередь заблокированных процессов. Каждый процесс, поступающий в систему для обработки, помещается в очередь готовых к выполнению процессов. Когда операционной системе приходит время выбрать для выполнения другой процесс, она выбирает его из этой очереди. Если схема приоритетов отсутствует, эта очередь может работать по принципу “первым вошел — первым вышел”. Когда выполнение процесса прерывается, он, в зависимости от обстоятельств, может либо завершиться, либо попасть в одну из двух очередей (готовых к выполнению или заблокированных процессов). И наконец, после того как произойдет событие, все ожидающие его процессы из очереди заблокированных перемещаются в очередь готовых к выполнению процессов.

Такая организация приводит к тому, что после любого события операционная система должна сканировать всю очередь заблокированных процессов, отыскивая среди них те, которые ожидают именно этого события. В большой операционной системе в подобной очереди может пребывать несколько сотен или даже тысяч процессов. Поэтому эффективнее организовать несколько очередей, для каждого события — свою. Тогда при каком-то событии все процессы из соответствующей очереди можно будет перевести в очередь готовых к выполнению процессов (см. рис. 3.8, б).

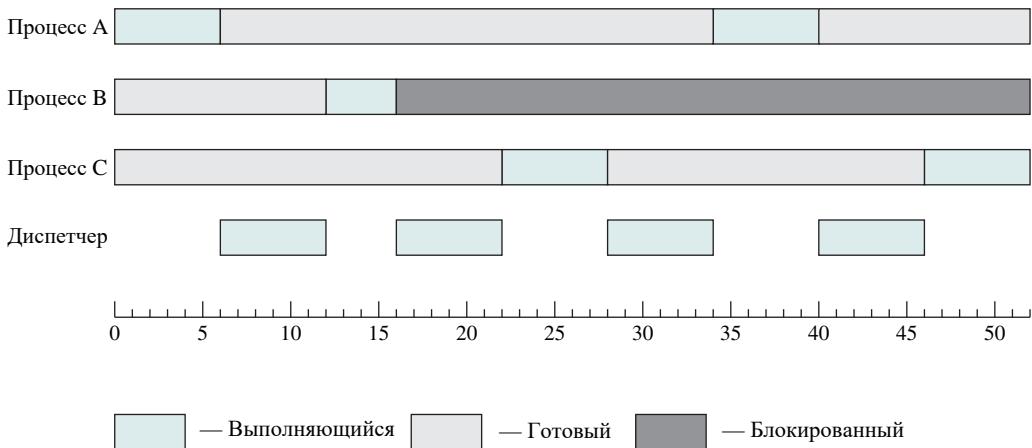
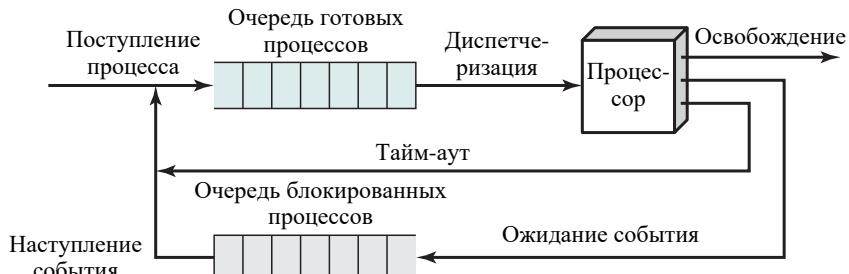
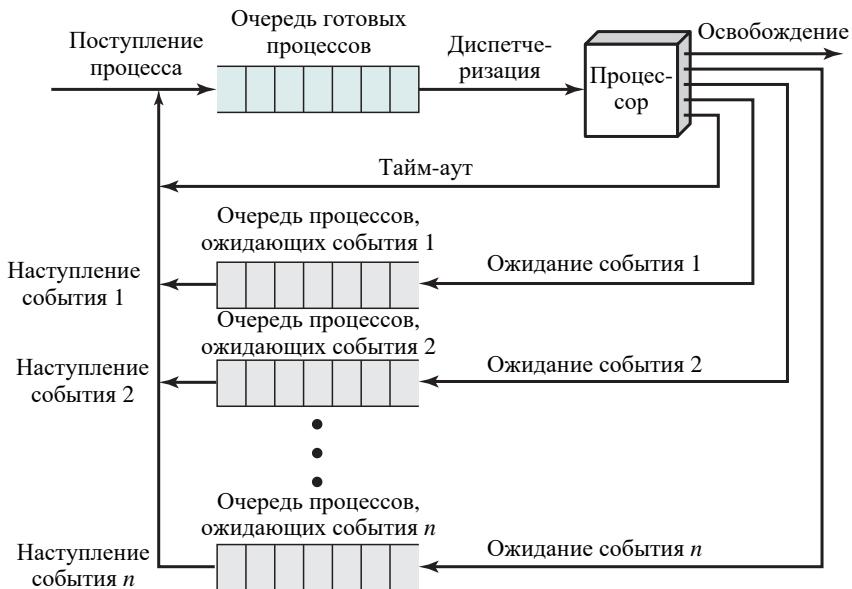


Рис. 3.7. Состояния процессов из рис. 3.4



а) Схема с одной очередью блокированных процессов



б) Схема с несколькими очередями блокированных процессов

**Рис. 3.8.** Модель, представленная на рис. 3.6, с использованием очередей

И последнее усовершенствование: если диспетчеризация осуществляется с использованием приоритетов, то было бы удобно организовать несколько очередей готовых к выполнению процессов. В такой схеме каждая очередь соответствовала бы своему уровню приоритета. Тогда операционной системе легко было бы определять готовый для выполнения процесс с наивысшим приоритетом, который ждет своей очереди дольше всех остальных.

## Приостановленные процессы

### Необходимость свопинга

Три основных состояния процессов, описанные в предыдущем разделе (готовый, выполняющийся и блокированный), позволяют смоделировать поведение процессов и получить представление о реализации операционной системы. Многие операционные системы разработаны на основе только этих трех состояний.

Однако можно привести убедительные аргументы в пользу добавления в модель и других состояний. Чтобы понять, какие выгоды могут дать эти новые состояния, рассмотрим систему, не использующую виртуальную память, в которой каждый процесс перед выполнением нужно загрузить в основную память. Таким образом, все процессы, представленные на рис. 3.8, б, должны находиться в основной памяти.

Теперь вспомним, что причиной разработки всех этих схем послужило более медленное, по сравнению с вычислениями, выполнение операций ввода-вывода, приводящее к простоям процессора в однозадачной системе. Однако организация работы в соответствии со схемой, помещенной на рис. 3.8, б, полностью эту проблему не решает. Конечно, при работе в соответствии с такой моделью в памяти находится несколько процессов, и пока одни процессы ожидают окончания операций ввода-вывода, процессор может перейти к выполнению других процессов. Но процессор работает настолько быстрее выполнения операций ввода-вывода, что вскоре все находящиеся в памяти процессы оказываются в состоянии ожидания. Таким образом, процессор может в основном проставлять даже в многозадачной системе.

Что же делать? Можно увеличить емкость основной памяти, чтобы в ней помещалось больше процессов. Но в таком подходе есть два недостатка. Во-первых, это ведет к резкому повышению стоимости памяти системы. Во-вторых, аппетит программиста в использовании памяти для своих программ возрастает пропорционально падению ее стоимости, так что увеличение объема памяти приводит к увеличению размера процессов, а не к росту их числа.

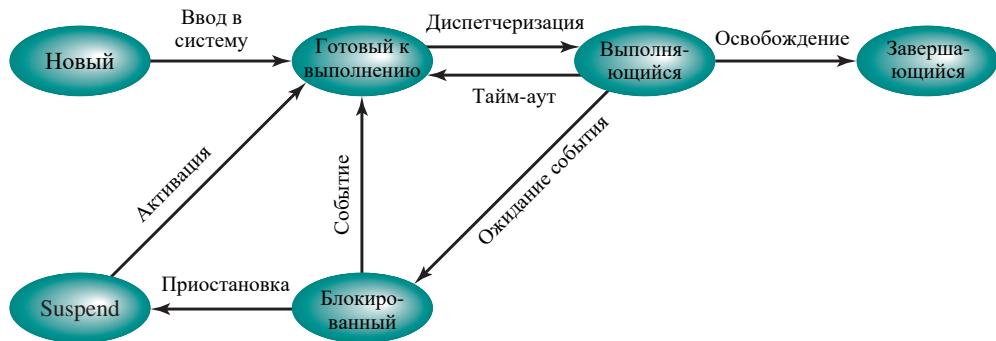
Другим решением проблемы является свопинг, который включает в себя перенос части (или всего) процесса из основной памяти на диск. Если в основной памяти нет ни одного готового к выполнению процесса, операционная система переносит один из блокированных процессов на диск (осуществляет его свопинг), помещая его в очередь приостановленных процессов, которые временно извлечены из основной памяти. Затем операционная система загружает другой процесс из очереди приостановленных, после чего продолжает его выполнение.

Однако свопинг сам по себе является операцией ввода-вывода, поэтому есть риск ухудшить ситуацию, вместо того чтобы улучшить ее. Однако благодаря тому что обмен информацией с диском обычно происходит быстрее прочих операций ввода-вывода (например, запись-считывание с ленты или вывод на принтер), свопинг чаще всего повышает производительность работы системы в целом.

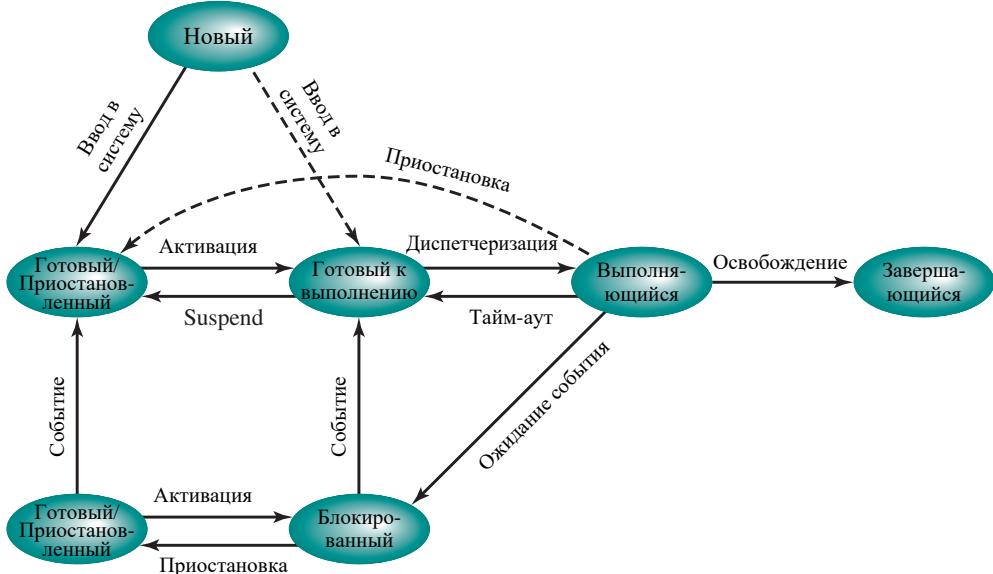
Если в модель поведения процессов ввести только что описанный свопинг, то нам придется ввести и новое состояние — состояние приостановленного (*suspended*) процесса (рис. 3.9, а). Когда все процессы в основной памяти находятся в блокированном состоянии, операционная система может приостановить один из процессов, переведя его в приостановленное состояние и сбросив на диск. Освободившееся в основной памяти пространство можно будет использовать для загрузки другого процесса.

После того как операционная система выгрузила один из процессов на диск, у нее есть две возможности выбора процесса для загрузки в основную память: либо создать новый процесс, либо загрузить процесс, который был приостановлен. Может показаться, что лучше было бы загрузить для обработки ранее приостановленный процесс, что не приведет к увеличению нагрузки на систему.

Однако это не совсем так. Все процессы, перед тем как они были приостановлены, находились в блокированном состоянии.



а) С одним приостановленным состоянием



б) С двумя приостановленными состояниями

Рис. 3.9. Диаграмма переходов с учетом приостановленных процессов

Ясно, что возвращение в память блокированного процесса не даст никаких результатов, потому что он по-прежнему не готов к выполнению. Вспомним также, что каждый процесс в приостановленном состоянии блокирован в ожидании какого-то определенного события, и если это событие происходит, процесс перестает быть блокированным и потенциально готов к выполнению.

Следовательно, этот аспект следует учесть при разработке операционной системы. Мы имеем дело с двумя независимыми ситуациями: ожидает ли процесс какого-либо события (т.е. блокирован он или нет) и выгружен ли процесс из основной памяти (т.е. приостановлен он или нет). Чтобы учесть 2×2 возможные комбинации, нужны четыре перечисленных ниже состояния.

1. **Готовый.** Процесс, который находится в основной памяти и готов к выполнению.
2. **Блокированный.** Процесс, находящийся в основной памяти и ожидающий какого-то события.
3. **Блокированный/Приостановленный.** Процесс, находящийся во вторичной памяти и ожидающий какого-то события.
4. **Готовый/Приостановленный.** Процесс, находящийся во вторичной памяти, но уже готовый к выполнению; для этого его нужно только загрузить в основную память.

Перед тем как составлять диаграмму переходов состояний, в которой учитываются два новых приостановленных состояния, следует упомянуть еще одно обстоятельство. До сих пор мы не учитывали существование виртуальной памяти; считалось, что процесс находится либо полностью в основной памяти, либо полностью вне ее. При наличии виртуальной памяти появляется возможность выполнять процесс, который загружен в основную память лишь частично. Если происходит обращение к отсутствующему в основной памяти адресу процесса, эта часть процесса может быть загружена. Казалось бы, использование виртуальной памяти избавляет от необходимости явного свопинга, потому что любой нужный адрес любого процесса можно перенести в основную память или из нее с помощью аппаратного обеспечения процессора, управляющего памятью. Однако, как мы увидим в главе 8, “Виртуальная память”, при наличии достаточно большого количества активных процессов, которые все частично находятся в основной памяти, производительность виртуальной памяти может оказаться недостаточной. Поэтому даже при наличии виртуальной памяти операционной системе время от времени требуется явно и полностью выгружать процессы из основной памяти ради повышения общей производительности.

А теперь рассмотрим модель переходов состояний, представленную на рис. 3.9, б (пунктирными линиями показаны переходы, которые возможны, но не являются обязательными). Среди новых переходов наиболее важными являются следующие.

- **Блокированный → Блокированный/Приостановленный.** Если к выполнению не готов ни один процесс, то по крайней мере один блокированный процесс выгружается из памяти, чтобы освободить место для другого процесса, который не является блокированным. Этот переход можно выполнять и при наличии готовых к выполнению процессов — в частности, если операционная система определит, что для выполняющегося в настоящее время процесса или процесса, управление к которому перейдет в ближайшее время, нужно увеличить объем основной памяти для обеспечения адекватной производительности.
- **Блокированный/Приостановленный → Готовый/Приостановленный.** Процесс в состоянии блокированного/приостановленного переходит в состояние готового к выполнению/приостановленного, если происходит событие, которого ожидал этот процесс. Заметим, что для такого перехода операционная система должна иметь доступ к информации о состоянии приостановленных процессов.
- **Готовый/Приостановленный → Готовый.** Когда в основной памяти нет готовых к выполнению процессов, операционной системе для продолжения вычислений требуется загрузить процесс в память. Может случиться и так, что у готового к

выполнению/приостановленного процесса окажется более высокий приоритет, чем у любого другого из готовых к выполнению процессов. В такой ситуации разработчик операционной системы может решить, что важнее обеспечить приоритет процесса, чем минимизировать свопинг.

- **Готовый → Готовый/Приостановленный.** Обычно операционная система предпочитает приостанавливать не готовый, а заблокированный процесс, поскольку к выполнению готового процесса можно приступить немедленно, а блокированный процесс только зря занимает основную память, поскольку не может быть выполнен. Однако иногда оказывается, что единственный способ освободить достаточно большой блок основной памяти — это приостановить готовый к выполнению процесс. Операционная система может также вместо блокированного процесса с более высоким приоритетом приостановить готовый к выполнению процесс с более низким приоритетом, если блокированный процесс достаточно скоро будет готов к выполнению.

Кроме того, заслуживают рассмотрения и другие переходы.

- **Новый → Готовый/Приостановленный и Новый → Готовый.** После создания нового процесса этот процесс может быть добавлен либо в очередь готовых к выполнению, либо в очередь готовых к выполнению/приостановленных процессов. В любом из этих случаев операционная система должна создать управляющий блок процесса и выделить ему адресное пространство. Может оказаться лучше выполнять эти действия на ранних этапах, чтобы иметь большой пул неблокированных процессов. Однако если придерживаться этой стратегии, то в основной памяти может не хватить места для нового процесса. По этой причине предусмотрен переход нового процесса в состояние готового к выполнению/приостановленного. С другой стороны, создание процесса в “по требованию” приводит к уменьшению непроизводительных затрат и позволяет операционной системе выполнять свои обязанности по созданию процессов даже тогда, когда она переполнена блокированными процессами.
- **Блокированный/Приостановленный → Блокированный.** На первый взгляд может показаться, что учитывать такой переход бессмысленно. Зачем, в конце концов, загружать в память процесс, который не готов к выполнению? Однако рассмотрим такой сценарий: завершился некоторый процесс, освободив при этом определенную часть основной памяти. В очереди заблокированных/приостановленных процессов находится процесс, приоритет которого выше, чем приоритет любого процесса из очереди готовых к выполнению/приостановленных процессов. Кроме того, операционная система располагает аргументами в пользу того, что довольно скоро произойдет событие, которое снимет блокировку с этого высокоприоритетного процесса. При таких обстоятельствах резонно отдать предпочтение блокированному процессу перед готовыми к выполнению, загрузив в основную память именно его.
- **Выполняющийся → Готовый/Приостановленный.** Обычно выполняющийся процесс, у которого вышло отведенное ему время, переходит в состояние готового к выполнению. Однако при наличии процесса с более высоким приоритетом, который находился в очереди заблокированных/приостановленных процессов и только

что был разблокирован, операционная система может отдать предпочтение именно ему. Чтобы освободить часть основной памяти, она может перевести выполняющийся процесс непосредственно в состояние готового к выполнению/приостановленного процесса.

- **Произвольное состояние → Завершение.** Обычно завершается выполняющийся в настоящий момент процесс — это происходит либо из-за того, что он выполнен до конца, либо из-за ошибок при его работе. Однако в некоторых операционных системах процесс может завершаться создавшим его процессом или вместе с завершением родительского процесса. Такое завершение возможно при условии, что процессы из любого состояния могут переходить в состояние завершения.

### Другие использования приостановки

До сих пор концепция временной остановки процесса ассоциировалась у нас с его отсутствием в основной памяти. Процесс, который отсутствует в основной памяти, не может быть запущен немедленно, независимо от того, ожидает ли он какого-то события.

Однако концепцию приостановленного процесса можно обобщить. Определим приостановленный процесс как такой, который удовлетворяет следующим критериям.

1. Процесс не может быть запущен в данный момент.
2. Процесс может как ожидать какого-то события, так и находиться в состоянии ожидания. Если он находится в состоянии ожидания, то блокирующее событие не связано с условием приостановки, а наступление события не означает, что процесс может быть выполнен.
3. Процесс приостанавливается самостоятельно, операционной системой или родительским процессом.
4. Процесс не может выйти из состояния приостановленного до тех пор, пока не будет явно выведен из этого состояния приостановившим его агентом.

В табл. 3.3 перечислены некоторые причины, по которым процессы могут быть приостановлены. Одной из ранее обсуждавшихся причин является необходимость выгрузить процесс на диск, чтобы вместо него можно было загрузить готовый к выполнению процесс или просто ослабить нагрузку на виртуальную память, предоставив каждому из оставшихся процессов дополнительную порцию основной памяти. Могут быть и другие причины для того, чтобы приостановить процесс. Рассмотрим, например, процесс, который используется для наблюдения за работой системы. Этот процесс может использоваться для фиксирования интенсивности использования различных ресурсов (процессора, памяти, каналов) и скорости выполнения в системе пользовательских процессов. Операционная система может время от времени включать и отключать такой процесс. Если операционная система выявит проблему (например, состояние взаимоблокировки, о котором рассказывается в главе 6, “Параллельные вычисления: взаимоблокировка и голодание”), она может приостановить процесс. Другим примером служат неполадки в линии связи. В данной ситуации оператор может отдать операционной системе команду приостановить процесс, использующий эту линию, чтобы выполнить необходимые тесты и исправить ситуацию.

**Таблица 3.3. Причины, по которым процессы переходят в состояние приостановленных**

<b>Свопинг</b>	Операционной системе нужно освободить пространство в основной памяти, чтобы загрузить готовый к выполнению процесс
<b>Другие причины, появляющиеся у операционной системы</b>	Операционная система может приостановить фоновый или служебный процесс, а также подозрительный процесс, послуживший вероятной причиной возникновения ошибок
<b>Запрос интерактивного пользователя</b>	Пользователь может захотеть приостановить процесс, чтобы приступить к отладке программы или в связи с использованием некоторого ресурса
<b>Временной режим выполнения</b>	Процесс может выполняться периодически (например, программа для учета использования ресурсов или работы системы); в промежутках между выполнением такой процесс может приостанавливаться
<b>Запрос родительского процесса</b>	Родительскому процессу может понадобиться возможность приостанавливать выполнение дочерних процессов для их проверки или модификации, а также для координации работы нескольких дочерних процессов

Другие причины временной остановки процессов связаны с действиями интерактивного пользователя. Например, если пользователь заподозрил, что в программе есть дефект, он может приступить к отладке программы, приостановив ее выполнение. При этом пользователь может тестировать и модифицировать программу или данные, а затем возобновить ее выполнение. Другим примером является фоновый процесс, собирающий информацию о системе. Не исключено, что пользователь захочет иметь возможность включать и выключать этот процесс.

Рассмотрение временного графика работы также может привести к решению о целесообразности свопинга. Например, процесс, который должен периодически активизироваться с большим интервалом времени между активизациями и долго простояивает, имеет смысл выгружать из основной памяти на то время, в течение которого он не используется. Примером такого процесса может быть программа, ведущая учет использования ресурсов или активности пользователей.

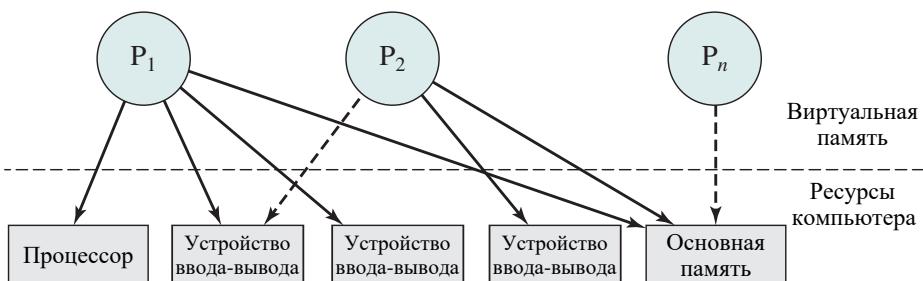
Наконец, родительский процесс может захотеть приостановить процесс, который он породил. Например, пусть процесс А породил процесс В, чтобы прочитать файл. Впоследствии при выполнении процесса В возникла ошибка чтения, и он сообщил об этом процессу А. Процесс А приостанавливает процесс В и пытается выяснить и устранить причину ошибки.

Во всех описанных выше случаях активизация приостановленного процесса происходит по запросу того агента, который перед этим вызвал временную остановку.

### 3.3. ОПИСАНИЕ ПРОЦЕССОВ

Операционная система управляет событиями, которые происходят в компьютерной системе. Она планирует и координирует выполнение процессов, выделяет им ресурсы и предоставляет по запросу системных и пользовательских программ основные сервисы. Мы можем представить себе операционную систему как некий механизм, управляющий тем, как процессы используют системные ресурсы.

Эта концепция проиллюстрирована на рис. 3.10. Пусть в многозадачной среде имеется несколько процессов ( $P_1, \dots, P_n$ ), которые уже созданы и загружены в виртуальную память. Каждому процессу для его функционирования нужен доступ к определенным системным ресурсам, в число которых входят процессор, устройства ввода-вывода и основная память. В ситуации, изображенной на рисунке, процесс  $P_1$  находится в состоянии выполнения, т.е. в основной памяти находится по крайней мере часть этого процесса. Кроме того, он осуществляет управление двумя устройствами ввода-вывода. Процесс  $P_2$  тоже находится в основной памяти, но он блокирован, ожидая, пока освободится устройство ввода-вывода, находящееся в распоряжении процесса  $P_1$ . Процесс  $P_n$  выгружен из основной памяти и, соответственно, приостановлен.

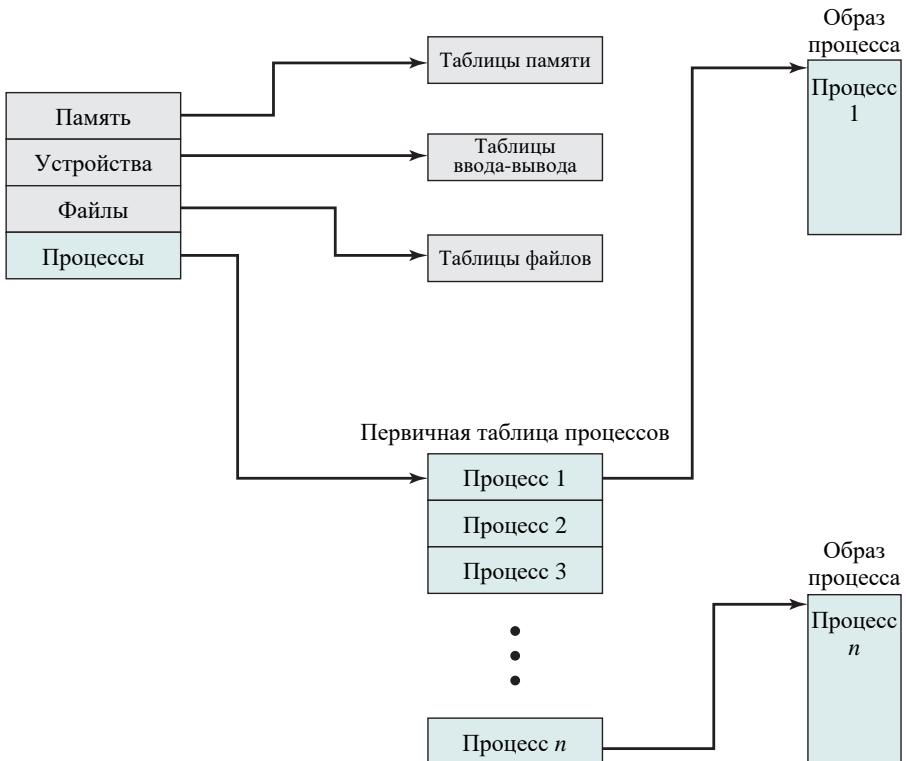


**Рис. 3.10.** Процессы и ресурсы в некоторый момент времени

Далее в этой главе мы подробно рассмотрим, как выглядит управление ресурсами со стороны операционной системы с точки зрения процессов. А пока что зададим себе более фундаментальный вопрос: какая информация нужна операционной системе, чтобы управлять процессами и выделяемыми для них ресурсами?

### Управляющие структуры операционной системы

Поскольку в задачи операционной системы входит управление процессами и ресурсами, она должна располагать информацией о текущем состоянии каждого процесса и ресурса. Универсальный подход к предоставлению такой информации прост: операционная система создает и поддерживает таблицы с информацией по каждому объекту управления. Общее представление об этом можно получить из рис. 3.11, на котором показаны четыре различных вида таблиц, поддерживающихся операционной системой: для памяти, устройств ввода-вывода, файлов и процессов. Хотя детали в разных операционных системах могут различаться, по сути, все операционные системы поддерживают информацию по этим четырем категориям.



**Рис. 3.11.** Общая структура управляющих таблиц операционной системы

**Таблицы памяти** (memory tables) используются для того, чтобы следить за основной (реальной) и вторичной (виртуальной) памятью. Некоторая часть основной памяти резервируется для операционной системы, оставшаяся же доступна для использования процессами. Процессы, которые находятся во вторичной памяти, используют некоторую разновидность виртуальной памяти либо простой механизм свопинга. Таблицы памяти должны включать такую информацию:

- объем основной памяти, отведенной процессу;
  - объем вторичной памяти, отведенной процессу;
  - все атрибуты защиты блоков основной или виртуальной памяти, как, например, указание, какой из процессов имеет доступ к той или иной совместно используемой области памяти;
  - вся информация, необходимая для управления виртуальной памятью.

Подробно эти информационные структуры, используемые для управления памятью, рассматриваются в части 3 книги.

**Таблицы ввода-вывода** (I/O tables) используются операционной системой для управления устройствами ввода-вывода и каналами компьютерной системы. В каждый момент времени устройство ввода-вывода может быть либо свободно, либо отдано в распоряжение какому-то определенному процессу. Если выполняется операция ввода-вывода, операционная система должна иметь информацию о ее состоянии и о том, какие адреса основной памяти задействованы в этой операции в качестве источника вывода или мес-

та, куда передаются данные при вводе. Управление вводом-выводом рассматривается в главе 11, “Управление вводом-выводом и планирование дисковых операций”.

Операционная система может также поддерживать **таблицы файлов** (file tables). В этих таблицах находится информация о существующих файлах, их расположении во вторичной памяти, текущем состоянии и других атрибутах. Большая часть этой информации, если не вся, может поддерживаться системой управления файлами. В этом случае операционная система мало знает (или совсем ничего не знает) о файлах. В операционных системах другого типа основная часть работы, связанной с управлением файлами, выполняется самой операционной системой. Эта тема обсуждается в главе 12, “Управление файлами”.

Наконец, операционная система должна поддерживать таблицы процессов, чтобы иметь возможность управлять ими. В оставшейся части данного раздела рассматриваются требования к **таблицам процессов** (process tables). Перед тем как продолжить рассмотрение, сделаем два замечания. Во-первых, хотя на рис. 3.11 и показаны четыре разных вида таблиц, ясно, что все они должны быть связаны между собой или иметь перекрестные ссылки. В конце концов, управление памятью, устройствами ввода-вывода и файлами осуществляется для того, чтобы могли выполняться процессы, поэтому в таблицах процессов должны быть явные или неявные ссылки на эти ресурсы. Например, доступ к файлам, информация о которых хранится в таблицах файлов, осуществляется через устройства ввода-вывода, и эти файлы или их части в определенные моменты времени будут находиться в основной или виртуальной памяти. Сами таблицы должны быть доступны для операционной системы, поэтому место для них выделяется системой управления памятью.

Во-вторых, какие сведения необходимы операционной системе для создания этих таблиц? Конечно же, у нее должна быть информация по основной конфигурации системы, в которую входят сведения об объеме основной памяти, количестве и виде устройств ввода-вывода, а также их идентификаторах и т.п. Таким образом, во время инициализации операционная система должна иметь доступ к определенным данным конфигурации, которые определяют основные параметры вычислительной среды. Эти данные могут создаваться как вне операционной системы с участием оператора, так и с помощью каких-то программ, определяющих конфигурацию вычислительной системы.

## Структуры управления процессами

Рассмотрим вопрос о том, какими сведениями должна располагать операционная система, чтобы управлять процессом. Во-первых, она должна знать, где находится этот процесс, а во-вторых, ей должны быть известны необходимые для управления атрибуты процесса (такие, как его идентификатор и состояние).

### Местоположение процесса

Прежде чем перейти к рассмотрению вопроса о размещении процесса или о его атрибутах, зададим себе еще более фундаментальный вопрос: в чем заключаются физические проявления процесса? Как минимум в процесс входит программа или набор программ, которые нужно выполнить. С этими программами связан набор ячеек памяти, в которых хранятся локальные и глобальные переменные и константы. Таким образом, процессу должен быть выделен такой объем памяти, в котором поместились бы программа и данные, принадлежащие процессу. Кроме того, при работе программы обычно используется стек (см. приложение П, “Управление процедурами”), с помощью которого реализуются вызовы процедур и передача параметров. Наконец, с каждым процессом

связано несколько атрибутов, которые используются операционной системой для управления этим процессом. Обычно такой набор атрибутов называется **управляющим блоком процесса** (process control block).<sup>8</sup> Коллекция, в которую входят программа, данные, стек и атрибуты, называется **образом процесса** (process image) (табл. 3.4).

**ТАБЛИЦА 3.4. ТИПИЧНЫЕ ЭЛЕМЕНТЫ ОБРАЗА ПРОЦЕССА**

<b>Данные пользователя</b>	Допускающая изменения часть пользовательского адресного пространства. Сюда могут входить данные программы, пользовательский стек и модифицируемый код
<b>Пользовательская программа</b>	Программа, которую нужно выполнить
<b>Системный стек</b>	С каждым процессом связан один или несколько системных стеков. Стек используется для хранения параметров, адресов вызова процедур и системных служб
<b>Управляющий блок процесса</b>	Данные, необходимые операционной системе для управления процессом (см. табл. 3.5)

Местоположение образа процесса зависит от используемой схемы управления памятью (в простейшем случае образ процесса имеет вид непрерывного блока памяти, который расположен во вторичной памяти, обычно на диске). Чтобы операционная система могла управлять процессом, по крайней мере небольшая часть его образа должна находиться в основной памяти. Чтобы можно было запустить процесс, его образ необходимо полностью загрузить в основную (или в виртуальную) память. Таким образом, операционной системе нужно знать местонахождение каждого процесса на диске, а для тех процессов, которые загружены в основную память, — их местонахождение в основной памяти. В главе 2, “Обзор операционных систем”, мы рассматривали несколько более сложную модификацию этой схемы, использующуюся в системах CTSS, в которых при свопинге процесс может выгружаться из основной памяти только частично. При этом операционная система должна следить за тем, какая часть образов каждого из процессов осталась в основной памяти.

Современные операционные системы предполагают наличие аппаратных систем подкачки, разрешающих для поддержки частично резидентных процессов использовать не последовательные блоки физической памяти<sup>9</sup>. В любой момент времени часть образа процесса может находиться в основной памяти, в то время как остальная его часть остается во вторичной памяти.<sup>10</sup> Поэтому таблицы процессов, поддерживаемые операцион-

<sup>8</sup> Часто используются другие названия этой структуры данных — “блок управления заданием”, “дескриптор процесса”, “дескриптор задания”.

<sup>9</sup> Краткий обзор концепций страничной организации памяти, сегментов и виртуальной памяти представлен в посвященном управлению памятью подразделе раздела 2.3.

<sup>10</sup> В этом кратком обсуждении опущены некоторые детали. Например, в операционной системе с использованием виртуальной памяти все образы активных процессов находятся во вторичной памяти. Когда в основную память загружается часть образа, она не переносится туда, а копируется. Таким образом, во вторичной памяти остаются копии всех сегментов и/или всех страниц. Однако, если часть образа в основной памяти модифицируется, копия на диске становится устаревшей до тех пор, пока измененная часть основной памяти не будет скопирована обратно на диск.

ной системой, должны содержать сведения о местонахождении каждого сегмента и/или каждой страницы всех образов процессов.

Структура сведений о местоположении процессов, изображенная на рис. 3.11, организована следующим образом. Имеется первичная таблица процессов, в которой каждому процессу соответствует одна запись. Все записи должны содержать по крайней мере указатель на образ процесса. Если образ процесса состоит из нескольких блоков, то эта информация либо содержится непосредственно в соответствующей записи первичной таблицы, либо может быть получена при помощи ссылок на записи в таблицах памяти. Конечно же, это описание носит общий характер; в каждой операционной системе используется собственный метод организации информации о расположении процессов.

## Атрибуты процессов

Сложная многозадачная система должна располагать обширными сведениями о каждом процессе. Как было сказано ранее, можно считать, что эта информация находится в управляющем блоке процесса. Различные системы организуют эту информацию по-разному; в конце настоящей и следующей глав приводятся несколько примеров такой организации. А пока что рассмотрим вопрос о том, информация какого типа может понадобиться операционной системе, не останавливаясь на схеме организации этой информации.

В табл. 3.5 перечислены типичные виды информации, требующейся операционной системе для каждого процесса. Возможно, читателя несколько удивит объем требуемой информации, однако этот список будет выглядеть гораздо убедительнее после более полного знакомства с функциями операционной системы.

Информацию, которая находится в управляющем блоке процесса, можно разбить на три основные категории:

1. информация по идентификации процесса;
2. информация по состоянию процесса;
3. информация, используемая при управлении процессом.

Что касается **идентификации процесса** (process identification), то почти во всех операционных системах каждому процессу присваивается числовой идентификатор, который может быть просто индексом в первичной таблице процессов (см. рис. 3.11); в противном случае должно иметься некоторое отображение, позволяющее операционной системе найти по идентификатору процесса соответствующие ему таблицы. Идентификаторы могут быть полезны в разных ситуациях. В частности, они используются для реализации перекрестных ссылок на таблицы процессов из других таблиц, находящихся под управлением операционной системы. Например, таблицы памяти могут предоставлять информацию об основной памяти с указанием всех областей, выделенных каждому из процессов, называемому посредством его идентификатора. Аналогичные ссылки могут быть и в таблицах ввода-вывода или таблицах файлов. Если процессы обмениваются между собой информацией, их идентификаторы указывают операционной системе участников такого обмена. При создании нового процесса идентификаторы указывают родительский и дочерние процессы.

**ТАБЛИЦА 3.5. ТИПИЧНЫЕ ЭЛЕМЕНТЫ УПРАВЛЯЮЩЕГО БЛОКА ПРОЦЕССА**

---

**Идентификация процессов**

---

**Идентификаторы**

Числовые идентификаторы, которые могут храниться в управляемом блоке процесса.

- Идентификатор данного процесса
  - Идентификатор родительского процесса
  - Идентификатор пользователя
- 

**Информация о состоянии процессора**

---

**Регистры, доступные пользователю**

Доступный пользователю регистр — это регистр, к которому можно обратиться с помощью машинных команд, выполняющихся процессором. Обычно имеется от 8 до 32 таких регистров, хотя в некоторых реализациях RISC (процессоров с ограниченным набором команд) встречается свыше 100 регистров.

**Управляющие регистры и регистры состояния**

В процессоре имеется несколько разновидностей регистров, которые используются для управления работой процессора. К ним относятся следующие.

- **Счетчик команд.** В этом регистре хранится адрес очередной извлекаемой команды.
- **Коды условия.** Отражают результат выполнения последней арифметической или логической операции (например, знак, равенство нулю, наличие переноса, равенство, переполнение).
- **Информация о состоянии.** Сюда входят флаги разрешения прерываний и информация о режиме выполнения.

**Указатели стеков**

С каждым процессом связан один или несколько системных стеков. В стеке хранятся параметры и адреса вызовов процедур и системных служб. Указатель стека указывает на его вершину.

---

**Управляющая информация процесса**

---

**Информация по планированию и состоянию**

Эта информация нужна операционной системе для выполнения планирования и обычно включает следующее.

- **Состояние процесса.** Определяет готовность планируемого процесса к выполнению (т.е. выполняющийся, готовый к выполнению, ожидающий какого-то события или приостановленный).
  - **Приоритет.** Одно или несколько полей могут использоваться для описания приоритета процесса. В некоторых системах могут требоваться несколько значений (таких, как приоритет по умолчанию, текущий приоритет, максимально возможный приоритет).
  - **Информация, связанная с планированием.** Эта информация зависит от используемого алгоритма планирования. В качестве примера можно привести такие показатели, как время ожидания или время, в течение которого процесс выполнялся при последнем запуске.
  - **Информация о событиях.** Идентификация события, наступление которого позволит продолжить выполнение процесса, находящегося в состоянии ожидания.
-

## Структурирование данных

Процесс может быть связан с другими процессами посредством очереди, кольца или какой-либо другой структуры. Например, все процессы в состоянии ожидания, имеющие один и тот же приоритет, могут находиться в одной очереди. Процессы могут иметь родственные отношения (быть родительскими или дочерними по отношению друг к другу). Для поддержания этих структур управляющий блок процесса может содержать указатели на другие процессы.

## Обмен информацией между процессами

Различные флаги, сигналы и сообщения могут иметь отношение к обмену информацией между двумя независимыми процессами. Некоторая часть этой информации, или вся она, может храниться в управляющем блоке процесса.

## Привилегии процессов

Процессы имеют привилегии, которые могут выражаться в предоставлении права доступа к определенной области памяти или возможности выполнять определенные виды команд. Кроме того, привилегии могут определять возможность использования различных системных утилит и служб.

## Управление памятью

Этот раздел может содержать указатели на таблицы сегментов и/или страниц, в которых описывается распределение процесса в виртуальной памяти.

## Владение ресурсами и их использование

Здесь могут быть указаны ресурсы, которыми управляет процесс (например, перечень открытых файлов). Кроме того, в данный раздел могут быть включены сведения по истории использования процессора и других ресурсов; эта информация может потребоваться при планировании.

Кроме того, процессу может быть присвоен идентификатор пользователя, который указывает, кто из пользователей отвечает за данное задание.

**Информация о состоянии процессора** (processor state information) состоит из содержащего его регистров. Во время выполнения процесса эта информация, конечно же, находится в регистрах. Прерывая процесс, всю содержащуюся в регистрах информацию необходимо сохранить, чтобы восстановить ее при возобновлении выполнения этого процесса. Характер и количество участвующих в сохранении регистров зависят от устройства процессора. Обычно в набор регистров входят регистры, доступные пользователю, управляющие регистры и регистры состояния, а также указатели стеков. Все они описаны в главе 1, “Обзор компьютерной системы”.

Следует заметить, что в процессорах любого вида имеется регистр или набор регистров, известных под названием **слово состояния программы** (program status word — PSW), в которых содержится информация о состоянии и кодах условий. Хорошим примером слова состояния процессора является регистр EFLAGS (показан на рис. 3.12 и описан в табл. 3.6), имеющийся в процессорах Intel x86. Эту структуру используют все операционные системы (включая UNIX и Windows), работающие на процессорах x86.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	I D	V I P F	V I C M	A V F	R 0	N T	I O P L	O F F	D F F	I T F	T S F	S Z F	0 0	A F 0	P F 1	C F							

X ID = Флаг идентификации  
X VIP = Флаг ожидания виртуального прерывания  
X VIF = Флаг виртуального прерывания  
X AC = Флаг проверки выравнивания  
X VM = Флаг режима виртуального 8086  
X RF = Флаг возобновления  
X NT = Флаг вложенной задачи  
X IOPL = Уровень привилегий ввода-вывода  
S OF = Флаг переполнения

C DF = Флаг направления  
X IF = Флаг разрешения прерываний  
X TF = Флаг ловушки  
S SF = Флаг знака  
S ZF = Флаг нуля  
S AF = Флаг вспомогательного переноса  
S PF = Флаг четности  
S CF = Флаг переноса

S — флаг состояния  
C — управляющий флаг  
X — системный флаг  
Заштрихованные биты зарезервированы

Рис. 3.12. Регистр EFLAGS процессора x86

ТАБЛИЦА 3.6. БИТЫ РЕГИСТРА EFLAGS ПРОЦЕССОРА x86

**Флаги состояний****Флаг коррекции, или вспомогательного переноса (Auxiliary carry flag — AF)**

Если данный флаг установлен, то это означает, что произошел перенос или заем единицы из одного полубайта в другой при выполнении операций с 8-битными арифметическими или логическими объектами с использованием регистра AL

**Флаг переноса (Carry flag — CF)**

Используется для индикации переноса единицы в старший разряд или займа единицы из этого разряда при арифметических операциях. Его содержимое также изменяется при некоторых операциях циклического сдвига

**Флаг переполнения (Overflow flag — OF)**

Указывает на переполнение результатов при сложении или вычитании

**Флаг четности (Parity flag — PF)**

Четность результата арифметической или логической операции. 1 указывает, что результат четный, а 0 — что результат нечетный

**Флаг знака (Sign flag — SF)**

Указывает знак результата арифметической или логической операции

**Флаг нуля (Zero flag — ZF)**

Указывает, равен ли нулю результат арифметической или логической операции

**Окончание табл. 3.6****Управляющий флаг****Флаг направления (Direction flag — DF)**

Задает порядок изменения (увеличение или уменьшение) содержимого 16-битовых полурегистров SI и DI (для работы в 16-битовом режиме) или 32-битовых регистров ESI и EDI (для работы в 32-битовом режиме), использующихся в командах обработки строк

**Системные флаги (прикладные программы не должны их изменять)****Флаг проверки выравнивания (Alignment check — AC)**

Флаг устанавливается при адресации одинарного или двойного слова, не выровненного на границу слова или двойного слова

**Флаг идентификации (Identification flag — IF)**

Если процессор в состоянии устанавливать и сбрасывать этот флаг, то данный процессор поддерживает команду CPUID. При выполнении данной команды выдается информация об изготовителе, серии и модели процессора

**Флаг возобновления (Resume flag — RF)**

Дает возможность программисту отменить отладочные исключения, так что команда после отладочного исключения может быть перезапущена без генерации другого исключения

**Уровень привилегий ввода-вывода (I/O privilege level — IOPL)**

Определяет уровень привилегий ввода-вывода выполняющегося процесса. При установленном флаге (низкий уровень привилегий) при каждом доступе к устройствам ввода-вывода генерируется исключение

**Флаг разрешения прерываний (Interrupt enable flag — IF)**

Если этот флаг установлен, процессор реагирует на внешние прерывания

**Флаг ловушки (Trap flag — TF)**

Если флаг установлен, то после выполнения каждой команды генерируется прерывание. Этот режим используется для отладки

**Флаг вложенной задачи (Nested task flag — NT)**

Указывает на то, что текущее задание вложено в другое задание при работе в защищенном режиме

**Флаг режима виртуального 8086 (Virtual 8086 mode — VM)**

Позволяет программисту включать и выключать режим виртуального 8086

**Флаг ожидания виртуального прерывания (Virtual interrupt pending — VIP)**

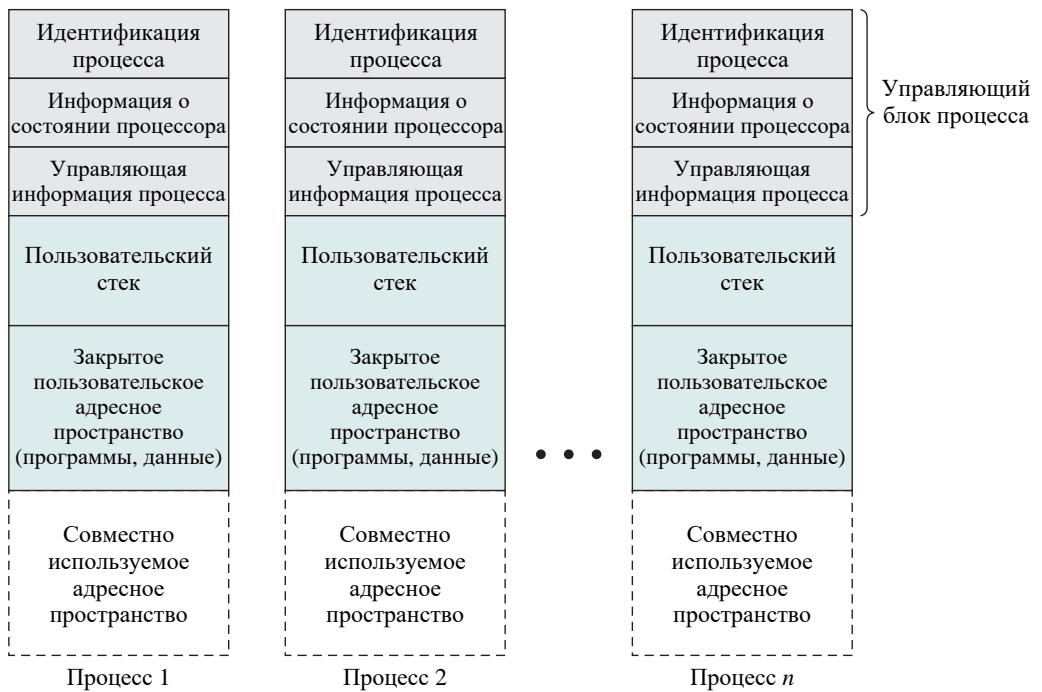
Используется в режиме виртуального 8086, чтобы указать, что одно или несколько прерываний ожидают обслуживания

**Флаг виртуального прерывания (Virtual interrupt flag — VIF)**

В режиме виртуального 8086 используется вместо флага IF

Третью основную категорию информации в управляющем блоке процесса можно назвать, за неимением лучшего термина, **управляющей информацией процесса** (process control information). Это дополнительная информация, необходимая операционной системе для того, чтобы управлять различными активными процессами и координировать их. В последней части табл. 3.5 приведена эта информация. Когда мы рассмотрим детали функционирования операционных систем в последующих главах, нам станет понятна необходимость различных пунктов этого списка.

На рис. 3.13 предложена структура образов процессов в виртуальной памяти. Каждый образ процесса состоит из управляющего блока процесса, стека пользователя, закрытого адресного пространства процесса и всех других адресных пространств, которые данный процесс использует совместно с другими процессами. На рисунке каждый образ процесса изображен в виде области непрерывных адресов, но в реальной реализации это может быть не так; размещение образа процесса в памяти зависит от схемы управления памятью и от способа организации управляющих структур в операционной системе.

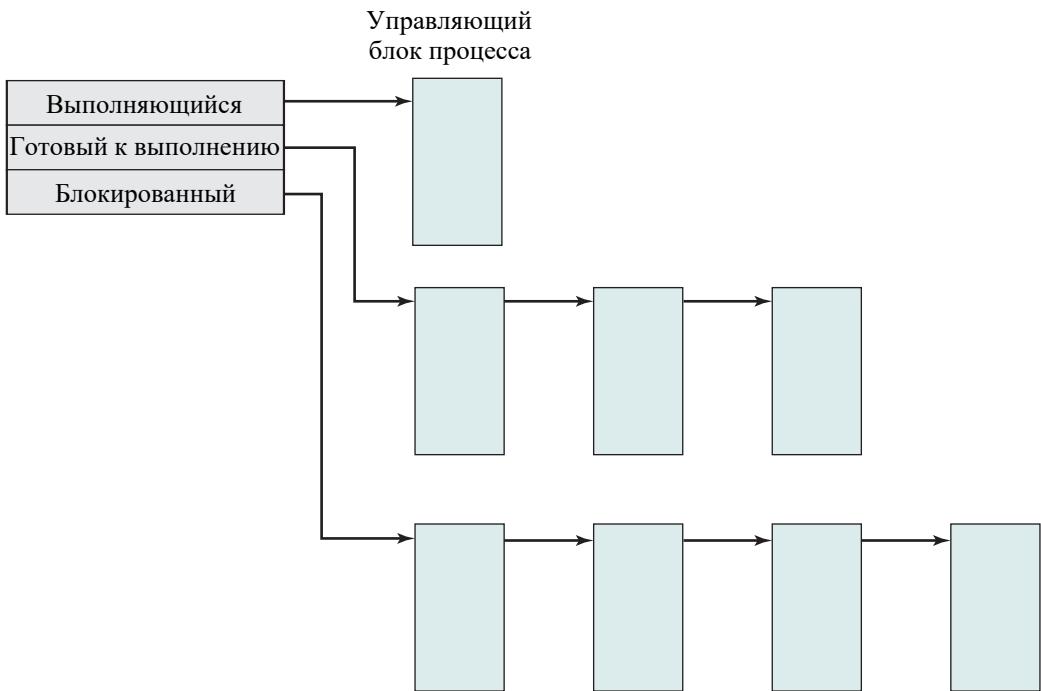


**Рис. 3.13.** Пользовательские процессы в виртуальной памяти

Как видно из табл. 3.5, управляющий блок процесса может содержать в себе структурную информацию, в которую входят указатели, позволяющие связывать между собой различные управляющие блоки процессов. Таким образом, описанные в предыдущем разделе очереди могут быть реализованы в виде связанных списков, элементами которых являются управляющие блоки процессов. Например, схема очередности, показанная на рис. 3.8, а, может быть реализована в соответствии со схемой, изображенной на рис. 3.14.

## *Роль управляющего блока процесса*

Управляющий блок процесса — это самая важная структура данных из всех имеющихся в операционной системе. В управляющем блоке каждого процесса входит вся необходимая операционной системе информация о нем. Информация в этих блоках считывается и/или модифицируется почти каждым модулем операционной системы, включая те, которые связаны с планированием, распределением ресурсов, обработкой прерываний, а также осуществлением контроля и анализа. Можно сказать, что состояние операционной системы задается совокупностью управляющих блоков процессов.



**Рис. 3.14.** Структура списков процессов

Это приводит к важной проблеме проектирования операционных систем. В состав операционной системы входит ряд подпрограмм, которым нужен доступ к управляющим блокам процессов. Предоставить прямой доступ к этим таблицам совсем не трудно — каждому процессу присваивается свой уникальный идентификатор, который может быть использован в качестве индекса в таблице указателей на управляющие блоки процессов. Трудность состоит не в том, чтобы предоставить доступ, а в том, чтобы обеспечить защиту, и в связи с этим возникают две проблемы.

- Ошибка в какой-нибудь одной подпрограмме (например, в обработчике прерываний) может привести к повреждению управляющего блока процесса, в результате чего система потеряет возможность управлять данным процессом.
- Изменение структуры или семантики управляющего блока процесса может повлиять на ряд модулей операционной системы.

В качестве возможного способа решения этих проблем можно потребовать, чтобы выполнение всех действий с управляющими блоками осуществлялось операционной системой только через программу-обработчик, единственной задачей которой будет защита управляющего блока процесса и которая в единоличном порядке отвечает за чтение информации из этих блоков и запись информации в них. Целесообразность использования такой программы определяется тем, насколько она влияет на производительность системы, а также степенью надежности остального программного обеспечения системы.

## 3.4. УПРАВЛЕНИЕ ПРОЦЕССАМИ

### Режимы выполнения

Перед тем как продолжить обсуждение способов, с помощью которых операционная система управляет процессами, нужно разобраться, в чем состоит различие между режимами работы процессора при выполнении кода операционной системы и при выполнении кодов пользовательских программ. Большинство процессоров поддерживают по крайней мере два режима работы. Определенные команды выполняются только в более привилегированном режиме. К ним относятся операции считывания или внесения изменений в управляющие регистры (например, операции со словом состояния программы), команды ввода-вывода, а также команды, связанные с управлением памятью. Кроме того, доступ к некоторым областям памяти может быть получен только в более привилегированном режиме.

Режим с меньшими привилегиями часто называют **пользовательским режимом** (user mode), потому что обычно в этом режиме выполняются пользовательские программы. Режим с более высокими привилегиями называется **системным режимом** (system mode), **режимом управления** (control mode) или **режимом ядра** (kernel mode). В последнем названии упоминается ядро, т.е. часть операционной системы, которая выполняет важнейшие ее функции. В табл. 3.7 перечислены те из функций операционной системы, которые обычно возлагаются на ее ядро.

Таблица 3.7. Типичные функции ядра операционной системы

#### Управление процессами

- Создание и завершение процессов
- Планирование и диспетчеризация процессов
- Переключение процессов
- Синхронизация и поддержка обмена информацией между процессами
- Организация управляющих блоков процессов

#### Управление памятью

- Выделение адресного пространства процессам
- Свопинг
- Управление страницами и сегментами

#### Управление вводом-выводом

- Управление буферами
- Выделение процессам каналов и устройств ввода-вывода

#### Функции поддержки

- Обработка прерываний
- Учет использования ресурсов
- Текущий контроль системы

Нетрудно понять, зачем нужны два вышеуказанных режима. Необходимо защитить операционную систему и ее основные таблицы, такие как управляющие блоки процессов, от воздействия пользовательских программ. Программы, работающие в режиме ядра, обладают полным контролем над процессором и всеми его командами и регистрами, а также имеют доступ ко всем ячейкам памяти. Такой уровень привилегий пользовательским программам не нужен, поэтому, исходя из соображений безопасности, лучше сделать его недоступным для пользовательских программ.

В связи с этим возникают два вопроса: каким образом процессор может определить, в каком режиме должна выполняться данная программа, и как происходит переключение из одного режима в другой? Что касается ответа на первый вопрос, то в слове состояния программы имеется бит, в котором указывается режим выполнения. При некоторых событиях происходит изменение этого бита. Например, если пользователь вызывает службу операционной системы, устанавливается режим ядра (обычно это происходит в результате выполнения команд изменения режима), а по окончании работы системной службы режим вновь переключается на пользовательский. В качестве примера рассмотрим процессор Intel Itanium, который реализует 64-разрядную архитектуру IA-64. Этот процессор имеет регистр состояния процессора (processor status register — psr), который включает двухбитное поле текущего уровня привилегий (current privilege level — cpl). Уровень 0 является наиболее привилегированным уровнем, в то время как уровень 3 — наименее привилегированным. Большинство операционных систем, таких как Linux, используют уровень 0 для ядра и один из прочих уровней для пользовательского режима. Когда происходит прерывание, процессор сбрасывает большую часть битов в регистре состояния процессора, включая поле текущего уровня привилегий. Это автоматически устанавливает cpl равным 0. В конце процедуры обработки прерываний выполняется команда возврата из прерывания int (interrupt return). Она заставляет процессор восстановить регистр состояния процессора прерванной программы, что восстанавливает ее уровень привилегий. Подобная последовательность действий выполняется, когда приложение выполняет системный вызов. В Itanium приложение осуществляет системный вызов, помещая идентификатор системного вызова и его аргументы в предопределенные области, а затем выполняется специальная команда, которая имеет тот же эффект, что и выполнение прерывания на пользовательском уровне, и передает управление ядру.

## Создание процессов

В разделе 3.2 обсуждались события, влекущие за собой создание новых процессов. Ознакомившись со структурами данных, связанными с процессами, опишем вкратце этапы создания процессов.

Если операционная система по какой-либо причине (см. табл. 3.1) приняла решение создать процесс, она может сделать это следующим образом.

- Присвоить новому процессу уникальный идентификатор.** На этом этапе в первичную таблицу процессов вносится новая запись.
- Выделить пространство для процесса.** Сюда включаются все элементы образа процесса. Операционная система должна знать, сколько места нужно для пользовательского адресного пространства (для программ и данных) и для пользовательского стека. Эти значения могут назначаться по умолчанию исходя из типа процесса или могут быть установлены на основе запроса пользователя при создании

задания. Если процесс порождается другим процессом, родительский процесс может передать операционной системе необходимые величины, поместив их в запрос на создание процесса. Если вновь создающийся процесс будет использовать какую-то часть адресного пространства совместно с другими процессами, необходимо установить соответствующие связи. И наконец, нужно выделить место для управляющего блока процесса.

3. **Инициализировать управляющий блок процесса.** Информация о процессе содержит его идентификатор, а также некоторые другие идентификаторы, например идентификатор родительского процесса. Информация о состоянии процессора обычно инициализируется нулевыми значениями, за исключением счетчика команд (который содержит точку входа в программу) и указателей системного стека (задающие границы стека процесса). Инициализация управляющей информации происходит на основе значений, установленных по умолчанию, с учетом атрибутов из запроса на создание процесса. Например, состояние процесса обычно инициализируется значением “готов” или “готов и приостановлен”. Что касается приоритета, то по умолчанию он может быть установлен минимальным, если не поступил явный запрос на установку более высокого приоритета. Поначалу процесс может не владеть никакими ресурсами (типа устройств ввода-вывода или файлов), если не сделан явный запрос или если эти ресурсы не были переданы по наследству от родительского процесса.
4. **Установить необходимые связи.** Если операционная система, например, поддерживает очередь планируемых заданий в виде списка со связями, то новый процесс необходимо поместить в список готовых или готовых/приостановленных процессов.
5. **Создать или расширить другие структуры данных.** Операционная система может, например, поддерживать для каждого процесса файл с учетом используемых ресурсов, который позже будет использован для учета или оценки производительности системы.

## Переключение процессов

Казалось бы, функция переключения процессов проста и понятна. В какой-то момент времени выполняющийся процесс прерывается и операционная система переводит в состояние выполнения другой процесс, передавая ему управление. Однако здесь возникают некоторые вопросы, касающиеся архитектуры операционной системы. Во-первых, какие события должны приводить к переключению процессов? Во-вторых, как установить различие между переключением режимов работы и переключением процессов? И наконец, что нужно делать операционной системе с различными структурами данных, находящимися под ее управлением, чтобы переключить процесс?

### Когда нужно переключать процессы

Переключение процесса может произойти в любой момент, когда управление от выполняющегося процесса переходит к операционной системе. В табл. 3.8 перечислены возможные причины, по которым управление может перейти к операционной системе.

**Таблица 3.8. Механизмы прерывания выполнения процесса**

Механизм	Причина	Что используется
Прерывание	Внешняя по отношению к выполнению текущей команды	Отклик на внешнее асинхронное событие
Ловушка	Связана с выполнением текущей команды	Обработку ошибки или исключительной ситуации
Вызов супервизора	Явный запрос	Вызов функции операционной системы

Сначала рассмотрим системные прерывания. Фактически имеются системные прерывания двух видов. Первый вид — обычные прерывания, а второй — ловушки (trap). Прерывания первого вида происходят из-за событий определенного типа, не связанных с выполняющимся процессом и являющихся внешними по отношению к нему (таким событием может быть, например, завершение операции ввода-вывода). Ловушки связаны с ошибкой или исключительной ситуацией, возникшей в результате выполнения текущего процесса. В качестве примера можно привести попытку получения незаконного доступа к файлу. При обычном прерывании управление сначала передается обработчику прерываний, который осуществляет некоторые подготовительные действия, а затем — функции операционной системы, отвечающей за прерывания данного вида. Приведем конкретные примеры прерываний.

- **Прерывание таймера.** Операционная система определяет, что текущий процесс выполняется в течение максимально разрешенного промежутка времени, именуемого **квантом времени** (time slice). Квант времени представляет собой максимальное количество времени, которое процесс может выполнять без прерывания. Если это так, то данный процесс нужно переключить в состояние готовности и передать управление другому процессу.
- **Прерывание ввода-вывода.** Операционная система определяет, что именно произошло, и если это то событие, которого ожидают один или несколько процессов, операционная система переводит все соответствующие блокированные процессы в состояние готовности (соответственно, блокированные/приостановленные процессы она переводит в состояние готовых/приостановленных процессов). Затем операционная система должна принять решение: возобновить выполнение текущего процесса или передать управление готовому к выполнению процессу с более высоким приоритетом.
- **Ошибка отсутствия блока в памяти.** Допустим, что процессор должен обратиться к слову виртуальной памяти, которое в настоящий момент отсутствует в основной памяти. При этом операционная система должна загрузить в основную память блок (страницу или сегмент), в котором содержится адресованное слово. Сразу же после запроса на загрузку блока операционная система может передать управление другому процессу, а процесс, для продолжения выполнения которого нужно загрузить блок в основную память, переходит в блокированное состояние. После загрузки нужного блока этот процесс переходит в состояние готовности.

В случае **ловушки** (trap) операционная система определяет, является ли ошибка или исключительная ситуация фатальной. Если это так, то выполняющийся в данный момент процесс переходит в состояние завершающегося, а управление переходит к другому процессу. В противном случае действия операционной системы будут зависеть от характера произошедшей ошибки, а также от конструкции самой операционной системы. Может быть предпринята попытка выполнить восстановительные процедуры или просто поставить пользователя в известность о произошедшей ошибке. Операционная система при этом может как выполнить переключение процессов, так и продолжить выполнение текущего процесса.

Наконец, операционная система может быть активирована в результате **вызыва супервизора** (supervisor call), который исходит от выполняемой программы. Например, пусть в ходе работы пользовательского процесса нужно выполнить команду, для которой требуется операция ввода-вывода, например открытие файла. Такой вызов приведет к тому, что управление перейдет к процедуре, являющейся частью кода операционной системы. Вообще говоря, использование таких системных вызовов приводит к переходу процесса в блокированное состояние.

### Переключение режимов

В главе 1, “Обзор компьютерной системы”, нами рассматривалась целесообразность включения цикла прерывания в цикл команды. Напомним, что в ходе цикла прерывания процессор проверяет, не поступили ли какие-нибудь прерывания, на наличие которых указывает сигнал прерывания. Если никаких прерываний нет, процессор переходит к циклу выборки, извлекая из памяти очередную команду, входящую в текущую программу. Если же имеются необработанные прерывания, то процессор выполняет следующие действия.

1. Устанавливает в счетчике команд начальный адрес программы — обработчика прерываний.
2. Переключается из пользовательского режима в режим ядра, чтобы можно было выполнять привилегированные команды, содержащиеся в коде обработки прерывания.

После этого процессор переходит к циклу выборки и выбирает первую команду из программы обработки прерываний, обслуживающей данное прерывание. Обычно в этот момент контекст прерванного процесса сохраняется в управляющем блоке процесса прерванной программы.

Может возникнуть вопрос: что входит в состав сохраняемого контекста? Этот контекст должен содержать информацию, которая может быть изменена в процессе работы обработчика прерываний и которая необходима для возобновления выполнения прерванной программы. Таким образом, должна быть сохранена часть управляющего блока процесса с информацией о состоянии процессора. В нее входит содержимое счетчика команд и других регистров процессора, а также информация о стеках.

Нужно ли сделать что-нибудь еще? Это зависит от дальнейших событий. Обычно обработчик прерываний является короткой программой, которая выполняет несколько базовых действий, связанных с прерыванием. Например, эта программа сбрасывает флаг состояния, сигнализирующий о наличии прерываний. Она может послать подтверждение о получении прерывания тому элементу аппаратного обеспечения, который генерировал это прерывание (например, контроллеру ввода-вывода). Кроме того, эта программа

может выполнить некоторые служебные действия, связанные с событием, сгенерировавшим прерывание. Например, если прерывание связано с вводом-выводом, обработчик прерываний проверит, не произошли ли ошибки при его выполнении. Если произошла ошибка, обработчик может послать сигнал процессу, первоначально выдавшему запрос на операцию ввода-вывода. Если прерывание сгенерировано таймером, программа обработки прерываний передаст управление диспетчеру. Диспетчер выполнит переключение процессора на обработку другого процесса, если промежуток времени, отведенный на выполнение текущего процесса, уже истек.

А как быть с остальной информацией, содержащейся в управляющем блоке процесса? Если вслед за данным прерыванием последует переключение на другой процесс, то нужно выполнить соответствующие действия по переключению. Однако в большинстве операционных систем прерывание не обязательно влечет за собой переключение процессов. По окончании работы обработчика прерываний возможно возобновление выполнения того процесса, который выполнялся и до прерывания. В некоторых случаях во время прерывания достаточно сохранить только информацию о состоянии процессора. После того как управление вновь возвратится к выполнявшейся перед прерыванием программе, нужно будет лишь восстановить эту информацию. Обычно функция сохранения и восстановления состояния процессора возлагается на аппаратное обеспечение.

### Изменение состояния процесса

Понятно, что переключение режима работы процессора и переключение процессов — это не одно и то же.<sup>11</sup> Переключение режима может происходить без изменения состояния процесса, выполняющегося в данное время. В этом случае сохранение контекста и его последующее восстановление не приведет к большим накладным расходам. Однако если выполняющийся в текущий момент времени процесс нужно перевести в другое состояние (состояние готовности, блокированное и т.д.), то операционная система должна произвести в своей среде определенные изменения. В случае переключения процессов должны быть выполнены следующие действия.

1. Сохранение контекста процессора, включая содержимое счетчика команд и других регистров.
2. Обновление управляющего блока выполняющегося в данное время процесса. Сюда входит изменение состояния процесса на одно из следующих: готовый, блокированный, готовый/приостановленный или завершающийся. Кроме того, должно быть обновлено содержимое других полей с указанием причины переключения процесса из состояния выполнения, а также с сохранением информации по учету используемых ресурсов.
3. Помещение управляющего блока данного процесса в соответствующую очередь (очередь готовых к выполнению процессов; процессов, блокированных событием  $i$ ; очередь готовых/приостановленных процессов).

<sup>11</sup> В литературе по операционным системам часто встречается термин *переключение контекста* (context switch). К сожалению, хотя в большинстве изданий этот термин используется для обозначения действий, которые в нашей книге называются переключением процессов, в других источниках он используется для обозначения переключения режима работы процессора или даже переключения потоков (о котором пойдет речь в следующей главе). Поэтому термин *переключение контекста* в данной книге во избежание его неоднозначного понимания не используется.

4. Выбор следующего процесса для выполнения; эта тема исследуется в части IV книги.
5. Обновление управляющего блока выбранного процесса. Для этого процесса нужно установить состояние выполнения.
6. Обновление структур данных по управлению памятью. Понадобится ли эта информация, зависит от того, каким образом выполняется преобразование адресов (подробно эта тема рассматривается в части III книги).
7. Восстановление состояния процессора, в котором он находился, когда выбранный процесс был последний раз переключен из состояния выполнения. Это происходит путем загрузки содержимого программного счетчика и других регистров процессора.

Таким образом, переключение процесса, включающее переключение его состояния, требует значительно больших усилий по сравнению с переключением режима работы процессора.

## 3.5. ВЫПОЛНЕНИЕ КОДА ОПЕРАЦИОННОЙ СИСТЕМЫ

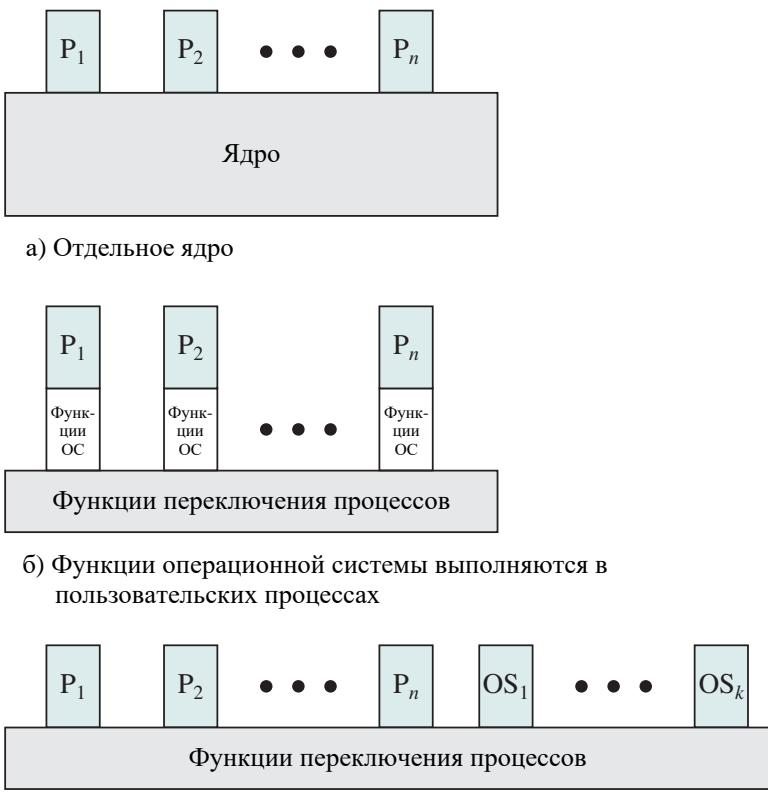
В главе 2, “Обзор операционных систем”, были отмечены два занимательных факта, касающиеся операционных систем.

- Операционная система работает точно так же, как и обычная программа, т.е. тоже является программой, которая выполняется процессором.
- Операционная система часто передает управление другим программам; возврат управления операционной системе зависит от процессора.

Если операционная система представляет собой всего лишь обычный набор программ и если она выполняется процессором точно так же, как и любая другая программа, то является ли операционная система процессом? Если это так, то как осуществляется управление этим процессом? Размышления над этими интересными вопросами стали причиной появления различных подходов к разработке операционных систем. На рис. 3.15 проиллюстрированы подходы, реализованные в различных операционных системах.

### Ядро вне процессов

Один из традиционных подходов, которые применялись во многих ранних операционных системах, состоит в том, чтобы выполнять ядро операционной системы вне всяких процессов (рис. 3.15, а). При таком подходе прерывание выполняющегося в данное время процесса или вызов управляющей программы приводит к сохранению контекста данного процесса и передаче управления ядру. Операционная система имеет собственную область памяти и собственный системный стек, который используется для управления вызовами процедур и возвратами из них. Операционная система может выполнить все необходимые функции и восстановить контекст прерванного процесса, что приведет к продолжению выполнения этого процесса. В качестве альтернативы операционная система может завершить сохранение контекста данного процесса и перейти к планированию и диспетчеризации другого процесса. Случится это или нет — зависит от того, что именно послужило причиной прерывания, и от ряда других обстоятельств.



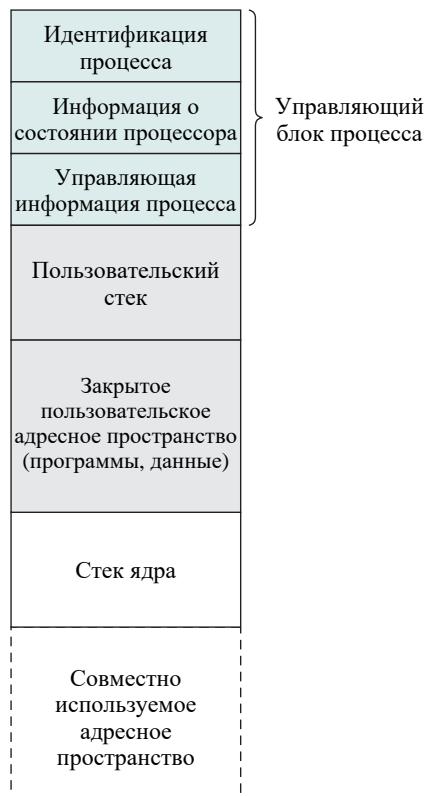
**Рис. 3.15.** Взаимосвязь между операционной системой и пользовательскими процессами

В любом случае основным моментом такой схемы является то, что концепция процесса рассматривается в ней лишь применительно к пользовательским программам. Код операционной системы выполняется как некий отдельный объект, работающий в привилегированном режиме.

## Выполнение в составе пользовательских процессов

На небольших машинах (персональных компьютерах, рабочих станциях) часто применяется альтернативный подход, при котором почти все программы операционной системы выполняются в контексте пользовательского процесса. Разработчики таких систем придерживаются той точки зрения, что операционная система — это в основном набор процедур, которые вызываются для выполнения различных функций пользовательского процесса. Этот подход проиллюстрирован на рис. 3.15, б. В любой момент операционная система управляет и образами процессов. Каждый образ включает в себя не только блоки, изображенные на рис. 3.13, но и области кода, данных и стека программ ядра.

На рис. 3.16 показана типичная структура образа процесса, принятая в такой стратегии. Для управления вызовом системных процедур, работающих в режиме ядра, и возврата из них используется отдельный стек ядра. Код и данные операционной системы находятся в совместно используемом адресном пространстве и доступны для использования всеми пользовательскими процессами.



**Рис. 3.16.** Образ процесса: код операционной системы выполняется в пользовательском пространстве

При прерывании, системном прерывании или вызове управляющей программы процессор переходит в режим ядра, а управление передается операционной системе. Чтобы это произошло, сохраняется контекст процесса и выполняется переключение режима с передачей управления процедуре операционной системы. Однако выполнение текущего пользовательского процесса продолжается. Таким образом, переключения процесса не происходит, переключается только режим работы процессора в рамках одного и того же процесса.

Если операционная система по завершении своей задачи придет к заключению, что следует продолжить текущий процесс, то с переключением режима процессора в предыдущее состояние возобновляется выполнение прерванной программы в рамках текущего процесса. Одно из основных преимуществ такого подхода состоит в следующем: если пользовательская программа прервась, чтобы выполнить некоторую процедуру

операционной системы, а затем возобновила свою работу, нам удается избежать двух излишних переключений процессов. Если же операционная система определит, что нужно переключить процесс, а не продолжать выполнение предыдущей программы, то управление переходит к процедуре, выполняющей переключение процессов. В зависимости от архитектуры операционной системы эта процедура может выполняться либо в составе текущего процесса, либо в составе некоторого другого процесса. В любом случае в какой-то момент текущий процесс нужно будет переключить, сняв его с выполнения, а в состояние выполнения перевести другой процесс. С точки зрения логики эту фазу удобнее всего рассматривать как нечто, происходящее вне всех процессов.

В некотором отношении такая точка зрения на операционную систему является довольно интересной. Выполняющийся процесс в определенный момент времени сам сохраняет информацию о своем состоянии, выбирает для выполнения другой процесс, находящийся в состоянии готовности, и передает ему управление. Причина того, что в такой ситуации не наступает хаос и произвол, заключается в том, что критичный код представляет собой не код пользовательской программы, а совместно используемый код операционной системы, выполняющийся в контексте процесса. В силу концепции пользовательского режима и режима ядра пользователь не может вмешиваться в работу системных процедур, хотя они и выполняются в среде пользовательского процесса. Это еще раз напоминает нам о различиях концепций процесса и программы и о том, что между ними нельзя ставить знак равенства. В ходе процесса могут выполняться и пользовательские программы, и программы операционной системы; с другой стороны, программы операционной системы, выполняемые в разных пользовательских процессах, являются идентичными.

## Операционная система на основе процессов

Еще одним вариантом построения операционной системы, проиллюстрированным на рис. 3.15, в, является ее реализация в виде набора системных процессов. Так же, как и при других подходах, программы, входящие в ядро, выполняются в режиме ядра, однако в этом случае основные функции ядра организованы как отдельные процессы. Здесь также возможно наличие небольшого кода, который является внешним по отношению ко всем процессам и осуществляет их переключение.

Такой подход обладает рядом преимуществ. Его применение дисциплинирует программистов и способствует разработке модульных операционных систем с минимальными, ясными межмодульными интерфейсами. Кроме того, некоторые второстепенные функции операционных систем удобно реализовывать в виде отдельных процессов. Например, ранее упоминалась программа-монитор, которая ведет запись интенсивности использования различных ресурсов (процессора, памяти, каналов) и скорости выполнения процессов в системе. Поскольку эта программа не обслуживает какой-то конкретный активный процесс, она может вызываться только операционной системой. В качестве процесса эта программа может выполняться с определенным приоритетом и чередоваться с другими программами под управлением диспетчера. И наконец, реализация операционной системы в виде набора процессов полезна в многопроцессорных и многокомпьютерных системах, в которых отдельные службы операционной системы могут быть переданы для выполнения специально предназначенным процессорам, что позволит повысить общую производительность системы.

## 3.6. УПРАВЛЕНИЕ ПРОЦЕССАМИ В ОПЕРАЦИОННОЙ СИСТЕМЕ UNIX SVR4

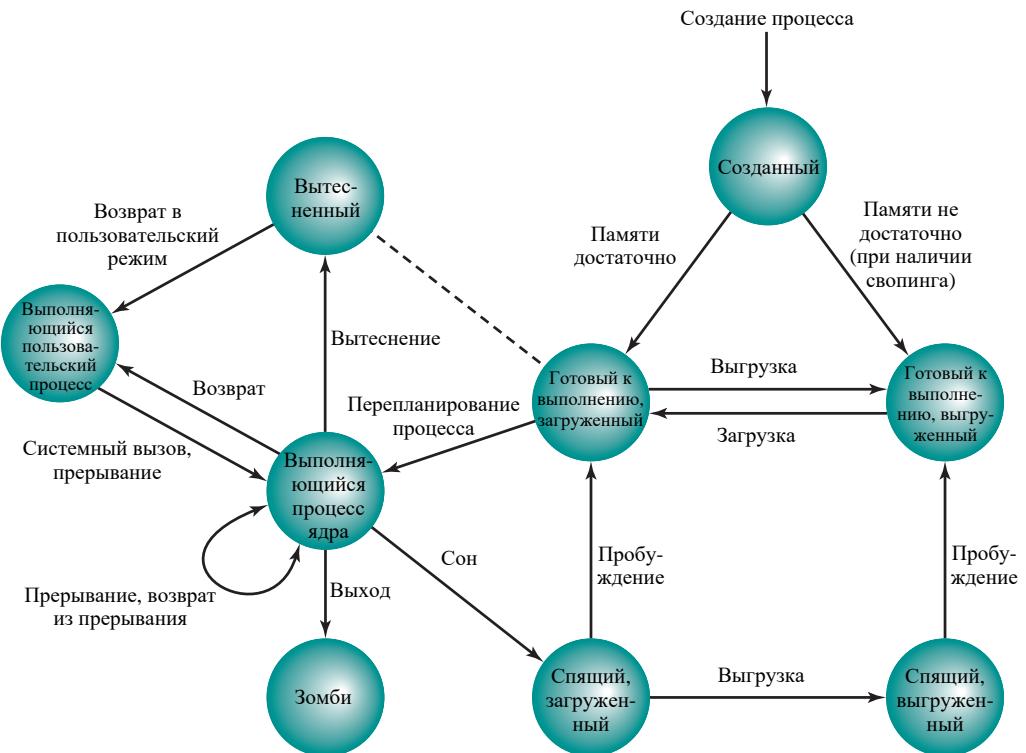
Операционная система UNIX System V использует простую, но обладающую широкими возможностями организацию процессов, широко доступную для пользователей. UNIX работает в соответствии с моделью, изображенной на рис. 3.15, б, согласно которой большинство программ операционной системы выполняется в среде пользовательских процессов. В операционной системе UNIX используются две категории процессов: системные и пользовательские. Системные процессы выполняют код операционной системы в режиме ядра, осуществляя различные административные функции, такие как выделение памяти или свопинг процессов. Пользовательские процессы в пользовательском режиме выполняют код пользовательских программ и утилит, а в режиме ядра — команды, принадлежащие ядру. Пользовательский процесс переключается в режим ядра при вызове системной функции, генерации исключения или при обработке прерывания.

### СОСТОЯНИЯ ПРОЦЕССОВ

Всего в операционной системе UNIX SVR4 распознается девять состояний процессов, перечисленных в табл. 3.9; соответствующая диаграмма переходов состояний показана на рис. 3.17 (в ее основе — рисунок из [14]). Этот рисунок похож на рис. 3.9, б; нужно только принять во внимание, что два спящих состояния в системе UNIX соответствуют двум блокированным состояниям. Кратко перечислим основные различия между диаграммами.

**Таблица 3.9. Состояния процессов в UNIX**

<b>Выполняющийся пользовательский</b>	Выполняющийся в пользовательском режиме
<b>Выполняющийся ядра</b>	Выполняющийся в режиме ядра
<b>Готовый к выполнению, загруженный</b>	Готов к выполнению, как только ядро решит передать ему управление
<b>Спящий, загруженный</b>	Не может выполняться, пока не произойдет некоторое событие; процесс находится в основной памяти (блокированное состояние)
<b>Готовый к выполнению, выгруженный</b>	Процесс готов к выполнению, но прежде чем ядро сможет спланировать его запуск, процесс свопинга должен загрузить этот процесс в основную память
<b>Спящий, выгруженный</b>	Процесс ожидает некоторого события; он выгружен из основной памяти (блокированное состояние)
<b>Вытесненный</b>	В момент переключения процессора из режима ядра в пользовательский режим ядро решает передать управление другому процессу
<b>Созданный</b>	Процесс только что создан и еще не готов к выполнению
<b>Зомби</b>	Самого процесса больше не существует, но записи о нем остались с тем, чтобы ими мог воспользоваться родительский процесс



**Рис. 3.17.** Диаграмма переходов между состояниями процессов в системе UNIX

- Для отражения того факта, что процесс может выполняться как в пользовательском режиме, так и в режиме ядра, в диаграмме имеется два состояния выполняющихся процессов.
  - Состояния процессов “Вытесненный” и “Готовый к выполнению загруженный” отличаются одно от другого. По сути, эти два состояния почти одинаковы (на что указывает соединяющая их пунктирная линия). Различие же делается, чтобы подчеркнуть, каким именно образом процесс может быть прерван в пользу другого. Если процесс выполняется в режиме ядра (в результате вызова диспетчера, прерывания по таймеру или прерывания по команде ввода-вывода), рано или поздно наступает момент, когда ядро завершает свою работу и готово возвратить управление пользовательской программе. В этот момент ядро может принять решение вытеснить текущий процесс и передать управление процессу с более высоким приоритетом, чем у выполнявшегося до этого. В таком случае текущий процесс переходит в состояние вытесненного, однако с точки зрения диспетчеризации эти процессы одинаковы. Процесс, прерванный в пользу другого, и процесс, находящийся в состоянии готового к выполнению загруженного, находятся в одной очереди.

Такое вытеснение процесса может произойти только в момент переключения режима выполнения из режима ядра в пользовательский режим. При выполнении процесса в режиме ядра он не может быть вытеснен, поэтому операционная система UNIX не

приспособлена для работы в режиме реального времени. Обсуждение требований к системам, выполняющим обработку запросов в реальном времени, можно найти в главе 10, “Многопроцессорное планирование и планирование реального времени”.

В UNIX есть два процесса, которых нет ни в каких других операционных системах. Процесс 0 — это специальный процесс, который создается при загрузке системы. В сущности, он предопределен как структура данных, которая загружается вместе с системой. Этот процесс является процессом свопинга. Кроме того, процесс 0 порождает процесс 1, который называется инициализирующим процессом (*init process*). Этот процесс является родительским по отношению ко всем остальным. Когда в систему входит новый интерактивный пользователь, именно процесс 1 создает для него новый процесс. Далее пользовательский процесс может создавать ветвящиеся дочерние процессы. Таким образом, каждое приложение может состоять из ряда взаимосвязанных процессов.

## Описание процессов

В операционной системе UNIX процессы представлены довольно сложными структурами данных, которые предоставляют операционной системе всю необходимую для управления и диспетчеризации процессов информацию. В табл. 3.10 приведены элементы образа процесса, разделенные на три части: контекст пользовательского уровня, контекст регистров и контекст системного уровня.

**Таблица 3.10. Образ процесса в UNIX**

<b>Контекст пользовательского уровня</b>	
<b>Текст процесса</b>	Выполняемые машинные команды программы
<b>Данные процесса</b>	Данные, доступные программе этого процесса
<b>Пользовательский стек</b>	Содержит аргументы, локальные переменные и указатели функций, выполняющихся в пользовательском режиме
<b>Совместно используемая память</b>	Область памяти, используемая совместно с другими процессами; применяется для обмена информацией между процессами
<b>Контекст регистров</b>	
<b>Счетчик команд</b>	Адрес очередной команды, которая будет выполняться; она может находиться как в пользовательском пространстве, так и в пространстве ядра
<b>Регистр состояния процессора</b>	Содержит данные о состоянии аппаратного обеспечения в момент передачи управления; содержимое и формат этих данных зависят от конкретного аппаратного обеспечения
<b>Указатель стека</b>	Указывает положение вершины стека ядра (или пользовательского стека, в зависимости от режима работы процессора)
<b>Регистры общего назначения</b>	Зависят от используемого аппаратного обеспечения

Окончание табл. 3.10

Контекст системного уровня	
<b>Запись таблицы процессов</b>	Определяет состояние процесса; эта информация всегда доступна операционной системе
<b>Пользовательская область</b>	Информация по управлению процессом, необходимая только в контексте данного процесса
<b>Таблица областей процесса</b>	Задает отображение виртуальных адресов в физические; содержит также поле полномочий, в котором указывается тип доступа, на который процесс имеет право: только для чтения, для чтения и записи или для чтения и выполнения
<b>Стек ядра</b>	Содержит кадр стека процедур ядра при работе процесса в режиме ядра

В контексте пользовательского уровня (user-level context) входят основные элементы пользовательских программ; он может генерироваться непосредственно из скомпилированных объектных файлов. Каждая пользовательская программа разделена на две части, одна из которых размещается в текстовой области, а другая — в области данных. Текстовая область предназначена только для чтения; в ней хранятся команды пользовательских программ.

Во время выполнения процессор использует пользовательский стек для вызовов и возвратов из процедур, а также для передачи параметров. Совместно используемая область памяти — это область данных, доступ к которой одновременно предоставляется различным процессам. Хотя в системе имеется только одна физическая копия совместно используемой области памяти, при использовании виртуальной памяти эта область находится в адресном пространстве каждого процесса, который ее использует. Когда процесс не выполняется, информация о состоянии процессора сохраняется в области **контекста регистров**.

В контексте системного уровня (system-level context) находится остальная информация, которая нужна операционной системе для управления процессом. Эта информация состоит из статической части фиксированного размера, который остается неизменным на протяжении всего времени жизни процесса, и динамической части, размер которой меняется.

Одним из компонентов статической части является запись таблицы процессов, которая фактически является частью таблицы процессов, поддерживаемой операционной системой, в которой каждому процессу соответствует одна запись. Запись таблицы процессов содержит информацию по управлению процессом, доступную ядру в любой момент времени. Таким образом, в системе виртуальной памяти все записи таблицы процессов постоянно остаются в основной памяти. В табл. 3.11 перечислены компоненты записи таблицы процессов. Пользовательская область содержит дополнительную управляющую информацию, которая нужна ядру при работе в контексте этого процесса; эта информация используется также при загрузке и выгрузке страниц процесса из основной памяти. В табл. 3.12 приведено содержимое этой таблицы.

**ТАБЛИЦА 3.11. ЭЛЕМЕНТ ТАБЛИЦЫ ПРОЦЕССОВ В СИСТЕМЕ UNIX**

<b>Состояние процесса</b>	Текущее состояние процесса
<b>Указатели</b>	Пользовательская область и область памяти процесса (текст, данные, стек)
<b>Размер процесса</b>	Дает возможность операционной системе определить, сколько памяти потребуется процессу
<b>Идентификаторы пользователя</b>	<b>Реальный идентификатор пользователя</b> (real user ID) указывает, кто из пользователей несет ответственность за выполняющийся процесс. <b>Фактический идентификатор пользователя</b> (effective user ID) может использоваться процессом для предоставления временных привилегий, связанных с определенной программой; на время выполнения этой программы в составе процесса последний использует фактический идентификатор пользователя
<b>Идентификаторы процесса</b>	Идентификатор данного и родительского процессов. Эти идентификаторы присваиваются процессу в состоянии создания
<b>Дескриптор событий</b>	Используется, когда процесс находится в спящем состоянии; с наступлением события процесс переходит в состояние готовности
<b>Приоритет</b>	Используется при планировании процессов
<b>Сигнал</b>	Перечисляет отправленные, но еще не обработанные сигналы
<b>Таймеры</b>	Включают время выполнения процесса, использование ресурсов ядром, а также пользовательские таймеры для отправки сигналов в определенное время
<b>Р-связь</b>	Указатель на следующий элемент в очереди готовых к выполнению процессов (используется, когда процесс находится в состоянии готовности)
<b>Состояние памяти</b>	Указывает, находится ли образ процесса в основной памяти или выгружен из нее. Если процесс загружен в память, в этом поле также указывается, можно ли его выгрузить или он временно блокирован в основной памяти

**ТАБЛИЦА 3.12. ПОЛЬЗОВАТЕЛЬСКАЯ ОБЛАСТЬ UNIX**

<b>Указатель таблицы процессов</b>	Указывает запись, соответствующую области пользователя
<b>Идентификаторы пользователя</b>	Реальный и фактический идентификаторы пользователя. Используются для определения пользовательских привилегий
<b>Таймеры</b>	Записывают время, затраченное на выполнение данного и дочерних процессов в пользовательском режиме и в режиме ядра
<b>Массив обработчиков сигналов</b>	Указывает, как будет реагировать процесс на каждый из пяти типов сигналов, заданных в системе (завершаться, игнорировать сигнал, выполнить заданную пользователем функцию)
<b>Управляющий терминал</b>	Указывает, с какого терминала был запущен процесс (если этот терминал существует)
<b>Поле ошибок</b>	Содержит записи об ошибках, произошедших во время системного вызова

Окончание табл. 3.12

<b>Возвращаемое значение</b>	Содержит результат выполнения системных вызовов
<b>Параметры ввода-вывода</b>	Задает объем передаваемых данных, адрес массива данных в пользовательском пространстве, а также смещения в файлах при вводе-выводе
<b>Файловые параметры</b>	Текущий и корневой каталоги описывают файловую систему процесса
<b>Таблица дескрипторов файлов пользователя</b>	Содержит записи об открытых файлах
<b>Границные поля</b>	Ограничивают размер процесса и размер файла, который он может записать
<b>Поля режимов доступа</b>	Установки режима доступа к создаваемым процессом файлам

Различия между записью таблицы процессов и пользовательской областью отражают тот факт, что ядро системы UNIX всегда выполняется в контексте какого-нибудь процесса. Большую часть времени ядро работает с контекстом текущего процесса, однако иногда ядру нужен доступ к информации и о других процессах. Например, когда ядро выполняет подготовку к алгоритму планирования для диспетчеризации другого процесса, ему необходим доступ к информации о других процессах. Доступ к информации в таблице процесса может быть получен, когда данный процесс не является текущим.

Третьей статической частью контекста системного уровня является таблица областей процесса, которая используется системой управления памятью. И наконец, стек ядра представляет собой динамическую часть контекста системного уровня. Этот стек используется при выполнении процесса в режиме ядра и содержит информацию, которую нужно сохранять и восстанавливать во время вызовов процедур и прерываний.

## Управление процессами

В операционной системе UNIX процессы создаются с помощью вызова системной функции ядра под названием `fork()`. При вызове этой функции процессом операционная система выполняет следующие действия [14].

1. Выделяет в таблице процессов место для нового процесса.
2. Назначает этому процессу уникальный идентификатор.
3. Создает копию образа родительского процесса, за исключением совместно используемых областей памяти.
4. Увеличивает показания счетчиков всех файлов, принадлежащих родительскому процессу, что отражает тот факт, что новый процесс также владеет этими файлами.
5. Назначает процессу состояние готовности к выполнению.
6. Возвращает родительскому процессу идентификатор дочернего процесса, а дочернему процессу — значение 0.

Все перечисленные выше действия выполняются в рамках родительского процесса в режиме ядра. После того как ядро закончит выполнение этих функций, оно может перейти к выполнению одного из следующих действий как части программы диспетчера.

- Оставаясь в рамках родительского процесса, переключить процессор в пользовательский режим; процесс будет продолжен с той команды, которая следует после вызова функции `fork()`.
- Передать управление дочернему процессу. Дочерний процесс начинает выполнять с того же места кода, что и родительский: с точки возврата после вызова функции `fork()`.
- Передать управление другому процессу. При этом и родительский и дочерний процессы переходят в состояние готовности к выполнению.

Возможно, такой метод создания процессов трудно изобразить наглядно, потому что и родительский, и дочерний процессы в момент создания выполняют один и тот же проход по коду. Различаются они возвращаемым функцией `fork()` значением: если оно равно нулю, то это дочерний процесс. Таким образом, можно выполнить команду ветвления, которая приведет к выполнению дочерней программы или к продолжению выполнения основной ветви.

## 3.7. РЕЗЮМЕ

Наиболее фундаментальной концепцией современных операционных систем является процесс. Основная функция операционной системы состоит в создании, управлении и завершении процессов. Операционная система должна следить за тем, чтобы каждому активному процессу выделялось время для выполнения на процессоре, координировать деятельность процессов, разрешать конфликтные ситуации и выделять процессам системные ресурсы.

Чтобы операционная система имела возможность управлять процессами, она поддерживает описание каждого процесса или образ процесса. В образ процесса входят адресное пространство, в котором этот процесс выполняется, и управляющий блок процесса. В управляющем блоке содержится вся информация, которая требуется операционной системе для управления процессом, включая его текущее состояние, выделенные ресурсы, приоритет и другие необходимые данные.

Во время своего существования процесс может переходить из одного состояния в другое. Наиболее важными из всех состояний являются состояние готовности, состояние выполняющегося процесса и блокированное состояние. Готовый к выполнению процесс — это процесс, который не выполняется в данный момент, но его выполнение может начаться сразу же, как только операционная система передаст ему управление. Выполняющийся процесс — это процесс, который в настоящее время выполняется процессором. В многопроцессорной системе в этом состоянии может находиться сразу несколько процессов. Блокированным является процесс, который ожидает наступления какого-то события, например завершения операции ввода-вывода.

Выполнение процессов время от времени прерывается. Прерывание наступает либо вследствие какого-то внешнего по отношению к процессу события, которое распознается процессором, либо вследствие вызова управляющей программы операционной сис-

темы. В любом случае происходит переключение процессора в другой режим работы и передача управления подпрограмме операционной системы. После выполнения своих функций операционная система может продолжить выполнение прерванного процесса или переключиться на выполнение другого процесса.

## 3.8. КЛЮЧЕВЫЕ ТЕРМИНЫ, КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАЧИ

### Ключевые термины

Блокированное состояние	Образ процесса	Свопинг
Вытеснение	Переключение процесса	Системный режим
Диспетчер	Переключение режимов	След
Дочерний процесс	Пользовательский режим	Слово состояния программы
Задание	Прерывание	Состояние выполнения
Запуск процесса	Привилегированный режим	Состояние готовности
Квант времени	Процесс	Состояние завершения
Ловушка	Режим ядра	Управляющая информация процесса
Новый процесс	Родительский процесс	Управляющий блок процесса

### Контрольные вопросы

- 3.1. Что такое след процесса?
- 3.2. В результате каких событий создаются процессы?
- 3.3. Дайте краткое описание каждого состояния, фигурирующего в модели обработки процессов, представленной на рис. 3.6.
- 3.4. Что такое вытеснение процесса?
- 3.5. Что такое свопинг и когда он применяется?
- 3.6. Зачем на рис. 3.9, б представлены два блокированных состояния?
- 3.7. Перечислите четыре характерных признака приостановленных процессов.
- 3.8. Для каких объектов операционная система поддерживает таблицы с управляющей информацией?
- 3.9. Перечислите три общие категории информации в управляющем блоке процесса.
- 3.10. Зачем нужны два режима работы процессора (пользовательский режим и режим ядра)?
- 3.11. Какие действия выполняет операционная система при создании нового процесса?
- 3.12. Чем различаются обычное прерывание и ловушка?
- 3.13. Приведите три примера прерываний.
- 3.14. Чем различаются переключение режима работы процессора и переключение процессов?

## Задачи

- 3.1. Приведенная далее таблица переходов представляет собой упрощенную модель управления процессами. Метки указывают переходы между состояниями.

	Готовый	Выполняющийся	Блокированный	Нерезидентный
Готовый	—	1	—	5
Выполняющийся	2	—	3	—
Блокированный	4	—	—	6

Приведите пример событий, вызывающих каждый из указанных в таблице переходов. Если это может помочь, нарисуйте диаграмму переходов.

- 3.2. Предположим, что в момент времени 5 не используются никакие системные ресурсы, за исключением процессора и памяти. Теперь рассмотрим следующие события.

В момент 5: Р1 выполняет команду чтения с дискового устройства 3

В момент 15: Истекает квант времени Р5

В момент 18: Р7 выполняет команду записи на дисковое устройство 3

В момент 20: Р3 выполняет команду чтения с дискового устройства 2

В момент 24: Р5 выполняет команду записи на дисковое устройство 3

В момент 28: Выполняется выгрузка процесса Р5 на диск

В момент 33: Прерывание от дискового устройства 2: чтение Р3 завершено

В момент 36: Прерывание от дискового устройства 3: чтение Р1 завершено

В момент 38: Процесс Р8 завершается

В момент 40: Прерывание от дискового устройства 3: запись Р5 завершена

В момент 44: Загрузка процесса Р5 с диска

В момент 48: Прерывание от дискового устройства 3: запись Р7 завершена

Для каждого из моментов времени 22, 37 и 47 укажите состояние, в котором находится каждый процесс. Если процесс блокирован, укажите событие, которое к этому привело.

- 3.3. На рис. 3.9, б имеется семь состояний. В принципе можно изобразить переходы между любыми двумя состояниями — всего 42 различных перехода.

- Перечислите все возможные переходы и приведите примеры, что именно могло вызвать каждый переход.
- Перечислите все невозможные переходы и поясните, почему они невозможны.

- 3.4. Для модели с семью состояниями, изображенной на рис. 3.9, б, изобразите диаграмму, подобную показанной на рис. 3.8, б.

**3.5.** Рассмотрим диаграмму переходов состояний, изображенную на рис. 3.9, б. Предположим, что для операционной системы пришло время переключать процесс. Пусть у нас имеются процессы как в состоянии готовности, так и готовые к выполнению/приостановленные. Кроме того, по крайней мере один из готовых к выполнению приостановленных процессов имеет более высокий приоритет по сравнению с приоритетом любого процесса в состоянии готовности. Можно действовать в соответствии с одной из двух экстремальных стратегий.

1. Всегда выбирать процесс в состоянии готовности, чтобы свести свопинг к минимуму.
2. Всегда отдавать предпочтение процессу с более высоким приоритетом, даже если для этого приходится выполнять свопинг, который не является необходимым.

Предложите промежуточную стратегию, в которой была бы предпринята попытка сбалансировать концепции приоритета и производительности.

**3.6.** В табл. 3.13 приведены состояния процессов, использующиеся в операционной системе VAX/VMS.

**Таблица 3.13. Состояния процессов в операционной системе VAX/VMS**

Выполняющееся	Выполняющийся процесс
Выполнимое (резидентное)	Готовый к выполнению процесс, находящийся в основной памяти
Выполнимое (выгруженное)	Готовый к выполнению процесс, выгруженный из основной памяти
Состояние ожидания загрузки страницы	Процесс обратился к странице, которая отсутствует в основной памяти, и должен ждать, пока она будет считана
Состояние ожидания разрешения конфликта доступа к странице	Процесс обратился к совместно используемой странице, загрузку которой уже ожидает другой процесс, или к персональной странице, для которой как раз выполняется чтение или запись
Состояние ожидания общего события	Ожидание переключения совместно используемого флага событий (флаги событий — это однобитовые механизмы передачи сигналов от одного процесса другому)
Состояние ожидания свободной страницы	Процесс ожидает, пока ко множеству выделенных ему и находящихся в основной памяти страниц (рабочему множеству процесса) будет добавлена свободная страница основной памяти
Состояние сна (резидентное)	Процесс переводит сам себя в состояние ожидания
Состояние сна (выгруженное)	Процесс в состоянии сна выгружается из основной памяти
Ожидание локального события (резидентное)	Процесс находится в основной памяти и ожидает наступления локального события (обычно этим событием является завершение ввода-вывода)
Ожидание локального события (выгруженное)	Процесс в ожидании локального события выгружен из основной памяти
Приостановленное (резидентное)	Процесс, переведенный другим процессом в состояние ожидания
Приостановленное (выгруженное)	Приостановленный процесс выгружается из основной памяти
Ожидание ресурса	Процесс, ожидающий, пока ему будет предоставлен некоторый системный ресурс

- a. Сможете ли вы найти обоснование наличию такого большого количества различных состояний ожидания?
  - б. Почему нет резидентных и приостановленных версий таких состояний, как ожидание загрузки страницы в основную память, ожидание разрешения конфликта доступа к странице, ожидание общего события, ожидание свободной страницы и ожидание ресурса?
  - в. Изобразите диаграмму переходов состояний и укажите на ней действие или событие, которое приводит к каждому из изображенных переходов.
- 3.7. В операционной системе VAX/VMS для облегчения защиты процессов и распределения системных ресурсов между ними используются четыре режима доступа к процессору. Режим доступа определяет такие характеристики.

- **Привилегии выполнения команд:** какие команды может выполнить процессор.
- **Привилегии доступа к памяти:** к каким ячейкам виртуальной памяти имеет доступ текущая команда.

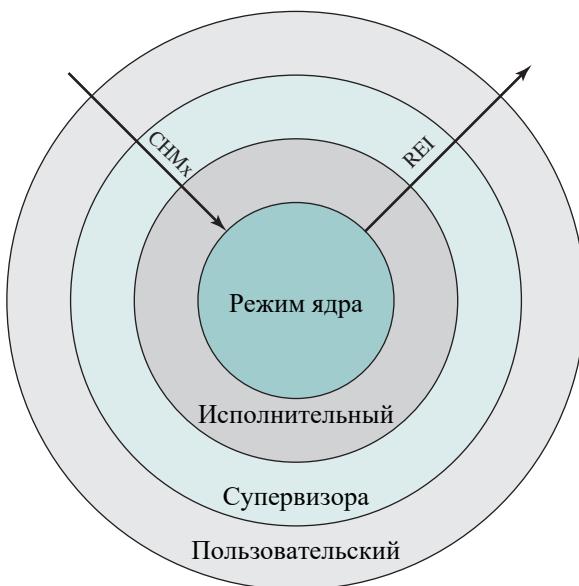
Перечислим эти четыре режима.

- **Режим ядра.** В нем выполняется ядро операционной системы VMS, которое включает в себя систему управления памятью, обработчик прерываний и операции ввода-вывода.
- **Исполнительный.** В этом режиме выполняются многие служебные программы операционной системы, в том числе программы для управления файлами и записями (на диск и магнитную ленту).
- **Режим супервизора.** В нем выполняются другие службы операционной системы, такие как функции обработки команд пользователя.
- **Пользовательский режим,** являющийся режимом выполнения пользовательских программ, а также таких утилит, как компиляторы, редакторы, компоновщики и отладчики.

Для выполнения процесса в менее привилегированном режиме часто требуется вызов процедуры, работающей в более привилегированном режиме, например когда пользовательская программа нуждается в сервисе операционной системы. Такие процедуры вызываются с использованием команды изменения режима (Change Mode — CHM), результатом выполнения которой является прерывание, передающее управление программе, работающей в новом режиме доступа. Возврат происходит с использованием команды REI (return from exception or interrupt — возврат после исключительной ситуации или прерывания).

- a. В некоторых операционных системах имеются в наличии только два режима работы процессора: режим ядра и пользовательский режим. Какие преимущества и недостатки наличия четырех режимов вместо двух?
  - б. Можете ли вы придумать пример использования большего количества режимов, чем четыре?
- 3.8. Схему работы операционной системы VMS, обсуждаемую в предыдущей задаче, часто называют кольцевой системой защиты (рис. 3.18). По аналогии можно сказать, что простая схема с двумя режимами (пользовательским и режимом ядра), описанная в разделе 3.3, является кольцевой структурой с двумя кольцами. Основной недостаток кольцевой (иерархической) структуры состоит в том, что она не позволяет реализовать принцип необходимого знания.

В [231] приведен такой пример: если объект должен быть доступен в домене  $D_j$ , но недоступен в домене  $D_i$ , то это возможно только при выполнении условия  $j < i$ . Однако это означает, что каждый объект, доступный в  $D_i$ , также доступен в домене  $D_j$ . Дайте подробное объяснение проблемы, приведенной в предыдущем абзаце.



**Рис. 3.18.** Режимы доступа в операционной системе VAX/VMS

- 3.9. На рис. 3.8,б предполагается, что каждый процесс в фиксированный момент времени может находиться только в одной очереди.
  - a. Можно ли реализовать такую схему, в которой процесс мог бы одновременно ожидать нескольких событий? Приведите соответствующий пример.
  - b. Каким образом следует модифицировать структуру очередей, чтобы в ней поддерживалась указанная возможность?
- 3.10. Раньше в некоторых компьютерах прерывание приводило к сохранению значений регистров в фиксированных ячейках, соответствующих данному сигналу прерывания. Когда такой метод является удобным? Объясните, в чем его недостаток в общем случае.
- 3.11. В разделе 3.4 утверждается, что операционная система UNIX не приспособлена для приложений, работающих в режиме реального времени, потому что в ней нельзя вытеснить процесс, выполняющийся в режиме ядра. Расшифруйте это утверждение.
- 3.12. Вы должны выполнить следующую программу на языке программирования С:

```
main()
{
    int pid;
    pid = fork();
    printf("%d\n", pid);
}
```

В предположении, что вызов `fork()` выполнился успешно, каким может быть вывод этой программы?



# ГЛАВА 4

---

# Потоки

В ЭТОЙ ГЛАВЕ...

## 4.1. Процессы и потоки

- Многопоточность
- Функциональность потоков
- Состояния потоков
- Синхронизация потоков

## 4.2. Типы потоков

- Потоки на пользовательском уровне и на уровне ядра
- Пользовательские потоки
- Потоки на уровне ядра
- Комбинированные подходы
- Другие схемы
  - Соответствие нескольких потоков нескольким процессам
  - Соответствие одного потока нескольким процессам

## 4.3. Многоядерность и многопоточность

- Производительность программного обеспечения в многоядерных системах
- Пример приложения: игровые программы Valve

## 4.4. Управление процессами и потоками в Windows

- Управление фоновыми задачами и жизненным циклом приложений
- Процессы в Windows
- Объекты процессов и потоков
- Многопоточность
- Состояния потоков
- Поддержка подсистем операционной системы

## 4.5. Управление потоками и SMP в Solaris

- Многопоточная архитектура
- Мотивация
- Структура процессов
- Выполнение потоков
- Прерывания в роли потоков

## 4.6. Управление процессами и потоками в Linux

Задания Linux

Потоки Linux

Пространства имен Linux

Пространство имен MNT

Пространство имен UTS

Пространство имен IPC

Пространство имен PID

Сетевое пространство имен

Пользовательское пространство имен

Подсистема Linux cgroup

## 4.7. Управление процессами и потоками в Android

Приложения Android

Операции

Состояния операций

Завершение приложения

Процессы и потоки

## 4.8. Mac OS X Grand Central Dispatch

## 4.9. Резюме

## 4.10. Ключевые термины, контрольные вопросы и задачи

Ключевые термины

Контрольные вопросы

Задачи

## УЧЕБНЫЕ ЦЕЛИ

- Различать процессы и потоки.
- Разбираться в фундаментальных вопросах проектирования потоков.
- Понимать различия между пользовательскими потоками и потоками ядра.
- Разбираться в основах управления потоками в Windows.
- Разбираться в основах управления потоками в Solaris.
- Разбираться в основах управления потоками в Linux.

В этой главе излагаются некоторые сложные концепции, связанные с управлением процессами, с которыми можно встретиться в современных операционных системах. Вы узнаете, что концепция процессов на самом деле сложнее и тоньше, чем можно себе представить исходя из материала предшествующих глав. По сути, эта концепция объединяет в себе две отдельные, потенциально независимые концепции, одна из которых имеет отношение к владению ресурсами, а другая — к выполнению процессов. Во многих операционных системах это различие ведет к появлению конструкции, известной под названием **поток** (thread).

## 4.1. ПРОЦЕССЫ И ПОТОКИ

До сих пор концепцию процесса можно было охарактеризовать двумя свойствами.

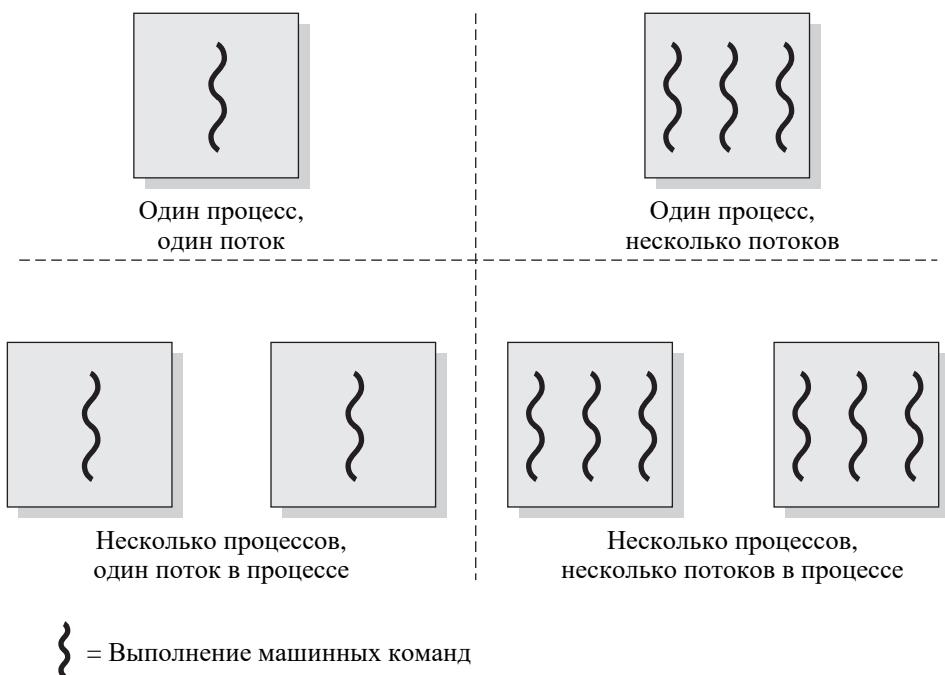
- **Владение ресурсами** (resource ownership). Процесс включает виртуальное адресное пространство, в котором содержится образ процесса; вспомните из главы 3, “Описание процессов и управление ими”, что образ процесса представляет собой коллекцию из кода, данных, стека и атрибутов, определенных в управляющем блоке процесса. Время от времени процесс может владеть такими ресурсами, как основная память, каналы и устройства ввода-вывода или файлы. Операционная система выполняет защитные функции, предотвращая нежелательные взаимодействия процессов на почве владения ресурсами.
- **Планирование/выполнение** (scheduling/execution). Выполнение процесса осуществляется путем выполнения кода одной или нескольких программ (рис. 1.5). Это выполнение процесса может чередоваться с выполнением других процессов. Поэтому процесс имеет такие параметры, как состояние (выполняющийся процесс, готовый к выполнению процесс и т.д.) и приоритет диспетчеризации, и представляет собой сущность, по отношению к которой операционная система выполняет планирование и диспетчеризацию.

Немного подумав, читатель может убедиться, что эти характеристики являются независимыми и что операционная система может рассматривать их по отдельности. Во многих операционных системах (в особенности современных) именно так и происходит. Чтобы различать две приведенные выше характеристики, единицу диспетчеризации

обычно называют **потоком** (thread) или **облегченным процессом** (lightweight process), а единицу владения ресурсами — **процессом** или **заданием** (task).<sup>1</sup>

## Многопоточность

Многопоточностью (multithreading) называется способность операционной системы поддерживать в рамках одного процесса несколько параллельных путей выполнения. Традиционный подход, при котором каждый процесс представляет собой единый поток выполнения, называется однопоточным подходом. В двух левых частях рис. 4.1 иллюстрируются однопоточные подходы. Примером операционной системы, способной поддерживать не более одного однопоточного пользовательского процесса, является MS-DOS. Другие операционные системы, такие как разнообразные разновидности UNIX, поддерживают процессы множества пользователей, но в каждом из этих процессов может содержаться только один поток. В правой половине рис. 4.1 представлены многопоточные подходы.



**Рис. 4.1.** Потоки и процессы

<sup>1</sup> Увы, последовательность в использовании терминологии не выдерживается даже в такой степени. В операционной системе для мейнфреймов IBM концепции адресного пространства и задания примерно соответствуют концепциям процесса и потока, описанным в этом разделе. Кроме того, термин *облегченный процесс* (lightweight process) используется в трех значениях: 1) эквивалентен термину *поток* (thread), 2) обозначает поток особого вида, известный как поток уровня ядра (kernel-level thread), 3) (в операционной системе Solaris) элемент, отображающий пользовательские потоки на потоки уровня ядра.

Примером системы, в которой один процесс может расщепляться на несколько потоков, является среда выполнения Java. В этом разделе нас будет интересовать использование нескольких процессов, каждый из которых поддерживает выполнение нескольких потоков. Подобный подход принят, среди прочих, в таких операционных системах, как Windows, Solaris и многих современных версиях UNIX. В этом разделе приведено общее описание многопоточного режима, а в последующих разделах будут подробно рассмотрены подходы, использующиеся в операционных системах Windows, Solaris и Linux.

В многопоточной среде процесс определяется как структурная единица распределения ресурсов, а также структурная единица защиты. С процессами связаны следующие элементы.

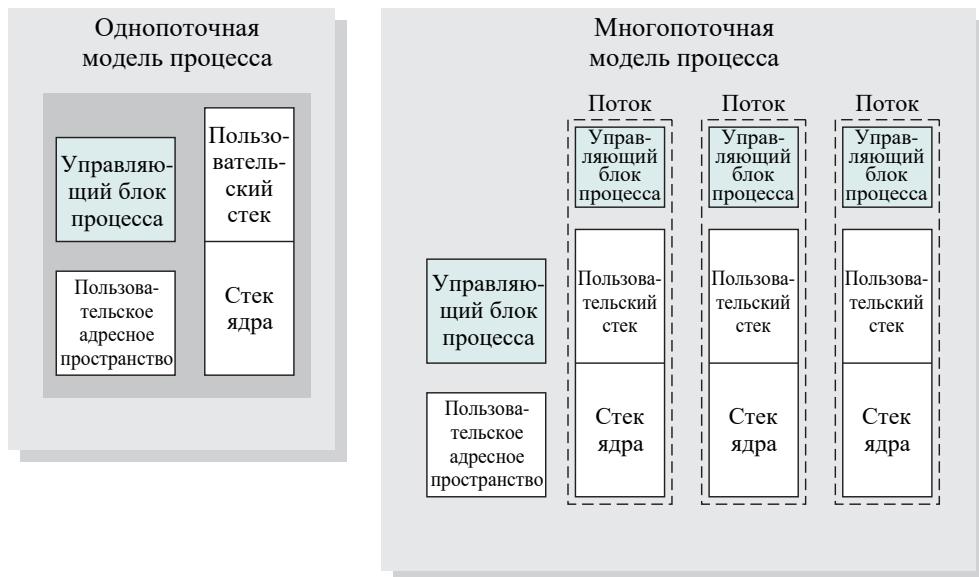
- Виртуальное адресное пространство, в котором содержится образ процесса.
- Защищенный доступ к процессорам, другим процессам (при межпроцессном обмене информацией), файлам и ресурсам ввода-вывода (устройствам и каналам).

В рамках процесса могут находиться один или несколько потоков, каждый из которых обладает следующими характеристиками.

- Состояние выполнения потока (выполняющийся, готовый к выполнению и т.д.).
- Сохраненный контекст не выполняющегося потока; один из способов рассмотрения потока — считать его независимым счетчиком команд, работающим в рамках процесса.
- Стек выполнения.
- Статическая память, выделяемая потоку для локальных переменных.
- Доступ к памяти и ресурсам процесса, которому этот поток принадлежит; этот доступ разделяется всеми потоками данного процесса.

На рис. 4.2 продемонстрировано различие между потоками и процессами с точки зрения управления последними. В однопоточной модели процесса (в которой не имеется концепции потока, отличной от концепции процесса) в его представление входят управляющий блок этого процесса и пользовательское адресное пространство, а также стеки ядра и пользователя, с помощью которых осуществляются вызовы процедур и возвраты из них при выполнении процесса. Пока процесс работает, он управляет регистрами процессора. Когда выполнение процесса прерывается, содержимое регистров процессора сохраняется в памяти. В многопоточной среде с каждым процессом тоже связаны единственный управляющий блок и адресное пространство, но теперь для каждого потока создаются свои отдельные стеки, а также свой управляющий блок, в котором содержатся значения регистров, приоритет и другая информация о состоянии потока.

Таким образом, все потоки процесса разделяют между собой состояние и ресурсы этого процесса. Они находятся в одном и том же адресном пространстве и имеют доступ к одним и тем же данным. Если один поток изменяет в памяти какие-то данные, то другие потоки во время своего доступа к этим данным имеют возможность отследить эти изменения. Если один поток открывает файл с правом чтения, другие потоки данного процесса тоже могут читать из этого файла.



**Рис. 4.2.** Однопоточная и многопоточная модели процесса

Основные преимущества использования потоков с точки зрения производительности таковы.

1. Для создания нового потока в уже существующем процессе необходимо намного меньше времени, чем для создания нового процесса. Исследования, проведенные разработчиками операционной системы Mach, показали, что скорость создания потоков в UNIX на порядок превышает скорость создания процессов [255].
2. Поток можно завершить намного быстрее, чем процесс.
3. Переключение между потоками в рамках одного и того же процесса происходит намного быстрее переключения между процессами.
4. При использовании потоков повышается эффективность обмена информацией между двумя выполняющимися программами. В большинстве операционных систем обмен между независимыми процессами происходит с участием ядра, в функции которого входит обеспечение защиты, и механизма, необходимого для осуществления обмена. Однако благодаря тому, что различные потоки одного и того же процесса используют одну и ту же область памяти и одни и те же файлы, они могут обмениваться информацией без участия ядра.

Таким образом, если приложение или функцию нужно реализовать в виде набора взаимосвязанных модулей, намного эффективнее реализовать ее в виде набора потоков, чем в виде набора отдельных процессов.

Примером приложения, в котором можно удачно применить потоки, является файловый сервер. При получении каждого нового файлового запроса программа управления файлами может порождать новый поток. Из-за того, что серверу придется обрабатывать очень большое количество запросов, за короткий промежуток времени будет создаваться и удаляться множество потоков. Если такая серверная программа работает на

многопроцессорной машине, то на разных процессорах в рамках одного процесса может одновременно выполняться несколько потоков. Кроме того, из-за того что процессы или потоки файлового сервера должны совместно использовать данные из файлов, а следовательно, координировать свои действия, рациональнее использовать потоки и общую область памяти, а не процессы и обмен сообщениями.

Потоковая конструкция процесса полезна и на однопроцессорных машинах. Она помогает упростить структуру программы, выполняющей несколько логически различных функций.

В [150] приводится четыре следующих примера использования потоков в однопользовательской многозадачной системе.

- 1. Работа в приоритетном и фоновом режимах.** В качестве примера можно привести программу электронных таблиц, в которой один из потоков может отвечать за отображение меню и считывать ввод пользователя, а другой — выполнять команды пользователя и обновлять таблицу. Такая схема часто увеличивает воспринимаемую пользователем скорость работы приложения, позволяя пользователю начать ввод следующей команды еще до завершения выполнения предыдущей.
- 2. Асинхронная обработка.** Элементы асинхронности в программе можно реализовать в виде потоков. Например, в качестве меры предосторожности на случай отключения электричества можно сделать так, чтобы текстовый редактор каждую минуту сбрасывал на диск содержимое буфера оперативного запоминающего устройства. Можно создать поток, единственной задачей которого будет создание резервной копии и который будет планировать свою работу непосредственно с помощью операционной системы. Это позволит обойтись без помещения в основную программу замысловатого кода, обеспечивающего проверку соблюдения временного графика или координацию ввода и вывода.
- 3. Скорость выполнения.** Многопоточный процесс может производить вычисления с одной порцией данных, одновременно считывая с устройства ввода-вывода следующую порцию. В многопроцессорной системе несколько потоков одного и того же процесса могут выполняться одновременно. Таким образом, даже несмотря на то, что один поток может быть заблокирован операцией ввода-вывода для чтения пакета данных, другой поток может продолжать выполняться.
- 4. Модульная структура программы.** Программы, осуществляющие разнообразные действия или выполняющие множество вводов из различных источников и выводов в разные места назначения, легче разрабатывать и реализовывать с помощью потоков.

В операционной системе, поддерживающей потоки, планирование и диспетчеризация осуществляются на основе потоков; таким образом, большая часть информации о состоянии процесса, имеющей отношение к его выполнению, поддерживается в структурах данных на уровне потоков. Однако есть несколько действий, которые затрагивают все потоки процесса и которые операционная система должна поддерживать именно на этом уровне. Если процесс приостанавливается, то при этом предполагается, что его адресное пространство будет выгружено из основной памяти. Поскольку все потоки процесса используют одно и то же адресное пространство, все они должны одновременно перейти в состояние приостановленных. Соответственно, прекращение процесса приводит к прекращению всех составляющих его потоков.

## ФУНКЦИОНАЛЬНОСТЬ ПОТОКОВ

Потоки, подобно процессам, характеризуются состояниями выполнения; кроме того, они могут быть синхронизированы между собой. Рассмотрим по очереди эти два аспекта.

### Состояния потоков

Основными состояниями потоков, как и процессов, являются состояние выполнения потока, состояние готовности и состояние блокировки. Вообще говоря, состояние приостановки нет смысла связывать с потоками, потому что такие состояния логичнее рассматривать на уровне процессов. В частности, если процесс приостанавливается, обязательно приостанавливаются все его потоки, потому что все они совместно используют адресное пространство этого процесса.

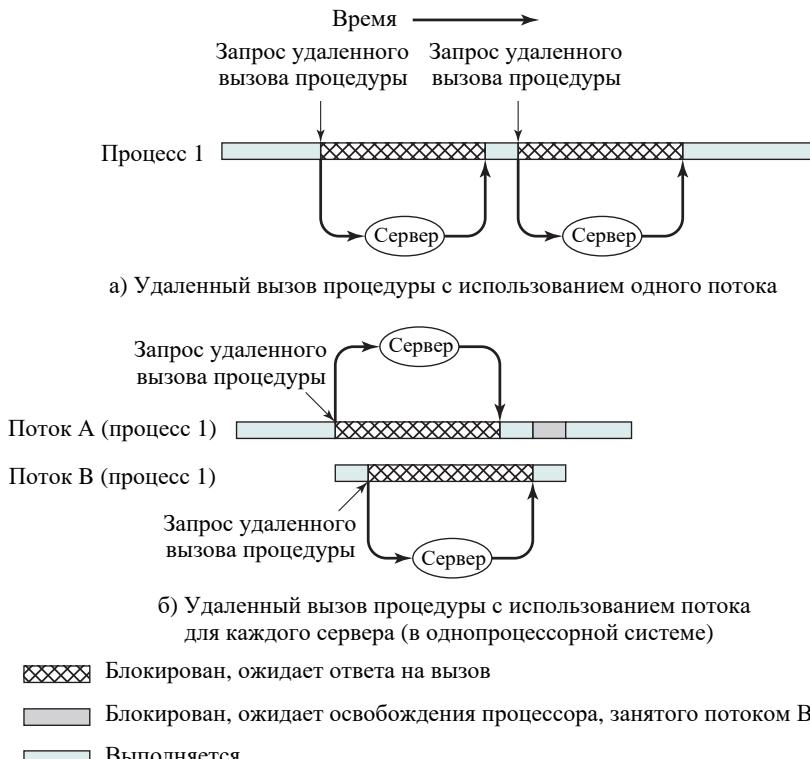
С изменением состояния потоков связаны такие четыре основных действия [6].

1. **Порождение.** Обычно одновременно с новым процессом создается его поток. Далее, в рамках одного и того же процесса один поток может породить другой поток, определив его указатель команд и аргументы. Новый поток создается со своим контекстом регистров и стековым пространством, после чего он помещается в очередь готовых к выполнению потоков.
2. **Блокирование.** Если потоку нужно подождать наступления некоторого события, он блокируется (при этом сохраняются содержимое его пользовательских регистров, счетчика команд, а также указатели стеков). После этого процессор может перейти к выполнению другого готового потока.
3. **Разблокирование.** Когда наступает событие, ожидание которого блокировало поток, последний переходит в состояние готовности.
4. **Завершение.** После завершения потока его контекст регистров и стеки удаляются.

Важно понять, должно ли блокирование потока обязательно приводить к блокированию всего процесса. Другими словами, могут ли выполняться какие-нибудь готовые к выполнению потоки процесса, если один из его потоков блокирован? Ясно, что если блокировка одного из потоков будет приводить к блокировке всего процесса, то это существенно уменьшит гибкость и эффективность потоков.

Мы еще вернемся к обсуждению этого вопроса при сравнении потоков на пользовательском уровне и потоков на уровне ядра, а пока что рассмотрим выигрыш в производительности при использовании потоков, которые не блокируют весь процесс. На рис. 4.3 (из [133]) показана программа, выполняющая два вызова удаленных процедур (*remote procedure call* — *RPC*)<sup>2</sup> на двух разных машинах, чтобы получить результат после их совместного выполнения. В однопоточной программе результаты получаются последовательно, поэтому программа должна ожидать, пока от каждого сервера по очереди будет получен ответ. Переписав программу так, чтобы для каждого вызова удаленной процедуры она использовала отдельный поток, можно получить существенный выигрыш в скорости.

<sup>2</sup> Вызов удаленной процедуры — это технология, при которой две программы, которые могут выполняться на разных машинах, взаимодействуют между собой с помощью синтаксиса и семантики вызовов и возвратов из процедур. Обе программы, вызывающая и вызываемая, ведут себя так, как будто они выполняются на одной и той же машине. Вызовы удаленных процедур часто применяются в приложениях, работающих по схеме “клиент/сервер”. Подробнее вызовы удаленных процедур обсуждаются в главе 16, “Облачные операционные системы и операционные системы Интернета вещей”.



**Рис. 4.3.** Удаленный вызов процедуры (RPC), в котором используются потоки

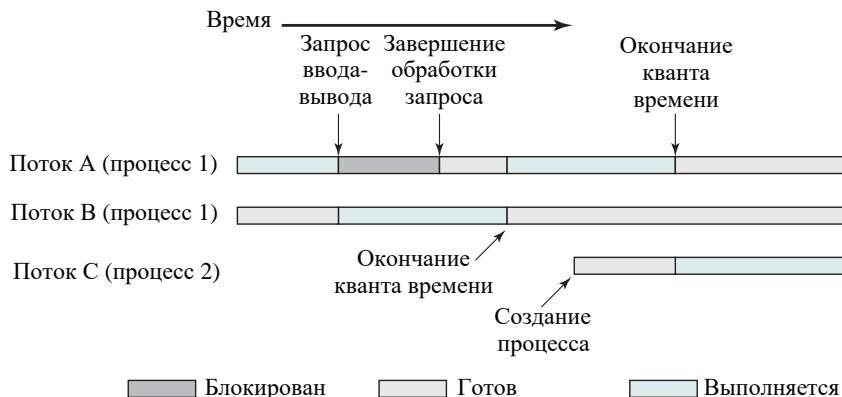
Заметим, что если такая программа работает на однопроцессорной машине, то запросы будут генерироваться последовательно; результаты тоже будут получены последовательно, однако программа будет ожидать двух ответов одновременно.

В однопроцессорных системах многозадачность позволяет чередовать различные потоки нескольких процессов. В примере, показанном на рис. 4.4, чередуются три потока, принадлежащие двум процессам. Передача управления от одного процесса другому происходит либо тогда, когда блокируется выполняющийся поток, либо когда заканчивается интервал времени, отведенный для его выполнения.<sup>3</sup>

### Синхронизация потоков

Все потоки процесса используют одно и то же адресное пространство, как и другие ресурсы, например открытые файлы. Любое изменение какого-нибудь ресурса одним из потоков процесса оказывает влияние на другие потоки этого же процесса. Поэтому действия различных потоков необходимо синхронизировать, чтобы они не мешали друг другу или чтобы не повредили структуры данных. Например, если каждый из двух потоков будет пытаться добавить свой элемент в двунаправленный список, может быть потерян один из элементов (или нарушена целостность списка).

<sup>3</sup> В этом примере поток С начинает выполняться, после того как оканчивается интервал времени, отведенный потоку А, несмотря на то что поток В тоже находится в состоянии готовности. Выбор между потоками В и С — это вопрос планирования; данная тема исследуется в части IV книги.



**Рис. 4.4.** Пример многопоточности в однопроцессорной системе

При рассмотрении синхронизации потоков возникают те же вопросы и используются те же методы, что и при синхронизации процессов. Эти вопросы и методы обсуждаются в главе 5, “Параллельные вычисления: взаимоисключения и многозадачность”, и главе 6, “Параллельные вычисления: взаимоблокировка и голодание”.

## 4.2. Типы потоков

### Потоки на пользовательском уровне и на уровне ядра

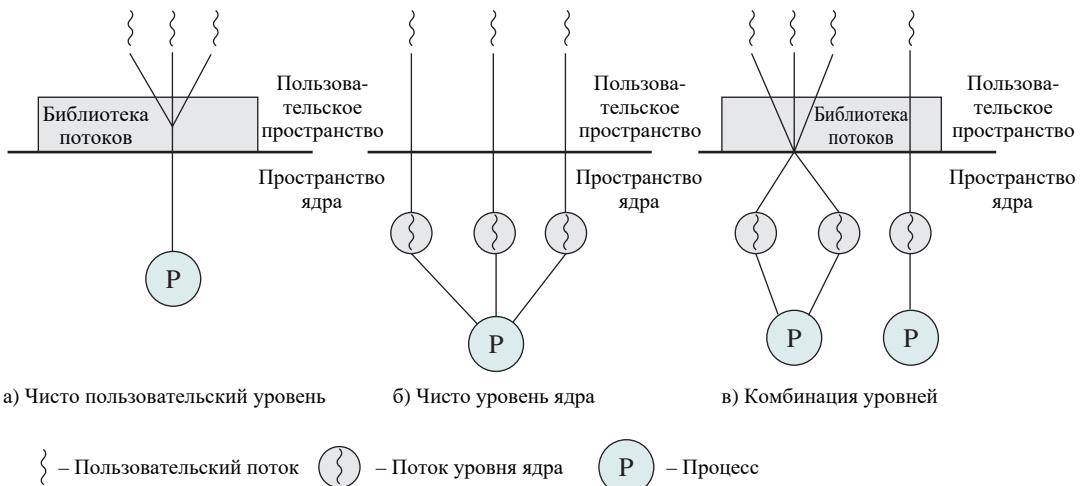
Обычно выделяют две общие категории потоков: пользовательские потоки, или потоки на уровне пользователя (user-level threads — ULT), и потоки на уровне ядра (kernel-level threads — KLT)<sup>4</sup>. Потоки второго типа в литературе иногда называются *потоками, поддерживаемыми ядром* или *облегченными процессами* (lightweight processes).

#### Пользовательские потоки

В программе, полностью состоящей из ULT-потоков, все действия по управлению потоками выполняются самим приложением; ядро, по сути, и не подозревает о существовании потоков. На рис. 4.5, а проиллюстрирован подход, при котором используются только потоки на уровне пользователя. Чтобы приложение было многопоточным, его следует создавать с применением специальной библиотеки, представляющей собой пакет программ для работы с потоками на уровне ядра. Такая библиотека для работы с потоками содержит код, с помощью которого можно создавать и удалять потоки, производить обмен сообщениями и данными между потоками, планировать их выполнение, а также сохранять и восстанавливать их контекст.

По умолчанию приложение в начале своей работы состоит из одного потока, и его выполнение начинается как выполнение этого потока. Такое приложение вместе с составляющим его потоком размещается в едином процессе, который управляется ядром. Выполняющееся приложение (процесс которого находится в состоянии выполнения) в любой момент времени может породить новый поток, который будет выполняться в пределах того же процесса.

<sup>4</sup> Эти аббревиатуры не являются широко распространенными и используются только для краткости.



**Рис. 4.5. Пользовательские потоки и потоки уровня ядра**

Новый поток создается с помощью вызова специальной подпрограммы из библиотеки, предназначенной для работы с потоками. Управление к этой подпрограмме переходит в результате вызова процедуры. Библиотека потоков создает структуру данных для нового потока, а потом передает управление одному из готовых к выполнению потоков данного процесса, руководствуясь некоторым алгоритмом планирования. Когда управление переходит к библиотечной подпрограмме, контекст текущего потока сохраняется, а когда управление возвращается к потоку, его контекст восстанавливается. Этот контекст в основном состоит из содержимого пользовательских регистров, счетчика команд и указателей стека.

Все описанные в предыдущих абзацах события происходят в пользовательском пространстве в рамках одного процесса. Ядро не подозревает об этой деятельности. Оно продолжает осуществлять планирование процесса как единого целого и приписывать ему единое состояние выполнения (состояние готовности, состояние выполняющегося процесса, состояние блокировки и т.д.). Приведенные ниже примеры должны прояснить взаимосвязь между планированием потоков и планированием процессов. Предположим, что выполняется поток 2, входящий в процесс В (рис. 4.6). Состояния этого процесса и составляющих его потоков на пользовательском уровне показаны на рис. 4.6, а. Впоследствии может произойти одно из следующих событий.

1. Приложение, в котором выполняется поток 2, может произвести системный вызов, например запрос ввода-вывода, который блокирует процесс В. В результате этого вызова управление перейдет к ядру. Ядро вызывает процедуру ввода-вывода, переводит процесс В в состояние блокировки и передает управление другому процессу. Тем временем поток 2 процесса В все еще находится в состоянии выполнения в соответствии со структурой данных, поддерживаемой библиотекой потоков. Важно отметить, что поток 2 не выполняется в том смысле, что он работает с процессором; однако библиотека потоков воспринимает его как выполняющийся. Соответствующие диаграммы состояний показаны на рис. 4.6, б.
2. В результате прерывания по таймеру управление может перейти к ядру; ядро определяет, что интервал времени, отведенный выполняющемуся в данный момент

процессу В, истек. Ядро переводит процесс В в состояние готовности и передает управление другому процессу. В это время, согласно структуре данных, которая поддерживается библиотекой потоков, поток 2 процесса В по-прежнему будет находиться в состоянии выполнения. Соответствующие диаграммы состояний показаны на рис. 4.6, в.

- Поток 2 достигает точки выполнения, когда ему требуется, чтобы поток 1 процесса В выполнил некоторое действие. Он переходит в заблокированное состояние, а поток 1 — из состояния готовности в состояние выполнения. Сам процесс остается в состоянии выполнения. Соответствующие диаграммы состояний показаны на рис. 4.6, г.

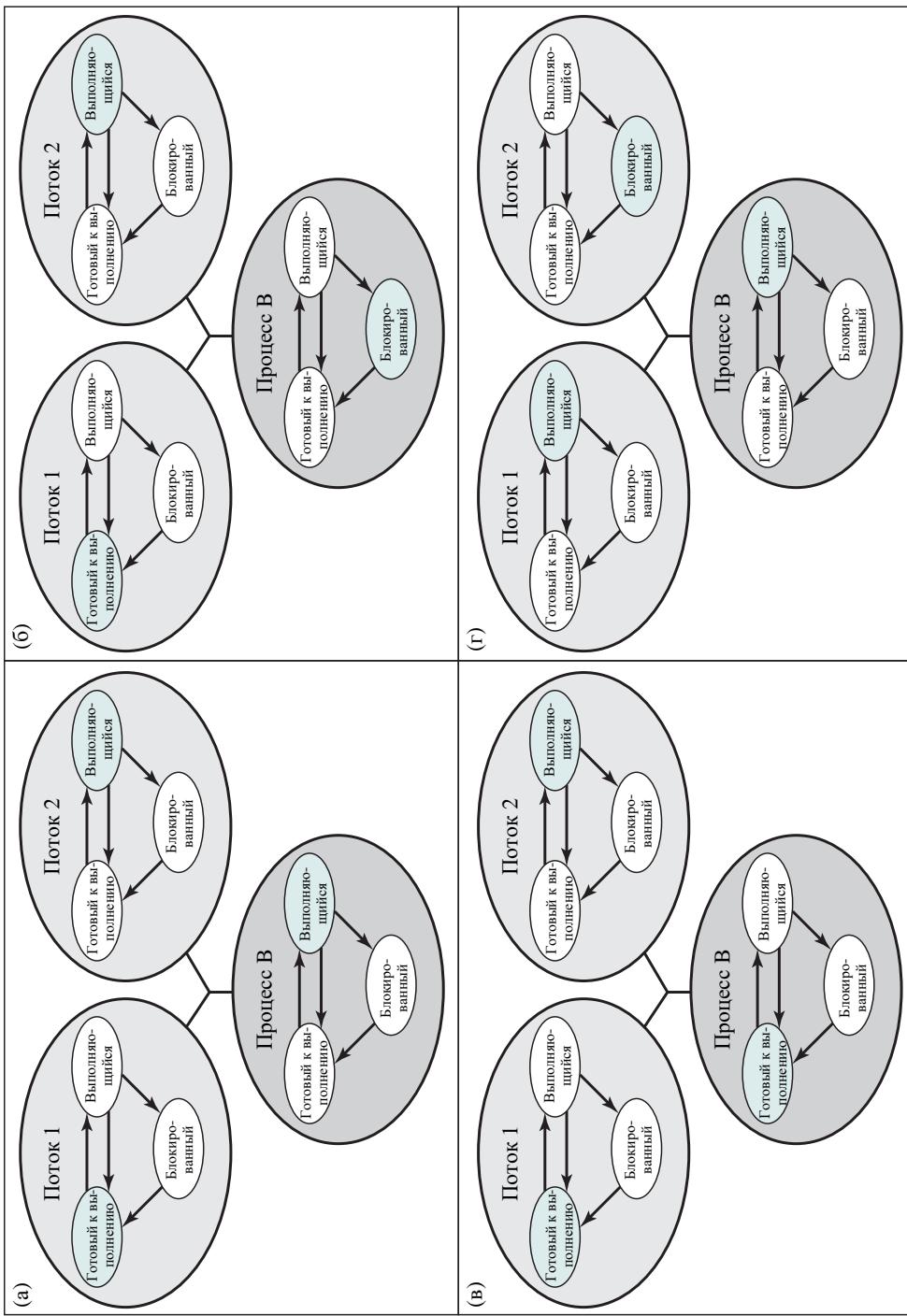
Обратите внимание: три представленных выше пункта предлагают альтернативные варианты событий, начинающиеся с диаграммы на рис. 4.6, а. Так что каждая часть рис. 4.6 (б, в и г) — представляет собой переход из ситуации, показанной в части а. В случаях 1 и 2 (рис. 4.7, б и в) при возврате управления процессу В возобновляется выполнение потока 2. Заметим также, что процесс, в котором выполняется код из библиотеки потоков, может быть прерван либо из-за того, что закончится отведенный ему интервал времени, либо из-за наличия процесса с более высоким приоритетом. Выполнение прерванного процесса продолжится в библиотеке потоков, которая завершит переключение потоков и передаст управление новому потоку процесса.

Применение потоков на пользовательском уровне обладает некоторыми преимуществами перед применением потоков на уровне ядра. К этим преимуществам относятся следующие.

- Переключение потоков не предусматривает переход в режим ядра, так как структуры данных для управления потоками находятся в адресном пространстве одного и того же процесса. Поэтому для управления потоками процессу не нужно переключаться в режим ядра. Благодаря этому обстоятельству удается избежать накладных расходов, связанных с двумя переключениями режимов (пользовательского режима в режим ядра и обратно).
- Планирование может выполняться с учетом специфики приложения. Для одних приложений может лучше подойти простой алгоритм планирования по круговому алгоритму, а для других — алгоритм планирования, основанный на использовании приоритета. Алгоритм планирования может подбираться для конкретного приложения, причем это не повлияет на алгоритм планирования, заложенный в операционной системе.
- Использование потоков на пользовательском уровне может работать в любой операционной системе. Для их поддержки в ядро системы не потребуется вносить никаких изменений. Библиотека потоков представляет собой набор утилит, работающих на уровне приложения и совместно используемых всеми приложениями.

Применение потоков на пользовательском уровне обладает также двумя явными недостатками по сравнению с применением потоков на уровне ядра.

- В типичной операционной системе многие системные вызовы являются блокирующими. Когда в потоке, работающем на пользовательском уровне, выполняется системный вызов, блокируется не только данный поток, но и все потоки того процесса, к которому он относится.



**Рис. 4.6.** Примеры взаимосвязей между состояниями потоков пользовательского уровня и состояниями процесса

2. В стратегии с наличием потоков только на пользовательском уровне приложение не может воспользоваться преимуществами многопроцессорной системы, так как ядро закрепляет за каждым процессом только один процессор. Поэтому несколько потоков одного и того же процесса не могут выполняться одновременно. Фактически у нас получается многозадачность на уровне приложения в рамках одного процесса. Несмотря на то что даже такая многозадачность может привести к значительному увеличению скорости работы приложения, имеются приложения, которые работали бы гораздо лучше, если бы различные части их кода могли выполняться одновременно.

Эти две проблемы разрешимы. Например, их можно преодолеть, если писать приложение не в виде нескольких потоков, а в виде нескольких процессов. Однако при таком подходе основные преимущества потоков сводятся на нет: каждое переключение становится не переключением потоков, а переключением процессов, что приводит к значительно большим накладным затратам.

Другим методом преодоления проблемы блокирования является преобразование блокирующего системного вызова в неблокирующий, известный как *jacketing*. Например, вместо непосредственного вызова системной процедуры ввода-вывода поток вызывает подпрограмму-оболочку, которая производит ввод-вывод на уровне приложения. В этой программе содержится код, который проверяет, занято ли устройство ввода-вывода. Если оно занято, поток передает управление другому потоку (что происходит с помощью библиотеки потоков). Когда наш поток вновь получает управление, он повторно осуществляет проверку занятости устройства ввода-вывода.

### **Потоки на уровне ядра**

В программе, работа которой полностью основана на потоках, работающих на уровне ядра, все действия по управлению потоками выполняются ядром. В области приложений отсутствует код, предназначенный для управления потоками. Вместо него используется интерфейс прикладного программирования (application programming interface — API) средств ядра, управляющих потоками. Примером такого подхода является операционная система Windows.

На рис. 4.5,б проиллюстрирована стратегия использования потоков на уровне ядра. Ядро поддерживает информацию контекста процесса как единого целого, а также контекстов каждого отдельного потока процесса. Планирование выполняется ядром на основе потоков. С помощью такого подхода удается избавиться от двух упомянутых ранее основных недостатков потоков пользовательского уровня. Во-первых, ядро может одновременно осуществлять планирование работы нескольких потоков одного и того же процесса на нескольких процессорах. Во-вторых, при блокировке одного из потоков процесса ядро может выбрать для выполнения другой поток этого же процесса. Еще одним преимуществом такого подхода является то, что сами процедуры ядра могут быть многопоточными.

Основным недостатком подхода с использованием потоков на уровне ядра по сравнению с использованием потоков на пользовательском уровне является то, что для передачи управления от одного потока другому в рамках одного и того же процесса приходится переключаться в режим ядра. Результаты исследований, проведенных на однопроцессорной машине VAX под управлением UNIX-подобной операционной системы, представ-

ленные в табл. 4.1, иллюстрируют различие между этими двумя подходами на однопроцессорном компьютере VAX под управлением UNIX-подобной операционной системы. Сравнивалось время выполнения таких двух задач, как нулевое ветвление (Null Fork) — время, затраченное на создание, планирование и выполнение процесса/потока, состоящего только из пустой процедуры (измеряются только накладные расходы, связанные с ветвлением процесса/потока) и ожидание сигнала (Signal-Wait) — время, затраченное на передачу сигнала от одного процесса/потока другому процессу/потоку, находящемуся в состоянии ожидания (накладные расходы на синхронизацию двух процессов/потоков). Мы видим, что различие во времени выполнения потоков на уровне ядра и потоков на пользовательском уровне более чем на порядок превосходит по величине различие во времени выполнения потоков на уровне ядра и процессов.

**Таблица 4.1. Время задержек потоков (ms)**

Операция	Пользовательские потоки	Потоки на уровне ядра	Процессы
Нулевое ветвление	34	948	11300
Ожидание сигнала	37	441	1840

Таким образом, создается впечатление, что как применение многопоточности на уровне ядра дает выигрыш по сравнению с процессами, так и многопоточность на пользовательском уровне дает выигрыш по сравнению с многопоточностью на уровне ядра. Однако на деле возможность этого дополнительного выигрыша зависит от характера приложений. Если для большинства переключений потоков приложения необходим доступ к ядру, то схема с потоками на пользовательском уровне может работать не намного лучше, чем схема с потоками на уровне ядра.

### Комбинированные подходы

В некоторых операционных системах применяется комбинирование потоков обоих видов (рис. 4.5, в). В комбинированных системах создание потоков выполняется полностью в пользовательском пространстве, там же, где и код планирования и синхронизации потоков в приложениях. Несколько потоков на пользовательском уровне, входящих в состав приложения, отображаются в такое же или меньшее число потоков на уровне ядра. Программист может изменять число потоков на уровне ядра, подбирая его таким, чтобы оно позволило достичь наилучших результатов.

При комбинированном подходе несколько потоков одного и того же приложения могут выполняться одновременно на нескольких процессорах, а блокирующие системные вызовы не приводят к блокировке всего процесса. При надлежащей реализации такой подход будет сочетать в себе преимущества подходов, в которых применяются только потоки на пользовательском уровне или только потоки на уровне ядра, сводя недостатки каждого из этих подходов к минимуму.

Хорошим примером операционной системы, использующей такой комбинированный подход, является Solaris. Текущая версия Solaris ограничивает отношение потоков ULT/KLT как один к одному.

## Другие схемы

Как уже упоминалось, понятия единицы распределения ресурсов и планирования традиционно отождествляются с понятием процесса. В такой концепции поддерживается однозначное соответствие между потоками и процессами. В последнее время наблюдается интерес к использованию нескольких потоков в одном процессе, когда выполняется соотношение “многие к одному”. Однако возможны и другие комбинации, а именно — соответствие нескольких потоков нескольким процессам и соответствие одного потока нескольким процессам. Примеры применения каждой из упомянутых комбинаций приводятся в табл. 4.2.

**Таблица 4.2. Соотношение между потоками и процессами**

Потоки: процессы	Описание	Примеры систем
1:1	Каждый поток реализован в виде отдельного процесса с собственным адресным пространством и со своими ресурсами	Традиционные реализации системы UNIX
M:1	Для процесса задаются адресное пространство и динамическое владение ресурсами. В рамках этого процесса может быть создано несколько потоков	OS/2, Windows NT, Solaris, Linux, OS/390, MACH
1:M	Поток может переходить из среды одного процесса в среду другого процесса. Это облегчает перенос потоков из одной системы в другую	Ra (Clouds), Emerald
M:N	Сочетает в себе подходы, основанные на соотношениях M:1 и 1:M	TRIX

### Соответствие нескольких потоков нескольким процессам

Идея реализации соответствия нескольких процессов нескольким потокам была исследована в экспериментальной операционной системе TRIX [187, 264]. В ней используются понятия домена и потока. Домен — это статический объект, состоящий из адресного пространства и портов, через которые можно отправлять и получать сообщения. Поток — это единая выполняемая ветвь, обладающая стеком выполнения и характеризующаяся состоянием процессора, а также информацией по планированию.

Как и в других рассматривавшихся ранее многопоточных подходах, в рамках одного домена могут выполняться несколько потоков. При этом удается получить уже описанное повышение эффективности работы. Однако имеется также возможность осуществлять деятельность одного и того же пользователя или приложения в нескольких доменах. В этом случае имеется поток, который может переходить из одного домена в другой.

По-видимому, использование одного и того же потока в разных доменах продиктовано желанием предоставить программисту средства структурирования. Например, рассмотрим программу, в которой используется подпрограмма ввода-вывода. В многозадачной среде, в которой пользователю позволено создавать процессы, основная программа может генерировать новый процесс для управления вводом-выводом, а затем продолжить свою работу. Однако если для дальнейшего выполнения основной программы необходимы результаты операции ввода-вывода, то она должна ждать, пока не закончится

работа подпрограммы ввода-вывода. Подобное приложение можно осуществить такими способами.

1. Реализовать всю программу в виде единого процесса. Такой прямолинейный подход является вполне обоснованным. Недостатки этого подхода связаны с управлением памятью. Эффективно организованный как единое целое процесс может занимать в памяти много места, в то время как для подпрограммы ввода-вывода требуется относительно небольшое адресное пространство. Из-за того что подпрограмма ввода-вывода выполняется в адресном пространстве более объемной программы, во время выполнения ввода-вывода весь процесс должен оставаться в основной памяти либо операция ввода-вывода будет выполняться с применением свопинга. То же самое происходит и в случае, когда и основная программа, и подпрограмма ввода-вывода реализованы в виде двух потоков в одном адресном пространстве.
2. Основная программа и подпрограмма ввода-вывода реализуются в виде двух отдельных процессов. Это приводит к накладным затратам, возникающим в результате создания подчиненного процесса. Если ввод-вывод производится достаточно часто, то необходимо будет либо оставить такой подчиненный процесс активным на все время работы основного процесса, что связано с затратами на управление ресурсами, либо часто создавать и завершать процесс с подпрограммой, что приведет к снижению эффективности.
3. Реализовать действия основной программы и подпрограммы ввода-вывода как единый поток. Однако для основной программы следует создать свое адресное пространство (свой домен), а для подпрограммы ввода-вывода — свое. Таким образом, поток в ходе выполнения программы будет переходить из одного адресного пространства в другое. Операционная система может управлять этими двумя адресными пространствами независимо, не затрачивая никаких дополнительных ресурсов на создание процесса. Более того, адресное пространство, используемое подпрограммой ввода-вывода, может использоваться совместно с другими простыми подпрограммами ввода-вывода.

Опыт разработчиков операционной системы TRIX свидетельствует о том, что третий вариант заслуживает внимания и для некоторых приложений может оказаться самым эффективным.

### **Соответствие одного потока нескольким процессам**

В области распределенных операционных систем (разрабатываемых для управления распределенными компьютерными системами) представляет интерес концепция потока как основного элемента, способного переходить из одного адресного пространства в другое.<sup>5</sup> Заслуживают упоминания операционная система Clouds и в особенности ее ядро, известное под названием “Ra” [57]. В качестве другого примера можно привести систему Emerald [245].

В операционной системе Clouds поток является единицей активности с точки зрения пользователя. Процесс имеет вид виртуального адресного пространства с относящим-

---

<sup>5</sup> В последние годы активно исследуется тема перехода процессов и потоков из одного адресного пространства в другое (миграция). Эта тема рассматривается в главе 18, “Распределенная обработка, вычисления «клиент/сервер» и кластеры”.

ся к нему управляющим блоком. После создания поток начинает выполнение в рамках процесса. Потоки могут переходить из одного адресного пространства в другое и даже выходить за рамки машины (т.е. переходить из одного компьютера в другой). При переходе в другое место поток должен нести с собой определенную информацию — такую, как управляющий терминал, глобальные параметры и сведения по его планированию (например, приоритет).

Такой подход является эффективным способом изоляции пользователя и программиста от деталей распределенной среды. Деятельность пользователя может ограничиваться одним потоком, а перемещение этого потока из одной машины в другую может быть обусловлено функционированием операционной системы, руководствующейся такими обстоятельствами, как необходимость доступа к удаленным ресурсам или выравнивание загрузки машин.

## 4.3. Многоядерность и многопоточность

Применение многоядерных систем для поддержки одного приложения с несколькими потоками (ситуация, которая может возникнуть на рабочей станции, игровой приставке или персональном компьютере при работе приложения, интенсивно использующего процессор) приводит к возникновению вопросов производительности и дизайна приложений. В этом разделе мы сначала рассмотрим некоторые следствия выполнения многопоточных приложений на многоядерных процессорах с точки зрения производительности, а затем опишем конкретный пример приложения, разработанного с учетом использования возможностей многоядерности.

### Производительность программного обеспечения в многоядерных системах

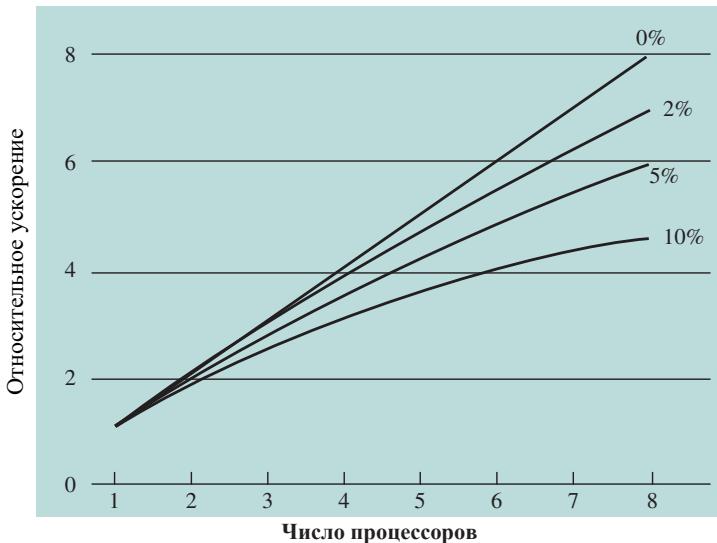
Потенциальные преимущества многоядерных систем в смысле производительности зависят от способности эффективно использовать параллельные ресурсы, доступные приложению. Давайте сначала сосредоточимся на одном приложении, работающем в многоядерной системе. Закон Амдала (см. приложение Д, “Закон Амдала”) гласит:

$$\text{Ускорение} = \frac{\text{Время выполнения программы на одном процессоре}}{\text{Время выполнения программы на } N \text{ параллельных процессорах}} = \frac{1}{(1 - f) + \frac{f}{N}}$$

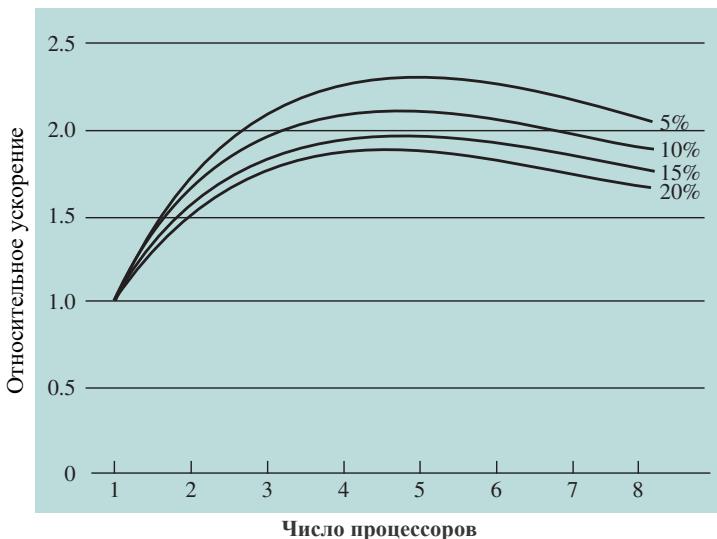
В законе предполагается, что часть времени выполнения программы ( $1-f$ ) расходуется на выполнение кода, который обязан выполняться последовательно, а часть  $f$  представляет собой код, который может быть бесконечно распараллелен без накладных расходов, связанных с планированием.

Казалось бы, этот закон должен сделать многоядерные системы очень привлекательной перспективой. Но, как показано на рис. 4.7, *a*, даже небольшое количество нераспараллеливаемого кода оказывает заметное влияние на ускорение программы. Если не могут быть распараллелены только 10% кода ( $f=0,9$ ), то выполнение программы в системе с восемью процессорами дает прирост производительности только в 4,7 раза. Кроме того, обычно имеются накладные расходы, связанные с обменом информацией и

распределением работы между несколькими процессорами, а также накладные расходы согласованности кешей. Все это приводит к кривой, имеющей пик производительности, а затем демонстрирующей ее ухудшение из-за увеличения накладных расходов, связанных с использованием нескольких процессоров. Типичный пример показан на рис. 4.7, б (взят из [167]).



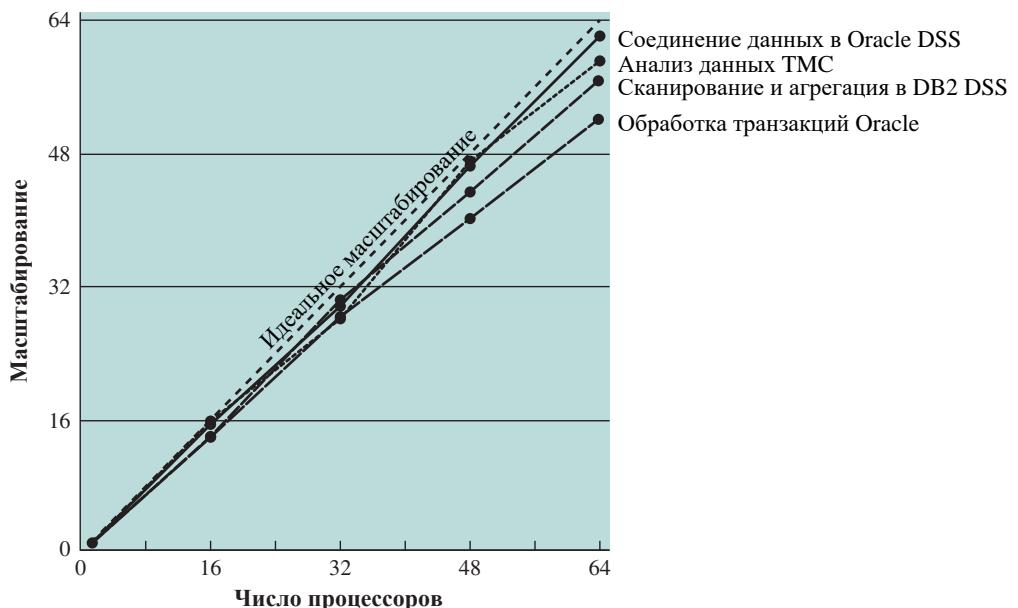
а) Ускорение для различного количества последовательного кода



б) Ускорение с учетом накладных расходов

**Рис. 4.7.** Зависимость производительности вычислений от количества ядер

Однако разработчики программного обеспечения решают эти проблемы, и существует множество приложений, которые эффективно используют возможности многоядерных систем. В [167] сообщается о наборе приложений для работы с базами данных, в которых большое внимание уделялось сокращению не распараллеливаемой части кода в аппаратных архитектурах, операционных системах и прикладном программном обеспечении. На рис. 4.8 показаны достигнутые результаты. Как демонстрирует этот пример, системы управления базами данных и приложения для работы с базами данных являются одной из областей, в которых могут эффективно использоваться многоядерные системы. Многие виды серверов также могут эффективно использовать параллельные многоядерные архитектуры, так как обычно серверы работают с многочисленными относительно независимыми операциями, которые могут быть обработаны параллельно.



**Рис. 4.8.** Масштабирование загрузок баз данных в многопроцессорных системах

Помимо серверного программного обеспечения общего назначения, имеется ряд классов приложений, получающих преимущества от возможности масштабирования пропускной способности с увеличением количества ядер. В работе [166] содержатся следующие примеры.

- **Изначально многопоточные приложения.** Многопоточные приложения характеризуются небольшим количеством процессов с большим количеством потоков. Примерами таких приложений являются Lotus Domino и Siebel CRM.
- **Многопроцессные приложения.** Многопроцессные приложения характеризуются большим количеством однопоточных процессов. Примерами многопроцессных приложений являются база данных Oracle, SAP и PeopleSoft.
- **Приложения Java.** Приложения Java тесно связаны с потоками. Язык Java не только значительно облегчает написание многопоточных приложений; сама виртуальная машина Java является многопоточным процессом, который обеспечивает

планирование и управления памятью Java-приложений. Приложения Java, могущие получить выгоду непосредственно от использования многоядерных систем, включают такие серверы приложений, как Oracle Java Application Server, BEA Weblogic, IBM Websphere и сервер приложений Tomcat с открытым исходным кодом. Все приложения, использующие серверы приложений Java 2 Platform Enterprise Edition (платформа J2EE), могут немедленно получить выгоду от использования многоядерных технологий.

- **Приложения в нескольких экземплярах.** Даже если отдельное приложение не масштабируется, чтобы получить преимущества от использования большого числа потоков, все еще возможно получить выгоды от многоядерной архитектуры, запуская несколько экземпляров приложения одновременно. Если несколько экземпляров приложения требуют определенной степени изоляции, для предоставления каждому из них собственной отдельной и безопасной среды выполнения можно использовать технологию виртуализации (для аппаратного обеспечения операционной системы).

## Пример приложения: игровые программы Valve

Valve — компания, разработавшая ряд популярных игр, а также популярный игровой механизм (“движок” — engine) Source, который представляет собой анимационный механизм, широко используемый Valve для своих игр и лицензированный для других разработчиков.

За последние годы Valve переписала программное обеспечение Source таким образом, чтобы использовать все преимущества многопоточности на многоядерных процессорах от Intel и AMD [200]. Переделанный механизм Source обеспечивает более мощную поддержку игр Valve, таких как *Half Life 2*.

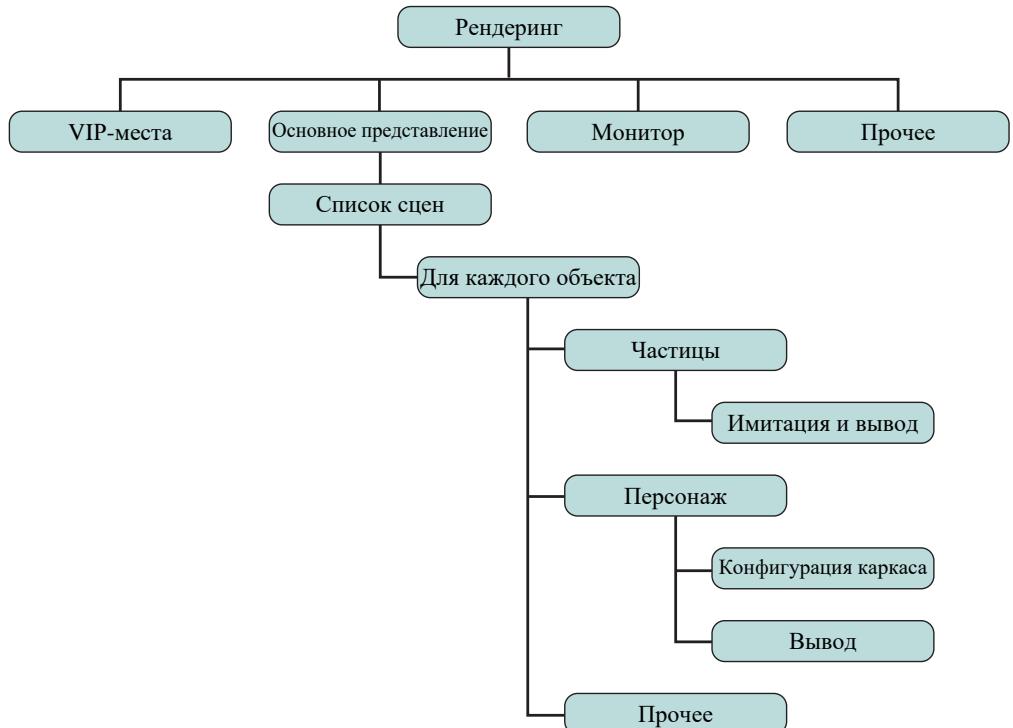
С точки зрения Valve имеются следующие варианты зернистости потоков [102].

- **Грубая многопоточность.** Отдельные модули, именуемые системами, назначаются отдельным процессорам. В случае механизма Source это будет означать рендеринг на одном процессоре, искусственный интеллект — на другом, физику — на третьем и так далее — простое и прямолинейное решение. В сущности, каждый модуль является однопоточным, а глобальная координация включает синхронизацию всех потоков с потоком временной шкалы.
- **Тонкая многопоточность.** Много похожих или идентичных задач распределено между несколькими процессорами. Например, цикл, выполняющий итерации над массивом данных, можно разделить на несколько небольших циклов в отдельных потоках, которые могут выполняться параллельно.
- **Гибридная многопоточность.** Данная многопоточность предполагает избирательное использование тонкой многопоточности для одних систем и однопоточность для других.

Компания Valve обнаружила, что при использовании грубой многопоточности можно достичь двукратного повышения производительности на двух процессорах по сравнению с выполнением на одном процессоре. Однако такой прирост производительности может быть достигнут только для искусственно созданных случаев. Для реальных игр улучшение оказывается примерно в 1,2 раза. Было также установлено, что эффективное

использование мелкозернистой многопоточности — достаточно трудная задача. Время выполнения единичного объема работы может быть переменным, так что глобальное управление временной шкалой приложения требует сложного программирования.

Выяснилось также, что гибридный подход является наиболее перспективным и обеспечивающим лучшее масштабирование для многоядерных систем с 8 или 16 процессорами. Valve определила системы, которые очень эффективно работают при назначении одному процессору. Примером может служить система микширования звука, которая мало взаимодействует с пользователями, не ограничена оконной конфигурацией и работает с собственным набором данных. Другие модули, такие как обеспечивающие рендеринг сцены, могут быть организованы в виде нескольких потоков так, что модуль может выполняться и на одном процессоре, но при этом чем больше процессоров ему доступно, тем большей оказывается его производительность.



**Рис. 4.9.** Гибридная многопоточность для модуля рендеринга

На рис. 4.9 показана структура потоков модуля рендеринга. В этой иерархической структуре потоки высокого уровня при необходимости создают потоки низкого уровня. Модуль основан на критической части механизма Source — на списке, который является представлением базы данных визуальных элементов в мире игры. Первая задача заключается в определении, какие области игрового мира должны быть отображены. Следующая задача состоит в выяснении, какие объекты находятся в сцене при рассмотрении ее под разными углами, с нескольких точек. Затем наступает время интенсивной работы процессора. Модуль рендеринга должен визуализировать каждый объект с нескольких точек зрения, таких как точка зрения игрока, монитора или отражения в воде.

Некоторые из ключевых элементов стратегии многопоточности для модуля рендеринга перечислены в [148] и включают следующее.

- Построение списков рендеринга сцен параллельно для нескольких сцен (например, мир и его отражение в воде).
- Симуляция перекрытия изображений.
- Параллельное вычисление трансформаций каркаса для всех персонажей во всех сценах.
- Разрешение параллельного вывода с помощью нескольких потоков.

Проектировщики обнаружили, что простая блокировка ключевых баз данных, таких как списки элементов мира, на уровне потоков слишком неэффективна. Более 95% времени поток пытается прочитать информацию из набора данных и не более 5% времени — выполнять запись. Таким образом, в этом случае эффективно работает механизм параллельных вычислений, известный как модель “один писатель — несколько читателей”.

## 4.4. УПРАВЛЕНИЕ ПРОЦЕССАМИ И ПОТОКАМИ В WINDOWS

Этот раздел начинается с обзора ключевых объектов и механизмов, которые поддерживают выполнение приложений в Windows. В оставшейся части раздела более подробно рассматривается управление процессами и потоками в Windows.

**Приложение** состоит из одного или нескольких процессов. Каждый **процесс** предоставляет ресурсы, необходимые для выполнения программы. Процесс имеет виртуальное адресное пространство, выполнимый код, открытые дескрипторы системных объектов, контекст безопасности, уникальный идентификатор процесса, переменные среды, класс приоритета, минимальный и максимальный размеры рабочего множества и по крайней мере один поток выполнения. Каждый процесс запускается с одним потоком, который часто называют основным или первичным, но любой из его потоков может создавать дополнительные потоки.

**Поток** является сущностью в рамках процесса, которая может быть запланирована к выполнению. Все потоки процесса разделяют его виртуальное адресное пространство и системные ресурсы. Кроме того, каждый поток поддерживает обработчики исключений, приоритет планирования, локальную память потока, уникальный идентификатор потока и набор структур, которые система будет использовать для сохранения контекста потока до тех пор, пока он не будет запущен планировщиком. В многопроцессорном компьютере система может одновременно выполнять столько потоков, сколько имеется процессоров.

**Объект задания** (*job object*) позволяет группировать процессы для управления ими как единым целым. Объекты заданий представляют собой именуемые, защищаемые, совместно используемые объекты, которые управляют атрибутами связанных с ними процессов. Операции, выполняемые над объектами заданий, влияют на все процессы, связанные с объектом задания. Примеры включают соблюдение ограничений, такие как размер рабочего множества или приоритет процесса, или завершение работы всех процессов, связанных с заданием.

**Пул потоков** — это коллекция рабочих потоков, которые эффективно выполняют асинхронные обратные вызовы от имени приложения. Пул потоков используется главным образом для сокращения числа потоков приложения и управления рабочими потоками.

**Волокно (fiber)** — это единица выполнения, которая должна планироваться приложением вручную. Волокна выполняются в контексте потоков, которые их планируют. Каждый поток может планировать несколько волокон. В общем случае волокна не предоставляют преимущества по сравнению с хорошо спроектированным многопоточным приложением. Однако применение волокон может упростить перенос приложений, которые были разработаны с применением планирования собственных потоков. С точки зрения системы волокно является тождественным потоку, который его выполняет. Например, если волокно обращается к локальной памяти потока, оно обращается к локальной памяти того потока, которые его выполняет. Кроме того, если волокно вызывает функцию `ExitThread`, то поток, который его выполняет, завершает свою работу. Однако волокно не имеет всей той же информации о состоянии, связанной с ним, что и информация, связанная с потоком. Единственная поддерживаемая волокном информация о состоянии — его стек, подмножество его регистров и данные волокна, предоставленные при его создании. Сохраненные регистры представляют собой набор регистров, обычно сохраняемых при вызове функции. Волокна не подлежат вытесняющему планированию. Поток выполняет планирование, переключаясь на одно волокно с другого. Система же по-прежнему занимается планированием выполнения потоков. Когда прерывается поток, выполняющий волокна, выполнение текущего волокна прерывается, но оно остается выбранным.

Пользовательский режим планирования (user-mode scheduling — UMS) представляет собой облегченный механизм, который приложения могут использовать для планирования собственных потоков. Приложение может переключаться между потоками UMS в пользовательском режиме без участия системного планировщика и восстанавливать контроль над процессором, если поток UMS блокируется в ядре. Каждый поток UMS имеет собственный контекст вместо совместного использования контекста единственного потока. Возможность переключения между потоками в пользовательском режиме делает UMS более эффективным, чем пул потоков для краткосрочной работы, которая требует нескольких системных вызовов. UMS полезен для приложений с высокими требованиями к производительности, которые должны эффективно выполнять несколько потоков одновременно на многопроцессорных или многоядерных системах. Чтобы воспользоваться преимуществами UMS, приложение должно реализовать компонент планировщика, который управляет UMS-потоками приложения и определяет, когда они должны быть выполнены.

## Управление фоновыми задачами и жизненным циклом приложений

Начиная с Windows 8 и до Windows 10 за управление состоянием отдельных приложений отвечают их разработчики. Предыдущие версии Windows всегда давали полный контроль над жизненным циклом процесса пользователю. В классической среде рабочего стола ответственность за закрытие приложения несет пользователь. При этом может открыться диалоговое окно, предлагающее сохранить сделанную работу. В новом интерфейсе Metro за процесс жизненного цикла приложения отвечает Windows. Хотя одновременно с пользовательским интерфейсом Metro с использованием функциональ-

ности SnapView параллельно с основным приложением может выполняться ограниченное количество приложений, в любой момент времени может выполняться только одно приложение Store. Это является прямым следствием нового дизайна. Windows Live Tiles показывает приложения, запущенные в системе, получая push-уведомления и не используя системные ресурсы для отображения предлагаемого динамического содержимого.

Приложение переднего плана в интерфейсе Metro имеет доступ ко всем процессорным, сетевым и дисковым ресурсам, доступным пользователю. Все прочие приложения приостанавливаются и не имеют доступа к этим ресурсам. Когда приложение входит в режим приостановки, должно быть запущено событие для сохранения состояния пользовательской информации. За это отвечает разработчик приложения. Windows может завершить фоновое приложение по ряду причин, например из-за потребности в ресурсах, или из-за тайм-аута. Это существенное отличие от предшествующих версий операционных систем Windows. Приложение должно сохранить любые введенные пользователем данные, измененные им настройки и т.д. Это означает, что вы должны сохранить состояние вашего приложения при его приостановке в случае, если Windows завершает его, так, чтобы восстановить его состояние позже. Когда приложение возвращается на передний план, запускается другое событие — для получения пользовательского состояния из памяти. При прекращении работы фонового приложения никакие события не инициируются. Вместо этого данные приложения остаются резидентными в системе (как будто приложение просто приостановлено) до тех пор, пока приложение не будет запущено снова. Пользователи находят приложение в том же состоянии, в котором они его оставили, независимо от того, было ли оно приостановлено или прекращено Windows или закрыто пользователем. Прикладные программисты могут использовать код для выяснения, следует ли восстанавливать сохраненное состояние приложения.

Некоторые приложения, такие как каналы новостей, могут смотреть на дату предыдущего выполнения и выбрасывать старые данные, освобождая место для новой информации. Это выяснение даты выполняется разработчиком приложения, а не операционной системой. Если приложение закрывает пользователя, то несохраненные данные теряются. Приложения переднего плана, занимающие все системные ресурсы и приводящие к голоданию фоновых приложений — печальная реальность Windows. Поэтому критически важным для успеха приложения в Windows является его разработка с учетом возможности изменений состояния.

Для обработки потребностей фоновых задач создан специальный интерфейс прикладного программирования (API) для фоновых задач, который позволяет приложениям выполнять небольшие задачи, пока они не находятся на переднем плане. В такой ограниченной среде приложения могут получать push-уведомления от сервера или пользователь может принять телефонный звонок. Push-уведомления являются шаблонными XML-строками. Они управляются посредством облачной службы, известной как служба уведомлений Windows (Windows Notification Service — WNS). Время от времени она будет передавать уведомления фоновым приложениям пользователя. API будет ставить эти запросы в очередь и обрабатывать их, когда будет получать достаточные для этого ресурсы процессора. Фоновые задачи серьезно ограничены в использовании процессора, получая только одну процессорную секунду из каждого процессорного часа. Такая стратегия гарантирует, что критические задачи будут получать необходимую им квоту ресурсов, но не гарантирует, что будут работать фоновые приложения.

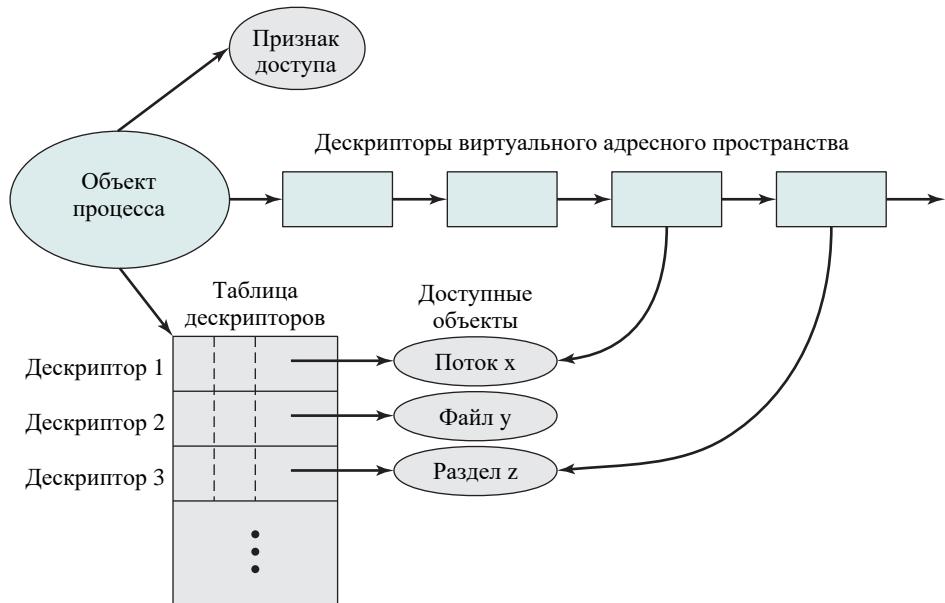
## Процессы в Windows

Важными характеристиками процессов Windows являются следующие.

- Процессы Windows реализованы как объекты.
- Процесс может быть создан как новый процесс или как копия существующего.
- Выполнимый процесс может содержать один или несколько потоков.
- Как объекты процессов, так и объекты потоков имеют встроенные возможности синхронизации.

На рис. 4.10 (основанном на [212]) проиллюстрирован способ взаимосвязи между процессом и ресурсами, которыми этот процесс управляет или которые использует. В целях безопасности каждому процессу присваивается признак доступа, который называется первичным токеном (token) или маркером процесса. При входе пользователя в Windows создается маркер доступа, в который входит идентификатор безопасности пользователя. Каждый процесс, который создается данным пользователем или запускается от его имени, содержит копию этого маркера. Указанный маркер используется операционной системой Windows для проверки возможности доступа пользователя к защищенным объектам или возможности выполнения специальных функций в системе и над защищенными объектами. Маркер доступа управляет возможностью изменения процессом собственных атрибутов. В этом случае процесс не имеет дескриптора, открытого для его маркера доступа. Если процесс попытается открыть такой дескриптор, система безопасности определит, разрешено ли это, а значит, может ли процесс изменить свои атрибуты.

С процессами связан и ряд блоков, которые определяют виртуальное адресное пространство, закрепленное в данный момент за процессом. Процесс не может непосредственно изменять эти структуры; в этом он должен полагаться на менеджер виртуальной памяти, предоставляющий сервис выделения памяти процессу.



**Рис. 4.10.** Процессы Windows и их ресурсы

Наконец, процесс включает таблицу объектов, которая управляет другими объектами, известными процессу. На рис. 4.10 показан один поток, но их может быть много. Кроме того, процесс имеет доступ к объектам файлов и разделов, которые определяют раздел совместно используемой памяти.

## Объекты процессов и потоков

Объектно-ориентированная структура операционной системы Windows облегчает разработку подсистемы общего назначения для работы с процессами. Windows использует два типа связанных с процессами объектов: процессы и потоки. Процесс — это объект, соответствующий заданию или приложению пользователя, который владеет собственными ресурсами, такими как память и открытые файлы. Поток — это диспетчеризуемая единица работы, которая выполняется последовательно и является прерываемой, что позволяет процессору переключаться на выполнение другого потока.

Каждый процесс в операционной системе Windows представлен объектом. Каждый объект процесса включает некоторое количество атрибутов и инкапсулирует ряд действий, или служб, которые он может выполнять. Процесс будет выполнять действия при вызове через ряд опубликованных методов интерфейса. При создании нового процесса операционная система Windows использует класс или тип объекта, определенный как шаблон процесса для генерации новых экземпляров объектов процессов Windows. Во время создания объекта его атрибутам присваиваются конкретные значения. В табл. 4.3 приводится краткое описание каждого атрибута процессов.

**Таблица 4.3. Атрибуты объекта процесса Windows**

<b>Идентификатор процесса</b>	Уникальное значение, идентифицирующее процесс в операционной системе
<b>Дескриптор защиты</b>	Описывает, кто создал объект, кто обладает правом доступа к нему или может им пользоваться и кто определяет права доступа к объекту
<b>Базовый приоритет</b>	Базовый приоритет выполнения потоков, принадлежащих процессу
<b>Процессор по умолчанию</b>	Заданный по умолчанию набор процессоров, на котором возможно выполнение потоков процесса
<b>Квоты</b>	Максимальное количество страничной и прочей системной памяти, объем в страничном файле и процессорное время, доступные данному процессу
<b>Время выполнения</b>	Суммарное время, затраченное на выполнение всех потоков процесса
<b>Счетчики ввода-вывода</b>	Переменные, в которые заносятся сведения о количестве и типе операций ввода-вывода, выполненных потоками процесса
<b>Счетчики операций с виртуальной памятью</b>	Переменные, в которые заносятся сведения о количестве и типе операций с виртуальной памятью, выполненных потоками процесса
<b>Порты исключений/отладки</b>	Каналы обмена информацией между процессами, в которые диспетчер процессов должен отправить сообщение при возникновении исключительной ситуации из-за одного из потоков процесса
<b>Состояние выхода</b>	Причина завершения процесса

Процесс Windows перед выполнением должен содержать хотя бы один поток, который затем может создавать другие потоки. В многопроцессорной системе несколько потоков одного и того же процесса могут выполняться параллельно. В табл. 4.4 определены атрибуты объекта потока. Заметим, что некоторые атрибуты потока подобны атрибутам процесса. Значения таких атрибутов потока извлекаются из значений соответствующих атрибутов процесса. Например, в многопроцессорной системе *сродные потоку процессоры* — это множество процессоров, на которых может выполняться данный поток; это множество совпадает с множеством *процессоров, сродных процессу*, или является его подмножеством.

Обратите внимание, что одним из атрибутов процесса является его контекст, который содержит значения регистров процессора при последнем выполнении потока. Эта информация позволяет операционной системе приостанавливать и возобновлять потоки. Более того, приостановив поток и изменив его контекст, можно изменить его поведение.

**Таблица 4.4. Атрибуты объекта потока Windows**

<b>Идентификатор потока</b>	Уникальное значение, идентифицирующее поток, когда он вызывает сервис
<b>Контекст потока</b>	Набор значений регистров и другие данные, которыми определяется состояние выполнения потока
<b>Динамический приоритет</b>	Приоритет выполнения потока в данный момент времени
<b>Базовый приоритет</b>	Нижний предел динамического приоритета потока
<b>Процессоры потока</b>	Множество процессоров, на которых может выполняться поток. Это множество является подмножеством процессоров, сродных процессу потока, или совпадает с ним
<b>Время выполнения потока</b>	Совокупное время, затраченное на выполнение потока в пользовательском режиме и в режиме ядра
<b>Статус оповещения</b>	Флаг, который указывает, следует ли потоку выполнять асинхронный вызов процедуры
<b>Счетчик приостановок</b>	В нем указывается, сколько раз выполнение потока было приостановлено без последующего возобновления
<b>Признак имперсонации</b>	Временный признак доступа, позволяющий потоку выполнять операции от имени другого процесса (используется подсистемами)
<b>Порт завершения</b>	Канал обмена информацией между процессами, на который диспетчер процессов должен отправить сообщение при завершении потока (используется подсистемами)
<b>Состояние выхода потока</b>	Причина завершения потока

## МНОГОПОТОЧНОСТЬ

Операционная система Windows поддерживает параллельное выполнение процессов, потому что потоки различных процессов могут выполняться параллельно (демонстрируя одновременное выполнение своей работы). Более того, нескольким потокам одного

и того же процесса могут быть выделены различные процессоры, и эти потоки также могут выполняться одновременно. Параллелизм достигается в многопоточном процессе без накладных расходов на использование нескольких процессов. Потоки одного и того же процесса могут обмениваться между собой информацией с помощью общего адресного пространства и имеют доступ к совместным ресурсам процесса. Потоки, принадлежащие разным процессам, могут обмениваться между собой информацией с помощью общей области памяти, установленной для этих двух процессов.

Объектно-ориентированный многопоточный процесс является эффективным средством реализации серверных приложений. Например, один обслуживающий процесс может обслуживать несколько клиентов. Каждый запрос клиента приводит к созданию в сервере нового потока.

## СОСТОЯНИЯ ПОТОКОВ

Поток, созданный в Windows, может находиться в одном из шести состояний (рис. 4.11). Перечислим эти состояния.

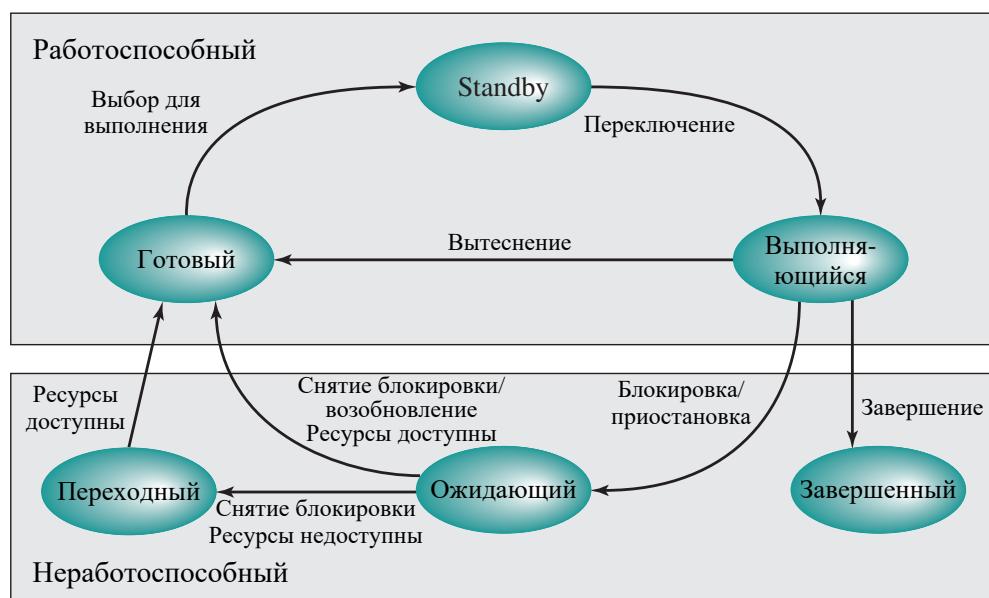


Рис. 4.11. Состояния потоков в операционной системе Windows

- Готовый к выполнению.** Поток, который может быть направлен на выполнение. Диспетчер ядра отслеживает все готовые к выполнению потоки и осуществляет их планирование в соответствии с приоритетом.
- Резервный.** Поток, который будет запущен следующим на данном процессоре. Поток находится в этом состоянии до тех пор, пока процессор не освободится. Если приоритет резервного потока достаточно высок, то он может вытеснить выполняющийся в данный момент поток. В противном случае резервный поток ждет, пока произойдет блокировка выполняющегося потока или пока истечет выделенный ему промежуток времени.

3. **Выполняющийся.** Как только диспетчер ядра выполнит переключение потоков, резервный поток перейдет в состояние выполнения и будет пребывать в нем до тех пор, пока не произойдет одно из следующих событий: поток будет вытеснен, закончится отведенный ему интервал времени, поток будет блокирован или завершен. В первых двух случаях поток снова переходит в состояние готовности.
4. **Ожидаящий.** Поток входит в состояние ожидания, если 1) он блокирован каким-то событием (например, операцией ввода-вывода), 2) он добровольно ждет синхронизации или 3) среда подсистемы предписывает потоку, чтобы он сам себя приостановил. После того как условия ожидания удовлетворены, поток переходит в состояние готовности, если все его ресурсы доступны.
5. **Переходный.** Поток переходит в это состояние, если он готов к выполнению, но ресурсы недоступны (например, страницы стека потока могут быть сброшены на диск). После того как необходимые ресурсы станут доступными, процесс передает в состояние готовности.
6. **Завершенный.** Завершение потока может быть инициировано самим потоком либо другим потоком или может произойти вместе с завершением родительского процесса. После завершения необходимых операций освобождения ресурсов и тому подобного поток удаляется из системы (или может быть сохранен исполнительной системой<sup>6</sup> для дальнейшей повторной инициализации).

## Поддержка подсистем операционной системы

Средства общего назначения для работы с процессами и потоками должны поддерживать структуры процессов и потоков, соответствующие различным средам операционной системы. В обязанности каждой подсистемы входит использование функций обработки процессов и потоков операционной системы Windows для эмуляции функций обработки процессов и потоков соответствующей подсистемы операционной системы. Система управления процессами и потоками довольно сложна; здесь мы приводим лишь ее краткий обзор.

Процесс создается по запросу приложения операционной системы, который поступает в соответствующую защищенную подсистему. Подсистема, в свою очередь, отправляет запрос на создание процесса исполнительной системе Windows, которая создает объект-процесс и возвращает подсистеме его дескриптор. Создавая процесс, операционная система Windows не создает поток автоматически (в отличие от Win32, создание нового процесса в которой всегда сопровождается созданием потока). Поэтому подсистема Win32 повторно обращается к менеджеру процессов Windows, чтобы создать поток нового процесса и получить его дескриптор. Затем соответствующая информация о потоке и процессе возвращается приложению. В POSIX потоки не поддерживаются. Поэтому подсистема POSIX получает поток для нового процесса от Windows, так что процесс может быть активирован, но приложению возвращается только информация о процессе. Тот факт, что процесс POSIX реализуется с помощью как процесса, так и потока исполнительной системой Windows, для приложения остается незамеченным.

<sup>6</sup> Исполнительная система Windows описана в главе 2, “Обзор операционных систем”. Она включает базовые службы операционной системы, такие как управление памятью, процессами и потоками, безопасностью, вводом-выводом и межпроцессным взаимодействием.

Когда исполнительной подсистемой создается новый процесс, он наследует многие атрибуты создавшего его процесса. Однако в среде Win32 процесс создается непрямым образом. Процесс клиентского приложения генерирует запрос на создание процесса в адрес подсистемы Win32; подсистема, в свою очередь, отправляет запрос на создание процесса исполнительной системе Windows. Так как новый процесс должен наследовать характеристики процесса-клиента, а не обслуживающего процесса, Windows позволяет подсистеме указывать родительский процесс нового процесса. Затем новый процесс наследует маркер доступа, квоты, базовый приоритет и принятые по умолчанию сродство процессоров родительского процесса.

## 4.5. УПРАВЛЕНИЕ ПОТОКАМИ И SMP В SOLARIS

В операционной системе Solaris реализован многоуровневый подход к управлению потоками, способствующий значительной гибкости использования процессорных ресурсов.

### Многопоточная архитектура

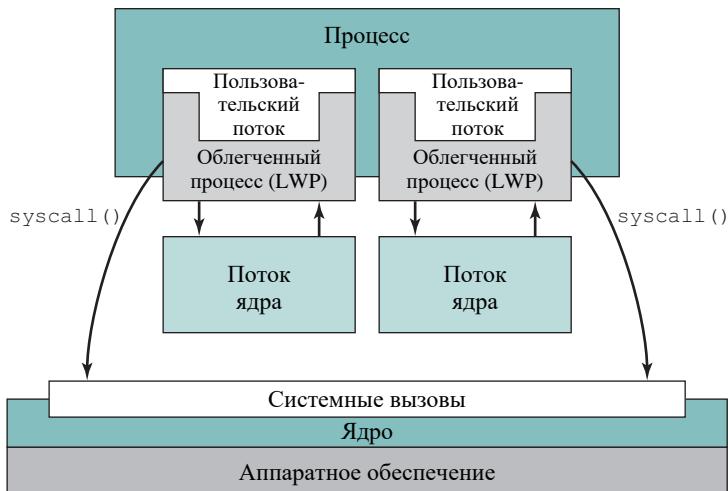
В операционной системе Solaris используются четыре отдельные концепции, связанные с потоками.

- Процесс.** Это обычный процесс UNIX, который включает в себя пользовательское адресное пространство, стек и управляющий блок процесса.
- Потоки на пользовательском уровне.** Эти потоки реализуются с помощью библиотеки потоков в адресном пространстве процесса; они невидимы для операционной системы. Потоки на пользовательском уровне (user-level thread —ULT<sup>7</sup>) играют роль интерфейса для параллелизма приложений.
- Облегченные процессы.** Облегченный процесс (lightweight process — LWP) можно рассматривать как отображение между потоками на пользовательском уровне и потоками ядра. Каждый из облегченных процессов поддерживает один или несколько потоков на пользовательском уровне и отображает их в один поток ядра. Планирование облегченных процессов производится ядром независимо. В много-процессорной системе облегченные процессы могут выполняться параллельно на нескольких процессорах.
- Потоки ядра.** Эти потоки являются фундаментальными элементами; планирование и выполнение каждого из них может осуществляться на одном из системных процессоров.

На рис. 4.12 проиллюстрирована взаимосвязь между этими четырьмя элементами. Заметим, что каждому облегченному процессу всегда соответствует один поток ядра. Облегченный процесс видим для приложения в рамках процесса. Таким образом, структуры данных облегченного процесса существуют в рамках адресного пространства соответствующего им процесса. В то же время каждый облегченный процесс связан с единственным диспетчеризуемым потоком ядра, а структуры данных этого потока ядра поддерживаются в адресном пространстве ядра.

---

<sup>7</sup> Эта аббревиатура используется только в данной книге; ее нет в литературе, посвященной Solaris.



**Рис. 4.12.** Процессы и потоки в Solaris [167]

Процесс может состоять из одного ULT, связанного с одним LWP. В этом случае имеется единственный поток выполнения, соответствующий традиционному процессу UNIX. Если в рамках одного процесса параллелизм не требуется, приложение использует эту структуру процесса. Если же приложению нужен параллелизм, его процесс содержит несколько потоков, каждый из которых связан с одним LWP, который, в свою очередь, связан с одним потоком ядра.

Кроме того, имеются потоки ядра, не связанные с LWP. Ядро создает, выполняет и уничтожает эти потоки ядра для выполнения определенных системных функций. Использование потоков ядра вместо процессов ядра для реализации системных функций уменьшает накладные расходы переключения в ядре (из процесса в поток).

## Мотивация

Трехуровневая структура потока (ULT, LWP, поток ядра) в Solaris предназначена для облегчения управления потоками со стороны операционной системы и обеспечения ясного интерфейса для приложений. ULT-интерфейс может быть библиотекой стандартных потоков. Определенные ULT отображаются на LWP, которые управляются операционной системой; те же взаимно однозначно связаны с потоками ядра. Таким образом, управление параллелизмом и выполнением происходит на уровне потоков ядра.

Кроме того, приложение имеет доступ к аппаратному обеспечению через интерфейс прикладного программирования, состоящий из системных вызовов. Этот API позволяет пользователю вызывать службы ядра для выполнения от имени вызывающего процесса привилегированных задач, таких как чтение или запись файлов, выдача команд управления устройствами, создание новых процессов или потоков, выделение памяти для процессов и т.д.

## Структура процессов

На рис. 4.13 приведено общее сравнение структуры процессов в традиционной операционной системе UNIX со структурой процессов в операционной системе Solaris.

В типичных реализациях UNIX в структуру процесса входят такие составляющие:

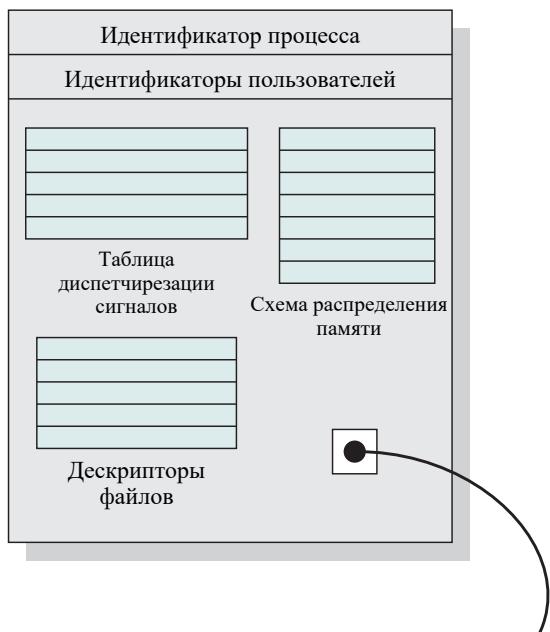
- идентификатор процесса,
- идентификаторы пользователя,
- таблица диспетчирования сигналов, которую ядро использует для того, чтобы принимать решение о том, что должно быть сделано при отправке сигнала процессу;
- дескрипторы файлов, описывающие состояние используемых процессом файлов;
- схема распределения памяти, определяющая адресное пространство процесса;
- структура состояния процессора, включающая стек ядра для процесса.

В операционной системе Solaris эта базовая структура остается, но в ней блок состояния процессора заменен списком структур, в котором для каждого облегченного процесса имеется свой блок данных.

Структура процесса в UNIX



Структура процесса в Solaris



LWP 2

Идентификатор LWP
Приоритет
Маска сигналов
Регистры
Стек
• • •

LWP 1

Идентификатор LWP
Приоритет
Маска сигналов
Регистры
Стек
• • •

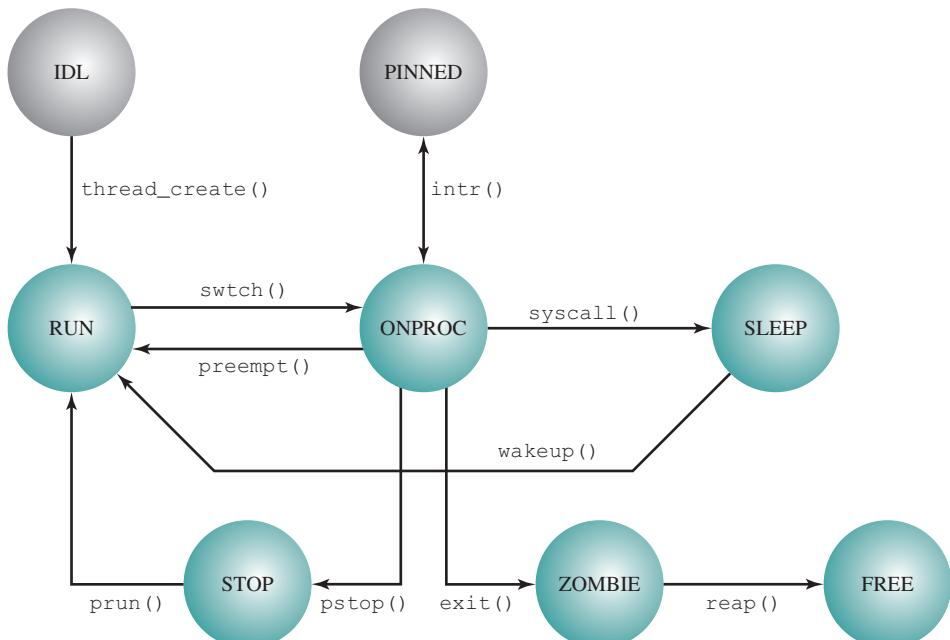
Рис. 4.13. Структура процесса в традиционном UNIX и в Solaris [154]

В структуру данных облегченного процесса входят такие элементы:

- идентификатор облегченного процесса;
- приоритет данного облегченного процесса (и следовательно, потока ядра, который его поддерживает);
- маска сигналов, предоставляющая ядру информацию о том, какие сигналы могут быть восприняты процессом;
- сохраненные значения регистров пользовательского уровня (когда облегченный процесс не выполняется);
- стек ядра данного облегченного процесса, в который входят аргументы системного вызова, результаты и коды ошибок каждого уровня;
- данные по использованию ресурсов и профилированию;
- указатель на соответствующий поток ядра;
- указатель на структуру процесса.

## Выполнение потоков

На рис. 4.14 показана упрощенная схема состояний выполнения потоков. Эти состояния отражают статус выполнения как потоков ядра, так и связанных с ними облегченных процессов. Как упоминалось ранее, некоторые потоки ядра не связаны с облегченными процессами; к ним применима та же диаграмма.



**Рис. 4.14.** Состояния потоков Solaris

Перечислим возможные состояния.

- **RUN:** поток работоспособен и готов к выполнению.
- **ONPROC:** поток выполняется процессором.
- **SLEEP:** поток заблокирован.
- **STOP:** поток остановлен.
- **ZOMBIE:** поток завершен.
- **FREE:** ресурсы потока освобождены и поток ожидает удаления из структуры данных потоков операционной системы.
- **PINNED:** контекст сохраняется, и поток приостанавливается до тех пор, пока не будет обработано прерывание.

Поток перемещается из состояния ONPROC в RUN при вытеснении потоком с более высоким приоритетом или из-за завершения кванта времени. Поток перемещается из состояния ONPROC в SLEEP, если он заблокирован и должен ожидать некоторого события для возвращения в состояние RUN. Блокирование выполняется, если поток осуществлял системный вызов и должен ожидать его завершения. Поток входит в состояние STOP, если его процесс останавливается; это может быть сделано в целях отладки.

## Прерывания в роли потоков

В большинстве операционных систем приняты две основные формы параллельной деятельности: процессы и прерывания. Процессы (или потоки) взаимодействуют один с другим и управляют использованием совместных структур данных с помощью различных примитивов, обеспечивающих взаимоисключения (когда в каждый момент времени только один процесс может выполнять определенный код или осуществлять доступ к определенным данным) и синхронизирующих их выполнение. Прерывания синхронизируются путем предотвращения их обработки на некоторое время. В операционной системе Solaris эти две концепции объединяются в одной модели потоков ядра; прерывания в такой модели преобразуются в потоки ядра.

Эти преобразования выполняются для сокращения накладных расходов. Обработчики прерываний часто манипулируют данными, которые используются совместно с остальной частью ядра. Поэтому во время работы процедуры ядра, осуществляющей доступ к этим данным, прерывания должны быть заблокированы, даже если большинство прерываний не оказывают влияния на эти данные. Обычно для этого приходится повышать уровень приоритета прерываний, чтобы блокировать прерывания на время выполнения подпрограммы. После завершения подпрограммы уровень приоритета понижается. Все эти операции отнимают время. В многопроцессорной системе проблема усиливается. Ядро должно защищать большее количество объектов, и ему может понадобиться блокировать прерывания на всех процессорах.

Решение, принятное в операционной системе Solaris, выглядит так.

1. Для обработки прерываний в системе Solaris используются потоки ядра. Как и любой другой поток ядра, поток прерывания обладает собственным идентификатором, приоритетом, контекстом и стеком.
2. Ядро управляет доступом к структурам данных и синхронизирует потоки прерываний с помощью примитивов взаимоисключений (рассматривающихся в главе 5,

“Параллельные вычисления: взаимоисключения и многозадачность”). Таким образом, для обработки прерываний используются обычные методы синхронизации потоков.

3. Потокам прерываний присваиваются более высокие приоритеты, чем всем другим типам потоков ядра.

Если происходит прерывание, оно передается определенному процессору, а выполняющийся на этом процессоре поток закрепляется. Закрепленный (pinned) поток не может перейти на другой процессор; его контекст сохраняется, и процесс приостанавливается до тех пор, пока не будет обработано прерывание. После этого процессор приступает к выполнению потока прерывания. В наличии всегда имеется запас деактивированных потоков прерываний, так что новый поток создавать не нужно. Затем выполняется поток, в котором происходит обработка прерывания. Если программе обработки понадобится доступ к структуре данных, которая каким-то образом заблокирована и используется другим потоком, поток прерывания должен ждать. Поток прерывания может быть вытеснен только другим потоком прерывания с более высоким приоритетом.

Опыт использования потоков прерываний в операционной системе Solaris свидетельствует о том, что такой подход обеспечивает производительность, превосходящую производительность традиционных методов обработки прерываний [132].

## 4.6. УПРАВЛЕНИЕ ПРОЦЕССАМИ И ПОТОКАМИ В LINUX

### Задания Linux

В операционной системе Linux процесс, или задание, представляется структурой данных `task_struct`. В этой структуре данных информация разбита на следующие категории.

- **Состояние.** Состояние выполнения процесса (выполняющийся, готовый к выполнению, приостановленный, остановленный, зомби).
- **Информация по планированию.** Информация, которая нужна операционной системе Linux для планирования процессов. Процесс может быть обычным или выполняющимся в реальном времени; кроме того, он обладает некоторым приоритетом. Процессы, выполняющиеся в реальном времени, планируются до обычных процессов; в каждой из категорий можно использовать относительные приоритеты. Счетчик ведет отсчет времени, отведенного процессу.
- **Идентификаторы.** Каждый процесс обладает собственным уникальным идентификатором процесса (PID), а также идентификаторами пользователя и группы. Идентификатор группы применяется для того, чтобы назначить группе пользователя права доступа к ресурсам.
- **Обмен информацией между процессами.** В операционной системе Linux используется такой же механизм межпроцессного взаимодействия (IPC), как и в операционной системе UNIX SVR4, описанной в главе 6, “Параллельные вычисления: взаимоблокировка и голодание”.

- Связи.** Каждый процесс содержит в себе связи с параллельными ему процессами, с родственными ему процессами (с которыми он имеет общий родительский процесс) и со всеми своими дочерними процессами.
  - Время и таймеры.** Сюда входят время создания процесса, а также количество процессорного времени, затраченного на данный процесс. С процессом также могут быть связаны интервальные таймеры (один или несколько). Интервальный таймер задается в процессе с помощью системного вызова; после истечения периода таймера процессу отправляется соответствующий сигнал. Таймер может быть создан для одноразового или периодического использования.
  - Файловая система.** Содержит в себе указатели на все файлы, открытые данным процессом.
  - Адресное пространство.** Определяет отведенную данному процессу виртуальную память.
  - Контекст, зависящий от процессора.** Информация по регистрам и стеку, составляющая контекст данного процесса.
- На рис. 4.15 показаны состояния выполнения процесса.
- Выполняющийся.** Это состояние отвечает на самом деле двум состояниям: текущий процесс либо выполняется, либо готов к выполнению.
  - Прерываемый.** Это состояние блокировки, в котором процесс ожидает наступления события, например завершения операции ввода-вывода, освобождения ресурса или сигнала от другого процесса.

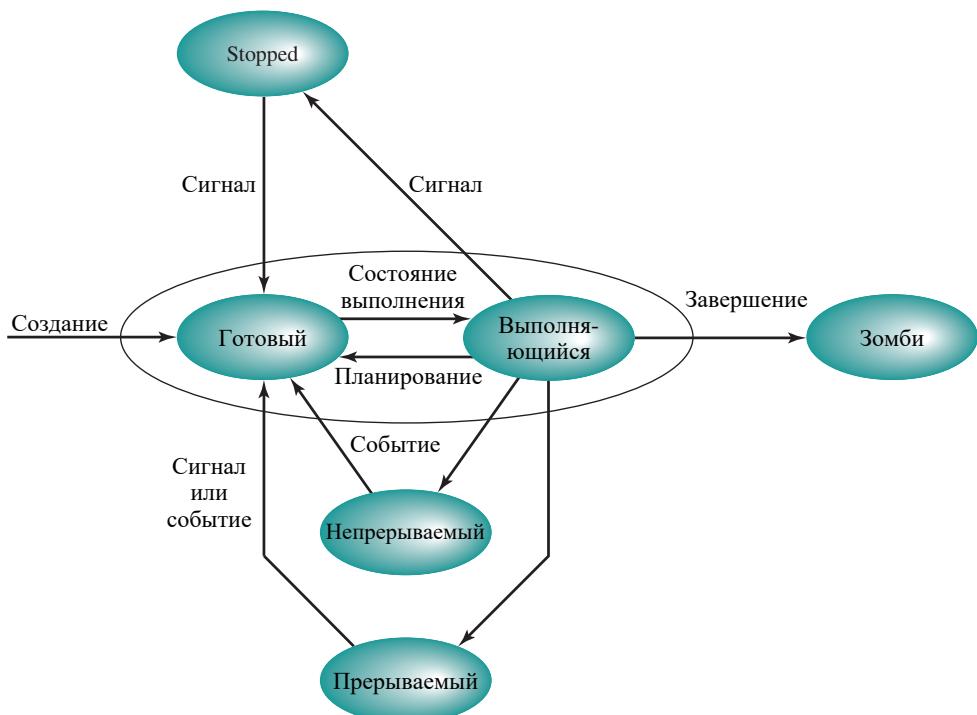


Рис. 4.15. Модель процессов и потоков Linux

- **Непрерываемый.** Это состояние блокировки другого рода. Его отличие от предыдущего состоит в том, что в непрерываемом состоянии процесс непосредственно ожидает выполнения какого-то аппаратного условия, поэтому он не воспринимает никаких сигналов.
- **Остановленный.** Процесс был остановлен и может быть продолжен только при соответствующем воздействии другого процесса. Например, процесс, который находится в состоянии отладки, может перейти в состояние остановки.
- **Зомби.** Процесс был прекращен, но по какой-то причине его структура остается в таблице процессов.

## Потоки Linux

Традиционные UNIX-системы поддерживают один поток выполнения для каждого процесса, в то время как современные UNIX-системы обычно предоставляют поддержку нескольких потоков на уровне ядра для каждого процесса. Как и традиционные системы UNIX, старые версии ядра Linux не поддерживают многопоточность. Вместо этого приложения необходимо было разрабатывать с набором функций библиотеки пользовательского уровня (наиболее популярной из которых является библиотека *pthread* (POSIX thread), причем все потоки отображаются на один процесс уровня ядра<sup>8</sup>). Мы видели, что современные версии UNIX предлагают потоки на уровне ядра. Linux предоставляет уникальное решение, состоящее в том, что оно не признает различие между потоками и процессами. Используя механизм, похожий на облегченные процессы Solaris, пользовательские потоки отображаются на процессы уровня ядра. Несколько потоков пользователяского уровня, которые составляют единый процесс уровня пользователя, отображаются на процессы уровня ядра Linux, которые разделяют один и тот же идентификатор группы. Это позволяет данным процессам совместно использовать ресурсы, такие как файлы и память, и избегать необходимости переключения контекста, когда планировщик выполняет переключение между процессами в одной и той же группе.

В Linux новый процесс создается путем копирования атрибутов текущего процесса. Можно клонировать новый процесс так, что он будет разделять такие ресурсы, как файлы, обработчики сигналов и виртуальная память. Когда два процесса разделяют одну и ту же виртуальную память, они функционируют как потоки в пределах одного процесса. Однако для потока не определяется отдельный тип структуры данных. Вместо обычной функции `fork()` процессы создаются в Linux с помощью вызова `clone()`. Эта команда включает в себя в качестве аргументов набор флагов. Традиционный системный вызов `fork()` реализуется в Linux как системный вызов `clone()` со сброшенными флагами.

Флаги клонирования включают следующие.

- `CLONE_NEWPID`: создание нового пространства имен PID.
- `CLONE_PARENT`:зывающий процесс и новое задание разделяют один и тот же родительский процесс.

---

<sup>8</sup> POSIX (Portable Operating Systems based on UNIX — переносимые операционной системы на основе UNIX) представляет собой стандарт интерфейса прикладного программирования IEEE, включающий стандарт потокового API. Библиотеки, реализующие стандарт POSIX Threads, часто именуются *Pthread*. Обычно они используются в UNIX-подобных системах POSIX, таких как Linux и Solaris, но имеются их реализации и для Microsoft Windows.

- **CLONE\_SYSVSEM**: применение семантики `SEM_UNDO` из System V.
- **CLONE\_THREAD**: вставка процесса в ту же группу потоков, что и родительский процесс. Если этот флаг имеет значение `true`, он неявно обеспечивает установку флага `CLONE_PARENT`.
- **CLONE\_VM**: совместное использование адресного пространства (дескриптор памяти и все таблицы страниц).
- **CLONE\_FS**: совместное использование информации о файловой системе (включая текущий рабочий каталог, корень файловой системы и маску `umask`).
- **CLONE\_FILES**: совместное использование таблицы дескрипторов файлов. Создание или закрытие файлового дескриптора распространяется на другой процесс, так же как и изменения флагов файлового дескриптора с помощью системного вызова `fcntl()`.

Когда ядро Linux выполняет переключение контекста от одного процесса к другому, оно проверяет, является ли адрес каталога страниц текущего процесса тем же, что и у планируемого процесса. Если да, то они разделяют одно адресное пространство, так что переключение контекста в основном состоит просто в переходе из одного места в коде в другое.

Хотя клонированные процессы, которые являются частью одной и той же группы процессов, могут разделить одно и то же пространство памяти, они не могут использовать одни и те же пользовательские стеки. Таким образом, вызов `clone()` создает отдельный стек для каждого процесса.

## Пространства имен Linux

С каждым процессом в Linux связан набор **пространств имен** (namespace). Пространство имен позволяет процессу (или нескольким процессам, которые разделяют одно и то же пространство имен) иметь представление системы, отличное от такого у других процессов, которые имеют другие связанные с ними пространства имен. Пространства имен и cgroups<sup>9</sup> (которые будут описаны в следующем разделе) являются основой облегченной виртуализации Linux, которая представляет собой функциональную возможность, предоставляющую процессу или группе процессов иллюзию, что они являются единственными процессами в системе. Эта функция широко используется проектами Linux Containers. В настоящее время имеется шесть пространств имен в Linux: `mnt`, `pid`, `net`, `ipc`, `uts` и `user`.

Пространства имен создаются путем системного вызова `clone()`, который получает в качестве параметра один из шести флагов пространств имен (`CLONE_NEWNS`, `CLONE_NEWPID`, `CLONE_NEWNET`, `CLONE_NEWIPC`, `CLONE_NEWUTS` и `CLONE_NEWUSER`). Процесс может также создать пространство имен с помощью системного вызова `unshare()` с одним из этих флагов; в отличие от вызова `clone()` в этом случае новый процесс не создается; создается только новое пространство имен, которое присоединяется к вызывающему процессу.

---

<sup>9</sup> Cgroups (от англ. control group) — механизм ядра Linux, который ограничивает и изолирует вычислительные ресурсы (процессорные, сетевые, ресурсы памяти, ресурсы ввода-вывода) для групп процессов. — Примеч. пер.

## Пространство имен MNT

Данное пространство имен обеспечивает процесс некоторым представлением иерархии файловой системы таким образом, что два процесса с различными пространствами имен mnt будут видеть различные иерархии файловых систем. Все файловые операции, которые выполняет процесс, применяются только к файловой системе, видимой процессу.

## Пространство имен UTS

Пространство имен UTS (UNIX timesharing) связано с системным вызовом Linux `uname()`. Этот вызов возвращает имя и информацию о текущем ядре, включая имя системы в рамках некоторой определяемой реализацией сети, а также имя домена NIS. NIS (Network Information Service — сетевая информационная служба) представляет собой стандартную схему, используемую всеми крупными UNIX и UNIX-подобными системами. Она позволяет группе машин в домене NIS совместно использовать общий набор конфигурационных файлов, что, в свою очередь, позволяет системному администратору настроить клиентские системы NIS с использованием только минимальной конфигурации данных и добавлять, удалять или изменять данные конфигурации из одного местоположения. Применение пространства имен UTS позволяет варьировать параметры инициализации и настройки для различных процессов в одной и той же системе.

## Пространство имен IPC

Пространство имен IPC изолирует некоторые ресурсы межпроцессного взаимодействия (IPC), такие как семафоры, POSIX-очереди сообщений и многое другое. Таким образом, программист может использовать механизмы параллелизма, обеспечивающие межпроцессное взаимодействие между процессами, которые разделяют одно и то же пространство имен IPC.

## Пространство имен PID

Пространства имен PID изолируют пространства идентификаторов процессов, так что процессы в разных пространствах имен PID могут иметь одинаковые PID. Эта возможность используется в программном инструментарии Linux Checkpoint/Restore In Userspace (CRIU). Используя этот инструмент, можно приостановить запущенное приложение (или его часть) и сбросить его на жесткий диск в виде набора файлов. Затем можно использовать эти файлы для восстановления и запуска приложения на этом же компьютере или на другом узле для продолжения его работы. Отличительной особенностью проекта CRIU является то, что он в основном реализован в пространстве пользователя (после ряда неудачных попыток его реализации в основном в ядре).

## Сетевое пространство имен

Сетевые пространства имен обеспечивают изоляцию системных ресурсов, связанных с сетью. Таким образом, каждое сетевое пространство имен имеет собственные сетевые устройства, IP адреса, таблицы маршрутизации, номера портов и т.д. Эти пространства имен виртуализируют весь доступ к сетевым ресурсам, что позволяет каждому процессу или группе процессов, принадлежащим к такому сетевому пространству имен, при необходимости получать доступ к сети. В любой момент времени сетевое устройство принадлежит только одному сетевому пространству имен. Кроме того, сокет также может принадлежать только одному пространству имен.

## Пользовательское пространство имен

Пользовательские пространства имен предоставляют контейнер со своим набором UID, полностью отделенным от таковых в родительском наборе. Таким образом, когда процесс клонирует новый процесс, он может назначить новое пользовательское пространство имен, а также новое пространство имен PID и все другие пространства имен. Клонированный процесс может иметь доступ ко всем ресурсам родительского процесса и соответствующие привилегии или подмножество ресурсов и привилегий родительского процесса. Пользовательские пространства имен считаются уязвимыми в смысле безопасности, так как позволяют создавать непривилегированные контейнеры (процессы, которые создаются пользователем, не являющимся корневым).

## Подсистема *Linux cgroup*

Подсистема *Linux cgroup* вместе с подсистемой пространств имен являются основой для виртуализации облегченных процессов, и таким образом, они формируют основу контейнеров *Linux*; почти каждый современный проект контейнеров *Linux* (такой, как *Docker*, *LXC*, *Kubernetes* и др.) основан на них обоих. Подсистема *cgroups* в *Linux* обеспечивает управление и учет ресурсов. Она обрабатывает такие ресурсы, как процессор, сеть, память и многое другое; главным образом это необходимо на “концах спектра” (во встраиваемых устройствах и серверах) и гораздо меньше — в настольных компьютерах. Разработка *cgroups* была начата в 2006 году инженерами *Google* под названием “контейнеры процессов”, которое позднее было изменено на “*cgroups*”, чтобы избежать путаницы с контейнерами *Linux*. Для реализации *cgroups* были добавлены не новые системные вызовы, а новая виртуальная файловая система (*VFS*), “*cgroups*” (которую также иногда называют *cgroupfs*), так как все ее операции основаны на файловой системе. Новая версия *cgroups*, под названием “*cgroups v2*”, была выпущена в версии ядра 4.5 (март 2016). Подсистема *cgroup v2* решает многие проблемы несоответствия среди контроллеров *cgroup v1* и делает *cgroup v2* лучше организованной путем применения строгих и последовательных интерфейсов.

В настоящее время имеется 12 контроллеров *cgroup v1* и 3 контроллера *cgroup v2* (памяти, ввода-вывода и PID); в настоящее время ведется работа над другими контроллерами *v2*.

Для того чтобы использовать файловую систему *cgroups* (т.е. просматривать ее, добавлять задания в *cgroups* и т.д.), сначала ее следует смонтировать, как при работе с почти любой другой файловой системой. Файловая система *cgroup* может быть смонтирована на любом пути файловой системы, так что многие пользовательские приложения и проекты контейнеров используют в качестве точки монтирования */sys/fs/cgroup*. После монтирования файловой системы *cgroups* можно создавать подгруппы, присоединять к ним процессы и задачи, устанавливать ограничения на различные системные ресурсы и делать многое другое. Реализация *cgroup v1*, вероятно, будет существовать с реализацией *cgroup v2* до тех пор, пока есть проекты, использующие ее. Подобное встречается и в других подсистемах ядра, когда новая реализация существующей подсистемы заменяет текущую; например, в настоящее время существуют *iptables* и новые *nftables*, а в прошлом *iptables* сосуществовали с *ipchains*.

## 4.7. УПРАВЛЕНИЕ ПРОЦЕССАМИ И ПОТОКАМИ В ANDROID

Прежде чем обсуждать детали подхода Android к управлению процессами и потоками, нам нужно описать концепции приложений и операций Android.

### Приложения Android

Приложение Android представляет собой программное обеспечение, которое реализует соответствующую функциональность. Каждое приложение Android состоит из одного или нескольких экземпляров одного или нескольких из четырех типов компонентов приложений. Каждый компонент выполняет определенную роль в поведении приложения в целом и каждый компонент может быть активирован в приложении независимо и даже другими приложениями. Ниже приведены четыре типа компонентов.

1. **Операции (activities).** Операция соответствует единому экрану, видимому в качестве пользовательского интерфейса. Например, приложение электронной почты может иметь одну операцию, которая показывает список новых сообщений электронной почты, другая операция состоит в составлении сообщения электронной почты, а еще одна — для чтения электронной почты. Хотя операции работают вместе, образуя единое приложение для работы с электронной почтой, каждая из них является независимой от других. Android различает внутренние и экспортируемые операции. Другие приложения могут запускать экспортируемые операции, которые обычно включают главный экран приложения. Однако сторонние приложения не могут запустить выполнение внутренних операций. Например, приложение камеры может запустить операцию электронной почты для составления нового письма, чтобы поделиться с кем-то снятым фото.
2. **Службы (services).** Службы обычно используются для выполнения фоновых операций, которым требуется значительное количество времени для завершения. Это обеспечивает малое время отклика главного потока приложения, с которым непосредственно взаимодействует пользователь (он же — поток пользовательского интерфейса). Например, служба может создать поток для воспроизведения музыки в фоновом режиме, пока пользователь находится в другом приложении, или же создать поток для получения данных по сети без блокировки взаимодействия пользователя с операцией. Служба может быть вызвана приложением. Кроме того, существуют системные службы, которые работают все время жизни системы Android — такие, как менеджер электропитания (Power Manager), аккумулятора (Battery) или вибратора (Vibrator). Эти системные службы создают потоки, которые являются частью процесса системного сервера (System Server).
3. **Провайдеры контента (content provider).** Провайдер контента выступает в качестве интерфейса к данным приложения, который может использоваться приложением. Одной из категорий управляемых данных являются закрытые данные, используемые только в приложении, содержащем провайдер контента. Например, приложение NotePad (блокнот) использует провайдер контента для сохранения заметки. Другая категория — общие данные, доступные нескольким приложениям. Эта категория включает данные, хранящиеся в файловых системах, базе данных

SQLite, в Интернете или любом другом месте постоянного хранения, к которому ваше приложение может получить доступ.

4. **Получатели широковещательных сообщений** (broadcast receiver). Получатель широковещательных сообщений отвечает на все оповещения уровня системы. Сообщения могут исходить из другого приложения, например, чтобы дать знать другим приложениям, что некоторые данные загружены на устройство и доступны для использования, или от системы, например предупреждение о низком заряде батареи.

Каждое приложение работает на собственной выделенной виртуальной машине в виде единственного собственного процесса, который включает в себя и приложение, и виртуальную машину (рис. 4.16). Этот подход, часто именуемый моделью “песочницы”, изолирует каждое приложение от других. Таким образом, ни одно приложение не может иметь доступа к ресурсам другого приложения без предоставления специального разрешения. Каждое приложение рассматривается как отдельный пользователь Linux со своим уникальным идентификатором пользователя, который применяется для предоставления разрешений.

## Операции

Операция представляет собой компонент приложения, который предоставляет экран взаимодействия с пользователями, которые могут с его помощью что-то сделать, например осуществить телефонный вызов, принять фотографии, отправить письмо по электронной почте или посмотреть карту. Каждой операции предоставляется окно, в котором оно выводит свой пользовательский интерфейс. Окно обычно заполняет экран, но может быть и меньшего размера, чем экран, и размещаться поверх других окон.

Как уже упоминалось, приложение может включать несколько операций. Когда приложение выполняется, одна операция находится на переднем плане, и именно она взаимодействует с пользователем. Операции расположены в стеке (“последним вошел — первым вышел”) в порядке открытия каждой операции. Если пользователь переключается к некоторой другой операции в пределах приложения, создается новая операция, которая помещается на вершину стека; предыдущая операция переднего плана становится вторым элементом стека данного приложения. Этот процесс может повторяться много-кратно, наращивая стек. Пользователь может вернуться к последней операции переднего плана, нажав кнопку Back (назад) или используя аналогичную функцию интерфейса.

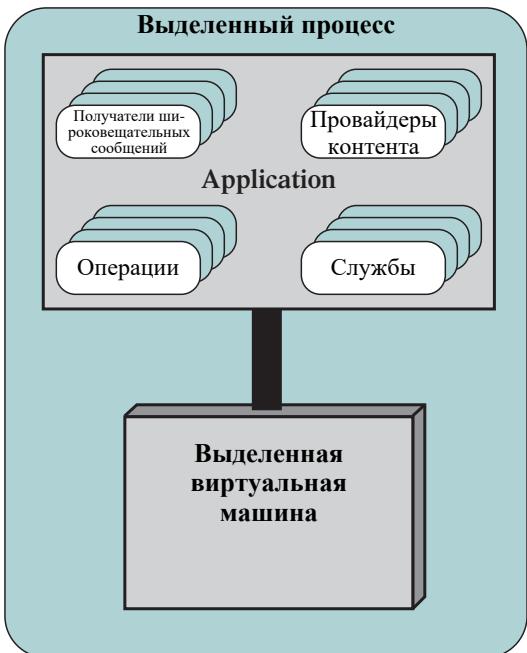


Рис. 4.16. Приложение Android

## Состояния операций

На рис. 4.17 показано упрощенное представление диаграммы переходов состояний операций. Имейте в виду, что в каждом приложении может быть много операций, и каждая из них может находиться в собственной точке на диаграмме переходов. При запуске новой операции программное обеспечение приложения выполняет ряд вызовов Activity Manager API (рис. 2.20): `onCreate()` выполняет статическую настройку операции, включающую инициализацию всех структур данных; `onStart()` делает операцию видимой на экране для пользователя; `onResume()` передает управление операции, так что вводимые пользователем данные поступают именно к ней. В этот момент операция находится в состоянии “Возобновленная” (Resumed), которое также называют *жизненным циклом переднего плана (foreground lifetime)*. В это время операция находится на экране на переднем плане, перед всеми иными операциями, и получает фокус ввода пользователя.

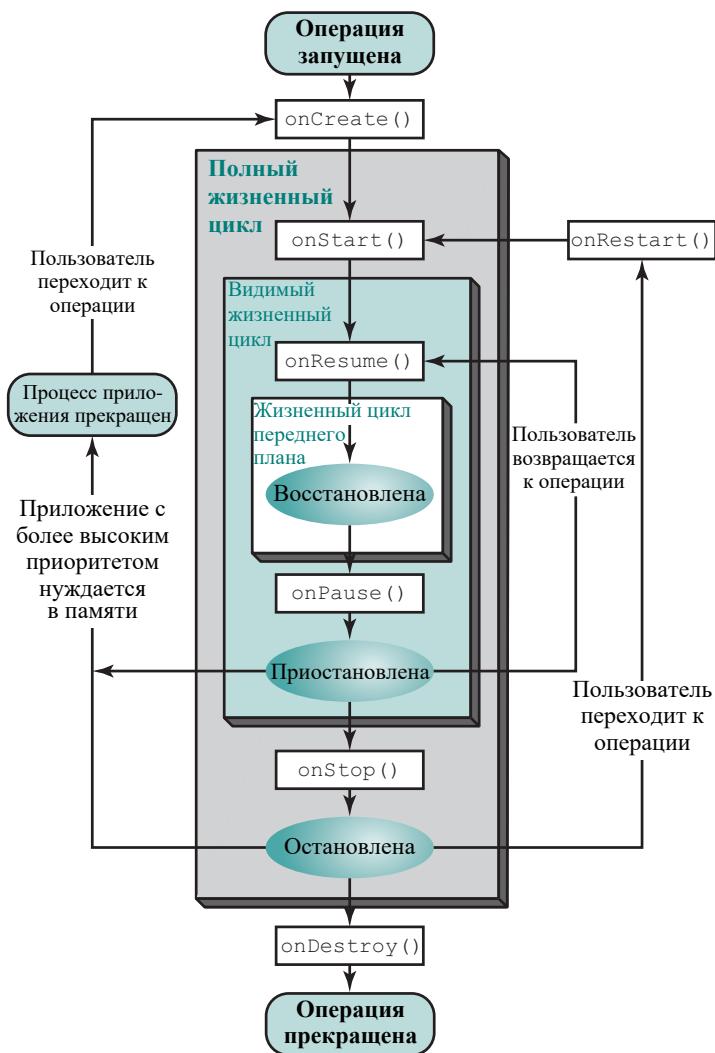


Рис. 4.17. Диаграмма переходов состояний операций

Действия пользователя могут привести к запуску другой операции в рамках приложения. Например, во время выполнения приложения электронной почты, когда пользователь выбирает определенное сообщение электронной почты, открывается новая операция для просмотра этого сообщения. Система реагирует на такую операцию системным вызовом `onPause()`, который помещает текущую операцию в стек, приводя ее в состояние приостановленной. Затем приложение создает новый вид операции, которая переходит в состояние возобновленной.

В любое время пользователь может прекратить текущую операцию с помощью кнопки Back (назад), закрытия окна или некоторых других действий, имеющих отношение к этой операции. Затем для прекращения операции приложение вызывает `onStop()`, после чего обращается к операции на вершине стека и возобновляет ее. Вместе состояния возобновления и приостановки составляют *видимый жизненный цикл* операции. В это время пользователь может видеть операцию на экране и взаимодействовать с ней.

Если пользователь оставляет одно приложение и переходит к другому, например, чтобы перейти к главному экрану, текущая работающая операция приостанавливается, а затем и полностью останавливается. Когда пользователь возобновляет приложение, остановленная операция, находящаяся на вершине стека, перезапускается и становится операцией переднего плана приложения.

## Завершение приложения

Если одновременно выполняется слишком много действий, системе для того, чтобы поддерживать необходимое время отклика, может потребоваться восстановить часть основной памяти. В этом случае система будет высвобождать память, завершая работу одной или нескольких операций внутри приложения, а также завершая процесс данного приложения. Это позволяет освободить память, используемую для управления процессом, а также память для управления операциями, которые были завершены. Однако само приложение по-прежнему существует, а пользователь не знает об изменении его статуса. Если пользователь возвращается к этому приложению, система должна восстановить все остановленные операции при их вызове.

Система завершает приложения в стек-ориентированном стиле: первыми закрываются приложения, которые не использовались дольше всех. Закрытие приложений со службами переднего плана является крайне маловероятным.

## Процессы и потоки

По умолчанию приложению выделяются единственный процесс и единственный поток. Все компоненты приложения выполняются в единственном потоке единственного процесса данного приложения. Чтобы избежать замедления пользовательского интерфейса при выполнении компонентом медленных или блокирующих операций, разработчик может создать несколько потоков в рамках процесса и/или несколько процессов в рамках приложения. В любом случае все процессы данного приложения и их потоки выполняются в пределах одной и той же виртуальной машины.

Чтобы высвободить память в сильно загруженной системе, последняя может принудительно завершить один или несколько процессов. Как обсуждалось в предыдущем разделе, одновременно с процессом завершаются одна или более операций, поддерживаемых этим процессом. Для определения процесса (или процессов), завершение которых должно высвободить необходимые ресурсы, используется иерархия приоритетов. Кажд-

дый процесс в каждый момент времени существует на определенном уровне иерархии, так что завершение процессов начинается с процессов с низким приоритетом. Уровни иерархии в порядке убывания приоритета перечислены далее.

- **Процесс переднего плана.** Процесс, необходимый для поддержки того, что в настоящее время делает пользователь. Процессом переднего плана может быть более чем один процесс одновременно. Например, процессами переднего плана являются одновременно как процесс, который выполняет операцию, с которой взаимодействует пользователь (операция в состоянии “Возобновленная”), так и процесс, который содержит службу, связанную с деятельностью операции, взаимодействующей с пользователем.
- **Видимый процесс.** Процесс, обеспечивающий работу компонента, который не находится на переднем плане, но является видимым для пользователя.
- **Процесс службы.** Процесс, выполняющий службу, которая не попадает ни в одну из приведенных выше категорий. Примеры таких процессов включают воспроизведение музыки в фоновом режиме или фоновую загрузку данных из сети.
- **Фоновый процесс.** Процессы операций, находящихся в состоянии остановленных.
- **Пустой процесс.** Процесс, не соответствующий ни одному из активных компонентов приложения. Единственная причина его существования — с целью кеширования, для снижения времени запуска, когда этот компонент потребуется в следующий раз.

## 4.8. Mac OS X Grand Central Dispatch

Как упоминалось ранее, в главе 2, “Обзор операционных систем”, Mac OS X Grand Central Dispatch (GCD) предоставляет пул доступных потоков. Проектировщики могут указывать части приложений, именуемые блоками, которые могут быть независимоdispatch'изованы, и работать параллельно. Операционная система будет предоставлять как можно большую степень параллелизма, основываясь на количестве доступных ядер и потоковой емкости системы. Хотя и другие операционные системы реализовывали пулы потоков, GCD обеспечивает качественное улучшение в смысле простоты использования и эффективности [152].

Блок представляет собой простое расширение к языку C или к другим языкам, например к C++. Целью определения блока является определение автономной единицы работы, включая код и данные. Вот простой пример определения блока:

```
x = ^{
    printf("hello world\n");
}
```

Блок обозначается с помощью знака вставки (^) в начале функции, которая заключена в фигурные скобки. Приведенное выше определение блока определяет x как способ вызова функции, так что вызов x() выведет слова *hello world*.

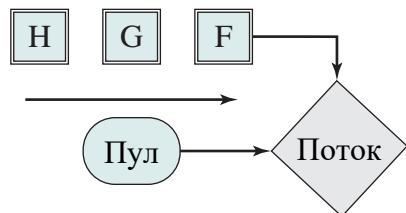
Блоки позволяют программисту инкапсулировать сложные функции вместе с их аргументами и данными, так что на них можно легко ссылаться в программе и передавать их, как переменную. Условно:

$$\boxed{F} = F + \text{Данные}$$

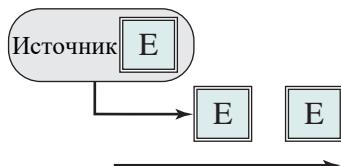
Блоки планируются и диспетчеризуются с использованием очередей. Приложение использует системные очереди, предоставляемые GCD, но может использовать и частные очереди. Блоки помещаются в очередь по мере встречи с ними в процессе выполнения программы. Затем GCD использует эти очереди для описания параллелизма, сериализации и обратных вызовов. Очереди представляют собой облегченные пользовательские структуры данных, что в общем случае делает их гораздо более эффективными, чем управление потоками и блокировками вручную. Например, показанная ниже очередь содержит три блока:



В зависимости от очереди и от того, как она определена, GCD рассматривает эти блоки либо как потенциально параллельные действия, либо как последовательные. В любом случае блоки диспетчеризуются по принципу “первым вошел — первым вышел”. Если это параллельная очередь, то диспетчер назначает F потоку, как только таковой оказывается доступным; затем назначается блок G, а после него — H. Если это последовательная очередь, диспетчер назначает F потоку, но затем назначает G потоку только после завершения F. Использование предопределенных потоков экономит затраты на создание нового потока для каждого запроса и сокращает задержки, связанные с обработкой блока. Размеры пулов потоков автоматически изменяются системой таким образом, чтобы максимизировать производительность приложений с использованием GCD при сведении к минимуму количества бездействующих или конкурирующих потоков.



Помимо непосредственного планирования блоков, приложение может связать один блок и очередь с источником событий, таким как таймер, сетевой сокет или файловый дескриптор. Каждый раз, когда источник генерирует событие, выполняется планирование блока, если он еще не запущен. Это обеспечивает быстрое реагирование без дорогостоящего постоянного опроса или связывания отдельного потока с источником события.



Пример из [233] демонстрирует легкость использования GCD. Рассмотрим приложение для работы с документом с кнопкой, после щелчка на которой будут выполняться анализ текущего документа и отображение некоторых статистических данных о нем. В общем случае этот анализ выполняется около секунды, поэтому для подключения действия к кнопке используется следующий код:

```
- (Inaction)analyzeDocument:(NSButton *)sender
{
    NSDictionary *stats = [myDoc analyze];
    [myModel setDict:stats];
    [myStatsView setNeedsDisplay:YES];
    [stats release];
}
```

Первая строка тела функции анализирует документ, вторая обновляет внутреннее состояние приложения, а третья указывает приложению, что просмотр статистики необходимо обновить, чтобы отразить новое состояние информации. Этот код, который следует распространенному шаблону, выполняется в основном потоке. Такой дизайн является приемлемым при условии, что анализ выполняется не слишком долго, потому что после того, как пользователь нажимает кнопку, от главного потока приложения требуется обработать ввод пользователя как можно быстрее, чтобы он мог вернуться к основному циклу событий для обработки следующего действия пользователя. Но если пользователь открывает очень большой или сложный документ, этап анализа может потребовать неприемлемо длительного времени. Разработчик вряд ли будет рад необходимости изменения кода для этого редкого события, при том что такое изменение может затрагивать глобальные объекты приложения, управление потоками, обратные вызовы, маршалинг аргументов, контекстные объекты, новые переменные и т.д. Но при работе с GCD скромное дополнение к коду тут же дает желаемый результат:

```
- (IBAction)analyzeDocument:(NSButton *)sender
{
    dispatch_async(dispatch_get_global_queue(0, 0), ^{
        NSDictionary *stats = [myDoc analyze];
        dispatch_async(dispatch_get_main_queue(), ^{
            [myModel setDict:stats];
            [myStatsView setNeedsDisplay:YES];
            [stats release];
        });
    });
}
```

Все функции GCD начинаются с префикса `dispatch_`. Внешний вызов `dispatch_async()` помещает задачу в глобальную параллельную очередь. Это сообщает операционной системе, что блок может быть переназначен отдельной параллельной очереди из основной очереди и выполнен параллельно. Таким образом, нет никакой задержки в главном потоке выполнения. После завершения анализа происходит внутренний вызов `dispatch_async()`. Это указывает операционной системе на необходимость поместить следующий блок кода в конец основной очереди для выполнения по достижении головы очереди. Так, с помощью очень небольшой работы со стороны программиста достигается выполнение желаемого требования.

## 4.9. РЕЗЮМЕ

В некоторых операционных системах различаются понятия процесса и потока; первый из них имеет отношение к владению ресурсами, а второй — к выполнению программы. Такой подход может привести к повышению эффективности программы и удобен при составлении кода. В многопоточной системе в рамках одного процесса есть возможность задавать несколько потоков. Для этого можно использовать либо потоки на пользовательском уровне, либо потоки на уровне ядра. Потоки на пользовательском уровне остаются невидимыми для операционной системы, они создаются и управляются библиотекой потоков, которая выполняется в пользовательском пространстве процесса. Потоки на пользовательском уровне очень эффективны, так как при их переключении не нужно переключать режим работы процессора. Однако в одном процессе в каждый момент времени может выполняться только один поток на пользовательском уровне. Если один такой поток будет заблокирован, это приведет к блокированию всего процесса. Потоки на уровне ядра — это потоки, которые управляются ядром. Благодаря тому, что такие потоки распознаются ядром, в многопроцессорной системе могут параллельно выполняться несколько потоков одного и того же процесса, а блокирование потока не приводит к блокированию всего процесса. Однако для переключения потоков на уровне ядра нужно переключать режим работы процессора.

## 4.10. КЛЮЧЕВЫЕ ТЕРМИНЫ, КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАЧИ

### Ключевые термины

Задание	Пользовательский поток	Приложение
Многопоточность	Пользовательский режим планирования	Пространства имен
Нить	Порт	Процесс
Облегченный процесс	Поток	Пул потоков
Объект задания	Поток уровня ядра	Сообщение

### Контрольные вопросы

- 4.1. В табл. 3.5 перечислены типичные элементы, встречающиеся в управляющем блоке процесса операционной системы, в которой не используются потоки. Какие из них следует отнести к управляющему блоку потока, а какие — к управляющему блоку процесса в многопоточной системе?
- 4.2. Перечислите причины, по которым переключение потоков обходится дешевле, чем переключение процессов.
- 4.3. Назовите две различные и потенциально независимые характеристики, содержащиеся в понятии процесса.
- 4.4. Приведите четыре общих примера использования потоков в однопользовательской многопроцессорной системе.
- 4.5. Какие ресурсы обычно совместно используются всеми потоками процесса?

- 4.6. Перечислите три преимущества потоков на пользовательском уровне над потоками на уровне ядра.
- 4.7. Приведите два недостатка потоков на пользовательском уровне по сравнению с потоками на уровне ядра.
- 4.8. Что такое *jacketing*?

## Задачи

- 4.1. Отмечено, что использование нескольких потоков в одном и том же процессе предоставляет следующие преимущества: 1) создание нового потока в уже существующем процессе требует меньших непроизводительных затрат, чем создание нового процесса, и 2) упрощается обмен информацией между потоками одного процесса. Входит ли в число преимуществ использования потоков также то, что переключение потоков одного процесса требует меньших затрат, чем переключение потоков разных процессов?
- 4.2. При сравнении потоков на пользовательском уровне и потоков на уровне ядра упоминалось, что недостаток потоков на пользовательском уровне состоит в том, что выполнение системного вызова блокирует не только вызвавший поток, но и все остальные потоки данного процесса. Почему так происходит?
- 4.3. То, что в других операционных системах воплощено в концепции процесса, в ныне незаслуженно забытой операционной системе OS/2 разделено на три составляющие: сессия, процессы и потоки. Сессия является набором одного или нескольких процессов, имеющих связь с интерфейсом пользователя (клавиатурой, дисплеем, мышью). Сессия представляет собой интерактивное пользовательское приложение, в роли которого может выступать текстовый редактор или электронная таблица. Эта концепция позволяет пользователю персонального компьютера запускать несколько приложений, открывая в каждом из них одно или несколько окон. Операционная система должна следить за тем, какое из окон, а следовательно, какая из сессий является активной. В зависимости от этого ввод, поступающий с клавиатуры и мыши, направляется в ту или иную сессию. В любой момент времени одна сессия обрабатывается на переднем плане, тогда как остальные сессии находятся в фоновом режиме. Все, что вводится с помощью клавиатуры и мыши, направляется в процесс, сессия которого в соответствии с состоянием приложений находится в приоритетном режиме. Когда сессия находится на переднем плане, процесс, выводящий видеосигнал, пересыпает его непосредственно в видеобуфер и, соответственно, на экран пользователя. При переходе сессии в фоновый режим содержимое физического видеобуфера сохраняется в логическом видеобуфере. Если какой-нибудь из потоков любого процесса, относящегося к сессии в фоновом режиме, производит вывод на экран, этот вывод направляется в соответствующий логический видеобуфер. Когда сессия возвращается на передний план, происходит обновление экрана, отражающее текущее содержимое логического видеобуфера этой сессии. Исключив сессии и связав интерфейс пользователя (клавиатуру, мышь, экран) с процессами, можно свести количество концепций, имеющих отношение к процессу, к двум. Таким образом, в каждый момент времени на переднем плане будет находиться один процесс.

В целях структурирования процессы можно разделить на потоки.

- a. Какие преимущества теряются при таком подходе?
- b. Если такая модификация будет реализована, как следует осуществлять назначение ресурсов (памяти, файлов и т.д.) — на уровне процесса или на уровне потока?

- 4.4. Рассмотрим среду, в которой осуществляется взаимно однозначное отображение между потоками на пользовательском уровне и потоками на уровне ядра. В такой системе один или несколько потоков одного и того же процесса могут производить блокирующие системные вызовы, в то время как другие будут продолжать выполняться. Объясните, почему на однопроцессорной машине в такой системе многопроцессорные программы могут выполняться быстрее, чем их однопоточные двойники.
- 4.5. Если процесс завершается, но какие-то его потоки все еще выполняются, то будут ли они выполнять и далее?
- 4.6. Структурирование операционной системы OS/390, предназначеннной для мейнфреймов, основано на концепциях адресного пространства и задания. В других операционных системах адресное пространство, грубо говоря, соответствует приложению и более или менее соответствует процессу. В рамках одного и того же адресного пространства можно одновременно создавать и выполнять несколько заданий, что приближенно соответствует концепции многопоточности. Для управления этими заданиями создается ряд структур данных, две из которых являются основными. Независимо от того, является ли данное адресное пространство выполняющимся, в соответствующем управляющем блоке адресного пространства (address space control block — ASCB) содержится необходимая операционной системе OS/390 информация. В ее состав входят текущий приоритет, размер выделенной данному адресному пространству реальной и виртуальной памяти, количество готовых к выполнению заданий в адресном пространстве, а также сведения о том, является ли каждое из этих заданий выгруженным из памяти. В управляющем блоке задания (task control block — TCB) отражается выполнение пользовательской программы. В нем содержится информация, необходимая для управления заданием в пределах адресного пространства, включая информацию о статусе процессора, указатели на входящие в состав задания программы и состояние выполнения задания. Блоки ASCB являются глобальными структурами, поддерживаемыми в системной памяти, а блоки TCB — локальными структурами, каждая из которых поддерживается в своем адресном пространстве. В чем состоит преимущество разделения управляющей информации на глобальную и локальную части?
- 4.7. Спецификации многих современных языков, таких как C и C++, являются недостаточными для многопоточных программ. Это может влиять на компиляторы и корректность кода, как показано ниже. Рассмотрим следующие объявления и определение функции:

```
int global_positives = 0;
typedef struct list
{
    struct list* next;
    double val;
}* list;
```

```
void count_positives(list l)
{
    list p;
    for (p = l; p; p = p -> next)
        if (p->val > 0.0)
            ++global_positives;
}
```

Теперь рассмотрим ситуацию, когда в потоке А выполняется код

```
count_positives(<список с только отрицательными значениями>);
```

в то время как в потоке В — код

```
+global_positives;
```

а. Что будет выполнять такая функция?

б. Язык С предназначен для однопоточного выполнения. Не приведет ли использование двух параллельных потоков к проблемам (или к потенциальным проблемам)?

**4.8.** Некоторые оптимизирующие компиляторы (включая относительно консервативный gcc) будут “оптимизировать” `count_positives` в код наподобие

```
void count_positives(list l)
{
    list p;
    register int r;
    r = global_positives;
    for (p = l; p; p = p -> next)
        if (p->val > 0.0) ++r;
    global_positives = r;
}
```

К каким проблемам (или к потенциальным проблемам) приведет эта версия кода при параллельном выполнении потоков А и В?

**4.9.** Рассмотрим следующий код, использующий POSIX Pthreads API:

```
thread2.c:
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
int myglobal;
void* thread_function(void* arg)
{
    int i, j;
    for (i = 0; i < 20; i++)
    {
        j = myglobal;
        j = j + 1;
        printf(".");
        fflush(stdout);
        sleep(1);
        myglobal = j;
    }
}
```

```
    return NULL;
}
int main(void)
{
    pthread_t mythread;
    int i;

    if (pthread_create(&mythread, NULL, thread_function, NULL))
    {
        printf("error creating thread.");
        abort();
    }

    for (i = 0; i < 20; i++)
    {
        myglobal = myglobal + 1;
        printf("o");
        fflush(stdout);
        sleep(1);
    }

    if (pthread_join(mythread, NULL))
    {
        printf("error joining thread.");
        abort();
    }

    printf("\nmyglobal equals %d\n", myglobal);
    exit(0);
}
```

В функции `main()` мы сначала объявляем переменную с именем `mythread`, которая имеет тип `pthread_t`. Это, по существу, идентификатор потока. Далее, инструкция `if` создает поток, связанный с `mythread`. Вызов `pthread_create()` возвращает нуль в случае успеха и ненулевое значение — в случае ошибки. Третьим аргументом `pthread_create()` является имя функции, которую новый поток будет выполнять при запуске. При выходе из `thread_function()` поток завершается. Тем временем основная программа сама представляет собой поток, так что имеется два потока выполнения. Функция `pthread_join()` позволяет основному потоку ждать до тех пор, пока не завершится новый поток.

a. Что делает приведенная программа?

Вот вывод данной программы:

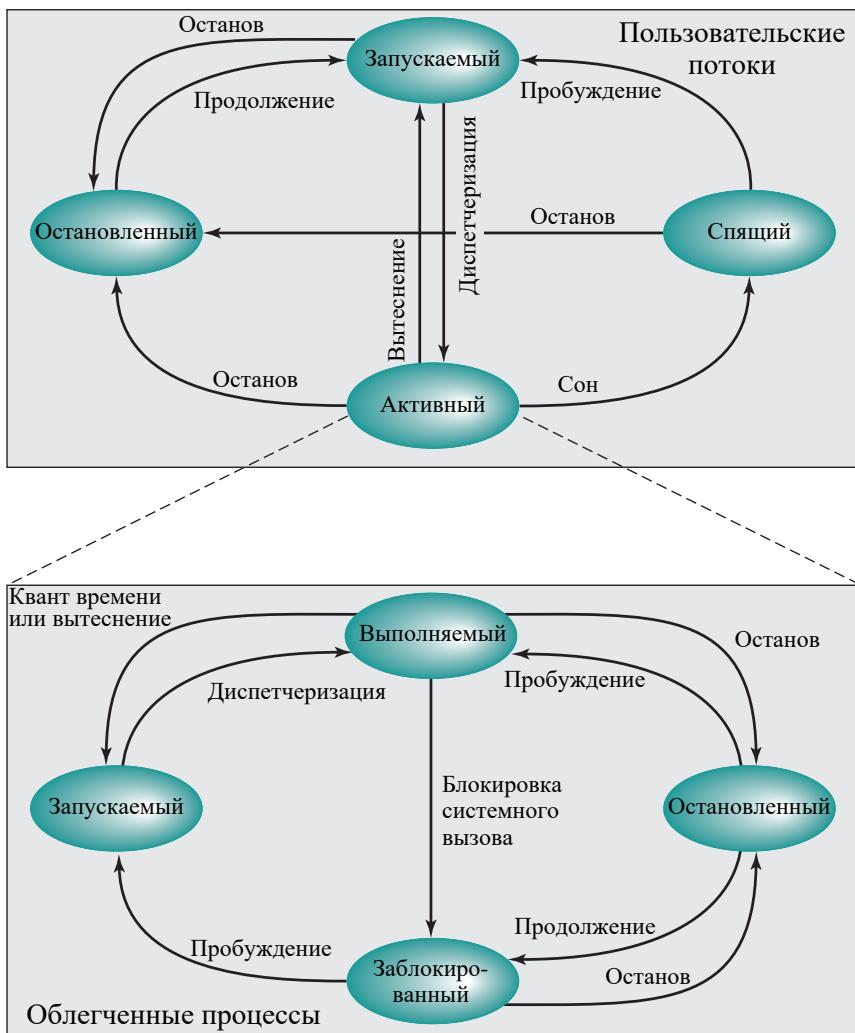
Это тот вывод, который вы ожидали? Если нет, что именно получилось не так?

4.10. В описании состояний потоков на пользовательском уровне в операционной системе Solaris говорилось о том, что поток пользовательского уровня может уступить управление другому потоку с таким же приоритетом. Возможна ли ситуация, когда в системе находится готовый к выполнению поток с более высоким приоритетом, и следовательно, выполняющийся поток уступит управление потоку с таким же или более высоким приоритетом?

**4.11.** В Solaris 9 и Solaris 10 имеется взаимно однозначное отображение между ULT и LWP. В Solaris 8 один LWP поддерживает один или несколько ULT.

- a. В чем состоит возможная выгода разрешения взаимно однозначного отображения между ULT и LWP?
- б. В Solaris 8 состояние выполнения потока ULT отличается от такового у его LWP. Поясните, почему.
- в. На рис. 4.18 приведена диаграмма переходов состояний для ULT и связанного с ним LWP в Solaris 8 и 9. Поясните показанные на диаграммах операции и их взаимоотношения.

**4.12.** Объясните причины наличия состояния “Непрерываемый” (Uninterruptible) в Linux.



**Рис. 4.18.** Состояния ULT и LWP в Solaris

# ГЛАВА 5

# ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ: ВЗАИМОИСКЛЮЧЕНИЯ И МНОГОЗАДАЧНОСТЬ

В ЭТОЙ ГЛАВЕ...

## 5.1. Взаимоисключения: программный подход

Алгоритм Деккера

- Первая попытка
- Вторая попытка
- Третья попытка
- Четвертая попытка
- Правильное решение

Алгоритм Петерсона

## 5.2. Принципы параллельных вычислений

Простой пример

Состояние гонки

Участие операционной системы

Взаимодействие процессов

Конкуренция процессов в борьбе за ресурсы

Сотрудничество процессов с применением совместного использования

Сотрудничество с использованием связи

Требования к взаимным исключениям

## 5.3. Взаимоисключения: аппаратная поддержка

Отключение прерываний

Специальные машинные команды

Команда сравнения и присваивания

Команда обмена

Свойства подхода, основанного на использовании машинных инструкций

## 5.4. Семафоры

Задача производителя/потребителя

Реализация семафоров

## 5.5. Мониторы

Мониторы с сигналами

Мониторы с оповещением и широковещанием

## 5.6. Передача сообщений

Синхронизация

Адресация

Формат сообщения

Принцип работы очереди

Взаимные исключения

## 5.7. Задача читателей/писателей

Приоритетное чтение

Приоритетная запись

## 5.8. Резюме

## 5.9. Ключевые термины, контрольные вопросы и задачи

Ключевые термины

Контрольные вопросы

Задачи

## УЧЕБНЫЕ ЦЕЛИ

- Понимать базовые концепции параллельных вычислений, такие как состояния гонки, взаимные исключения и др.
- Разбираться в аппаратных подходах к поддержке взаимоисключений.
- Понимать семафоры и работать с ними.
- Понимать мониторы и работать с ними.
- Уметь разъяснить задачу читателей/писателей.

Основные вопросы, на которых сосредоточивается внимание разработчиков операционных систем, связаны с управлением процессами и потоками.

- **Многозадачность:** управление множеством процессов в однопроцессорной системе.
- **Многопроцессорность:** управление множеством процессов в многопроцессорной системе.
- **Распределенные вычисления:** управление множеством процессов, выполняемых в распределенной вычислительной системе с множеством компьютеров. Основным примером таких систем являются широко распространенные в последнее время кластеры.

Фундаментальной концепцией этой области, да и разработки операционных систем в целом, является концепция **параллельных вычислений** (concurrency). Параллельность охватывает множество вопросов разработки, включая вопросы обмена информацией между процессами, разделения ресурсов (таких, как память, файлы, система ввода-вывода), синхронизацию работы процессов и распределение процессорного времени между различными процессами. Вы увидите, что эти вопросы возникают не только в многопроцессорной или распределенной вычислительной среде, но и в случае многозадачных систем на базе одного процессора.

Параллельность проявляется в трех различных контекстах.

1. **Множественные приложения.** Многозадачность разработана для того, чтобы позволить динамически разделять процессорное время между рядом активных приложений.
2. **Структурность приложений.** В качестве развития парадигмы модульной разработки и структурного программирования некоторые приложения могут быть разработаны как множество **параллельно работающих процессов**.
3. **Структура операционной системы.** Преимущества структурного программирования доступны не только прикладным, но и системным программистам, и, как вы знаете, операционные системы также зачастую реализуются в виде набора процессов или потоков.

В силу важности данного вопроса ему посвящены четыре главы данной книги. В настоящей и следующей главах рассматривается параллельность в контексте многопроцессорности и многозадачности; в главах 16, “Облачные операционные системы и операционные системы Интернета вещей”, и 18, “Распределенная обработка, вычисления

«клиент/сервер» и кластеры», изложены вопросы параллельных вычислений в контексте распределенных вычислений.

Данная глава начинается с введения в концепции параллельных вычислений и параллельного выполнения нескольких процессов<sup>1</sup>. Вы узнаете, что основным требованием поддержки параллельных процессов является возможность обеспечения взаимоисключений, т.е. возможность обеспечить работу только одного процесса с остановкой выполнения всех остальных. В следующем далее разделе будут рассмотрены различные подходы к обеспечению взаимоисключений. Все они являются программными решениями и требуют использования технологии, известной как *пережидание занятости* (*busy waiting*). Затем мы остановимся на некоторых аппаратных механизмах, способных обеспечить поддержку взаимоисключений, а также на решениях, которые не используют пережидание занятости и могут поддерживаться операционной системой или компиляторами. Мы рассмотрим три подхода: семафоры, мониторы и передачу сообщений.

Для иллюстрации концепций и сравнения представленных в этой главе подходов используются две классические задачи. Сначала мы познакомимся с задачей производителей/потребителей, а затем — с задачей читателей/писателей.

Наше изучение параллельных вычислений продолжится в главе 6, “Параллельные вычисления: взаимоблокировка и голодание”, и обсуждение механизмов параллельности рассматриваемых нами операционных систем мы отложим до ее конца.

В приложении А, “Вопросы параллельности”, охвачены дополнительные темы, связанные с параллельными вычислениями. В табл. 5.1 перечислены некоторые ключевые термины, связанные с параллельными вычислениями. Набор анимаций, иллюстрирующих концепции этой главы, доступен на веб-сайте данной книги.

**Таблица 5.1. Ключевые термины, связанные с параллельными вычислениями**

<b>Атомарная операция</b>	Функция или действие, реализованное как последовательность из одной или нескольких инструкций, представляющихся неделимыми; т.е. никакой другой процесс не может увидеть промежуточное состояние или прервать выполняющуюся операцию. Гарантируется выполнение последовательности инструкций как единой группы (либо не выполнение ни одной из них, без влияния на видимое состояние системы). Атомарность гарантирует изоляцию от параллельно выполняющихся процессов
<b>Критический участок</b>	Участок кода в рамках процесса, который требует доступа к общим ресурсам и который не должен выполняться в то время, когда в этом участке кода находится другой процесс
<b>Взаимоблокировка (клинич)</b>	Ситуация, когда два и более процессов не в состоянии работать, поскольку каждый из процессов ожидает выполнения некоторого действия другим процессом
<b>Динамическая взаимоблокировка</b>	Ситуация, когда два и более процессов постоянно изменяют свои состояния в ответ на изменения в других процессах без выполнения полезной работы

<sup>1</sup> Для простоты мы говорим о параллельном выполнении *процессов*; в действительности в ряде систем фундаментальной единицей параллельности является не процесс, а поток.

Окончание табл. 5.1

<b>Взаимоисключение</b>	Требование, чтобы, когда один процесс находится в критическом участке, который получает доступ к общим ресурсам, никакой другой процесс не мог находиться в критическом участке, который обращается к любому из этих общих ресурсов
<b>Состояние гонки</b>	Ситуация, когда несколько потоков или процессов читают и записывают элемент общих данных и конечный результат зависит от относительного времени выполнения этих потоков
<b>Голодание</b>	Ситуация, когда запуск процесса пропускается планировщиком бесконечное количество раз; хотя процесс готов работать, он никогда не выбирается

## 5.1. ВЗАИМОИСКЛЮЧЕНИЯ: ПРОГРАММНЫЙ ПОДХОД

Программный подход может быть реализован для параллельных процессов, которые выполняются как в однопроцессорной, так и в многопроцессорной системе с общей основной памятью. Обычно такие подходы предполагают элементарные взаимоисключения на уровне доступа к памяти ([140], см. также задачу 5.3). То есть одновременный доступ (чтение и/или запись) к одной и той же ячейке основной памяти упорядочивается при помощи некоторого механизма (хотя при этом порядок предоставления доступа не определяется порядком обращения процессов за доступом к памяти). Никакой иной поддержки со стороны аппаратного обеспечения, операционной системы или языка программирования не предполагается.

### Алгоритм Деккера

Дейкстра в [66] сообщил об алгоритме для взаимных исключений для двух процессов, предложенном голландским математиком Деккером (Dekker). Следуя Дейкстру, мы разработаем этот алгоритм поэтапно. Главное преимущество такого подхода — в демонстрации множества узких мест и возможных ошибок при создании параллельно работающих программ.

#### Первая попытка

Как упоминалось ранее, любая попытка взаимного исключения должна опираться на некий фундаментальный механизм исключений аппаратного обеспечения. Наиболее общим механизмом может служить ограничение, согласно которому к некоторой ячейке памяти в определенный момент времени может осуществляться только одно обращение. Воспользовавшись этим ограничением, зарезервируем глобальную ячейку памяти, которую назовем *turn*. Процесс (P0 или P1), который намерен выполнить критический участок, сначала проверяет содержимое ячейки памяти *turn*. Если значение *turn* равно номеру процесса, то процесс может войти в критический участок; в противном случае он должен ждать, постоянно опрашивая значение *turn* до тех пор, пока оно не позволит процессу войти в критический участок. Такая процедура известна как *пережидание занятости* (*busy waiting*), поскольку процесс вынужден, по сути, не делать ничего полезного до

тех пор, пока не получит разрешение на вход в критический участок. Более того, он постоянно опрашивает значение переменной и тем самым потребляет процессорное время.

После того как процесс, получивший право на вход в критический участок кода, выходит из него по завершении работы, он должен обновить значение `turn`, присвоив ему номер другого процесса.

Говоря формально, имеется глобальная переменная

```
int turn = 0;
```

На рис. 5.1, а показана программа для двух процессов. Это решение гарантирует корректную работу взаимоисключения, однако имеет два недостатка. Во-первых, при входе в критический участок процессы должны строго чередоваться; тем самым скорость работы диктуется более медленным из двух процессов. Если процессу P0 вход в критический участок требуется раз в час, а процессору P1 — 1000 раз в час, то темп работы P1 будет таким же, как и у процесса P0. Во-вторых, гораздо более серьезная ситуация возникает в случае сбоя одного из процессов — при этом второй процесс оказывается заблокированным (при этом не важно, происходит сбой процесса в критическом участке или нет).

Описанная конструкция представляет собой **сопрограмму** (coroutine). Сопрограммы разрабатываются таким образом, чтобы быть способными передавать управление друг другу (см. задачу 5.5). Однако хотя эта технология структурирования и весьма полезна для отдельно взятого процесса, для поддержки параллельных вычислений она не подходит.

## *Вторая попытка*

Проблема при первой попытке заключается в том, что в ней хранилось имя процесса, который имел право входа в критический участок, в то время как в действительности нам требуется информация об обоих процессах. По сути, каждый процесс должен иметь собственный ключ к критическому участку, так что если даже произойдет сбой одного процесса, второй все равно сможет получить доступ к критическому участку. Для удовлетворения этого условия определен логический вектор `flag`, в котором `flag[0]` соответствует процессу P0, а `flag[1]` — процессу P1. Каждый процесс может ознакомиться с флагом другого процесса, но не может его изменить. Когда процессу требуется войти в критический участок, он периодически проверяет состояние флага другого процесса до тех пор, пока тот не примет значение `false`, указывающее, что другой процесс покинул критический участок. Процесс немедленно устанавливает значение собственного флага равным `true` и входит в критический участок. После выхода из критического участка процесс сбрасывает свой флаг, присваивая ему значение `false`.

Теперь общие переменные<sup>2</sup> выглядят следующим образом:

```
enum    boolean { false = 0, true = 1; };
boolean flag[2] = { false, false };
```

Этот алгоритм показан на рис. 5.1, б. Теперь, если произойдет сбой одного из процессов вне критического участка (включая код установки значения флага), второй процесс заблокирован не будет. Этот второй процесс в таком случае сможет входить в критический участок всякий раз, как только это потребуется, поскольку флаг другого процесса всегда будет иметь значение `false`. Однако если сбой произойдет в критическом участке (или перед входом в него, но после установки значения флага равным `true`), то другой процесс окажется заблокированным навсегда.

---

<sup>2</sup> Здесь объявление перечисления `enum` использовано для объявления типа данных (`boolean`) и его значений.

```
/* Процесс 0 */          /* Процесс 1 */
.
.
.
while (turn != 0)        while (turn != 1)
/* Ничего не делаем */; /* Ничего не делаем */;
/* Критический участок */; turn = 0;
turn = 1;
.
```

*a) Первая попытка*

```
/* Процесс 0 */          /* Процесс 1 */
.
.
.
while (flag[1])        while (flag[0])
/* Ничего не делаем */; /* Ничего не делаем */;
flag[0] = true;          flag[1] = true;
/* Критический участок */; /* Критический участок */;
flag[0] = false;          flag[1] = false;
.
.
```

*б) Вторая попытка*

```
/* Процесс 0 */          /* Процесс 1 */
.
.
.
flag[0] = true;          flag[1] = true;
while (flag[1])          while (flag[0])
/* Ничего не делаем */; /* Ничего не делаем */;
/* Критический участок */; /* Критический участок */;
flag[0] = false;          flag[1] = false;
.
.
```

*в) Третья попытка*

```
/* Процесс 0 */          /* Процесс 1 */
.
.
.
flag[0] = true;          flag[1] = true;
while (flag[1]) {          while (flag[0]) {
    flag[0] = false;          flag[1] = false;
    /* Задержка */;          /* Задержка */;
    flag[0] = true;          flag[1] = true;
}
/* Критический участок */; /* Критический участок */;
flag[0] = false;          flag[1] = false;
.
.
```

*г) Четвертая попытка***Рис. 5.1.** Попытки взаимных исключений

Описанное решение, по сути, оказывается еще хуже предложенного ранее, поскольку даже не гарантирует взаимного исключения. Рассмотрим такую последовательность действий:

P0 выполняет инструкцию `while` и находит, что значение `flag[1]` равно `false`;

P1 выполняет инструкцию `while` и находит, что значение `flag[0]` равно `false`;

P0 устанавливает значение `flag[0]` равным `true` и входит в критический участок;

P1 устанавливает значение `flag[1]` равным `true` и входит в критический участок.

Поскольку после этого оба процесса одновременно оказываются в критическом участке, программа некорректна. Проблема заключается в том, что предложенное решение не является независимым от относительной скорости выполнения процессов.

### Третья попытка

Поскольку процесс может изменить свое состояние после того, как другой процесс ознакомится с ним, но до того, как этот другой процесс войдет в критический участок, вторая попытка также оказалась неудачной. Возможно, нам удастся исправить ситуацию путем внесения в код небольшого изменения, показанного на рис. 5.1,в.

Как и ранее, если происходит сбой одного процесса в критическом участке, включая код установки значения флага, то второй процесс окажется заблокированным (и соответственно, если сбой произойдет вне критического участка, то второй процесс блокирован не будет).

Далее проверим гарантированность взаимоисключения, проследив за происходящим с точки зрения процесса P0. После того как процесс P0 установит `flag[0]` равным `true`, P1 не сможет войти в критический участок до тех пор, пока туда не войдет и затем не покинет его процесс P0. Может оказаться так, что процесс P1 уже находится в критическом участке в тот момент, когда P0 устанавливает свой флаг. В этом случае процесс P0 будет заблокирован инструкцией `while` до тех пор, пока P1 не покинет критический участок. Аналогичные действия происходят при рассмотрении происходящего с точки зрения процесса P1.

Тем самым взаимное исключение гарантировается; однако третья попытка порождает еще одну проблему. Если оба процесса установят значения флагов равными `true` до того, как один из них выполнит инструкцию `while`, то каждый из процессов будет считать, что другой находится в критическом участке, и тем самым осуществляться взаимоблокировка.

### Четвертая попытка

В третьей попытке установка процессом флага состояния выполнялась без учета информации о состоянии другого процесса. Взаимоблокировка возникала по той причине, что каждый процесс мог добиваться своих прав на вход в критический участок и не было никакой возможности отступить назад из имеющегося положения. Можно попытаться исправить ситуацию, делая процессы более “уступчивыми”: каждый процесс, устанавливая свой флаг равным `true`, указывает о своем желании войти в критический участок, но готов отложить свой вход, уступая другому процессу, как показано на рис. 5.1,г.

Это уже совсем близко к корректному решению, хотя все еще и неверно. Взаимоисключение гарантировается (в чем можно убедиться, применяя те же рассуждения, что и при третьей попытке), однако рассмотрим следующую возможную последовательность событий:

P0 устанавливает значение flag[0] равным true;  
P1 устанавливает значение flag[1] равным true;  
P0 проверяет flag[1];  
P1 проверяет flag[0];  
P0 устанавливает значение flag[0] равным false;  
P1 устанавливает значение flag[1] равным false;  
P0 устанавливает значение flag[0] равным true;  
P1 устанавливает значение flag[1] равным true.

Эту последовательность можно продолжать до бесконечности — и ни один из процессов до бесконечности так и не сможет войти в критический участок. Строго говоря, это не взаимоблокировка, так как любое изменение относительной скорости двух процессов разорвет замкнутый круг и позволит одному из процессов войти в критический участок. Назовем такую ситуацию **динамической взаимоблокировкой** (livelock). Вспомним, что обычная взаимоблокировка осуществляется, когда несколько процессов желают войти в критический раздел, но ни одному из них это не удается. В случае неустойчивой взаимоблокировки существует приводящая к успеху последовательность действий, но вместе с тем возможна и такая (такие), при которой ни один из процессов не сможет войти в критический участок.

Хотя описанный сценарий маловероятен и вряд ли такая последовательность продлится сколь-нибудь долго, теоретически такая возможность имеется. Поэтому мы вынуждены отвергнуть как неудачную и четвертую попытку.

### Правильное решение

У нас должна быть возможность следить за состоянием обоих процессов, что обеспечивается массивом flag. Но, как показала четвертая попытка, этого недостаточно. Мы должны навязать определенный порядок действий двум процессам, чтобы избежать проблемы “взаимной вежливости”, с которой только что столкнулись. С этой целью можно использовать переменную turn из первой попытки. В нашем случае эта переменная указывает, какой из процессов имеет право на вход в критический участок.

Мы можем описать это решение (известное как алгоритм Dekker) следующим образом. Когда процесс P0 намерен войти в критический участок, он устанавливает свой флаг равным true, а затем проверяет состояние флага процесса P1. Если он равен false, P0 может немедленно входить в критический участок; в противном случае P0 обращается к переменной turn. Если turn=0, это означает, что сейчас — очередь процесса P0 на вход в критический участок, и P0 периодически проверяет состояние флага процесса P1. Этот процесс, в свою очередь, в некоторый момент времени обнаруживает, что сейчас не его очередь для входа в критический участок, и устанавливает свой флаг равным false, давая возможность войти в критический участок процессу P0. После того как P0 выйдет из критического участка, он установит свой флаг равным false для освобождения критического участка и присвоит переменной turn значение 1 для передачи прав на вход в критический участок процессу P1.

Алгоритм Dekker'a приведен на рис. 5.2. Конструкция **parbegin(P<sub>1</sub>, P<sub>2</sub>, ..., P<sub>n</sub>)** означает следующее: приостановка выполнения основной программы; инициализация параллельного выполнения процедур P<sub>1</sub>, P<sub>2</sub>, ..., P<sub>n</sub>; по окончании работы процедур P<sub>1</sub>, P<sub>2</sub>, ..., P<sub>n</sub> — возобновление выполнения основной программы. Доказательство корректности алгоритма Dekker'a оставляется читателю в качестве упражнения (см. задачу 5.1).

```

boolean flag [2];
int turn;
void P0()
{
    while (true)
    {
        flag [0] = true;
        while (flag [1])
        {
            if (turn == 1)
            {
                flag [0] = false;
                while (turn == 1) /* Ничего не делать */;
                flag [0] = true;
            }
            /* Критический участок */;
            turn = 1;
            flag [0] = false;
            /* Остальной код */;
        }
    }
}

void P1( )
{
    while (true)
    {
        flag [1] = true;
        while (flag [0])
        {
            if (turn == 0)
            {
                flag [1] = false;
                while (turn == 0) /* Ничего не делать */;
                flag [1] = true;
            }
            /* Критический участок */;
            turn = 0;
            flag [1] = false;
            /* Остальной код */;
        }
    }
}

void main ()
{
    flag [0] = false;
    flag [1] = false;
    turn = 1;
    parbegin (P0, P1);
}

```

Рис. 5.2. Алгоритм Деккера

## Алгоритм Петерсона

Алгоритм Деккера решает задачу взаимных исключений, но достаточно сложным путем, корректность которого не так легко доказать. Петерсон (Peterson) предложил простое и элегантное решение [190]. Как и ранее, глобальная переменная `flag` указывает положение каждого процесса по отношению к взаимоисключению, а глобальная переменная `turn` разрешает конфликты одновременности. Алгоритм представлен на рис. 5.3.

```

boolean flag [2];
int turn;
void P0()
{
    while (true)
    {
        flag [0] = true;
        turn = 1;
        while (flag [1] && turn == 1) /* Ничего не делать */;
        /* Критический раздел */;
        flag [0] = false;
        /* Остальной код */;
    }
}
void P1()
{
    while (true) {
        flag [1] = true;
        turn = 0;
        while (flag [0] && turn == 0) /* Ничего не делать */;
        /* Критический раздел */;
        flag [1] = false;
        /* Остальной код */
    }
}
void main()
{
    flag [0] = false;
    flag [1] = false;
    parbegin (P0, P1);
}

```

**Рис. 5.3.** Алгоритм Петерсона для двух процессов

Выполнение условий взаимоисключения легко показать. Рассмотрим процесс `P0`. После того как `flag[0]` установлен им равным `true`, `P1` войти в критический участок не может. Если же `P1` уже находится в критическом участке, то `flag[1]=true` и для `P0` вход в критический участок заблокирован. Однако взаимная блокировка в данном алгоритме предотвращена. Предположим, что `P0` заблокирован в своем цикле `while`. Это означает, что `flag[1]` равен `true`, а `turn=1`. `P0` может войти в критический

участок, когда либо `flag[1]` становится равным `false`, либо `turn` становится равным 0. Рассмотрим три исчерпывающих случая.

1. Р1 не намерен входить в критический участок. Такой случай невозможен, поскольку при этом выполнялось бы условие `flag[1] = false`.
2. Р1 ожидает входа в критический участок. Такой случай также невозможен, поскольку если `turn = 1`, то Р1 способен войти в критический участок.
3. Р1 циклически использует критический участок, монополизировав доступ к нему. Этого не может произойти, поскольку Р1 вынужден перед каждой попыткой входа в критический участок дать возможность входа процессу Р0, устанавливая значение `turn` равным 0.

Таким образом, у нас имеется простое решение проблемы взаимных исключений для двух процессов. Впрочем, алгоритм Петерсона легко обобщается на случай *n* процессов.

## 5.2. ПРИНЦИПЫ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ

В однопроцессорной многозадачной системе процессы чередуются для создания иллюзии одновременного выполнения (см. рис. 2.12, *a*). Несмотря на то что при этом не достигается реальная параллельная работа процессов и, более того, имеются определенные накладные расходы, связанные с переключением между процессами, такое чередующееся выполнение обеспечивает немалые выгоды с точки зрения эффективности и структуризации программ. В многопроцессорных системах возможно не только чередование процессов, но и их перекрытие (см. рис. 2.12, *b*).

На первый взгляд может показаться, что чередование и перекрытие представляют собой принципиально различающиеся режимы работы и, соответственно, при этом возникают различные проблемы. Однако в действительности обе технологии, которые можно рассматривать как примеры параллельных вычислений, порождают одинаковые проблемы. В однопроцессорных системах проблемы вытекают из основных характеристик многозадачных систем: невозможно предсказать относительную скорость выполнения процессов. Она зависит от других процессов, способа обработки прерываний операционной системой и стратегий планирования операционной системы. При этом возникают следующие трудности.

1. Совместное использование глобальных ресурсов чревато опасностями. Например, если два процесса используют одну глобальную переменную и оба выполняют чтение и запись этой переменной, то критическим оказывается порядок чтения и записи этой переменной разными процессами. Пример такой проблемы приведен в следующем подразделе.
2. Операционной системе трудно управлять распределением ресурсов оптимальным образом. Например, процесс А может потребовать и получить контроль над некоторым каналом ввода-вывода, после чего временно приостановить работу. Нежелательно, чтобы операционная система при этом блокировала канал и не давала другим процессам возможности использовать его, — такая политика может привести к возникновению условий взаимоблокировки, описываемых в главе 6, “Параллельные вычисления: взаимоблокировка и голодание”.

3. Становится очень трудно обнаружить программную ошибку, поскольку обычно результат работы программы перестает быть детерминированным и воспроизводимым (см., например, [146, 39, 225]).

Все описанные трудности имеются в наличии и в многопроцессорной системе, поскольку и в этом случае относительная скорость выполнения процессов непредсказуема. Многопроцессорная система, кроме того, должна быть в состоянии разрешить проблемы, возникающие вследствие одновременного выполнения нескольких процессов. Однако в основе своей эти проблемы те же, что и в однопроцессорной системе. Окончательно ясно это станет в процессе работы с материалом данной главы.

## Простой пример

Рассмотрим следующую процедуру:

```
void echo()
{
    chin = getchar();
    chout = chin;
    putchar(chout);
}
```

Она демонстрирует основные элементы отображения вводимых символов. Входной символ, получаемый с клавиатуры, сохраняется в переменной `chin`; после этого значение переменной `chin` присваивается переменной `chout` и выводится на экран. Данная процедура может неоднократно вызываться любым процессом для получения ввода пользователя и отображения его на экране.

Теперь представим, что у нас имеется однопроцессорная многозадачная система, поддерживающая единственного пользователя. Пользователь может переходить от одного приложения к другому; при этом все приложения используют одну и ту же клавиатуру и экран. Поскольку рассматриваемая нами процедура нужна всем приложениям, имеет смысл сделать ее совместно используемой процедурой, загружаемой в часть памяти, глобальную для всех приложений (таким образом, имеется только одна копия процедуры, и тем самым экономится память).

Совместное использование основной памяти процессами способствует эффективному и тесному взаимодействию процессов. Однако такое совместное использование может привести к проблемам. Рассмотрим приведенную ниже последовательность событий.

1. Процесс P1 вызывает процедуру `echo` и прерывается немедленно по выполнении функции `getchar`. В этот момент последний введенный символ `x` сохранен в переменной `chin`.
2. Активируется процесс P2, который вызывает процедуру `echo`. Эта процедура выполняется до конца; при этом считывается с клавиатуры и выводится на экран очередной символ `y`.
3. Продолжается выполнение процесса P1. Однако к этому моменту значение `x` в переменной `chin` перезаписано — теперь эта переменная содержит значение `y`, которое присваивается переменной `chout` и выводится на экран.

Таким образом, первый введенный символ благополучно теряется, зато второй оказывается выведенным на экран дважды. Проблема заключается в совместно используемой

глобальной переменной `chin`, к которой обращаются несколько процессов; если один процесс изменяет глобальную переменную и затем прерывается, другой может успеть изменить ее значение, перед тем как первый процесс им воспользуется. Предположим теперь, что выполнять процедуру одновременно процессы не могут. В таком случае описанная ранее последовательность действий выглядит иначе.

1. Процесс P1 вызывает процедуру `echo` и прерывается немедленно по выполнении функции `getchar`. В этот момент последний введенный символ `x` сохраняется в переменной `chin`.
2. Активируется процесс P2, который вызывает процедуру `echo`. Однако, поскольку приостановленный процесс P1 находится в процедуре `echo`, P2 блокируется от входа в данную процедуру. Следовательно, выполнение P2 приостанавливается до тех пор, пока процедура `echo` не окажется свободной.
3. В некоторый более поздний момент продолжается выполнение процесса P1, который завершает выполнение процедуры `echo`, выводя на экран верный символ — `x`.
4. После того как P1 покидает `echo`, блокировка P2 удаляется и позже, при возобновлении работы процесса P2, им успешно выполняется процедура `echo`.

Урок, который следует извлечь из данного примера, заключается в том, что совместно используемые глобальные переменные (как и другие совместно используемые глобальные ресурсы) нуждаются в защите, и единственный способ обеспечить ее состоит в управлении кодом, осуществляющим доступ к этим переменным. Если мы добьемся того, что в определенный момент времени только один процесс сможет входить в процедуру `echo` и она обязательно будет полностью выполнена вошедшим в нее процессом до того, как станет возможным ее выполнение другим процессом, то мы будем застрахованы от возникновения рассматриваемой ошибки. Каким образом этого добиться — основной вопрос данной главы.

Мы предполагали, что имеем дело с однопроцессорной многозадачной операционной системой. Пример продемонстрировал, что проблема может возникнуть даже в этом случае. В многопроцессорной системе возникает та же проблема защиты совместно используемых ресурсов, которая имеет аналогичное решение. Сначала предположим, что механизм управления доступом к разделяемой глобальной переменной отсутствует.

1. Одновременно выполняются процессы P1 и P2 — каждый на своем процессоре. Оба процесса вызывают процедуру `echo`.
2. Происходят следующие события (события в одной строке происходят параллельно).

Процесс P1	Процесс P2
•	•
<code>chin = getchar();</code>	<code>chin = getchar();</code>
•	•
<code>chout = chin;</code>	<code>chout = chin;</code>
<code>putchar(chout);</code>	<code>putchar(chout);</code>
•	•
•	•

В результате символ, введенный в процессе P1, теряется до того, как будет выведен на экран, и обоими процессами выводится символ, считанный процессом P2. Теперь добавим в систему механизм, гарантирующий, что в процедуре `echo` в любой момент времени может находиться только один процесс. В этом случае последовательность событий становится такой.

1. Одновременно выполняются процессы P1 и P2 — каждый на своем процессоре. Процесс P1 вызывает процедуру echo.
2. В то время как процесс P1 находится в процедуре echo, эту же процедуру вызывает процесс P2. Однако, поскольку процесс P1 находится в процедуре echo (не важно, выполняется ли в этот момент процесс P1 или приостановлен), P2 блокируется от входа в данную процедуру. Следовательно, выполнение P2 приостанавливается до тех пор, пока процедура echo не окажется свободной.
3. В некоторый более поздний момент времени выполнение процессом P1 процедуры echo завершается, после чего немедленно продолжается выполнение процесса P2 и им успешно выполняется процедура echo.

В однопроцессорной системе причина возникновения проблемы заключается в том, что прерывание может остановить выполнение процесса в произвольном месте. В многопроцессорной системе условия работы те же, но проблема может возникнуть и из-за того, что два выполняющихся одновременно процесса могут в один момент времени обратиться к одной и той же глобальной переменной. Однако решение проблем обоих типов одинаково: управление доступом к совместно используемым ресурсам.

## СОСТОЯНИЕ ГОНКИ

Состояние гонки возникает, когда несколько процессов или потоков читают и записывают элементы данных так, что конечный результат зависит от порядка выполнения инструкций в нескольких процессах. Давайте рассмотрим два простых примера.

В качестве первого примера предположим, что два процесса, P1 и P2, совместно используют глобальную переменную a. В какой-то момент своего времени выполнения P1 обновляет значение a, делая его равным 1, а процесс P2 в некоторый момент своего выполнения обновляет значение a, делая его равным 2. Таким образом, две задачи находятся в “гонке”, призом которой является запись переменной. В данном примере “проигравший” гонку процесс (который обновляет переменную последним) определяет конечное значение переменной a.

В качестве второго примера рассмотрим два процесса, P3 и P4, которые совместно используют глобальные переменные b и c, с начальными значениями  $b=1$  и  $c=2$ . В какой-то момент времени выполнения P3 выполняет присваивание  $b=b+c$ , а процесс P4 в некоторый момент своего выполнения выполняет присваивание  $c=b+c$ . Обратите внимание, что эти два процесса обновляют разные переменные. Однако их конечные значения зависят от порядка, в котором данные процессы выполняют присваивания. Если первым выполнит присваивание процесс P3, конечными значениями переменных будут  $b=3$  и  $c=5$ . Если же первым выполнит присваивание процесс P4, то конечными значениями переменных будут  $b=4$  и  $c=3$ .

В приложении А, “Вопросы параллельности”, в качестве примера рассматриваются вопросы состояния гонки с использованием семафоров.

## Участие операционной системы

Можно перечислить следующие вопросы конструирования и управления операционных систем, возникающие из-за наличия параллельных вычислений.

1. Операционная система должна быть способна отслеживать различные процессы. Это осуществляется при помощи управляющих блоков процессов, как описано в главе 4, “Потоки”.
2. Операционная система должна распределять и освобождать различные ресурсы для каждого активного процесса, в том числе следующие.
  - **Процессорное время.** Это функция планирования, рассматриваемая в части IV, “Планирование”.
  - **Память.** Большинство операционных систем используют схему виртуальной памяти. Этот вопрос рассматривается в части III, “Память”.
  - **Файлы.** Обсуждаются в главе 12, “Управление файлами”.
  - **Устройства ввода-вывода.** Обсуждаются в главе 11, “Управление вводом-выводом и планирование дисковых операций”.
3. Операционная система должна защищать данные и физические ресурсы каждого процесса от непреднамеренного воздействия других процессов, что включает использование технологий, применяющихся для работы с памятью, файлами и устройствами ввода-вывода.
4. Функционирование процесса и результат его работы не должны зависеть от скорости его выполнения по отношению к другим параллельно выполняющимся процессам. Этому вопросу посвящена данная глава.

Чтобы лучше понять вопросы независимости работы процессов от относительной скорости выполнения, рассмотрим сначала способы взаимодействия процессов.

## Взаимодействие процессов

Способы взаимодействия процессов можно классифицировать по степени осведомленности одного процесса о существовании другого. В табл. 5.2 перечислены три возможные степени осведомленности.

- **Процессы не осведомлены о наличии друг друга.** Это независимые процессы, не предназначенные для совместной работы. Наилучшим примером такой ситуации может служить многозадачность множества независимых процессов. Это могут быть пакетные задания, интерактивные сессии или комбинация тех и других. Хотя эти процессы и не работают совместно, операционная система должна решать вопросы конкурентного использования ресурсов. Например, два независимых приложения могут потребовать доступ к одному и тому же диску или к принтеру. Операционная система должна регулировать такие обращения.
- **Процессы косвенно осведомлены о наличии друг друга.** Эти процессы не обязательно должны быть осведомлены о наличии друг друга с точностью до идентификатора процесса, однако они совместно обращаются к некоторому объекту, например к буферу ввода-вывода. Такие процессы демонстрируют **сотрудничество** при совместном использовании общего объекта.
- **Процессы непосредственно осведомлены о наличии друг друга.** Такие процессы способны общаться один с другим с использованием идентификаторов процессов и изначально созданы для совместной работы. Также они демонстрируют **сотрудничество** при работе.

Условия работы процессов не всегда можно определить так ясно и четко, как указано в табл. 5.2; более того, некоторые процессы одновременно проявляют способность и к конкуренции, и к сотрудничеству. Тем не менее имеет смысл рассмотреть приведенный список и определить участие операционной системы в перечисленных в нем ситуациях.

**Таблица 5.2. Взаимодействие процессов**

Степень осведомленности	Взаимосвязь	Влияние одного процесса на другой	Потенциальные проблемы
Процессы не осведомлены один о другом	Конкуренция	<ul style="list-style-type: none"> <li>Результат работы одного процесса не зависит от действий других</li> <li>Возможно влияние одного процесса на время работы другого</li> </ul>	<ul style="list-style-type: none"> <li>Взаимоисключения</li> <li>Взаимоблокировки (возобновляемые ресурсы)</li> <li>Голодание</li> </ul>
Процессы косвенно осведомлены один о другом	Сотрудничество с использованием общих ресурсов	<ul style="list-style-type: none"> <li>Результат работы одного процесса может зависеть от информации, полученной от других процессов</li> <li>Возможно влияние одного процесса на время работы другого</li> </ul>	<ul style="list-style-type: none"> <li>Взаимоисключения</li> <li>Взаимоблокировки (возобновляемые ресурсы)</li> <li>Голодание</li> <li>Связь данных</li> </ul>
Процессы непосредственно осведомлены один о другом	Сотрудничество с использованием связи	<ul style="list-style-type: none"> <li>Результат работы одного процесса может зависеть от информации, полученной от других</li> <li>Возможно влияние одного процесса на время работы другого</li> </ul>	<ul style="list-style-type: none"> <li>Взаимоблокировки (расходуемые ресурсы)</li> <li>Голодание</li> </ul>

### **Конкуренция процессов в борьбе за ресурсы**

При необходимости использовать один и тот же ресурс параллельные процессы вступают в конфликт друг с другом. В чистом виде ситуацию можно описать следующим образом. В процессе работы два или более процессов нуждаются в доступе к некоторому ресурсу. Каждый из процессов не подозревает о наличии остальных и не подвергается никакому воздействию с их стороны. Отсюда следует, что каждый процесс не должен изменять состояние любого ресурса, с которым он работает. Примерами таких ресурсов могут быть устройства ввода-вывода, память, процессорное время и часы.

Между конкурирующими процессами не происходит никакого обмена информацией. Однако выполнение одного процесса может повлиять на поведение конкурирующего процесса. В частности, если два процесса желают получить доступ к одному ресурсу, то операционная система выделит этот ресурс одному из процессов, в то время как второй процесс вынужден будет ожидать завершения работы с ресурсом первого. Таким образом, скорость работы процесса, которому отказано в немедленном доступе к ресурсу, уменьшается. В предельном случае блокированный процесс может никогда не получить доступ к ресурсу и, следовательно, никогда не сможет успешно завершиться.

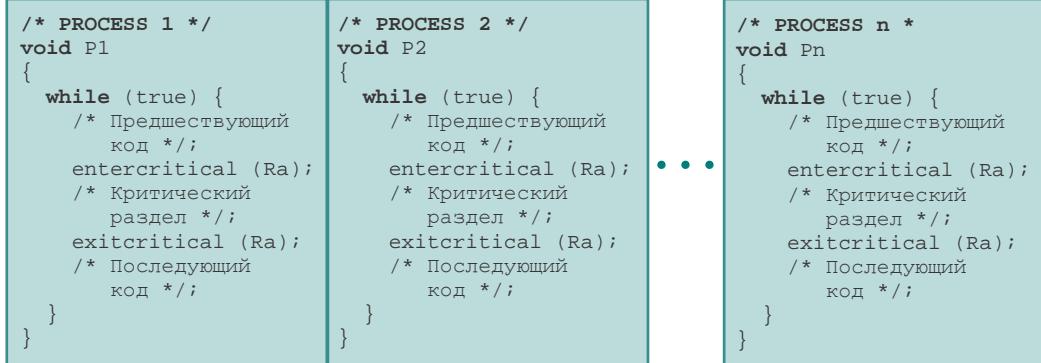
В случае конкуренции процессов мы сталкиваемся с тремя проблемами. Первая из них — необходимость **взаимных исключений** (*mutual exclusion*). Предположим, что два или больше процессов требуют доступа к одному неразделимому ресурсу, такому как принтер. При выполнении каждый процесс посыпает команды в устройство ввода-вывода, получает информацию о его состоянии, посыпает и/или получает данные. Мы будем говорить о таком ресурсе как о *критическом ресурсе*, а о части программы, которая его использует, — как о *критическом участке* (*critical section*) программы. Крайне важно, чтобы в критическом участке в любой момент времени могла находиться только одна программа. Мы не можем полагаться на то, что операционная система распознает ситуацию и выполнит это условие, поскольку полные требования к ресурсу могут оказаться не очевидными. Например, во время печати файла требуется, чтобы отдельный процесс имел полный контроль над принтером, иначе на бумаге можно получить чередование строк двух файлов.

Осуществление взаимных исключений создает две дополнительные проблемы. Одна из них — **взаимная блокировка** (*deadlock*). Рассмотрим, например, два процесса ( $P_1$  и  $P_2$ ) и два ресурса ( $R_1$  и  $R_2$ ). Предположим, что каждому процессу для выполнения части своих функций требуется доступ к обоим ресурсам. Тогда возможно возникновение следующей ситуации: операционная система выделяет ресурс  $R_1$  процессу  $P_2$ , а ресурс  $R_2$  — процессу  $P_1$ . В результате каждый процесс ожидает получения одного из двух ресурсов; при этом ни один из них не освобождает уже имеющийся у него ресурс, ожидая получения второго ресурса для выполнения функций, требующих наличия двух ресурсов. В результате процессы оказываются взаимно заблокированными.

Последняя проблема — **голодание** (*starvation*). Предположим, что у нас имеются три процесса ( $P_1$ ,  $P_2$ ,  $P_3$ ), каждому из которых периодически требуется доступ к ресурсу  $R$ . Представим ситуацию, в которой  $P_1$  обладает ресурсом, а  $P_2$  и  $P_3$  приостановлены в ожидании освобождения ресурса. После выхода  $P_1$  из критического раздела доступ к ресурсу будет получен одним из процессов  $P_2$  и  $P_3$ . Пусть операционная система предоставила доступ к ресурсу  $R$  процессу  $P_3$ . Пока он работает с ресурсом, доступ к ресурсу вновь требуется процессу  $P_1$ . В результате после освобождения ресурса процессом  $P_3$  может оказаться, что операционная система вновь предоставила доступ к ресурсу процессу  $P_1$ ; тем временем процессу  $P_3$  вновь требуется доступ к ресурсу  $R$ . Таким образом, теоретически возможна ситуация, в которой процесс  $P_2$  никогда не получит доступа к требуемому ему ресурсу, несмотря на то что никакой взаимной блокировки в этом случае нет.

Управление конкуренцией неизбежно приводит к участию операционной системы в этом процессе, поскольку именно она распределяет ресурсы. Кроме того, процессам необходима возможность запрашивать взаимоисключение, такое как блокировка ресурса перед его использованием. Любое решение этого вопроса требует поддержки операционной системы, например такой, как обеспечение возможности блокировки. На рис. 5.4 показан абстрактный механизм взаимоисключений. Здесь параллельно выполняются *n* процессов. Каждый процесс включает 1) критический участок, работающий с некоторым ресурсом  $R_a$ , и 2) остальную часть процедуры, в которой нет обращения к ресурсу. Для обеспечения взаимоисключения имеются две функции: `entercritical` и `exitcritical`. Каждая из них принимает в качестве аргумента имя ресурса, являющегося предметом конкуренции. Любой процесс, который пытается войти в критический участок в то время, как в нем находится другой процесс, будет приостановлен.

Нам остается только рассмотреть механизм, обеспечивающий работу функций `entercritical` и `exitcritical`. Однако пока что мы отложим этот вопрос и приступим к рассмотрению других случаев взаимодействия процессов.



**Рис. 5.4.** Иллюстрация взаимоисключений

### Сотрудничество процессов с применением совместного использования

Случай сотрудничества процессов с применением совместного использования охватывает процессы, взаимодействующие с другими процессами без наличия явной информации о них. Например, несколько процессов могут обращаться к совместно используемым переменным или к совместно используемым файлам или базам данных. Процессы могут использовать и обновлять общие данные без обращения к другим процессам, но с учетом того, что последние также могут обращаться к тем же данным. Таким образом, процессы должны сотрудничать, для того чтобы гарантировать корректную работу с совместно используемыми данными. Механизм управления доступом должен гарантировать целостность общих данных.

Поскольку данные хранятся в ресурсах (устройствах, памяти), в этом случае также наличествуют проблемы взаимоблокировок, взаимоисключений и голодания. Единственное отличие заключается в том, что доступ к данным может осуществляться в двух режимах — чтения и записи, и взаимоисключающими должны быть только операции записи.

Однако в данном случае вносится новое требование — согласованности данных. В качестве простейшего примера рассмотрим бухгалтерское приложение, в котором могут обновляться различные данные. Предположим, что два элемента данных,  $a$  и  $b$ , должны быть связаны соотношением  $a = b$ , так что любая программа, изменяющая одно значение, обязана изменить и другое, с тем чтобы это соотношение продолжало выполняться. Теперь рассмотрим следующие два процесса:

```

P1:
    a = a + 1;
    b = b + 1;

P2:
    b = 2 * b;
    a = 2 * a;

```

Если изначально состояние данных согласованное, то каждый процесс в отдельности не нарушает согласованности данных. Но что если при параллельном вычислении будет выполнена такая последовательность действий, которая соблюдает условия взаимоисключений при работе с каждым элементом данных ( $a$  и  $b$ )?

```
a = a + 1;
b = 2 * b;
b = b + 1;
a = 2 * a;
```

После выполнения этой последовательности действий условие  $a = b$  становится неверным. Например, если изначально  $a = b = 1$ , то по завершении вычислений  $a = 4$  и  $b = 3$ . Проблема решается путем объявления критическим участком каждой из последовательностей инструкций.

Таким образом, значение концепции критических участков не уменьшается и в случае сотрудничества с применением совместного использования. Здесь также могут использоваться рассмотренные нами ранее (см. рис. 5.4) абстрактные функции `entercritical` и `exitcritical`. В данном случае аргументами этих функций могут быть переменные, файлы или любые другие общие объекты. Более того, если критические участки используются для обеспечения целостности данных, то выступающего в роли аргумента функции определенного ресурса или определенной переменной может и не существовать. В таком случае мы можем рассматривать аргумент как идентификатор, совместно используемый параллельными процессами и определяющий критический участок кода, который должен быть защищен взаимным исключением.

### **Сотрудничество с использованием связи**

В рассмотренных нами случаях каждый процесс имел собственное изолированное окружение, не включающее в себя другие процессы. Взаимодействие между процессами было сугубо косвенным, и в обоих случаях наблюдалось совместное использование. В случае конкуренции процессы совместно использовали ресурсы, не имея информации о существовании друг друга; в случае сотрудничества процессы, не будучи осведомленными явно о наличии других процессов, тем не менее принимают меры к поддержанию целостности данных. При сотрудничестве с использованием связи различные процессы принимают участие в общей работе, которая и объединяет их. Связь обеспечивает возможность синхронизации, или координации, различных действий процессов.

Обычно можно считать, что связь состоит из сообщений определенного вида. Примитивы для отправки и получения сообщений могут быть предоставлены языком программирования или ядром операционной системы.

Поскольку в процессе передачи сообщений не происходит какого-либо совместного использования ресурсов, в этом случае сотрудничества взаимоисключений не требуются (хотя проблемы взаимоблокировок и голодания остаются актуальными). В качестве примера взаимоблокировки можно привести ситуацию, при которой каждый из двух процессов заблокирован ожиданием сообщения от другого процесса. Голодание можно проиллюстрировать следующим примером. Рассмотрим три процесса: P1, P2 и P3. Процесс P1 многократно пытается связаться с процессами P2 и P3, а те, в свою очередь, пытаются связаться с процессом P1. Может возникнуть ситуация, когда процессы P1 и P2 постоянно связываются друг с другом, а процесс P3 остается заблокированным, ожидая связи с процессом P1. Это не взаимоблокировка, поскольку процесс P1 при этом остается активными.

## Требования к взаимным исключениям

Любая возможность обеспечения поддержки взаимных исключений должна соответствовать следующим требованиям.

1. Взаимоисключения должны осуществляться в принудительном порядке. В любой момент времени из всех процессов, имеющих критический участок для одного и того же ресурса или общего объекта, в этом участке может находиться лишь только один процесс.
2. Процесс, завершающий работу в некритическом участке, не должен влиять на другие процессы.
3. Не должна возникать ситуация бесконечного ожидания доступа к критическому участку (т.е. не должны появляться взаимоблокировки и голодание).
4. Когда в критическом участке нет ни одного процесса, любой процесс, запросивший возможность входа в него, должен немедленно ее получить.
5. Не делаются никаких предположений о количестве процессов или их относительных скоростях работы.
6. Процесс остается в критическом участке только в течение ограниченного времени.

Имеется ряд способов удовлетворения перечисленных условий. Одним из них является передача ответственности за соответствие требованиям самому процессу, который должен выполняться параллельно. Таким образом, процесс, независимо от того, является ли он системной программой или приложением, должен координировать свои действия с другими процессами для работы взаимоисключений без поддержки со стороны языка программирования или операционной системы. Мы можем говорить о таком подходе как о программном. Хотя этот подход чреват большими накладными расходами и возможными ошибками, чрезвычайно полезно рассмотреть его для лучшего понимания сложностей, связанных с параллельными вычислениями. Этот вопрос был рассмотрен в предыдущем разделе. Другой подход, рассматриваемый в разделе 5.3, включает использование машинных команд специального назначения. Преимущество этого подхода заключается в снижении накладных расходов, но такой подход в общем случае проблему не решает. Еще один подход заключается в предоставлении определенного уровня поддержки со стороны операционной системы или языка программирования. Наиболее важные варианты такого подхода к решению проблемы взаимоисключений рассматриваются в разделах 5.4–5.6.

## 5.3. Взаимоисключения: аппаратная поддержка

В этом разделе мы рассмотрим несколько интересных аппаратных подходов к решению вопроса взаимоисключений.

### Отключение прерываний

Когда в машине имеется лишь один процессор, параллельные процессы не могут перекрываться, они способны только чередоваться. Кроме того, процесс будет продолжаться до тех пор, пока не будет вызвана служба операционной системы или пока процесс

не будет прерван. Следовательно, для того чтобы гарантировать взаимное исключение, достаточно защитить процесс от прерывания. Эта возможность может быть обеспечена в форме примитивов, определенных ядром операционной системы для запрета и разрешения прерываний. Процесс в таком случае может обеспечить взаимоисключение следующим образом (сравните с рис. 5.4):

```
while (true)
{
    /* Запрет прерываний */;
    /* Критический участок */;
    /* Разрешение прерываний */;
    /* Остальной код */;
}
```

Поскольку критический раздел не может быть прерван, выполнение взаимоисключения гарантируется. Однако цена такого подхода высока. Эффективность работы может заметно снизиться, поскольку при этом ограничена возможность процессора по чередованию программ. Другая проблема заключается в том, что такой подход не будет работать в многопроцессорной архитектуре. Если вычислительная система включает несколько процессоров, то вполне возможно (и обычно так и бывает), что несколько процессов выполняются одновременно. В этом случае запрет прерываний не гарантирует взаимоисключение.

## Специальные машинные команды

В многопроцессорной конфигурации несколько процессоров разделяют доступ к общей основной памяти. В этом случае отсутствует отношение “ведущий/ведомый” (master/slave) — процессоры работают независимо, “на равных”, и не существует механизма прерывания, на котором могли бы основываться взаимоисключения.

На уровне аппаратного обеспечения, как уже упоминалось, обращение к ячейке памяти исключает любые другие обращения к той же ячейке. Основываясь на этом принципе, разработчики процессоров предлагают ряд машинных команд, которые за один цикл выборки команды атомарно<sup>3</sup> выполняют над ячейкой памяти два действия, такие как чтение и запись или чтение и проверка значения. Поскольку эти действия выполняются в одном цикле выборки, на них не в состоянии повлиять никакие другие инструкции.

В этом разделе мы рассмотрим две из наиболее часто реализуемых инструкций (с остальными инструкциями вы можете познакомиться в [198]).

### Команда сравнения и присваивания

Команда сравнения и присваивания, compare&swap, может быть определена следующим образом [104]:

```
int compare_and_swap(int* word, int testval, int newval)
{
    int oldval;
    oldval = *word;
    if (oldval == testval) *word = newval;
    return oldval;
}
```

---

<sup>3</sup> Термин *атомарно* (atomic) означает, что команда рассматривается как единое действие, которое не может быть прервано.

Данная версия команды проверяет ячейку памяти (`*word`), сравнивая ее значение с тестовым (`testval`). Если текущее значение ячейки памяти равно `testval`, оно заменяется значением `newval`; в противном случае значение ячейки памяти остается неизменным. Всегда возвращается старое значение ячейки памяти; таким образом, ячейка памяти обновляется, если возвращаемое значение совпадает с тестовым. Эта атомарная команда состоит из двух частей (сравнение значений в ячейке памяти и тестового значения), и, если эти значения совпадают, выполняется присваивание. Вся функция выполняется атомарно, т.е. она не может быть прервана.

Другая версия данной команды возвращает логическое значение: `true`, если присваивание состоялось, и `false` в противном случае. Та или иная версия команды доступна практически на всех семействах процессоров (x86, IA64, sparc, IBM z series и т.д.), и большинство операционных систем использует эту команду для поддержки параллельности.

На рис. 5.5, а показан протокол взаимного исключения, основанный на использовании описанной команды<sup>4</sup>.

Совместно используемая переменная `bolt` инициализируется значением 0. Только тот процесс, который обнаруживает, что значение `bolt` равно 0, может войти в критический участок. Все другие процессы, пытающиеся войти в критический участок, переходят в режим пережидания занятости.

```
/* program mutual exclusion */

const int n = /* Количество процессов */;
int bolt;

void P(int i)
{
    while (true) {
        while (compare_and_swap(bolt, 0, 1)
               == 1)
            /* Ничего не делать */;

        /* Критический раздел */;
        bolt = 0;
        /* Остальной код */;
    }
}

void main() {
    bolt = 0;
    parbegin (P(1), P(2), ..., P(n));
}
```

а) Команда сравнения и обмена

```
/* program mutual exclusion */

int const n = /* Количество процессов */;
int bolt;

void P(int i)
{
    while (true)
        int keyi = 1;
        do exchange (&keyi, &bolt)
        while (keyi != 0);
        /* Критический раздел */;
        bolt = 0;
        /* Остальной код */;
    }
}

void main() {
    bolt = 0;
    parbegin (P(1), P(2), ..., P(n));
}
```

б) Команда обмена

**Рис. 5.5.** Аппаратная поддержка взаимоисключений

<sup>4</sup> Конструкция `parbegin(P1, P2, ..., Pn)` означает следующее: приостановка выполнения основной программы; инициализация параллельного выполнения процедур P<sub>1</sub>, P<sub>2</sub>, ..., P<sub>n</sub>; по окончании работы процедур P<sub>1</sub>, P<sub>2</sub>, ..., P<sub>n</sub> — возобновление выполнения основной программы.

Термин **пережидание занятости** (*busy waiting*, *spin waiting*) относится к методике, в соответствии с которой процесс до тех пор, пока не получит разрешение войти в критический участок, не может ничего делать, кроме как выполнять инструкции по проверке соответствующей переменной для получения разрешения на вход. Когда процесс покидает критический участок, он сбрасывает значение переменной *bolt* в 0; в этот момент один и только один из ожидающих процессов получает доступ к критическому участку. Выбор процесса зависит от того, какой процесс первым выполнит команду сравнения и присваивания.

### Команда обмена

Команда обмена может быть определена следующим образом:

```
void exchange(int* register, int* memory)
{
    int temp;
    temp = *memory;
    *memory = *register;
    *register = temp;
}
```

Эта команда обменивает содержимое регистра и ячейки памяти. Команда XCNG имеется как в архитектуре Intel IA-32 (Pentium), так и в архитектуре IA-64 (Itanium).

На рис. 5.5, б показан протокол взаимного исключения, основанный на использовании этой команды. Совместно используемая переменная *bolt* инициализируется нулевым значением. У каждого процесса имеется локальная переменная *key<sub>i</sub>*, инициализированная значением 1. В критический участок может войти только один процесс, который обнаруживает, что значение *bolt* равно 0. Этот процесс запрещает вход в критический участок всем другим процессам путем установки значения *bolt* равным 1. По окончании работы в критическом участке процесс вновь сбрасывает значение *bolt* в 0, тем самым позволяя другому процессу войти в критический участок.

Заметим, что при использовании рассмотренного алгоритма всегда выполняется следующее соотношение:

$$\text{bolt} + \sum_i \text{key}_i = n$$

Если *bolt* = 0, то в критическом участке нет ни одного процесса. Если *bolt* = 1, то в критическом участке находится ровно один процесс, а именно тот, переменная *key<sub>i</sub>* которого имеет нулевое значение.

### Свойства подхода, основанного на использовании машинных инструкций

Подход, основанный на использовании специальной машинной инструкции для осуществления взаимных исключений, имеет ряд преимуществ.

- Применим к любому количеству процессов при наличии как одного, так и нескольких процессоров, совместно использующих основную память.
- Очень прост, а потому легко проверяем.
- Может использоваться для поддержки множества критических участков; каждый из них может быть определен при помощи собственной переменной.

Однако у такого подхода имеются и серьезные недостатки.

- **Используется пережидание занятости.** Следовательно, в то время как процесс находится в ожидании доступа к критическому участку, он продолжает потреблять процессорное время.
- **Возможно голодание.** Если процесс покидает критический участок, а входа в него ожидают несколько других процессов, то выбор ожидающего процесса произведен. Следовательно, может оказаться, что какой-то из процессов будет ожидать входа в критический участок бесконечно.
- **Возможна взаимоблокировка.** Рассмотрим следующий сценарий в однопроцессорной системе. Процесс P1 выполняет специальную инструкцию (т.е. compare\_and\_swap или exchange) и входит в критический раздел. После этого процесс P1 прерывается процессом P2 с более высоким приоритетом. Если P2 попытается обратиться к тому же ресурсу, что и P1, ему будет отказано в доступе в соответствии с механизмом взаимоисключений, и он войдет в цикл пережидания занятости. Однако в силу того, что процесс P1 имеет более низкий приоритет, он не получит возможности продолжить работу, так как в наличии имеется активный процесс с высоким приоритетом.

Из-за наличия недостатков в случае использования как программных, так и аппаратных решений нам следует рассмотреть и другие механизмы обеспечения взаимоблокировок.

## 5.4. СЕМАФОРЫ

Теперь мы вернемся к механизмам операционных систем и языков программирования, обеспечивающим параллельные вычисления. В табл. 5.3 представлены основные широко распространенные механизмы. Этот раздел мы начнем с рассмотрения семафоров; следующие разделы будут посвящены мониторам и передаче сообщений. Прочие механизмы в табл. 5.3 будут рассмотрены в главах 6, “Параллельные вычисления: взаимоблокировка и голодание”, и 13, “Встроенные операционные системы”.

**Таблица 5.3. Основные механизмы параллельных вычислений**

<b>Семафор (Semaphore)</b>	Целочисленное значение, используемое для передачи сигналов между процессами. Над семафором могут быть выполнены только три операции (все они являются атомарными): инициализация, уменьшение (декремент) и увеличение (инкремент) значения. Операция уменьшения может привести к блокировке процесса, а операция увеличения — к разблокированию. Известен также как семафор со счетчиком (counting semaphore) или обобщенный семафор (general semaphore)
<b>Бинарный семафор (Binary semaphore)</b>	Семафор, который может принимать только два значения — 0 и 1
<b>Мьютекс (Mutex)</b>	Аналогичен бинарному семафору. Ключевым отличием является то, что процесс, блокирующий мьютекс (устанавливающий его значение равным 0), должен и разблокировать его (установить его значение равным 1)

Окончание табл. 5.3

<b>Условная переменная (Condition variable)</b>	Тип данных, используемый для блокировки процесса или потока до тех пор, пока не станет истинным некоторое условие
<b>Монитор (Monitor)</b>	Конструкция языка программирования, инкапсулирующая переменные, процедуры доступа и код инициализации, в абстрактном типе данных. Переменные монитора могут быть доступны только через его процедуры доступа, и в любой момент времени только один процесс может активно работать с монитором. Процедуры доступа представляют собой <i>критические участки</i> . Монитор может иметь очередь процессов, ожидающих доступа к нему
<b>Флаги событий (Event flags)</b>	Слово памяти, используемое как механизм синхронизации. Код приложения может связать с каждым битом флага свое событие. Поток может ждать либо одного события, либо сочетания событий путем проверки одного или нескольких битов в соответствующем флаге. Поток блокируется до тех пор, пока все необходимые биты не будут установлены (И) или пока не будет установлен хотя бы один из битов (ИЛИ)
<b>Почтовые ящики/ сообщения (Mailboxes/messages)</b>	Средство обмена информацией между двумя процессами, которое может быть использовано для синхронизации
<b>Спин-блокировки (Spinlocks)</b>	Механизм взаимоисключения, в котором процесс выполняется в бесконечном цикле, ожидая, когда значение блокирующей переменной укажет доступность критического участка

Первой большой работой, посвященной вопросам параллельных вычислений, стала монография Дейкстры [66], который рассматривал разработку операционной системы как построение множества сотрудничающих последовательных процессов и создание эффективных и надежных механизмов поддержки этого сотрудничества. Эти же механизмы легко применяются и пользовательскими процессами — если процессор и операционная система делают их общедоступными.

Фундаментальный принцип заключается в том, что два или более процессов могут сотрудничать посредством простых сигналов, так что в определенном месте процесс может приостановить работу до тех пор, пока не дождется соответствующего сигнала. Требования кооперации любой степени сложности могут быть удовлетворены соответствующей структурой сигналов. Для сигнализации используются специальные переменные, называемые семафорами. Для передачи сигнала через семафор `s` процесс выполняет примитив `semSignal(s)`, а для его получения — примитив `semWait(s)`. В последнем случае процесс приостанавливается до тех пор, пока не осуществляется передача соответствующего сигнала.<sup>5</sup>

Для достижения желаемого эффекта мы можем рассматривать семафор как переменную, имеющую целое значение, над которой определены три операции.

<sup>5</sup> В статье Дейкстры и во многих других источниках вместо `wait` используется буква `P`, а вместо `signal` — `V`; это первые буквы голландских слов *роверка* (*proberen*) и *увеличение* (*verhogen*). В других источниках встречаются термины `wait` и `signal`. В этой книге для ясности и во избежание коллизии с аналогичными операциями мониторов использованы термины `semWait` и `semSignal`.

1. Семафор может быть инициализирован неотрицательным целочисленным значением.
2. Операция `semWait` уменьшает значение семафора. Если это значение становится отрицательным, процесс, выполняющий операцию `semWait`, блокируется.
3. Операция `semSignal` увеличивает значение семафора. Если это значение меньше или равно нулю, заблокированный операцией `semWait` процесс (если таковой имеется) деблокируется.

Не имеется никаких иных способов получения информации о значении семафора или изменения его значения, кроме перечисленных.

Объяснить эти операции можно следующим образом. В начале работы семафор имеет значение нуль или некоторое положительное значение. Если значение положительное, то оно равно количеству процессов, которые могут вызвать операцию получения сигнала и немедленно продолжить выполнение. Если же значение равно нулю (полученное либо при инициализации, либо потому, что количество процессов, равное первоначальному значению семафора, вызвало операцию ожидания), очередной ожидающий процесс блокируется, а значение семафора становится отрицательным. Каждое последующее ожидание уменьшает значение семафора, так что оно имеет отрицательное значение, по модулю равное числу процессов, ожидающих разблокирования. Когда значение семафора отрицательное, каждый сигнал разблокирует один из ожидающих процессов.

Есть три интересных следствия определения семафора.

1. В общем случае до выполнения операции декремента нет никакого способа узнать, будет ли процесс заблокирован.
2. После того как один процесс увеличивает значение семафора, а другой процесс активируется, оба процесса продолжают выполняться одновременно. Нет никакого способа узнать, какой процесс (если таковой имеется) будет немедленно продолжать выполнение в однопроцессорной системе.
3. При передаче сигнала семафору вам не обязательно знать, находится ли другой процесс в состоянии ожидания, поэтому количество разблокированных процессов может быть равно нулю или одному.

На рис. 5.6 приведено более формальное определение примитивов семафоров. Предполагается, что примитивы `semWait` и `semSignal` атомарны, т.е. не могут быть прерваны. Более ограниченная версия семафора, известная как **бинарный семафор**, представлена на рис. 5.7. Бинарный семафор может принимать только значения 0 или 1 и может быть определен с помощью следующих трех операций.

1. Бинарный семафор может быть инициализирован значением 0 или 1.
2. Операция `semWaitB` проверяет значение семафора. Если это значение нулевое, процесс, выполняющий `semWaitB`, блокируется. Если значение равно 1, оно изменяется, становясь равным 0, и выполнение процесса продолжается.
3. Операция `semSignalB` проверяет, имеется ли процесс, заблокированный этим семафором (значение семафора равно 0). Если есть, то процесс, заблокированный операцией `semWaitB`, разблокируется. Если заблокированных процессов нет, значение семафора устанавливается равным 1.

```

struct semaphore {
    int count;
    queueType queue;
};

void semWait(semaphore s)
{
    s.count--;
    if (s.count < 0) {
        /* Процесс помещается в s.queue */;
        /* Блокировка данного процесса */;
    }
}

void semSignal(semaphore s)
{
    s.count++;
    if (s.count<= 0) {
        /* Удаление процесса P из s.queue */;
        /* Перемещение процесса P в список готовности */;
    }
}

```

**Рис. 5.6.** Определение примитивов семафора

```

struct binary_semaphore {
    enum {zero, one} value;
    queueType queue;
};

void semWaitB(binary_semaphore s)
{
    if (s.value == one)
        s.value = zero;
    else {
        /* Процесс помещается в s.queue */;
        /* Блокировка данного процесса */;
    }
}

void semSignalB(semaphore s)
{
    if (s.queue is empty())
        s.value = one;
    else {
        /* Удаление процесса P из s.queue */;
        /* Перемещение процесса P в список готовности */;
    }
}

```

**Рис. 5.7.** Определение примитивов бинарного семафора

С бинарным семафором тесно связана концепция **мьютекса** (взаимоисключающей блокировки — *mutual exclusion lock (mutex)*). Мьютекс представляет собой программный флаг, используемый для захвата и освобождения объекта. При захвате данных, не могущих быть общими, или при запуске обработки, которая не может одновременно выполняться в других местах в системе, мьютекс устанавливается в состояние блокировки (обычно 0), которое предотвращает другие попытки его использования. Мьютекс разблокируется, когда данные больше не нужны или процедура закончена. Ключевое отличие мьютекса от бинарного семафора заключается в том, что процесс, блокирующий мьютекс (устанавливающий его значение равным нулю) должен быть тем же, что и разблокирующий его (устанавливающий его значение равным 1). В случае бинарного семафора он может быть заблокирован одним процессом, а разблокирован — другим<sup>6</sup>.

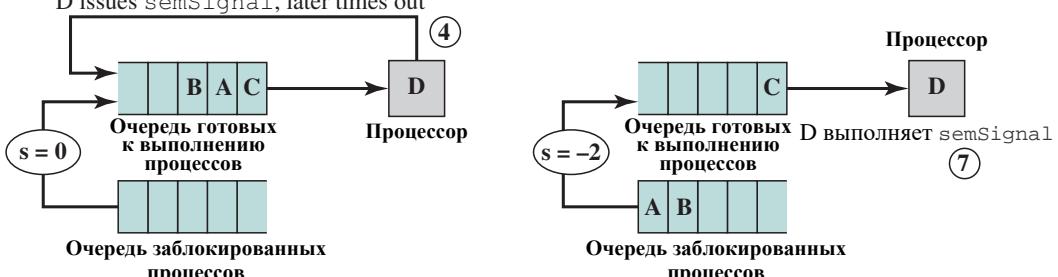
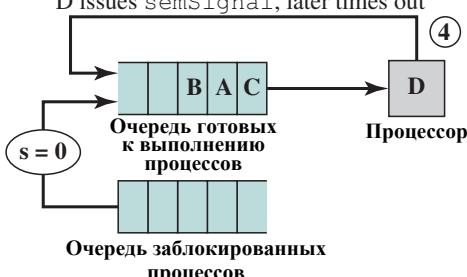
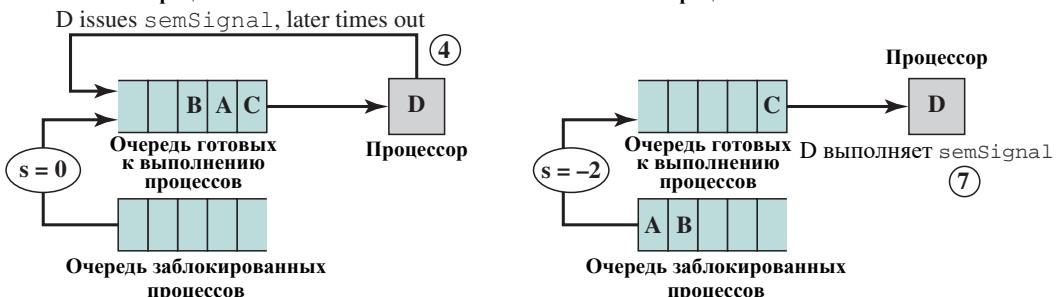
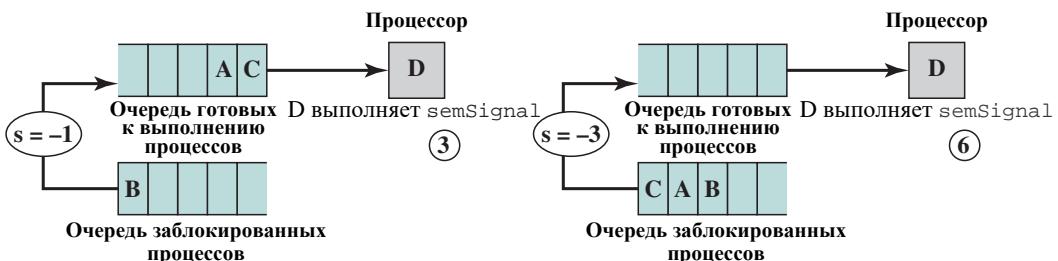
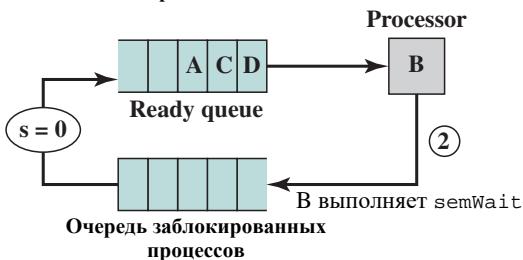
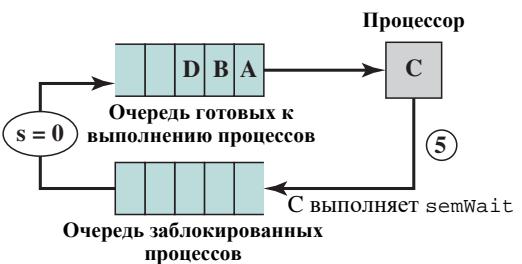
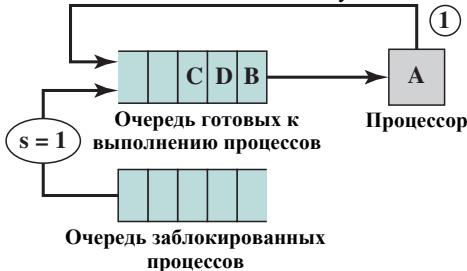
Для хранения процессов, ожидающих как обычных, так и бинарных семафоров, используется очередь. При этом возникает вопрос о порядке извлечения процессов из данной очереди. Наиболее корректный способ — использование принципа “первым вошел — первым вышел” (*first-in-first-out — FIFO*). При этом первым из очереди освобождается процесс, который был заблокирован дольше других. Семафор, использующий данный метод, называется **сильным семафором** (*strong semaphore*). Семафор, порядок извлечения процессов из очереди которого не определен, называется **слабым семафором** (*weak semaphore*). На рис. 5.8 приведен пример работы сильного семафора. Здесь процессы А, В и С зависят от результатов работы процесса D. Изначально работает процесс А (①); процессы В, С и D находятся в списке активных процессов, ожидая своей очереди. Значение семафора равно 1, это указывает на то, что один из результатов работы процесса D имеется в наличии. Когда процесс А выполняет инструкцию `semWait`, семафор уменьшается до 0, а процесс А продолжает выполнять и позже вновь становится в очередь на выполнение в списке активных процессов. Затем приступает к работе процесс В (②), который в конечном счете также выполняет инструкцию `semWait`, в результате чего процесс приостанавливается, давая возможность приступить к работе процессу D (③). Когда процесс D завершает работу над получением нового результата, он выполняет инструкцию `semSignal`, которая позволяет процессу В перейти из списка приостановленных процессов в список активных (④). Процесс D присоединяется к очереди активных процессов, и к выполнению приступает процесс С (⑤), но тут же приостанавливается при выполнении инструкции `semWait`. Точно так же приостанавливается и выполнение процессов А и В, давая возможность процессу D приступить к работе (⑥). После того как получается новый результат процесса D, им выполняется инструкция `semSignal`, которая переводит процесс С из списка приостановленных в очередь активных процессов. Последующие циклы выполнения процесса D деблокируют процессы А и В.

В следующем подразделе рассматривается алгоритм взаимоисключений (рис. 5.9), использование сильного семафора в котором гарантирует невозможность голодания, в то время как слабый семафор такой гарантии не дает. Далее мы будем считать, что работают сильные семафоры, поскольку они более удобны и обычно именно этот вид семафоров используется операционной системой.

---

<sup>6</sup> В ряде книг и учебников не делается никаких различий между мьютексом и бинарным семафором. Однако на практике ряд операционных систем, таких как Linux, Windows или Solaris, предоставляют мьютексы, отвечающие определениям из данной книги.

А выполняет semWait с последующим тайм-аутом



**Рис. 5.8.** Пример работы семафора

```

/* program mutual_exclusion */
const int n = /* Количество процессов */;
semaphore s = 1;
void P(int i)
{
    while (true) {
        semWait(s);
        /* Критический участок */;
        semSignal(s);
        /* Остальной код */;
    }
}
void main()
{
    parbegin (P(1), P(2), . . . , P(n));
}

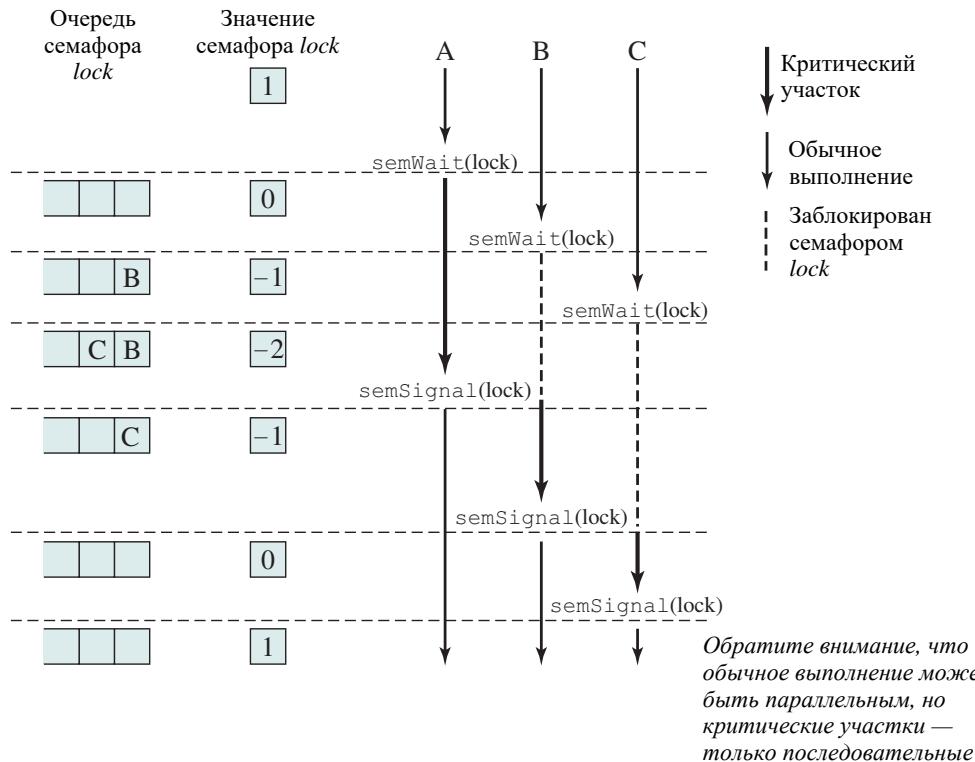
```

**Рис. 5.9.** Взаимоисключений с использованием семафоров

На рис. 5.9 показано простое решение задачи взаимоисключений с использованием семафора *s* (сравните с рис. 5.4). Пусть у нас имеется *n* процессов, идентифицируемых массивом *P(i)*. В каждом из процессов перед входом в критический раздел выполняется вызов *semWait(s)*. Если значение *s* становится отрицательным, процесс приостанавливается. Если же значение равно 1, оно уменьшается до нуля и процесс немедленно входит в критический участок; поскольку *s* больше не является положительным, ни один другой процесс не может войти в критический участок.

Семафор инициализируется значением 1. Следовательно, первый процесс, выполняющий инструкцию *semWait*, сможет немедленно попасть в критический участок, устанавливая при этом значение семафора равным 0. Любой другой процесс при попытке войти в критический участок обнаружит, что он занят. Соответственно, произойдет блокировка процесса, а значение семафора будет уменьшено до -1. Пытаться войти в критический участок может любое количество процессов; каждая неудачная попытка уменьшает значение семафора. После того как процесс, вошедший в критический участок первым, покидает его, *s* увеличивается, и один из заблокированных процессов (если таковые имеются) удаляется из связанной с семафором очереди заблокированных процессов и активизируется. Таким образом, как только планировщик операционной системы предоставит ему возможность выполнения, процесс тут же сможет войти в критический раздел.

На рис. 5.10 ([15]) показана возможная последовательность действий трех процессов при использовании технологии взаимоисключений, представленной на рис. 5.9. В этом примере три процесса (A, B, C) обращаются к совместно используемому ресурсу, защищенному семафором *lock*. Процесс A выполняет *semWait(lock)*; поскольку в этот момент семафор имеет значение 1, процесс A может немедленно войти в критический участок и значение семафора становится равным 0. Пока A находится в критическом участке, и B, и C выполняют операцию *semWait*, после чего в заблокированном состоянии ожидают доступности критического участка. Когда процесс A покидает критический участок и выполняет операцию *semSignal(lock)*, процесс B (являющийся первым в очереди) получает возможность войти в критический участок.



**Рис. 5.10.** Доступ процессов к общим данным, защищенным семафором

Программа, приведенная на рис. 5.9, может так же хорошо работать и в том случае, когда одновременно в критическом участке находятся несколько процессов. Для этого достаточно инициализировать семафор соответствующим значением. Таким образом, в любой момент времени значение `s.count` интерпретируется следующим образом.

- $s.count \geq 0$ : значение `s.count` определяет количество процессов, которые могут выполнить `semWait(s)` без приостановки процесса (подразумевается, что промежуточные вызовы `semSignal(s)` отсутствуют). Это позволяет семафорам поддерживать как синхронизацию, так и взаимоисключений.
- $s.count < 0$ : абсолютное значение `s.count` определяет количество приостановленных процессов в очереди `s.queue`.

## Задача производителя/потребителя

Сейчас мы рассмотрим одну из распространенных задач параллельных вычислений — задачу производителя/потребителя. Вот ее обобщенная формулировка. Имеется один или несколько производителей, генерирующих данные некоторого типа (записи, символы и т.п.) и помещающих их в буфер, а также единственный потребитель, который извлекает помещенные в буфер элементы по одному. Требуется защитить систему от перекрытия операций с буфером, т.е. обеспечить, чтобы одновременно получить доступ к буферу мог только один процесс (производитель или потребитель). Проблема заключа-

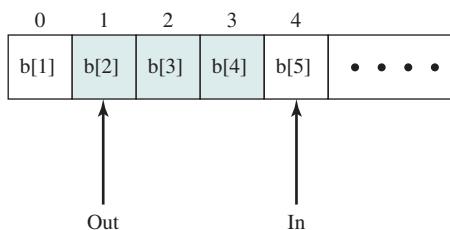
ется в том, чтобы гарантировать, что производитель не будет пытаться добавить данные в буфер, если он заполнен, и что потребитель не будет пытаться удалить данные из пустого буфера. Мы рассмотрим несколько решений этой задачи, с тем чтобы проиллюстрировать как мощь семафоров, так и встречающиеся при их использовании ловушки.

Для начала предположим, что буфер бесконечен и представляет собой линейный массив элементов. Говоря абстрактно, мы можем определить функции производителя и потребителя следующим образом:

```
/* Производитель */
while(true)
{
    /* Производство
    элемента v */
    b[in] = v;
    in++;
}

/* Потребитель */
while(true)
{
    while (in <= out)
        /* Бездействие */;
    w = b[out];
    out++;
    /* Потребление элемента w*/
}
```

На рис. 5.11 показана структура буфера  $b$ . Производитель может генерировать элементы и сохранять их в буфере со своей индивидуальной скоростью. Всякий раз при сохранении увеличивается индекс  $in$ . Потребитель поступает аналогично, с тем отличием, что он не должен считывать данные из пустого буфера. Следовательно, перед выполнением считывания он должен убедиться, что производитель его обогнал ( $in > out$ ).



Примечание: занятая часть буфера заштрихована

**Рис. 5.11.** Бесконечный буфер задачи “производитель/потребитель”

Попытаемся реализовать нашу систему с использованием бинарных семафоров. На рис. 5.12 приведена первая попытка реализации. Вместо работы с индексами  $in$  и  $out$  мы можем просто отслеживать количество элементов в буфере посредством целочисленной переменной  $n = in - out$ . Для осуществления взаимного исключения используется семафор  $s$ ; семафор  $delay$  применяется для ожидания потребителя при пустом буфере.

Решение представляется достаточно простым и очевидным. Производитель может добавлять данные в буфер в любой момент времени. Перед добавлением он выполняет `semWaitB(s)`, а после добавления — `semSignalB(s)`, чтобы предотвратить обращение к буферу других производителей или потребителя на все время операции добавления данных в буфер. Кроме того, работая в критическом участке, производитель увеличивает значение  $n$ . Если  $n = 1$ , то перед этим добавлением данных в буфер он был пуст, так что производитель выполняет `semSignalB(delay)`, чтобы сообщить об этом потребителю.

```

/* program producerconsumer */
int n;
binary_semaphore s = 1, delay = 0;

void producer()
{
    while (true) {
        produce();
        semWaitB(s);
        append();
        n++;
        if (n==1) semSignalB(delay);
        semSignalB(s);
    }
}

void consumer()
{
    semWaitB(delay);
    while (true) {
        semWaitB(s);
        take();
        n--;
        semSignalB(s);
        consume();
        if (n==0) semWaitB(delay);
    }
}

void main()
{
    n = 0;
    parbegin (producer, consumer);
}

```

**Рис. 5.12.** Неверное решение задачи "производитель/потребитель" с использованием бинарных семафоров

Потребитель начинает с ожидания производства первого элемента, используя вызов `semWaitB(delay)`. Затем потребитель получает данные из буфера и уменьшает значение `n` в своем критическом участке. Если производители опережают потребителя (достаточно распространенная ситуация), то потребитель будет редко блокирован семафором `delay`, поскольку `n` обычно положительно. Следовательно, благополучно работают и производитель, и потребитель.

Тем не менее в предложенной программе имеется изъян. Когда потребитель исчерпывает буфер, он должен сбросить семафор `delay` с помощью инструкции `if (n == 0) semWaitB(delay);`, чтобы дождаться размещения данных в буфере.

производителем. Рассмотрим сценарий, приведенный в табл. 5.4. В строке 14 потребитель не выполняет операцию `semWaitB`. Он действительно исчерпал буфер и установил `n` равным 0 в строке 8, но до проверки значения `n` в строке 14 оно было изменено производителем. В результате `semSignalB` не соответствует предшествующему `semWaitB`. Значение `n`, равное `-1` в строке 20, означает, что потребитель пытается извлечь из буфера несуществующий элемент. Простое перемещение проверки в критический участок потребителя недопустимо, так как может привести к взаимоблокировке (например, после строки 8 в табл. 5.4).

Таблица 5.4. Возможный сценарий работы программы на рис.5.12

	Производитель	Потребитель	s	n	Delay
1			1	0	0
2	<code>semWaitB(s)</code>		0	0	0
3	<code>n++</code>		0	1	0
4	<code>if (n==1) (semSignalB(delay))</code>		0	1	1
5	<code>semSignalB(s)</code>		1	1	1
6		<code>semWaitB(delay)</code>	1	1	0
7		<code>semWaitB(s)</code>	0	1	0
8		<code>n--</code>	0	0	0
9		<code>semSignalB(s)</code>	1	0	0
10	<code>semWaitB(s)</code>		0	0	0
11	<code>n++</code>		0	1	0
12	<code>if (n==1) (semSignalB(delay))</code>		0	1	1
13	<code>semSignalB(s)</code>		1	1	1
14		<code>if (n==0) (semWaitB(delay))</code>	1	1	1
15		<code>semWaitB(s)</code>	0	1	1
16		<code>n--</code>	0	0	1
17		<code>semSignalB(s)</code>	1	0	1
18		<code>if (n==0) (semWaitB(delay))</code>	1	0	0
19		<code>semWaitB(s)</code>	0	0	0
20		<code>n--</code>	0	-1	0
21		<code>semSignalB(s)</code>	1	-1	0

Белые ячейки представляют критический участок, управляемый семафором `s`.

Решение проблемы заключается во введении вспомогательной переменной, значение которой устанавливается в критическом участке и используется вне его, как показано на рис. 5.13. Внимательно рассмотрев приведенный код, вы убедитесь в отсутствии возможных взаимоблокировок.

```

/* program producerconsumer */
int n;
binary_semaphore s = 1, delay = 0;
void producer()
{
    while (true) {
        produce();
        semWaitB(s);
        append();
        n++;
        if (n==1) semSignalB(delay);
        semSignalB(s);
    }
}

void consumer()
{
    int m; /* Локальная переменная */
    semWaitB(delay);
    while (true) {
        semWaitB(s);
        take();
        n--;
        m = n;
        semSignalB(s);
        consume();
        if (m==0) semWaitB(delay);
    }
}

void main()
{
    n = 0;
    parbegin (producer, consumer);
}

```

**Рис. 5.13.** Верное решение задачи “производитель/потребитель” с использованием бинарных семафоров

Несколько более простое решение (приведенное на рис. 5.14) можно получить при использовании обобщенных семафоров (именуемых также семафорами со счетчиками). Переменная *n* в этом случае является семафором; ее значение остается равным количеству элементов в буфере. Предположим теперь, что при переписывании этой программы произошла ошибка, и операции *semSignal(s)* и *semSignal(n)* оказались взаимозамененными. Это может привести к тому, что операция *semSignal(n)* будет выполняться в критическом участке производителя без прерывания потребителя или другого производителя. Повлияет ли это на выполнение программы? Нет, поскольку потребитель в любом случае должен ожидать установки обоих семафоров перед продолжением работы.

```

/* program producerconsumer */
semaphore n = 0, s = 1;
void producer()
{
    while (true) {
        produce();
        semWait(s);
        append();
        semSignal(s);
        semSignal(n);
    }
}
void consumer()
{
    while (true) {
        semWait(n);
        semWait(s);
        take();
        semSignal(s);
        consume();
    }
}
void main()
{
    parbegin (producer, consumer);
}

```

**Рис. 5.14.** Решение задачи “производитель/потребитель” с использованием семафоров

Теперь предположим, что случайно взаимозаменены операции `semWait(n)` и `semWait(s)`. Это приведет к фатальным последствиям. Если пользователь войдет в критический участок, когда буфер пуст (`n.count = 0`), то ни один производитель не сможет добавить данные в буфер и система окажется в состоянии взаимной блокировки. Это хороший пример тонкости работы с семафорами и сложности корректной разработки параллельно работающих процессов.

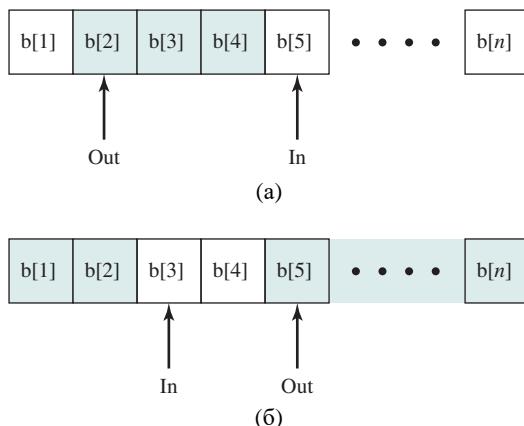
А теперь добавим к нашей задаче новое, достаточно реалистичное ограничение — конечность буфера. Буфер рассматривается нами как циклическое хранилище (рис. 5.15), в ходе работы с которым значения указателей должны выражаться по модулю размера буфера. При этом выполняются следующие условия.

#### Блокировка

#### Деблокирование

Производитель: вставка в полный буфер  
Потребитель: удаление из пустого буфера

Потребитель: вставка элемента в буфер  
Производитель: удаление элемента из буфера



**Рис. 5.15.** Конечный циклический буфер для задачи “производитель/потребитель”

Функции производителя и потребителя при этом могут быть записаны следующим образом (переменные *in* и *out* инициализированы значением 0, а *n* представляет собой размер буфера).

<p><b>Производитель:</b></p> <pre>while(true) {     /* Производство        элемента v*/;     while((in+1)%n == out)         /* Бездействие */;     b[in] = v;     in = (in+1) % n; }</pre>	<p><b>Потребитель:</b></p> <pre>while(true) {     while(in == out)         /* Бездействие */;     w = b[out];     out = (out+1) % n;     /* Потребление        элемента w */;</pre>
--	---

На рис. 5.16 приведено решение с использованием обобщенных семафоров. Для отслеживания пустого места в буфере в программу добавлен семафор *e*. Еще одним поучительным примером использования семафоров является задача о парикмахерской, описанная в приложении А, “Вопросы параллельности”. В нем также имеются дополнительные примеры, связанные с состоянием гонки при использовании семафоров.

## Реализация семафоров

Как упоминалось ранее, главное условие корректности работы семафоров заключается в требовании атомарности операций *semWait* и *semSignal*. Один из очевидных путей выполнения этого условия состоит в реализации семафоров в аппаратном или микропрограммном обеспечении. Если этот путь недоступен, применяются различные программные подходы. Суть проблемы заключается в реализации взаимных исключений: в определенный момент времени работать с семафором посредством операций *semWait* или *semSignal* может только один процесс. Следовательно, подойдет любая из рассматривавшихся программных схем, такая как алгоритмы Деккера или Петерсона (см. раздел 5.1); но это может привести к определенным накладным расходам.

```

/* program boundedbuffer */
const int sizeofbuffer = /* Размер буфера */;
semaphore s = 1, n = 0, e = sizeofbuffer;
void producer()
{
    while (true) {
        produce();
        semWait(e);
        semWait(s);
        append();
        semSignal(s);
        semSignal(n);
    }
}
void consumer()
{
    while (true) {
        semWait(n);
        semWait(s);
        take();
        semSignal(s);
        semSignal(e);
        consume();
    }
}
void main()
{
    parbegin (producer, consumer);
}

```

**Рис. 5.16.** Решение задачи “производитель/потребитель” с ограниченным буфером

Можно также использовать одну из схем поддержки взаимоисключений на аппаратном уровне. Так, на рис. 5.17, а показано, как можно использовать команду сравнения и присваивания значения. В этой реализации, как и на рис. 5.6, семафор представляет собой структуру; однако теперь он включает новый целочисленный компонент `s.flag`. Конечно, при таком способе реализации семафоров неизбежно пережидание занятости, но поскольку операции `semWait` и `semSignal` относительно небольшие, время ожидания минимально.

В однопроцессорной системе можно воспользоваться запретом прерываний на время выполнения операций `semWait` и `semSignal`, как предложено на рис. 5.17, б. Повторимся еще раз: малое время ожидания занятости этих операций означает целесообразность применения предложенного подхода.

```

semWait(s)
{
    while (compare_and_swap(s.flag, 0 , 1)
           == 1)
        /* Ничего не делать */;
    s.count--;
    if (s.count < 0)
    {
        /* Поместить данный
         * процесс в s.queue*/;
        /* Блокировать этот процесс
         * (следует также установить
         * s.flag равным 0) */;
    }
    s.flag = 0;
}

semSignal(s)
{
    while (compare_and_swap(s.flag, 0 , 1)
           == 1)
        /* Ничего не делать */;
    s.count++;
    if (s.count<= 0)
    {
        /* Убрать процесс
         * Р из s.queue */;
        /* Поместить процесс Р в список
         * готовых к выполнению
         * процессов*/;
    }
    s.flag = 0;
}

```

```

semWait(s)
{
    inhibit interrupts;
    s.count--;
    if (s.count < 0)
    {
        /* Поместить данный
         * процесс в s.queue*/;
        /* Блокировать этот процесс
         * и разрешить прерывания */;
    }
    else
        allow interrupts;
}

semSignal(s)
{
    inhibit interrupts;
    s.count++;
    if (s.count<= 0)
    {
        /* Убрать процесс
         * Р из s.queue */;
        /* Поместить процесс Р в список
         * готовых к выполнению
         * процессов*/;
    }
    allow interrupts;
}

```

а) Команда сравнения и присваивания

б) Прерывания

Рис. 5.17. Две возможные реализации семафоров

## 5.5. Мониторы

Семафоры обеспечивают достаточно мощный и гибкий инструмент для осуществления взаимных исключений и координации процессов. Однако, как вы видели на рис. 5.12, создать корректно работающую программу с использованием семафоров не всегда легко. Сложность заключается в том, что операции semWait и semSignal могут быть разбросаны по всей программе, и не всегда можно сразу отследить их воздействие на контролируемые ими семафоры.

Монитор представляет собой конструкцию языка программирования, которая обеспечивает функциональность, эквивалентную функциональности семафоров, но более легкую в управлении. Впервые формальное определение концепции мониторов было дано в [107]. Мониторы реализованы во множестве языков программирования, включая такие, как Concurrent Pascal, Pascal-Plus, Modula-2, Modula-3 и Java. Мониторы также

реализуются как программные библиотеки. Это позволяет использовать мониторы, блокирующие любые объекты. В частности, например, для связанного списка можно заблокировать все связанные списки одной блокировкой либо иметь отдельные блокировки для каждого списка, а возможно — и для каждого элемента списка.

Рассмотрение мониторов мы начнем с версии Хоара (Hoare).

## Мониторы с сигналами

Монитор представляет собой программный модуль, состоящий из инициализирующей последовательности, одной или нескольких процедур и локальных данных. Основные характеристики монитора таковы.

1. Локальные переменные монитора доступны только его процедурам; внешние процедуры доступа к локальным данным монитора не имеют.
2. Процесс входит в монитор путем вызова одной из его процедур.
3. В мониторе в определенный момент времени может выполняться только один процесс; любой другой процесс, вызвавший монитор, будет приостановлен в ожидании доступности монитора.

Первые две характеристики сразу заставляют нас вспомнить о объектах в объектно-ориентированном программировании. Фактически объектно-ориентированные операционные системы или языки программирования могут легко реализовать монитор как объект со специальными характеристиками.

Соблюдение условия выполнения только одного процесса в определенный момент времени позволяет монитору обеспечить взаимоисключения. Данные монитора доступны в этот момент только одному процессу, следовательно, защитить совместно используемые структуры данных можно, просто поместив их в монитор. Если данные в мониторе представляют некий ресурс, то монитор обеспечивает взаимоисключение при обращении к ресурсу.

Для широкого применения в параллельных вычислениях мониторы должны включать инструменты синхронизации. Предположим, например, что процесс вызывает монитор и, находясь в мониторе, должен быть приостановлен до выполнения некоторого условия. При этом нам требуется некий механизм, который не только приостанавливает процесс, но и освобождает монитор, позволяя войти в него другому процессу. Позже, когда условие окажется выполненным, а монитор доступным, приостановленный процесс сможет продолжить свою работу с того места, где он был приостановлен.

Монитор поддерживает синхронизацию при помощи **условных переменных** (*condition variable*), содержащихся в мониторе и доступных только в нем. Работать с этими переменными могут две функции.

- `cwait(c)`. Приостанавливает выполнение вызывающего процесса по условию `c`. Монитор при этом доступен для использования другим процессом.
- `csignal(c)`. Возобновляет выполнение некоторого процесса, приостановленного вызовом `cwait` с тем же условием. Если имеется несколько таких процессов, выбирается один из них; если таких процессов нет, функция не делает ничего.

Обратите внимание на то, что операции `wait/signal` монитора отличаются от соответствующих операций семафора. Если процесс в мониторе передает сигнал, но при этом нет ни одного ожидающего его процесса, то сигнал просто теряется.

На рис. 5.18 показана структура монитора. Хотя процесс может войти в монитор посредством вызова любой из его процедур, мы все же будем рассматривать монитор как имеющий единственную точку входа, которая позволяет обеспечить наличие в мониторе не более одного процесса в любой момент времени. Другие процессы, которые пытаются войти в монитор, присоединяются к очереди процессов, приостановленных в ожидании доступности монитора. После того как процесс вошел в монитор, он может временно приостановиться для ожидания условия  $x$ , выполнив вызов `cwait(x)`; после этого процесс помещается в очередь процессов, ожидающих повторного входа в монитор при выполнении условия, и возобновляет работу в точке программы, следующей за вызовом `cwait(x)`.

Если процесс, выполняющийся в мониторе, обнаруживает изменение переменной условия  $x$ , он выполняет операцию `csignal(x)`, которая сообщает об обнаруженном изменении соответствующей очереди. В качестве примера использования монитора вернемся к задаче “производитель/потребитель” с ограниченным буфером.

На рис. 5.19 показано решение задачи с использованием монитора.

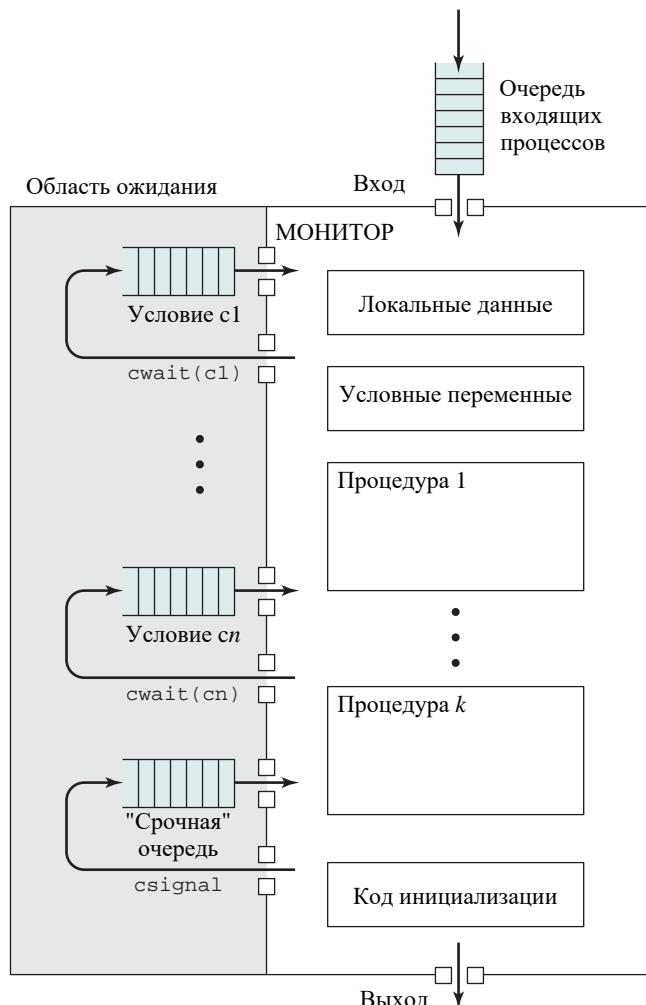


Рис. 5.18. Структура монитора

```

/* program producerconsumer */
monitor boundedbuffer;
char buffer [N];           /* Пространство для N элементов */
int nextin, nextout;       /* Указатели буфера */
int count;                 /* Количество элементов в буфере */
cond notfull, notempty;   /* Условные переменные для синхронизации */
void append (char x)
{
    if (count == N) cwait(notfull); /* Буфер полон; избегаем
                                      переполнения */

    buffer[nextin] = x;
    nextin = (nextin + 1) % N;
    count++;
    /* Еще один элемент в буфере */
    csignal (notempty); /* Возобновление любого ожидающего
                           потребителя */
}
void take (char x)
{
    if (count == 0) cwait(notempty); /* Буфер пуст; избегаем
                                      опустошения */

    x = buffer[nextout];
    nextout = (nextout + 1) % N;
    count--;
    /* На один элемент в буфере меньше */
    csignal (notfull); /* Возобновление любого ожидающего
                           производителя */
}
/* Тело монитора */
nextin = 0; nextout = 0; count = 0; /* Буфер изначально пустой */
}

```

```

void producer()
{
    char x;
    while (true) {
        produce(x);
        append(x);
    }
}
void consumer()
{
    char x;
    while (true) {
        take(x);
        consume(x);
    }
}
void main()
{
    parbegin (producer, consumer);
}

```

**Рис. 5.19.** Решение задачи “производитель/потребитель” с ограниченным буфером с использованием монитора

Модуль монитора `boundedbuffer` управляет буфером, использующимся для хранения и получения символов. Монитор включает две условные переменные (объявленные в конструкции `cond`): `notfull` истинно, если в буфере имеется место как минимум для одного символа, и `notempty`, если в буфере имеется по крайней мере один символ.

Производитель может добавить символы в буфер только из монитора при помощи процедуры `append`; прямого доступа к буферу у него нет. Сначала процедура проверяет условие `notfull`, чтобы выяснить, имеется ли в буфере пустое место. Если его нет, процесс приостанавливается, и в монитор может войти другой процесс (производитель или потребитель). Позже, когда буфер оказывается заполненным не до конца, приостановленный процесс извлекается из очереди и возобновляет свою работу. После того как процесс помещает символ в буфер, он сигнализирует о выполнении условия `notempty`, что разблокирует процесс потребителя (если последний был приостановлен).

Этот пример иллюстрирует разделение ответственности при работе с монитором и при использовании семафоров. Монитор автоматически обеспечивает взаимоисключение: одновременное обращение производителя и потребителя к буферу невозможно. Однако программист должен корректно разместить внутри монитора примитивы `cwait` и `csignal`, для того чтобы предотвратить размещение элемента в заполненном буфере или выборку из пустого буфера. В случае использования семафоров ответственность как за синхронизацию, так и за взаимоисключения полностью лежит на программисте.

Обратите внимание, что на рис. 5.19 процесс покидает монитор немедленно после выполнения функции `csignal`. Если вызов `csignal` осуществляется не в конце процедуры, то, по предложению Хоара, вызвавший эту функцию процесс приостанавливается, для того чтобы освободить монитор для другого процесса, помещается в очередь и остается там до тех пор, пока монитор вновь не освободится. Процесс можно поместить во входную очередь монитора вместе с другими процессами, еще не вошедшими в монитор. Однако поскольку рассматриваемый процесс уже частично выполнил свою задачу в мониторе, имеет смысл дать ему приоритет перед только входящими в монитор, для чего использовать дополнительную, “срочную” очередь (см. рис. 5.18). Заметим, что один из использующих мониторы языков, а именно `Concurrent Pascal`, требует, чтобы вызов `csignal` был последней операцией процедуры монитора.

Если выполнения условия `x` не ожидает ни один процесс, вызов `csignal(x)` не выполняет никаких действий.

Во время работы как с семафорами, так и с мониторами очень легко допустить ошибку в функции синхронизации. Например, если опустить любой из вызовов `csignal` в мониторе, то процесс, попавший в соответствующую очередь, останется там навсегда. Преимущество мониторов по сравнению с семафорами в том, что все синхронизирующие функции заключены в мониторе. Таким образом, проверить корректность синхронизации и отловить возможные ошибки при использовании мониторов оказывается проще, чем при использовании семафоров. Кроме того, при правильно разработанном мониторе доступ к защищенным ресурсам корректен независимо от запрашивающего процесса; при использовании же семафоров доступ к ресурсу корректен, только если правильно разработаны все процессы, обращающиеся к ресурсу.

## Мониторы с оповещением и широковещанием

Определение мониторов, данное Хоаром [107], требует, чтобы в случае, если в очереди ожидания выполнения условия есть хотя бы один процесс, при выполнении каким-

либо иным процессом операции `csignal` для этого условия был немедленно запущен процесс, находящийся в указанной очереди. Таким образом, выполнивший операцию `csignal` процесс должен либо немедленно выйти из монитора, либо быть приостановленным.

У такого подхода имеется два недостатка.

- Если выполнивший операцию `csignal` процесс не завершил свое пребывание в мониторе, то требуются два дополнительных переключения процессов: одно — для приостановки данного процесса и второе — для возобновления его работы, когда монитор станет доступным.
- Планировщик процессов, связанный с сигналом, должен быть идеально надежным. При выполнении `csignal` процесс из соответствующей очереди должен быть немедленно активизирован, причем планировщик должен гарантировать, что до активизации никакой другой процесс не войдет в монитор (в противном случае условие, в соответствии с которым активизируется процесс, может успеть измениться). Так, например, на рис. 5.19, когда выполняется `csignal(notempty)`, процесс из очереди `nonempty` должен быть активизирован до того, как новый потребитель войдет в монитор. Вот и другой пример: сбой процесса производителя может произойти непосредственно после того, как он добавит символ к пустому буферу, так что операция `csignal` не будет выполнена. В результате процессы в очереди `notempty` окажутся навечно заблокированными.

Лэмпсон (Lampson) и Ределл (Redell) разработали другое определение монитора для языка Mesa [143]. Их подход позволяет преодолевать описанные проблемы, а кроме того, предоставляет ряд полезных расширений концепции мониторов. Структура монитора Mesa использована и в языке программирования Modula-3 [178]. В Mesa примитив `csignal` заменен примитивом `cnotify`, который интерпретируется следующим образом. Когда процесс, выполняющийся в мониторе, вызывает `cnotify(x)`, об этом оповещается очередь условия `x`, но выполнение вызвавшего `cnotify` процесса продолжается. Результат оповещения состоит в том, что процесс в начале очереди условия возобновит свою работу в ближайшем будущем, когда монитор окажется свободным. Однако поскольку нет гарантии, что некий другой процесс не войдет в монитор до упомянутого ожидающего процесса, при возобновлении работы наш процесс должен еще раз проверить, выполнено ли условие. В случае использования такого подхода процедуры монитора `boundedbuffer` будут иметь следующий вид, как показано на рис. 5.20.

Инструкции `if` заменены циклами `while`; таким образом, будет выполняться как минимум одно лишнее вычисление условной переменной. Однако в этом случае отсутствуют ненужные переключения процессов и нет ограничений на момент запуска ожидающего процесса после вызова `cnotify`.

Одной из полезных особенностей такого рода мониторов может быть связанное с каждым примитивом условия `cnotify` предельное время ожидания. Процесс, который ожидал уведомления в течение предельного времени, помещается в список активных независимо от того, было ли уведомление о выполнении условия. При активизации процесс проверяет, выполнено ли условие, и, если выполнено, продолжает свою работу. Такая возможность предотвращает бесконечное голодание процесса в случае, когда другие процессы сбоят перед уведомлением о выполнении условия.

```

void append (char x)
{
    while (count == N)
        cwait(notfull); /* Буфер полон; избегаем переполнения */
    buffer[nextin] = x;
    nextin = (nextin + 1) % N;
    count++;           /* Еще один элемент в буфере */
    cnotify(notempty); /* notify any waiting consumer */
}

void take (char x)
{
    while (count == 0)
        cwait(notempty); /* Буфер пуст; избегаем опустошения */
    x = buffer[nextout];
    nextout = (nextout + 1) % N;
    count--;           /* На один элемент в буфере меньше */
    cnotify(notfull); /* Уведомление ожидающего производителя */
}

```

**Рис. 5.20.** Код монитора Mesa для буфера ограниченного размера

При использовании правила, согласно которому происходит уведомление процесса, а не его насильственная активизация, в систему команд можно включить примитив `cbroadcast`, который вызывает активизацию всех ожидающих процессов. Это может быть удобно в ситуациях, когда процесс не осведомлен о количестве ожидающих процессов. Предположим, например, что в программе “производитель/потребитель” функции `append` и `take` могут работать с символьными блоками переменной длины. В этом случае, когда производитель добавляет в буфер блок символов, он не обязан знать, сколько символов готов потребить каждый из ожидающих процессов. Он просто выполняет инструкцию `cbroadcast`, и все ожидающие процессы получают уведомление о том, что они могут попытаться получить свою долю символов из буфера.

Кроме того, широковещательное сообщение может использоваться в том случае, когда процесс не в состоянии точно определить, какой именно процесс из ожидающих должен быть активирован. Хорошим примером такой ситуации может служить диспетчер памяти. Допустим, у нас имеется  $j$  байт свободной памяти и некоторый процесс освобождает дополнительно  $k$  байт. Диспетчеру не известно, какой именно из ожидающих процессов сможет работать с  $k+j$  байт свободной памяти; следовательно, он должен использовать вызов `cbroadcast`, и все ожидающие процессы сами проверят, достаточно ли им освободившейся памяти.

Преимуществом монитора Лэмпсона–Ределла по сравнению с монитором Хоара является его меньшая подверженность ошибкам. При подходе Лэмпсона–Ределла, поскольку каждая процедура после получения сигнала проверяет переменную монитора с использованием цикла `while`, процесс может послать неверное уведомление или широковещательное сообщение, и это не приведет к ошибке в программе, получившей сигнал (попросту убедившись, что ее зря активизировали, программа вновь перейдет в состояние ожидания).

Другим преимуществом монитора Лэмпсона–Ределла является то, что он способствует использованию модульного подхода при создании программ. Рассмотрим, например, реализацию выделения памяти для буфера. Имеются два уровня условий, которые должны удовлетворяться для сотрудничающих последовательных процессов.

1. Согласованные структуры данных. Таким образом, монитор обеспечивает взаимное исключение и завершает операцию ввода или вывода, прежде чем разрешить другую операцию над буфером.
2. Условия первого уровня плюс достаточное количество памяти для этого процесса, чтобы завершить запрос выделения памяти.

В мониторе Хоара каждый сигнал не только передает условие уровня 1, но и несет неявное сообщение “я освободил достаточное количество байтов, чтобы мог работать вызов вашего конкретного аллокатора”. Таким образом, сигнал неявно выполняет условие уровня 2. Если позднее программист изменит определение условие уровня 2, будет необходимо перепрограммировать все процессы, работающие с сигналами. Если программист изменяет предположения, на которых основывается любой конкретный ожидающий процесс (т.е. ожидающий немного отличающийся инвариант уровня 2), может оказаться необходимым перепрограммирование всех процессов. Это приводит к снижению уровня модульности и может привести к ошибкам синхронизации (например, ошибочной активизации процесса) при внесении изменений в код. Программист должен помнить о необходимости внесения изменений во все процедуры в мониторе каждый раз, когда выполняется небольшое изменение условия уровня 2. При работе с монитором Лэмпсона–Ределла широковещание обеспечивает условие уровня 1 и несет подсказку о том, что может выполняться условие уровня 2; каждый процесс должен самостоятельно проверять условие уровня 2. Если изменения условия уровня 2 вносятся в ожидающем или сигнализирующем процессе, ошибочное пробуждение процесса невозможно, поскольку каждая процедура сама проверяет состояние уровня 2. Таким образом, условие уровня 2 может быть скрыто внутри каждой процедуры. В случае монитора Хоара условие уровня 2 должно быть перенесено из ожидающей процедуры в код каждого сигнализирующего процесса, что нарушает принципы абстракции данных и межпроцедурной модульности.

## 5.6. ПЕРЕДАЧА СООБЩЕНИЙ

При взаимодействии процессов между собой должны удовлетворяться два фундаментальных требования: синхронизации и коммуникации. Процессы должны быть синхронизированы, с тем чтобы обеспечить выполнение взаимных исключений; сотрудничающие процессы должны иметь возможность обмениваться информацией. Одним из подходов к обеспечению обеих указанных функций является передача сообщений. Важным достоинством передачи сообщений является ее пригодность для реализации как в однопроцессорных, так и в многопроцессорных системах с общей памятью, так и в распределенных системах.

Системы передачи сообщений могут быть различных типов; в этом разделе мы обратимся только к наиболее общим возможностям и свойствам таких систем. Обычно функции передачи сообщений представлены в виде пары примитивов

```
send(получатель, сообщение)
receive(отправитель, сообщение)
```

Это — минимальный набор операций, необходимый для работы процессов с системами передачи сообщений. Процесс посылает информацию в виде *сообщения* другому процессу, определенному как *получатель*, вызовом *send*. Получает информацию процесс при помощи выполнения примитива *receive*, которому указывает *отправителя* сообщения.

При разработке систем передачи сообщений следует решить ряд вопросов, которые перечислены в табл. 5.5. В оставшейся части данного раздела мы вкратце коснемся каждого из этих вопросов.

**Таблица 5.5. ХАРАКТЕРИСТИКИ СИСТЕМ ПЕРЕДАЧИ СООБЩЕНИЙ**

Синхронизация	Формат
Отправление Блокирующее Неблокирующее	Содержимое Длина Фиксированная
Получение Блокирующее Неблокирующее Проверка доставки	Переменная
Адресация	Принцип работы очереди
Прямая Отправление Получение Явное Неявное Косвенная Статическая Динамическая Владение	FIFO Приоритетная

## Синхронизация

Передача сообщения между двумя процессами предполагает наличие определенной степени их синхронизации: получатель не в состоянии получить сообщение до тех пор, пока оно не послано другим процессом. Кроме того, мы должны определить, что происходит после того, как процесс вызывает примитивы *send* и *receive*.

Рассмотрим сначала примитив *send*. При его выполнении имеются две возможности: посылающий сообщение процесс либо блокируется, либо продолжает работу. Аналогично две возможности есть и у процесса, выполняющего примитив *receive*.

1. Если сообщение было отправлено ранее, то процесс получает его и продолжает работу.

## 2. Если сообщения, ожидающего получение, нет, то:

- а) либо процесс блокируется до тех пор, пока сообщение не будет получено;
- б) либо процесс продолжает выполнение, отказываясь от дальнейших попыток получить его.

Таким образом, и отправитель, и получатель могут быть блокируемыми или неблокируемыми. Обычно встречаются три комбинации (хотя в реальных системах реализуется, как правило, только одна или две).

1. **Блокирующее отправление, блокирующее получение.** И отправитель, и получатель блокируются до тех пор, пока сообщение не будет доставлено по назначению. Такую ситуацию иногда называют *рандеву* (*rendezvous*). Эта комбинация обеспечивает тесную синхронизацию процессов.
2. **Неблокирующее отправление, блокирующее получение.** Хотя отправитель и может продолжать работу, получатель блокируется до получения сообщения. Эта комбинация, пожалуй, встречается чаще всего. Она позволяет процессу посыпать одно или несколько сообщений различным получателям с максимальной скоростью. Процесс, который должен получить сообщение перед тем, как приступить к выполнению каких-то действий, будет заблокирован, пока не получит необходимое сообщение. Примером такого рода системы может быть серверный процесс, существующий для предоставления сервисов или ресурсов другим процессам.
3. **Неблокирующее отправление, неблокирующее получение.** Не блокируется ни один из процессов.

Неблокирующий примитив `send` является более естественным выбором для множества задач с использованием параллельных вычислений. Например, если он используется для запроса на выполнение операции вывода (скажем, на принтер), то данный запрос может быть отправлен в виде сообщения, после чего работа процесса продолжится. Потенциальная опасность неблокирующего отправления сообщений состоит в том, что возможна ситуация, когда некоторая ошибка приведет к непрерывной генерации сообщений. Поскольку блокировка не предусмотрена, эти сообщения могут привести к потреблению значительной части системных ресурсов, в том числе процессорного времени и памяти, нанеся вред другим процессам и самой операционной системе. Кроме того, при таком подходе на программиста возлагается задача отслеживания успешной доставки сообщения адресату (процесс-получатель должен, в свою очередь, послать ответ с подтверждением получения сообщения).

В случае использования примитива `received` для большинства задач естественной представляется блокирующая технология. Вообще говоря, процесс, запросивший информацию, нуждается в ней для продолжения работы. Конечно, если сообщение теряется (что не такая уж редкость в распределенных системах) или происходит сбой процесса перед отправкой сообщения, то процесс-получатель может оказаться навсегда заблокированным. Решить эту проблему можно с помощью неблокирующего примитива `receive`; однако у этого варианта имеется свое слабое место: если сообщение послано после того, как процесс выполнил соответствующую операцию `receive`, то оно оказывается потерянным. Еще один возможный подход к решению проблемы заключается в том, чтобы позволить процессу перед тем, как выполнять `receive`, проверить, не имеется ли ожидающего получения сообщения, а также позволить процессу указывать несколько отправителей в примитиве `receive`. Последнее решение особенно удобно, если процесс ожидает сообщения из нескольких источников и может продолжать работу при получении любого из них.

## Адресация

Ясно, что совершенно необходимо иметь возможность указать в примитиве `send` процесс, являющийся получателем данного сообщения. Аналогично большинство реализаций позволяют получателю указать, сообщение от какого отправителя должно быть принято.

Различные схемы определения процессов в примитивах `send` и `receipt` разделяются на две категории: прямую (`direct`) и косвенную (`indirect`) адресацию. При **прямой адресации** примитив `send` включает идентификатор процесса-получателя. Когда применяется примитив `receive`, можно пойти двумя путями. Первый путь состоит в требовании явного указания процесса-отправителя, т.е. процесс должен знать заранее, от какого именно процесса он ожидает сообщение. Такой путь достаточно эффективен, если параллельные процессы сотрудничают. Однако во многих случаях невозможно предсказать, какой процесс будет отправителем ожидаемого сообщения (в качестве примера можно привести процесс сервера печати, который принимает сообщения — запросы на печать от любого другого процесса). Для таких приложений более эффективным будет подход с использованием **неявной адресации**. В этом случае параметр `отправитель` получает значение, возвращаемое после выполнения операции получения сообщения.

Еще одним распространенным подходом является **косвенная адресация**. Она предполагает, что сообщения передаются не прямо от отправителя получателю, а в совместно используемую структуру данных, состоящую из очередей для временного хранения сообщений (такие очереди обычно именуют *почтовыми ящиками* (*mailbox*)). Таким образом, для связи между двумя процессами один из них посыпает сообщение в соответствующий почтовый ящик, из которого его заберет второй процесс.

Эффективность косвенной адресации, в первую очередь, заключается в гибкости использования сообщений. При такой схеме работы с сообщениями отношения между отправителем и получателем могут быть любыми — “один к одному”, “один ко многим”, “многие к одному” или “многие ко многим”. Отношение **один к одному** обеспечивает закрытую связь, установленную между двумя процессами, изолируя их взаимодействие от постороннего вмешательства. Отношение **многие к одному** полезно при взаимодействии “клиент/сервер” — один процесс при этом представляет собой сервер, обслуживающий множество клиентов. В таком случае о почтовом ящике часто говорят как о *порте* (рис. 5.21). Отношение **один ко многим** обеспечивает рассылку от одного процесса множеству получателей, позволяя осуществить широковещательное сообщение множеству процессов. Отношение **многие ко многим** позволяют множеству процессов сервера параллельно обслуживать множество клиентов.

Связь процессов с почтовыми ящиками может быть как статической, так и динамической. Порты чаще всего статически связаны с определенными процессами, т.е. порт создается и назначается процессу навсегда. То же наблюдается и в случае использования отношения “один к одному” — закрытые каналы связи, как правило, также определяются статически, раз и навсегда. При наличии множества отправителей их связи с почтовым ящиком могут осуществляться динамически, с использованием для этой цели примитивов типа `connect` и `disconnect`.

С этим вопросом также тесно связан вопрос владения почтовым ящиком. В случае использования порта он, как правило, создается процессом-получателем и принадлежит ему. Таким образом, при уничтожении процесса порт также уничтожается. При исполь-

зование обобщенного почтового ящика операционная система может предложить специальный сервис по созданию почтовых ящиков. Такие ящики могут рассматриваться и как принадлежащие создавшему их процессу (и соответственно, уничтожаться при завершении работы процесса), и как принадлежащие операционной системе (в этом случае для уничтожения почтового ящика требуется явная команда).

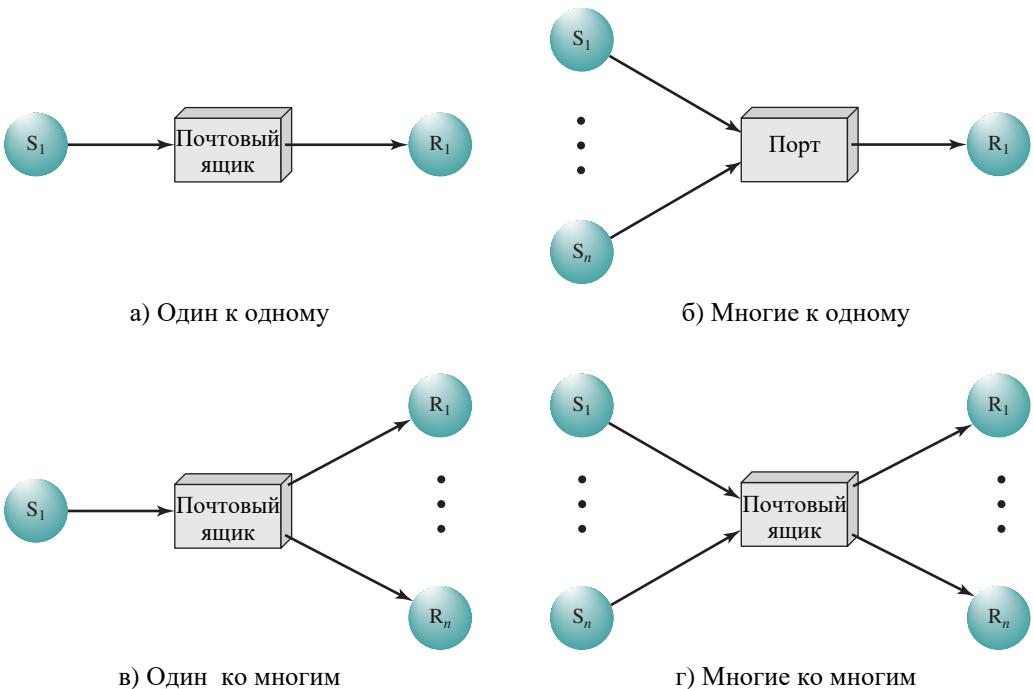
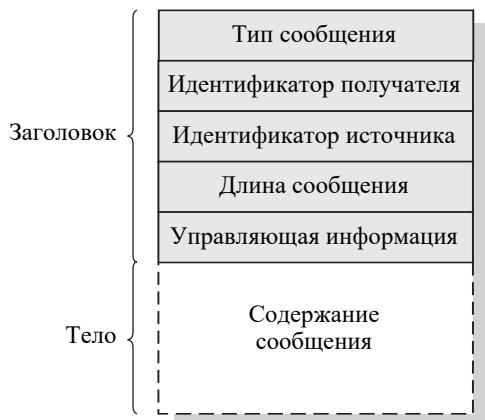


Рис. 5.21. Косвенная связь между процессами

## Формат сообщения

Формат сообщения зависит от преследуемых целей и от того, где работает система передачи сообщений — на одном компьютере или в распределенной системе. В некоторых операционных системах разработчики предпочитают короткие сообщения фиксированной длины, что позволяет минимизировать обработку и уменьшить расходы памяти на их хранение. При передаче больших объемов данных они могут размещаться в файле, а само сообщение — просто содержать ссылку на этот файл. Более гибкий подход позволяет использовать сообщения переменной длины.

На рис. 5.22 показан формат типичного сообщения операционной системы, которая поддерживает сообщения переменной длины. Сообщение разделено на две части: заголовок, содержащий информацию о сообщении, и тело с собственно содержанием сообщения. Заголовок может включать идентификаторы отправителя и получателя сообщения, поля длины и типа сообщения. В заголовке, кроме того, может находиться дополнительная управляющая информация, например указатель, позволяющий объединить создаваемые сообщения в связанный список; номер, позволяющий упорядочить передаваемые сообщения, или поле приоритета.



**Рис. 5.22.** Обобщенный формат сообщения

## Принцип работы очереди

Простейший принцип работы очереди — “первым вошел — первым вышел”, но он может оказаться неадекватным, если одни сообщения будут более срочными, чем другие. В этом случае очередь должна учитывать приоритет сообщений, основываясь либо на типе сообщения, либо на непосредственном указании приоритета отправителем. Можно также позволить получателю просматривать всю очередь сообщений и выбирать, какое письмо должно быть получено следующим.

## Взаимные исключения

На рис. 5.23 показан один из способов реализации взаимных исключений с использованием системы передачи сообщений (сравните с рис. 5.4, 5.5 и 5.9). В данной программе предполагается использование блокирующего `receive` и неблокирующего `send`. Множество параллельно выполняющихся процессов совместно используют почтовый ящик `box` как для отправки сообщений, так и для их получения. Почтовый ящик после инициализации содержит единственное сообщение с пустым содержимым. Процесс, намеревающийся войти в критический участок, сначала пытается получить сообщение. Если почтовый ящик пуст, процесс блокируется. Как только процесс получает сообщение, он тут же входит в критический участок, выполняет его код, а затем отсылает сообщение обратно в почтовый ящик. Таким образом, сообщение работает в качестве переходящего флага, передающегося от процесса к процессу.

В рассмотренном решении предполагается, что если операция `receive` выполняется параллельно более чем одним процессом, то:

- если имеется сообщение, оно передается только одному из процессов, а остальные процессы блокируются;
- если очередь сообщений пуста, блокируются все процессы; когда в очереди появляется сообщение, его получает только один из заблокированных процессов.

Это предположение выполняется практически для всех средств передачи сообщений.

В качестве другого примера использования сообщений на рис. 5.24 приведено решение задачи “производитель/потребитель” с ограниченным буфером.

```

/* program mutual exclusion */
const int n = /* Количество процессов */
void P(int i)
{
    message msg;
    while (true) {
        receive (box, msg);
        /* Критический раздел */;
        send (box, msg);
        /* Остальная часть кода */;
    }
}
void main()
{
    create mailbox (box);
    send (box, null);
    parbegin (P(1), P(2), . . . , P(n));
}

```

**Рис. 5.23.** Реализация взаимных исключений с использованием сообщений

```

const int capacity = /* Емкость буферов */ ;
null = /* Пустое сообщение */ ;
int i;
void producer()
{
    message pmsg;
    while (true) {
        receive (mayproduce,pmsg);
        pmsg = produce();
        send (mayconsume,pmsg);
    }
}
void consumer()
{
    message cmsg;
    while (true) {
        receive (mayconsume,cmsg);
        consume (cmsg);
        send (mayproduce,null);
    }
}
void main()
{
    create_mailbox (mayproduce);
    create_mailbox (mayconsume);
    for (int i = 1;i<= capacity;i++) send (mayproduce,null);
    parbegin (producer,consumer);
}

```

**Рис. 5.24.** Решение задачи “производитель/потребитель” с ограниченным буфером с использованием сообщений

Эту задачу можно решить способом, аналогичным приведенному на рис. 5.16, если воспользоваться реализацией взаимоисключений на базе сообщений. Однако в программе на рис. 5.24 используется другая возможность сообщений, а именно — передача данных в дополнение к сигналам. В программе используются два почтовых ящика. Когда производитель генерирует данные, он посыпает их в качестве сообщения в почтовый ящик `tauyconsume`. Пока в этом почтовом ящике имеется хотя бы одно письмо, потребитель может получать данные, — следовательно, почтовый ящик `tauyconsume` служит буфером, данные в котором организованы в виде очереди сообщений. “Емкость” этого буфера определяется глобальной переменной `caracity`. Почтовый ящик `taурproduce` изначально заполнен пустыми сообщениями в количестве, равном емкости буфера. Количество сообщений в этом почтовом ящике уменьшается при каждом поступлении новых данных и увеличивается при их использовании.

Такой подход достаточно гибкий — он может работать с любым количеством производителей и потребителей; главное, чтобы они имели доступ к обоим почтовым ящикам. Более того, система производителей и потребителей может быть распределенной, когда все производители и почтовый ящик `taурproduce` находятся на одной машине, а потребители и почтовый ящик `tauyconsume` — на другой.

## 5.7. ЗАДАЧА ЧИТАТЕЛЕЙ/ПИСАТЕЛЕЙ

При разработке механизмов синхронизации и параллельных вычислений зачастую полезно связать имеющуюся у вас задачу с уже известными и получить возможность проверить применимость вашего решения к известной задаче. В литературе довольно часто встречается рассмотрение таких “эталонных” задач, представляющих собой примеры часто возникающих перед разработчиком проблем. С одной из них — задачей производителя/потребителя — мы уже встречались; в этом разделе мы рассмотрим еще одну классическую задачу — читателей/писателей.

Определить ее можно следующим образом. Имеются данные, совместно используемые рядом процессов. Данные могут находиться в файле, в блоке основной памяти или даже в регистрах процессора. Имеется несколько процессов, которые только читают эти данные (читатели), и несколько других, которые только записывают данные (писатели). При этом должны удовлетворяться следующие условия.

1. Любое число читателей могут одновременно читать файл.
2. Записывать информацию в файл в определенный момент времени может только один писатель.
3. Когда писатель записывает данные в файл, ни один читатель не может его читать.

Таким образом, читатели представляют собой процессы, которые не обязаны быть взаимоисключающими, в то время как писатели являются процессами, которые должны исключить выполнение любых других процессов, как читателей, так и писателей.

Перед тем как приступить к работе, рассмотрим отличия этой задачи от двух других: обобщенной задачи взаимоисключений и задачи производителя/потребителя. В задаче читателей/писателей читатели не записывают данные, а писатели их не читают. Более общей является ситуация, когда каждый процесс может как читать, так и писать дан-

ные (и которая включает рассматриваемую задачу как частный случай). В таком случае мы можем объявить любую часть процесса, которая обращается к данным, критическим участком и использовать простейшее решение на основе взаимоисключений. Причина, по которой мы рассматриваем частный случай более общей задачи, заключается в том, что общее решение значительно замедляет работу, в то время как для частного случая имеется гораздо более эффективное решение. Представим себе библиотечный каталог, в котором читатели могут искать нужную им литературу, а один или несколько работников библиотеки могут этот каталог обновлять. При общем решении читатели будут вынуждены входить в каталог по одному, что, конечно, приведет к неоправданным задержкам и очередям. Кроме того, работники библиотеки при внесении изменений не должны мешать друг другу, а также не должны допускать читателей к данным в момент их изменения, чтобы предотвратить получение читателем несогласованной информации.

Можно ли рассматривать задачу производителя/потребителя как частный случай задачи читателей/писателей с единственным писателем (производитель) и единственным читателем (потребитель)? Оказывается, нет. Производитель — не просто писатель. Он должен считывать значение указателя очереди, чтобы определять, куда следует вносить очередную порцию информации, и выяснять, не заполнен ли буфер. Аналогично и потребитель является не просто читателем, так как он изменяет значение указателя очереди, указывая, что элемент удален из буфера.

Теперь мы рассмотрим два решения поставленной задачи.

## Приоритетное чтение

На рис. 5.25 приведено решение задачи с использованием семафоров для варианта, в котором имеются один читатель и один писатель (решение не требует изменений, если имеется много читателей и писателей). Процесс писателя очень прост. Для обеспечения взаимного исключения используется семафор `wsem`. Когда один писатель записывает данные, ни другие писатели, ни читатели не могут получить к ним доступ. Процесс читателя также использует семафор `wsem` для обеспечения взаимоисключений. Однако чтобы обеспечить возможность одновременной работы многих читателей, состояние семафора проверяет только читатель, входящий в критический участок, в котором нет других читателей. Если в критическом разделе уже находится хоть один читатель, то другой читатель приступает к работе, не ожидая семафора `wsem`. Для отслеживания количества читателей в критическом разделе используется глобальная переменная `readcount`, а для гарантии корректного ее обновления — семафор `x`.

## Приоритетная запись

В предыдущем решении приоритетной операцией являлось чтение данных. Если один читатель получил доступ к данным, то возможна ситуация, когда писатель в течение долгого времени не получит возможности внести изменения — пока хотя бы один из читателей будет находиться в критическом разделе (голодание писателя).

На рис. 5.26 показано решение, обеспечивающее выполнение следующего условия: ни один читатель не сможет обратиться к данным, если хотя бы один писатель объявил о своем намерении произвести запись.

```

/* program readersandwriters */
int readcount;
semaphore x = 1,wsem = 1;
void reader()
{
    while (true)
    {
        semWait (x);
        readcount++;
        if(readcount == 1)
            semWait (wsem);
        semSignal (x);
        READUNIT();
        semWait (x);
        readcount--;
        if(readcount == 0)
            semSignal (wsem);
        semSignal (x);
    }
}

void writer()
{
    while (true){
        semWait (wsem);
        WRITEUNIT();
        semSignal (wsem);
    }
}

void main()
{
    readcount = 0;
    parbegin (reader,writer);
}

```

**Рис. 5.25.** Решение задачи читателей/писателей с использованием семафоров (приоритетное чтение)

Для этого к имеющимся в программе семафорам и переменным процесса писателя добавлены следующие:

- семафор rsem, запрещающий вход читателей, если хотя бы один писатель объявил о намерении произвести запись;
- переменная writecount, обеспечивающая корректность установки значения rsem;
- семафор y, управляющий обновлением переменной writecount.

```
/* program readersandwriters */
int readcount,writecount;
semaphore x = 1, y = 1, z = 1, wsem = 1, rsem = 1;

void reader()
{
    while (true)
    {
        semWait(z);
        semWait(rsem);
        semWait(x);
        readcount++;
        if (readcount == 1)
            semWait(wsem);
        semSignal(x);
        semSignal(rsem);
        semSignal(z);
        READUNIT();
        semWait(x);
        readcount--;
        if (readcount == 0) semSignal(wsem);
        semSignal(x);
    }
}

void writer ()
{
    while (true){
        semWait(y);
        writecount++;
        if (writecount == 1)
            semWait(rsem);
        semSignal(y);
        semWait(wsem);
        WRITEUNIT();
        semSignal(wsem);
        semWait(y);
        writecount--;
        if (writecount == 0) semSignal(rsem);
        semSignal(y);
    }
}

void main()
{
    readcount = writecount = 0;
    parbegin(reader, writer);
}
```

**Рис. 5.26.** Решение задачи читателей/писателей с использованием семафоров (приоритетная запись)

В процессе читателя требуется один дополнительный семафор. На основании семафора `rsem` нельзя создавать длинную очередь, иначе сквозь нее не смогут “прорваться” писатели. Для осуществления ограничения, когда семафор `rsem` приостанавливает только один процесс чтения, служит семафор `z`, блокирующий остальных читателей. В табл. 5.6 приведены возможные ситуации в данной системе.

**Таблица 5.6. Состояния очередей процессов в программе из рис. 5.26**

---

<b>В системе имеются только читатели</b>	<ul style="list-style-type: none"> <li>• Устанавливается <code>wsem</code></li> <li>• Очередей нет</li> </ul>
<b>В системе имеются только писатели</b>	<ul style="list-style-type: none"> <li>• Устанавливаются <code>wsem</code> и <code>rsem</code></li> <li>• Очередь писателей на <code>wsem</code></li> </ul>
<b>В системе имеются как читатели, так и писатели; первым выполняется чтение</b>	<ul style="list-style-type: none"> <li>• <code>wsem</code> устанавливается читателем</li> <li>• <code>rsem</code> устанавливается писателем</li> <li>• Все писатели становятся в очередь на <code>wsem</code></li> <li>• Один читатель становится в очередь на <code>rsem</code></li> <li>• Остальные читатели становятся в очередь на <code>z</code></li> </ul>
<b>В системе имеются как читатели, так и писатели; первой выполняется запись</b>	<ul style="list-style-type: none"> <li>• <code>wsem</code> устанавливается писателем</li> <li>• <code>rsem</code> устанавливается писателем</li> <li>• Все писатели становятся в очередь на <code>wsem</code></li> <li>• Один читатель становится в очередь на <code>rsem</code></li> <li>• Остальные читатели становятся в очередь на <code>z</code></li> </ul>

---

На рис. 5.27 приведено еще одно решение задачи с приоритетной записью, реализованное с использованием системы передачи сообщений. В этом случае имеется управляющий процесс с правом доступа к совместно используемым данным. Прочие процессы запрашивают право доступа к данным, посыпая сообщение управляющему процессу. Запрашивающий процесс получает право доступа посредством ответного сообщения “OK”, а после завершения работы сообщает об этом специальным сообщением `finished`. У управляющего процесса имеются три почтовых ящика, по одному для каждого типа получаемых им сообщений.

Первыми управляющий процесс обслужит запросы на запись, давая, таким образом, приоритет операции записи; кроме того, управляющий процесс обеспечивает выполнение взаимоисключений, для чего используется переменная `count`, инициализируемая неким числом, которое заведомо больше максимального количества читателей. В нашем примере использовано значение 100. Действия управляющего процесса можно описать следующим образом.

- Если `count > 0`, значит, ожидающих писателей нет; активные читатели могут как присутствовать, так и отсутствовать. Управляющий процесс обслуживает сначала все сообщения типа `finished`, а затем запросы от писателей и читателей.
- Если `count = 0`, это означает, что у нас имеется только запрос на запись. Управляющий процесс дает писателю “добро” на выполнение своих действий и ожидает от него сообщения о завершении работы.

- Если  $count < 0$ , значит, писатель сделал запрос и ожидает завершения работы всех активных читателей, так что управляющий процесс при этом принимает только сообщения о завершении работы читателей.

```

void reader(int i)
{
    message rmsg;
    while (true)
    {
        rmsg = i;
        send(readrequest, rmsg);
        receive(mbox[i], rmsg);
        READUNIT();
        rmsg = i;
        send(finished, rmsg);
    }
}

void writer(int j)
{
    message msg;
    while (true)
    {
        rmsg = j;
        send(writerequest, rmsg);
        receive(mbox[j], rmsg);
        WRITEUNIT();
        rmsg = j;
        send(finished, rmsg);
    }
}

```

```

void controller()
{
    while (true)
    {
        if (count > 0) {
            if (!empty(finished))
            {
                receive(finished, msg);
                count++;
            }
        else if (!empty(writerequest))
        {
            receive(writerequest, msg);
            writer_id = msg.id;
            count = count - 100;
        }
        else if (!empty(readrequest))
        {
            receive(readrequest, msg);
            count--;
            send (msg.id, "OK");
        }
    }
    if (count == 0)
    {
        send (writer_id, "OK");
        receive (finished, msg);
        count = 100;
    }
    while (count < 0)
    {
        receive (finished, msg);
        count++;
    }
}

```

**Рис. 5.27.** Решение задачи читателей/писателей с использованием системы передачи сообщений

## 5.8. РЕЗЮМЕ

Центральными вопросами современных операционных систем являются многозадачность, многопроцессорность и распределенные вычисления. Фундаментальным понятием этих тем (как и технологий проектирования операционных систем в целом) являются параллельные вычисления. При параллельном выполнении нескольких задач (не важно, при реальном ли параллельном выполнении на разных процессорах или виртуальном в однопроцессорной системе) перед разработчиком возникают вопросы разрешения конфликтов и сотрудничества процессов.

Параллельно выполняющиеся процессы могут взаимодействовать друг с другом различными путями. Процессы, не осведомленные о наличии других процессов, могут, тем не менее, участвовать в конкурентной борьбе за использование ресурсов, например за процессорное время или устройства ввода-вывода. Процессы могут быть косвенно осведомлены о наличии других процессов при совместном использовании объектов, таких как блок основной памяти или файл. И наконец, процессы могут быть осведомлены о наличии других процессов непосредственно и сотрудничать с ними путем обмена информацией. Ключевыми вопросами при таком взаимодействии процессов становятся взаимные исключения и взаимоблокировки.

Взаимное исключение представляет собой выполнение условия, при котором из множества параллельно выполняющихся процессов одновременно получить доступ к некоторому ресурсу или выполнить определенную функцию в определенный момент времени может только один из них. Технология взаимоисключений может применяться для разрешения конфликтов, таких как конкуренция за использование ресурсов, и для синхронизации сотрудничающих процессов. Примером последнего может служить модель производителя/потребителя, в которой один процесс помещает данные в буфер, а другой (или несколько процессов) извлекает их оттуда.

Один из подходов к решению задачи взаимоисключений состоит в использовании специализированных машинных команд. Этот подход несколько снижает накладные расходы, однако все равно имеет низкую эффективность в связи с использованием технологии пережидания занятости.

Еще один подход к решению задачи поддержки взаимных исключений состоит в обеспечении специализированных возможностей в операционной системе. Наиболее широко распространенными технологиями являются семафоры и система передачи сообщений. Семафоры используются для передачи сигналов между процессами и могут быть легко применены для решения задачи взаимоисключений. Сообщения, кроме обеспечения взаимоисключений, являются эффективным средством связи между процессами.

## 5.9. КЛЮЧЕВЫЕ ТЕРМИНЫ, КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАЧИ

### Ключевые термины

Атомарность	Критический ресурс	Прямая адресация
Бинарный семафор	Критический участок	Семафор
Блокировка	Монитор	Семафор со счетчиком
Взаимное исключение	Мьютекс	Сильный семафор
Взаимоблокировка	Параллельность	Слабый семафор
Голодание	Параллельные процессы	Сопрограмма
Динамическая взаимоблокировка	Передача сообщений	Состояние гонки
Косвенная адресация	Пережидание занятости	Условная переменная

### Контрольные вопросы

- 5.1. Перечислите основные проблемы, связанные с концепцией параллельных вычислений.
- 5.2. Каковы три различных контекста, в которых проявляются параллельные вычисления?
- 5.3. Каковы основные требования к выполнению параллельных процессов?
- 5.4. Перечислите три степени осведомленности процесса о наличии других процессов и вкратце опишите их.
- 5.5. В чем основное различие между конкурирующими и сотрудничающими процессами?
- 5.6. Перечислите основные проблемы, связанные с конкуренцией процессов, и вкратце опишите их.
- 5.7. Перечислите требования к взаимоисключениям.
- 5.8. Какие операции могут выполняться над семафорами?
- 5.9. В чем различие между бинарными и обобщенными семафорами?
- 5.10. В чем различие между сильными и слабыми семафорами?
- 5.11. Что такое монитор?
- 5.12. В чем состоит различие между *блокирующими* и *неблокирующими* операциями в системе передачи сообщений?
- 5.13. Какие условия обычно связаны с задачей читателей/писателей?

## Задачи

**5.1.** Докажите корректность алгоритма Деккера.

- Покажите, что при этом обеспечиваются взаимные исключения. Указание: покажите, что, когда  $P_i$  входит в критический участок, истинно следующее выражение:  $\text{flag}[i] \text{ and } (\text{not } \text{flag}[1-i])$ .
- Покажите, что процесс, запрашивающий доступ к критическому участку, не может быть задержан бесконечно. Указание: рассмотрите следующие случаи: 1) в критический участок пытается войти единственный процесс; 2) оба процесса пытаются войти в критический участок и при этом 2, а)  $\text{turn}=0$  и  $\text{flag}[0]=\text{false}$  и 2, б)  $\text{turn}=0$  и  $\text{flag}[0]=\text{true}$ .

**5.2.** Рассмотрим алгоритм Деккера, записанный для произвольного количества процессов путем замены выполняемой при выходе из критического раздела инструкции

```
turn = 1 - i           /* P0 устанавливает turn равным 1,
                      а P1 – равным 0 */
```

инструкцией

```
turn = (turn + 1) % n /* n – количество процессов */
```

Проследите работу алгоритма при количестве параллельных процессов, превышающем два.

**5.3.** Покажите, что следующие программные подходы к реализации взаимоисключений не зависят от элементарных взаимоисключений на уровне доступа к памяти.

- Алгоритм булочной (см. задачу 5.10).
- Алгоритм Петерсона.

**5.4.** В начале раздела 5.2 говорилось, что многозадачность и многопроцессность имеют в отношении параллелизма одни и те же проблемы. В принципе это верно. Однако укажите два различия между многозадачностью и многопроцессностью, касающиеся параллелизма.

**5.5.** Процессы и потоки представляют мощный инструмент структурирования при реализации существенно более сложных, чем простые последовательные, программ. Цель данного упражнения — введение сопрограмм и сравнение их с процессами. Рассмотрим простую задачу из [50].

Прочесть 80-столбцовую перфокарту и вывести ее в виде 125-символьных строк; при этом после каждой перфокарты следует добавить пробел, а каждую пару звездочек (\*\*) следует заменить символом ↑.

- a. Разработайте решение поставленной задачи в виде обычной последовательной программы. Вы увидите, что решение задачи не такое простое, как кажется. Взаимодействие различных элементов программы оказывается достаточно сложным в связи с преобразованием длины строк от 80 до 125; кроме того, длина считанной с перфокарты строки меняется в зависимости от того, сколько пар звездочек удалось в ней обнаружить. Чтобы сделать решение поставленной задачи более понятным и минимизировать количество потенциальных ошибок, имеет смысл реализовать программу как три различные процедуры. Первая считывает строки перфокарты, добавляет к каждой из них пробел и записывает поток символов во временный файл. После того как будут считаны все перфокарты, вторая процедура приступает к чтению временного файла, выполняет замену `**` на `↑` и записывает результат во второй временный файл. Третья процедура читает второй временный файл и выводит его строками по 125 символов.
- b. Последовательное решение непривлекательно хотя бы в силу больших накладных расходов на операции ввода-вывода и создание временных файлов. Конвой (Conway) предложил новый вид структуры программы — сопрограмму, которая позволяет написать приложение как три программы, присоединенные к односимвольному буферу (рис. 5.28). При использовании традиционных процедур между вызывающей и вызываемой процедурами устанавливается отношение “ведущий/ведомый”. Вызывающая процедура может в любой момент осуществить вызов; вызываемая процедура начинает работу со своей точки входа и возвращает управление вызывающей процедуре в точке вызова. Сопрограмма демонстрирует более симметричные отношения. При каждом вызове выполнение продолжается от последней активной точки вызванной процедуры. Поскольку в таком случае невозможно говорить о том, что вызывающая процедура “выше” вызываемой, возврата из процедуры нет. Вместо этого любая сопрограмма может передать управление любой другой сопрограмме при помощи команды `resume`. При первом вызове сопрограммы она “продолжает выполнение” со своей точки входа. Затем сопрограммы возобновляют выполнение в точках своих последних вызовов команды `resume`. Заметим, что одновременно может выполняться только одна сопрограмма основной программы и что передача управления явным образом указывается в коде сопрограммы, так что говорить о параллельных вычислениях в этом случае не приходится. Рассмотрите и поясните работу программы, приведенной на рис. 5.28.
- c. В программе нет условия окончания работы. Полагая, что подпрограмма ввода-вывода `READCARD` возвращает значение `true`, если успешно считывает и размещает 80-символьный образ в `inbuf` (и `false` в противном случае), измените программу так, чтобы она учитывала возможность окончания чтения карты (заметим, что в этом случае последняя выведенная строка может содержать менее 125 символов).
- d. Перепишите это решение в виде множества, состоящего из трех процессов, с использованием семафоров.

```

char rs, sp;
char inbuf[80], outbuf[125];
void read()
{
    while (true) {
        READCARD (inbuf);
        for (int i=0; i < 80; i++)
        {
            rs = inbuf [i];
            RESUME squash
        }
        rs = "↑";
        RESUME squash;
    }
}

void print()
{
    while (true) {
        for (int j = 0; j < 125; j++)
        {
            outbuf [j] = sp;
            RESUME squash
        }
        OUTPUT (outbuf);
    }
}

```

```

void squash()
{
    while (true) {
        if (rs != "*") {
            sp = rs;
            RESUME print;
        }
        else {
            RESUME read;
            if (rs == "*")
            {
                sp = "↑";
                RESUME print;
            }
            else {
                sp = "*";
                RESUME print;
                sp = rs;
                RESUME print;
            }
        }
        RESUME read;
    }
}

```

**Рис. 5.28.** Применение сопрограмм

## 5.6. Рассмотрим следующую программу:

P1: {         shared int x;         x = 10;         while (1) {             x = x - 1;             x = x + 1;             if (x != 10)                 printf("x is %d",x)         }     }	P2: {         shared int x;         x = 10;         while (1) {             x = x - 1;             x = x + 1;             if (x!=10)                 printf("x is %d",x)         }     }
--	--

Обратите внимание, что планировщик в однопроцессорной системе будет реализовывать псевдопараллельное выполнение этих двух параллельных процессов путем чередования их команд, без ограничений на порядок чередования.

- a. Покажите последовательность чередования инструкций, которая приведет к выводу строки "x is 10".
- b. Покажите последовательность чередования инструкций, которая приведет к выводу строки "x is 8". Вы должны помнить, что увеличение/уменьшение на уровне исходного языка не является атомарным, т.е. код на языке ассемблера, реализующий одну инструкцию  $x = x + 1$  на языке программирования C, имеет следующий вид:

```
LD R0,X    /* Загрузка R0 из ячейки памяти x */
INCR R0    /* Увеличение R0 */
STO R0,X   /* Сохранение увеличенного значения в x */
```

### 5.7. Рассмотрим следующую программу:

```
const int n = 50;
int tally;
void total()
{
    int count;
    for(count = 1; count <= n; count++)
    {
        tally++;
    }
}
void main()
{
    tally = 0;
    parbegin(total(), total());
    write(tally());
}
```

- a. Определите нижнюю и верхнюю границы окончательного значения разделяемой переменной tally на выходе из приведенной программы в предположении, что процессы могут выполняться с любой относительной скоростью и что значение может быть увеличено только после того, как оно загружено в регистр отдельной машинной командой.
- b. Предположим, что одновременно может выполняться произвольное количество процессов (при этом остаются в силе предположения о работе программы из предыдущего задания). Как это повлияет на границы окончательного значения переменной tally?

### 5.8. Всегда ли пережидание занятости менее эффективно (с точки зрения процессорного времени), чем блокирующее ожидание? Поясните свой ответ.

### 5.9. Рассмотрим следующую программу:

```

boolean blocked[2];
int turn;
void P(int id)
{
    while(true)
    {
        blocked[id] = true;
        while(turn != id)
        {
            while(blocked[1-id])
                /* Ничего не делаем */ ;
            turn = id;
        }
        /* Критический участок */
        blocked[id] = false;
        /* Остальной код */
    }
}
void main()
{
    blocked[0] = false;
    blocked[1] = false;
    turn = 0;
    parbegin(P(0), P(1));
}

```

Это программное решение задачи взаимных блокировок предложено в [114]. Найдите контрпример, демонстрирующий некорректность данного решения. Интересно, что даже редакция журнала *Communications of the ACM* сочла это решение корректным.

Еще одним программным подходом к задаче взаимоисключений является **алгоритм булочной** Лампорта (Lamport) [139], названный так из-за того, что он основан на практике булочных (и других магазинов), в которых каждый покупатель при входе получает нумерованный билет, позволяющий обслужить всех покупателей по очереди.

```

boolean choosing[n];
int number[n];
while(true)
{
    choosing[i] = true;
    number[i] = 1+getmax(number[],n);
    choosing[i] = false;
    for(int j = 0; j < n; j++)
    {
        while(choosing[j]) {};
        while((number[j] != 0) &&
              (number[j],j) < (number[i],i))
            ) {};
        /* Критический участок */
        number[i] = 0;
        /* Остальная часть кода */
    }
}

```

**5.10.** Массивы `choosing` и `number` инициализируются соответственно значениями 0 и `false`.  $i$ -й элемент каждого массива может быть прочитан и перезаписан процессом  $i$ , но другие процессы могут только читать соответствующие значения. Запись  $(a, b) < (c, d)$  означает

$$(a < c) \text{ или } (a = c \text{ и } b < d)$$

- Опишите алгоритм словами.
- Покажите, что данный алгоритм позволяет избежать взаимоблокировок.
- Покажите, что данный алгоритм обеспечивает взаимные исключения.

**5.11.** Рассмотрим версию алгоритма булочной без выбора переменной. В таком случае мы имеем

```

1 int number[n];
2 while (true) {
3     number[i] = 1 + getmax(number[], n);
4     for (int j = 0; j < n; j++) {
5         while((number[j] != 0) && (number[j], j) < (number[i], i)) {};
6     }
7     /* Критический участок */;
8     number [i] = 0;
9     /* Остальная часть кода */;
10 }
```

Не нарушает ли эта версия требование взаимных исключений? Почему да или почему нет?

**5.12.** Рассмотрим следующий программный подход ко взаимным исключениям:

```
integer array control[1:N]; integer k
```

Здесь  $1 \leq k \leq N$ , и каждый элемент массива `control` представляет собой 0, 1 либо 2. Все элементы изначально нулевые; начальное значение `k` не имеет значения.

Программа  $i$ -го ( $1 \leq i \leq N$ ) процесса имеет вид

```

begin
    integer j;
L0: control [i] := 1;
LI: for j := k step 1 until N, 1 step 1 until k do
begin
    if j = i then goto L2;
    if control [j] ≠ 0 then goto L1
end;
L2: control [i] := 2;
    for j := 1 step 1 until N do
        if j ≠ i and control [j] = 2 then goto L0;
L3: if control [k] ≠ 0 and k ≠ i then goto L0;
L4: k := i;
    Критический участок;
L5: for j := k step 1 until N, 1 step 1 until k do
    if j ≠ k and control [j] ≠ 0 then
```

```

begin
    k := j;
    goto L6
end;
L6: control [i] := 0;
L7: Оставшаяся часть цикла;
    goto L0;
end

```

Это алгоритм Айзенберга–Мак-Гуайра (Eisenberg–McGuire). Поясните, как он работает и каковы его ключевые возможности.

### 5.13. Рассмотрим первую инструкцию `bolt = 0` на рис. 5.5, б.

- Получите тот же результат с помощью команды обмена.
- Какой метод предпочтительнее?

### 5.14. При использовании для реализации взаимоисключений специальных машинных команд на рис. 5.5 никакой контроль над продолжительностью ожидания доступа к критическому участку не осуществлялся. Разработайте алгоритм, который использует команду `compare&swap` и гарантирует, что любой процесс, ожидающий входа в критический участок, дождется своей очереди в пределах $n-1$ цикла, где $n$ — количество процессов, которые могут запросить доступ на вход в критический участок, а “цикл” — событие, состоящее в том, что один процесс покидает критический участок, а другой получает право входа в него.

### 5.15. Рассмотрим следующее определение семафоров:

```

void semWait(s)
{
    if (s.count > 0)
    {
        s.count--;
    }
    else
    {
        Поместить этот процесс в s.queue
        Заблокировать
    }
}
void semSignal(s)
{
    if (Имеется по крайней мере один
        приостановленный семафором s процесс)
    {
        Удалить процесс P из s.queue
        Поместить P в список активных процессов
    }
    else
        ...
    s.count++;
}

```

Сравните это определение с приведенным на рис. 5.6. Обратите внимание на одно отличие: в приведенном здесь определении семафор никогда не принимает отрицательное значение. Насколько это отличие влияет на работу семафора в программе? Можно ли заменить один семафор другим без изменения работы программы?

## 5.16. Рассмотрим совместно используемый ресурс со следующими характеристиками:

1) до тех пор, пока имеется меньше трех процессов, использующих этот ресурс, новые процессы могут начать использовать его немедленно; 2) когда есть три процесса, использующие ресурс, все три должны прекратить работу с ресурсом, прежде чем любые новые процессы смогут начать использовать его. Мы понимаем, что для отслеживания ждущих и активных процессов необходимы счетчики и что эти счетчики сами являются общими ресурсами, которые должны быть защищены с помощью взаимного исключения. Таким образом, можно было бы создать следующее решение:

```

1 semaphore mutex = 1, block = 0; /* Общие переменные: семафоры, */
2 int active = 0, waiting = 0;    /* счетчики и информация */
3 boolean must_wait = false;     /* о состоянии */
4
5 semWait(mutex);             /* Вход во взаимоисключение      */
6 if(must_wait) {              /* Если есть (или было) 3, то      */
7     ++waiting;               /* нужно ждать, но сначала нужно */
8     semSignal(mutex);        /* покинуть взаимоисключение      */
9     semWait(block);          /* Ожидание, пока уйдут все       */
10    semWait(mutex);          /* Повторный вход во взаимоисключение */
11    --waiting;               /* и обновление счетчика ожидания */
12 }
13 ++active;                  /* Обновление количества активных и */
14 must_wait=active==3;        /* запоминание достижения значения 3 */
15 semSignal(mutex);          /* Покидаем взаимоисключение      */
16
17 /* Критический участок */
18
19 semWait(mutex);             /* Вход во взаимоисключение      */
20 --active;                  /* и обновление счетчика active   */
21 if(active == 0) {           /* Покидает последний? */
22     int n;
23     if (waiting < 3) n = waiting;
24     else n = 3;              /* Если да – разблокируем до 3   */
25     while( n > 0 ) {         /* ждущих процессов */
26         semSignal(block);
27         --n;
28     }
29     must_wait = false; /* Покинули все активные процессы */
30 }
31 semSignal(mutex);          /* Покинули взаимоисключение      */

```

Решение представляется верным: все доступы к общим переменным защищены взаимоисключением, процессы во взаимоисключении не блокируют сами себя, новые процессы лишены возможности использования ресурсов, если есть (или были) три активных пользователя, а последний процесс, уходя, разблокирует до трех ожидающих процессов.

- Тем не менее программа некорректна. Поясните, почему.
- Предположим, что в строке 6 мы заменим `if` конструкцией `while`. Решает ли это все проблемы программы? Остаются ли при этом какие-либо проблемы?

### 5.17. Теперь рассмотрим корректное решение предыдущей задачи.

```

1 semaphore mutex = 1, block = 0; /* Общие переменные: семафоры,*/
2 int active = 0, waiting = 0;    /* счетчики и информация */
3 boolean must_wait = false;     /* о состоянии */
4
5 semWait(mutex);           /* Вход во взаимоисключение */
6 if(must_wait) {           /* Если есть (или было) 3, то */
7     ++waiting;            /* нужно ждать, но сначала нужно */
8     semSignal(mutex);    /* покинуть взаимоисключение */
9     semWait(block);      /* Ожидание, пока уйдут все */
10 } else {
11     ++active;             /* Обновление количества активных и */
11 }
12 must_wait=active==3; /* запоминаем достижение значения 3 */
13 semSignal(mutex);    /* Покидаем взаимоисключение */
14 }
15
16 /* Критический участок */
17
18 semWait(mutex);           /* Вход во взаимоисключение */
19 --active;                 /* и обновление счетчика active */
20 if(active == 0) {         /* Покидает последний? */
21     int n;
22     if (waiting < 3) n = waiting;
23     else n = 3;           /* Если да, смотрим, сколько процессов*/
24     waiting -= n;        /* разблокировать. Выводим значение из*/
25     active = n;          /* ожидающих и присваиваем active */
26     while( n > 0 ) {     /* Разблокируем процессы по одному */
27         semSignal(block);
28         --n;
29     }
30     must_wait=active==3; /* Запоминаем, если значение - 3 */
31 }
32 semSignal(mutex);    /* Покинули взаимоисключение */

```

- Поясните, как работает эта программа и почему она правильная.
- Это решение не предотвращает полностью вновь приходящие процессы от прохода вне очереди, хотя и делает его менее вероятным. Приведите пример такой ситуации.
- Эта программа является примером общего проектного шаблона, который является стандартным средством реализации многих задач, связанных с параллельными вычислениями, с помощью семафоров. Он известен как шаблон **Я сделаю это вместо тебя** (I'll Do It For You). Опишите этот шаблон.

### 5.18. Рассмотрим еще одно корректное решение предыдущей задачи.

```

1 semaphore mutex = 1, block = 0; /* Общие переменные: семафоры, */
2 int active = 0, waiting = 0;    /* счетчики и информация */
3 boolean must_wait = false;     /* о состоянии */
4
5 semWait(mutex);               /* Вход во взаимоисключение */
6 if(must_wait) {               /* Если есть (или было) 3, то */
7     ++waiting;                /* нужно ждать, но сначала нужно */
8     semSignal(mutex);         /* покинуть взаимоисключение */
9     semWait(block);          /* Ожидание, пока уйдут все */
10    --waiting;                /* Взаимоисключение; обновление count */
11 }
12 ++active;                   /* Обновление количества активных, */
13 must_wait = active == 3; /* запись по достижении 3 */
14 if(waiting > 0 && !must_wait) /* Если есть иные ожидающие */
15     semSignal(block);        /* и нет трех активных, разблоки-
16                                /* рование ожидающего процесса */
17 else semSignal(mutex);      /* В противном случае открываем */
18                                /* взаимоисключение */
19 /* Критический участок */
20
21 semWait(mutex);               /* Вход во взаимоисключение */
22 --active;                    /* и обновление счетчика active */
23 if(active == 0)              /* Покидает последний? */
24     must_wait = false;        /* Настройка входа новых процессов */
25 if(waiting == 0 && !must_wait) /* Если есть иные ожидающие */
26     semSignal(block);        /* и нет трех активных, разблоки-
27                                /* рование ожидающего процесса */
28 else semSignal(mutex);      /* В противном случае открываем */
29                                /* взаимоисключение */

```

- Поясните, как работает эта программа и почему она правильная.
- Отличается ли это решение от предыдущего в плане количества процессов, которые могут быть разблокированы одновременно? Поясните.
- Эта программа является примером общего проектного шаблона, который является стандартным средством реализации многих задач, связанных с параллельными вычислениями, с помощью семафоров. Он известен как шаблон **Передача эстафеты** (Pass The Baton). Опишите этот шаблон.

### 5.19. Бинарных семафоров должно быть достаточно для реализации обобщенных семафоров. Мы можем использовать для этого операции `semWaitB` и `semSignalB` и два бинарных семафора, `delay` и `mutex`. Рассмотрим следующий код.

```

void semWait(semaphore s)
{
    semWaitB(mutex);
    s--;
    if (s < 0)
    {
        semSignalB(mutex);
        semWaitB(delay);
    }
}

```

```

        else
            semSignalB(mutex);
    }
void semSignal(semaphore s)
{
    semWaitB(mutex);
    s++;
    if (s <= 0)
        semSignalB(delay);
    semSignalB(mutex);
}

```

Изначально семафор *s* инициализирован необходимым нам значением. Каждая операция *semWait* уменьшает значение *s*, а *semSignal* — увеличивает. Бинарный семафор *mutex*, инициализированный значением 1, обеспечивает взаимные исключения при обновлении значения *s*, а бинарный семафор *delay*, инициализированный нулевым значением, использован для блокировки процесса.

В приведенном коде имеется один дефект. Найдите его и предложите метод исправления. Указание: рассмотрите ситуацию, когда два процесса вызывают *semWait(s)* в тот момент, когда *s* изначально равно 0, и сразу после того, как первый выполнит *semSignalB(mutex)*, но до *semWaitB(delay)*, вызов *semWait(s)* вторым процессом дойдет до той же точки. Все, что надо сделать для решения поставленной задачи, — это переместить одну строку кода.

- 5.20.** В 1978 году Дейкстру выдвинул предположение о том, что не имеется решения задачи взаимных исключений неизвестного, но конечного числа процессов с использованием конечного числа слабых семафоров, которое позволяло бы избежать проблемы голодаания. В 1979 году Моррис (J. M. Morris) опроверг это предположение, опубликовав решение с использованием трех слабых семафоров. Алгоритм можно описать следующим образом. Если один или несколько процессов находятся в состоянии ожидания в операции *semWait(s)*, а другой процесс выполняет операцию *semSignal(s)*, то значение семафора *s* не изменяется и один из ожидающих процессов деблокируется независимо от *semWait(s)*. Кроме этих трех семафоров, алгоритм использует две неотрицательные целочисленные переменные в качестве счетчиков числа процессов, находящихся в некоторых участках алгоритма. Таким образом, семафоры A и B инициализированы значением 1, а семафор M и счетчики NA и NM — значением 0. Семафор B обеспечивает взаимоисключения при доступе к совместно используемой переменной NA. Процесс, пытающийся войти в критический участок, должен пройти через два барьера, представленные семафорами A и M. Счетчики NA и NM содержат, соответственно, число процессов, готовых пересечь барьер A и пересекших A, но еще не пересекших M. Во второй части протокола NM процессов, заблокированных семафором M, будут входить в критический участок один за другим, с использованием каскадной методики, аналогичной использованной в первой части. Напишите код алгоритма, соответствующего данному описанию.

### 5.21. Следующая задача однажды была предложена на экзамене.

Парк Юрского периода состоит из музея динозавров и безвольерного зоопарка. Имеется  $m$  пассажиров и  $n$  одноместных машин. Пассажиры некоторое время проводят в музее, а затем на машине посещают зоопарк. Если имеется свободная машина, в ней размещается один пассажир, который совершает поездку по парку в течение некоторого (случайного) времени. Если все  $n$  машин заняты, то пассажиры, желающие посетить парк, ожидают, пока машины освободятся; если есть свободные машины, но нет ожидающих их пассажиров, то в состоянии ожидания находятся машины. Используйте семафоры для синхронизации  $m$  процессов пассажиров и  $n$  процессов машин.

Приведенный далее набросок кода был найден на обрывке бумаги в экзаменационной аудитории. Проверьте корректность наброска, не обращая внимания на синтаксис и отсутствие объявлений переменных. Не забывайте, что P и V соответствуют semWait и semSignal.

```
resource Jurassic_Park()
    sem car_avail := 0, car_taken := 0, car_filled := 0,
        passenger_released := 0;
    process passenger(i := 1 to num_passengers)
        do true -> nap(int(random(1000*wander_time)))
            P(car_avail); V(car_taken); P(car_filled)
            P(passenger_released)
        od
    end passenger
    process car(j := 1 to num_cars)
        do true -> V(car_avail); P(car_taken); V(car_filled)
            nap(int(random(1000*ride_time)))
            V(passenger_released)
        od
    end car
end Jurassic_Park
```

5.22. В комментариях к рис. 5.12 и табл. 5.4 говорится: “Простое перемещение проверки в критический участок потребителя недопустимо, так как может привести к взаимоблокировке”. Продемонстрируйте это при помощи таблицы, аналогичной табл. 5.4.

5.23. Рассмотрим решение задачи производителя/потребителя с бесконечным буфером, приведенное на рис. 5.13. Предположим, что скорости работы производителя и потребителя примерно одинаковы (достаточно распространенная ситуация). Тогда сценарий работы может выглядеть примерно следующим образом:

Производитель: append;semSignal;produce;...  
                  ;append;semSignal;produce;...

Потребитель: consume;...;take;semWait;consume;...;take;semWait;...

Производитель всегда ухитряется добавлять новый элемент в буфер и сигнализировать об этом в тот момент, когда потребитель изымает из буфера предыдущий элемент. При этом производитель всегда добавляет новый элемент в пустой буфер, а потребитель изымает из буфера единственный имеющийся там элемент. Хотя потребитель никогда не блокируется семафором, к нему при этом выполняется достаточно большое количество обращений, что приводит к существенным накладным расходам.

Разработайте новую, более эффективно работающую в этих условиях программу. Указание: позвольте *n* принимать значение  $-1$ , означающее, что буфер пуст и что потребитель распознал это состояние буфера и заблокирован до тех пор, пока производитель не добавит в буфер новые данные. Это решение не требует использования локальной переменной *m*, которая имеется на рис. 5.13.

**5.24.** Рассмотрите рис. 5.16. Изменится ли смысл программы при взаимной замене приведенных далее инструкций?

- a. semWait(e)  
    semWait(s)
- b. semSignal(s)  
    semSignal(n)
- c. semWait(n)  
    semWait(s)
- d. semSignal(s)  
    semSignal(e)

**5.25.** Приведенный далее псевдокод представляет собой корректную реализацию задачи производителя/потребителя с ограниченным буфером.

```

item[3] buffer;           // Изначально пустой
semaphore empty;         // Инициализирован значением +3
semaphore full;          // Инициализирован значением 0
binary_semaphore mutex;   // Инициализирован значением 1
void producer()
{
    ...
    while (true) {
        item = produce();
        p1: wait(empty);
        /  wait(mutex);
        p2: append(item);
        \  signal(mutex);
        p3: signal(full);
    }
}
void consumer()
{
    ...
    while (true) {
        c1: wait(full);
        /  wait(mutex);
        c2: item' = take();
        \  signal(mutex);
        c3: signal(empty);
        consume(item);
    }
}

```

Метки p1, p2, p3 и c1, c2, c3 относятся к строкам приведенного выше кода (p2 и c2 охватывают по три строки кода). Семафоры empty и full представляют собой линейные семафоры, которые могут принимать неограниченные отрицательные и положительные значения. Имеется несколько процессов производителей, именуемых Pa, Pb, Pc, ..., и несколько процессов потребителей, именуемых Ca, Cb, Cc, ... Каждый семафор поддерживает очередь FIFO (“первым вошел — первым вышел”) заблокированных процессов. В диаграмме планирования, показанной ниже, каждая строка представляет состояние буфера и семафоров после запланированного выполнения. Для простоты мы предполагаем, что планирование таково, что процессы никогда не будут прерваны во время выполнения части кода p1, p2, ..., c3. Ваша задача заключается в том, чтобы закончить приведенную ниже диаграмму.

Планируемый шаг выполнения	Состояние full и очереди	Буфер	Состояние empty и очереди
Инициализация	full = 0	000	empty = +3
Ca выполняет c1	full = -1 (Ca)	000	empty = +3
Cb выполняет c1	full = -2 (Ca, Cb)	000	empty = +3
Pa выполняет p1	full = -2 (Ca, Cb)	000	empty = +2
Pa выполняет p2	full = -2 (Ca, Cb)	X00	empty = +2
Pa выполняет p3	full = -1 (Cb) Ca	X00	empty = +2
Ca выполняет c2	full = -1 (Cb)	000	empty = +2
Ca выполняет c3	full = -1 (Cb)	000	empty = +3
Pb выполняет p1	full =		empty =
Pa выполняет p1	full =		empty =
Pa выполняет	full =		empty =
Pb выполняет	full =		empty =
Pb выполняет	full =		empty =
Pc выполняет p1	full =		empty =
Cb выполняет	full =		empty =
Pc выполняет	full =		empty =
Cb выполняет	full =		empty =
Pa выполняет	full =		empty =
Pb выполняет p1-p3	full =		empty =
Pc выполняет	full =		empty =
Pa выполняет p1	full =		empty =
Pd выполняет p1	full =		empty =
Ca выполняет c1-c3	full =		empty =
Pa выполняет	full =		empty =
Cc выполняет c1-c2	full =		empty =
Pa выполняет	full =		empty =
Cc выполняет c3	full =		empty =
Pd выполняет p2-p3	full =		empty =

- 5.26.** Эта задача демонстрирует использование семафоров для согласования процессов трех типов.<sup>7</sup> Дед Мороз спит в своей избушке на Северном полюсе и может быть разбужен только 1) пятью сестрами-Снегурочками, вернувшимися из отпуска, или 2) несколькими Снеговиками, у которых возникли проблемы при изготовлении игрушек. Чтобы дать Деду Морозу поспать, Снеговики будят его только в том случае, когда проблемы возникают у троих из них. Когда трое Снеговиков решат свои вопросы, все остальные Снеговики, желающие посетить Деда Мороза, должны ждать, пока вернется побывавшая у него тройка. Если проснувшийся Дед Мороз обнаруживает у дверей избушки и Снеговиков, и последнюю из Снегурочек, то он просит Снеговиков обождать со своими проблемами до окончания празднования Нового года, поскольку куда важнее ехать поздравлять малышей. (Снегурочки не спешат вернуться из отпуска и остаются там до последнего момента.) Прибывшая последней Снегурочка идет будить Деда Мороза, пока остальные занимаются макияжем в своих комнатах. Решите эту задачу с помощью семафоров.
- 5.27.** Покажите, что система передачи сообщений и семафоры обладают эквивалентной функциональностью, для чего выполните следующее.
- Реализуйте передачу сообщений с использованием семафоров. Указание: сделайте это с помощью совместно используемого буфера для хранения почтовых ящиков, каждый из которых представляет собой массив слотов для сообщений.
  - Реализуйте семафоры с использованием передачи сообщений. Указание: добавьте в систему синхронизирующий процесс.
- 5.28.** Объясните, в чем заключается проблема в этой реализации задачи с одним писателем и многими читателями?

```

int readcount;           // Общая переменная, инициализирована 0
Semaphore mutex, wrt;   // Общие семафоры, инициализированы 1;

// Писатель:
semWait(mutex);
semWait(wrt);
/* Выполнение записи */
semSignal(wrt);

// Читатели:
semWait(mutex);
readcount := readcount + 1;
if readcount == 1 then semWait(wrt);
semSignal(mutex);
/* Выполнение чтения */
semWait(mutex);
readcount := readcount - 1;
if readcount == 0 then Up(wrt);
semSignal(mutex);

```

<sup>7</sup> Я признателен Джону Троно (John Trono) из колледжа Св. Михаила в Вермонте за предложение этой задачи.

# ГЛАВА 6

---

# ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ: ВЗАИМОБЛОКИРОВКА И ГОЛОДАНИЕ

В ЭТОЙ ГЛАВЕ...

## 6.1. Принципы взаимного блокирования

- Повторно используемые ресурсы
- Расходуемые ресурсы
- Графы распределения ресурсов
- Условия возникновения взаимоблокировок

## 6.2. Предотвращение взаимоблокировок

- Взаимоисключения
- Удержание и ожидание
- Отсутствие перераспределения
- Циклическое ожидание

## 6.3. Устранение взаимоблокировок

- Запрещение запуска процесса
- Запрет выделения ресурса

## 6.4. Обнаружение взаимоблокировок

- Алгоритм обнаружения взаимоблокировки
- Восстановление

## 6.5. Интегрированные стратегии разрешения взаимоблокировок

## 6.6. Задача об обедающих философах

- Решение с использованием семафоров
- Решение с использованием монитора

**6.7. Механизмы параллельных вычислений в UNIX**

- Каналы
- Сообщения
- Совместно используемая память
- Семафоры
- Сигналы

**6.8. Механизмы параллельных вычислений ядра Linux**

- Атомарные операции
- Циклические блокировки
  - Базовые циклические блокировки
  - Циклические блокировки читателя/писателя
- Семафоры
  - Бинарные семафоры и семафоры со счетчиками
  - Семафоры читателя/писателя
- Барьеры
- RCU (Read-Copy-Update)

**6.9. Примитивы синхронизации потоков Solaris**

- Блокировки взаимоисключений
- Семафоры
- Блокировки "читатели/писатель"
- Условные переменные

**6.10. Механизмы параллельных вычислений в Windows**

- Функции ожидания
- Объекты диспетчера
- Критические участки
- Гибкие блокировки читателя/писателя и условные переменные
- Синхронизация без участия блокировок

**6.11. Межпроцессное взаимодействие в Android****6.12. Резюме****6.13. Ключевые термины, контрольные вопросы и задачи**

- Ключевые термины
- Контрольные вопросы
- Задачи

## УЧЕБНЫЕ ЦЕЛИ

- Перечислить и пояснить условия взаимоблокировки.
- Описать стратегии предотвращения взаимоблокировок, связанные с каждым из условий взаимоблокировки.
- Объяснить разницу между предотвращением взаимоблокировки и устранением взаимоблокировки.
- Объяснить разницу между двумя подходами к устранению взаимоблокировки.
- Объяснить фундаментальную разницу между обнаружением взаимоблокировок и предотвращением и устранением взаимоблокировок.
- Пояснить принципы разработки комплексных стратегий для взаимоблокировок.
- Проанализировать задачу обедающих философов.
- Пояснить методы параллельных вычислений и синхронизации, используемые в UNIX, Linux, Solaris, Windows и Android.

В этой главе продолжается рассмотрение параллельных вычислений. Кроме того, она посвящена двум проблемам, доставляющим основные неприятности при работе с параллельными вычислениями: взаимоблокировкам и голоданию. Глава начнется с изложения основных принципов взаимоблокировок и связанный с ними проблемы голодания. Затем мы узнаем о трех основных подходах к работе с взаимоблокировками: предотвращении, обнаружении и устранении. Под конец мы рассмотрим еще одну классическую задачу, иллюстрирующую вопросы синхронизации и взаимоблокировки, а именно — задачу об обедающих философах.

Как и в главе 5, “Параллельные вычисления: взаимоисключения и многозадачность”, здесь мы ограничимся рассмотрением проблем в единой системе; распределенным системам посвящена глава 18, “Распределенная обработка, вычисления «клиент/сервер» и кластеры”. Анимации, иллюстрирующие взаимоблокировки, можно найти на сайте данной книги.

## 6.1. ПРИНЦИПЫ ВЗАЙМНОГО БЛОКИРОВАНИЯ

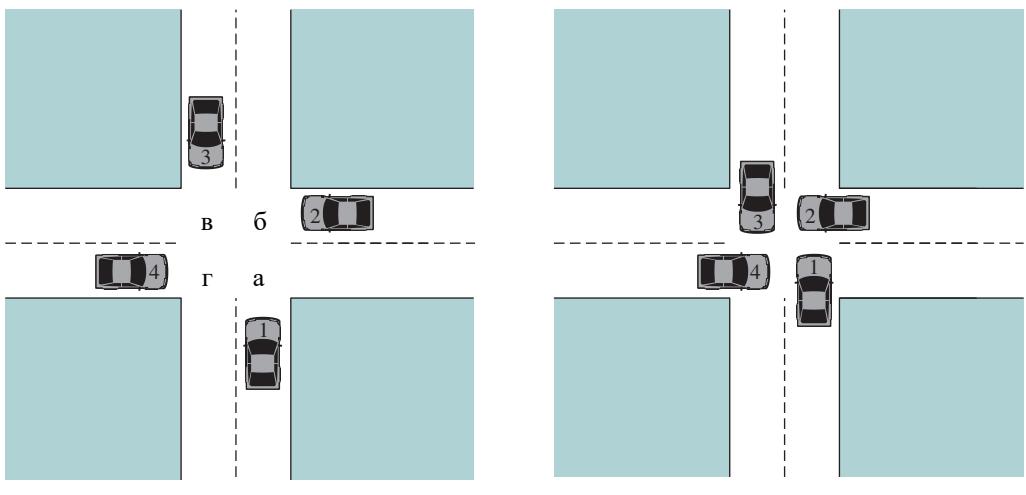
Взаимное блокирование (deadlock<sup>1</sup>) можно определить как *перманентное* блокирование множества процессов, которые либо конкурируют в борьбе за системные ресурсы, либо сообщаются один с другим. Множество процессов оказывается взаимно блокированным, если каждый процесс множества заблокирован в ожидании события (обычно — освобождения некоторого запрашиваемого ресурса), которое может быть вызвано только другим блокированным процессом множества. В отличие от других проблем, возникающих в процессе управления параллельными вычислениями, данная проблема в общем случае эффективного решения не имеет.

---

<sup>1</sup> Дословно — “мертвые объятия”. В русскоязычной литературе встречаются термины “тупик” и “клинич”, однако наиболее полно отражает суть происходящего именно термин “взаимоблокировка”. — Примеч. пер.

Все взаимоблокировки предполагают наличие конфликта в борьбе за ресурсы между двумя или несколькими процессами. Наиболее ярким примером может служить транспортная взаимоблокировка. На рис. 6.1 показана ситуация, когда четыре автомобиля должны примерно одновременно пересечь перекресток. Четыре квадранта перекрестка представляют собой ресурсы, которые требуются процессам. В частности, для успешного пересечения перекрестка всеми четырьмя автомобилями необходимые ресурсы выглядят следующим образом.

- Автомобилю 1, движущемуся на север, нужны квадранты *a* и *b*.
- Автомобилю 2, движущемуся на запад, нужны квадранты *b* и *c*.
- Автомобилю 3, движущемуся на юг, нужны квадранты *c* и *d*.
- Автомобилю 4, движущемуся на восток, нужны квадранты *d* и *a*.



а) Взаимоблокировка возможна

б) Взаимоблокировка произошла

**Рис. 6.1.** Пример взаимоблокировки

Обычное правило пересечения перекрестка состоит в том, что автомобиль должен уступить дорогу движущемуся справа. Это правило работает, когда перекресток пересекают два или три автомобиля. Например, если на перекрестке встретятся автомобили, движущиеся на север и на запад, то автомобиль, движущийся на север, уступит дорогу автомобилю, движущемуся на запад. Но если перекресток пересекают одновременно четыре автомобиля, каждый из которых согласно правилу воздержится от въезда на перекресток, возникнет взаимоблокировка.

Она возникнет и в том случае, если все четыре машины проигнорируют правило и осторожно въедут на перекресток, поскольку при этом каждый автомобиль захватит один ресурс (один квадрант) и останется на вечной стоянке в ожидании, когда другой автомобиль освободит следующий требующийся для пересечения перекрестка квадрант. Итак, мы опять получили взаимоблокировку.

Рассмотрим теперь картину взаимоблокировки с участием процессов и компьютерных ресурсов. На рис. 6.2 показана **диаграмма совместного выполнения** (joint progress diagram) двух процессов, конкурирующих в борьбе за два ресурса. Каждый из процессов

требует исключительного владения обоими ресурсами на некоторое время. Процессы P и Q имеют общий вид:

Процесс P	Процесс Q
...	...
Получение A	Получение B
...	...
Получение B	Получение A
...	...
Освобождение A	Освобождение B
...	...
Освобождение B	Освобождение A
...	...

На рис. 6.2 ось x представляет выполнение процесса P, а ось y — выполнение процесса Q. Таким образом, совместное выполнение двух процессов представлено путем, идущим из начала координат в северо-восточном направлении.

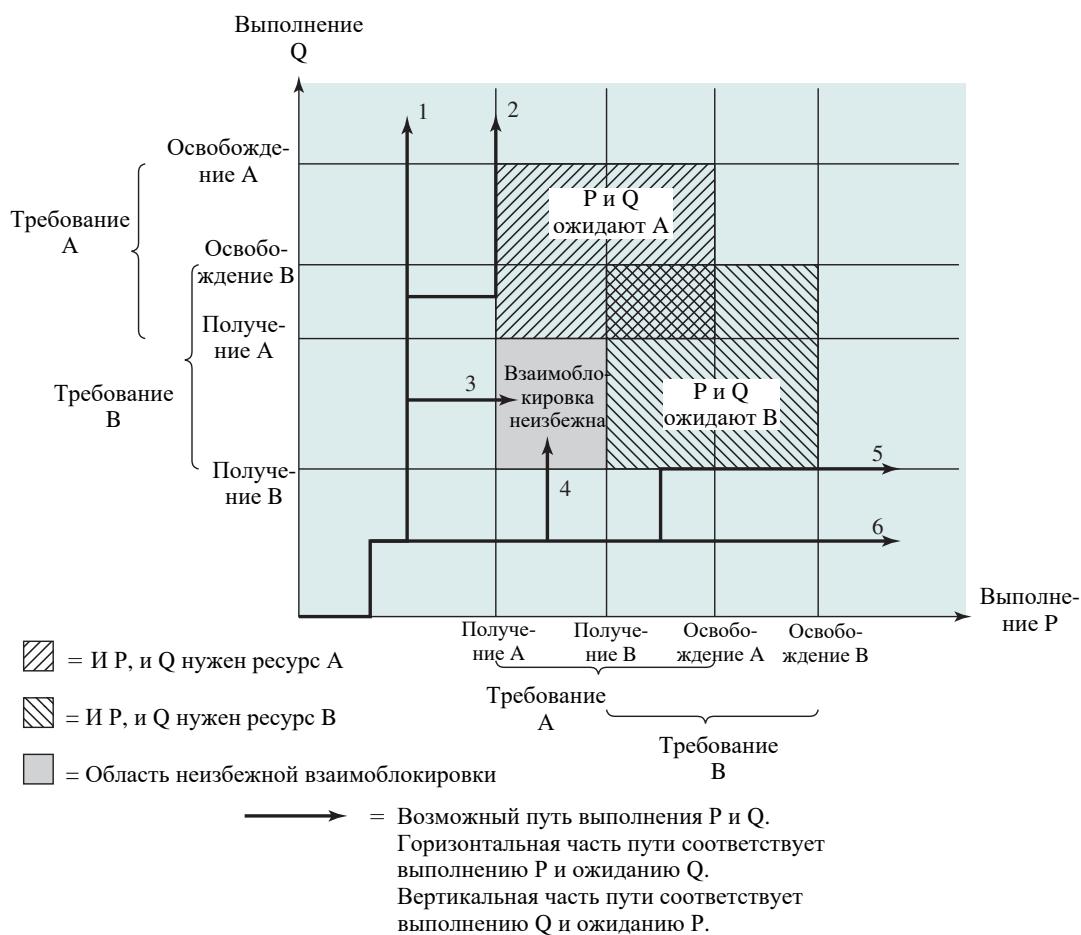


Рис. 6.2. Пример взаимоблокировки

В однопроцессорной системе в каждый момент времени может выполняться только один процесс, так что путь состоит из чередующихся горизонтальных и вертикальных отрезков, причем горизонтальные отрезки представляют работу процесса P, а вертикальные — процесса Q. На рисунке показаны области, в которых процессам P и Q требуется ресурс A, процессам P и Q — ресурс B, процессам P и Q — оба ресурса. Поскольку мы предполагаем, что каждый процесс требует исключительного управления любым ресурсом, все эти области — запрещенные; никакой путь, представляющий ход совместного выполнения P и Q, не может входить в эти области.

На рис. 6.2 показаны шесть различных путей выполнения процессов.

1. Q получает ресурс B, затем — ресурс A, затем освобождает ресурсы B и A. Когда процесс P продолжает выполнение, он может получить оба ресурса.
2. Q получает ресурс B, а затем — ресурс A. Процесс P начинает работу и блокируется при запросе ресурса A. Q освобождает ресурсы B и A. Когда процесс P продолжает выполнение, он может получить оба ресурса.
3. Q получает ресурс B, затем P получает ресурс A. Взаимоблокировка неизбежна, поскольку выполнение Q заблокировано при запросе ресурса A, а выполнение процесса P — при запросе ресурса B.
4. P получает ресурс A, затем Q получает ресурс B. Взаимоблокировка неизбежна, поскольку выполнение Q заблокировано при запросе ресурса A, а выполнение процесса P — при запросе ресурса B.
5. P получает ресурс A, а затем — ресурс B. Процесс Q начинает работу и блокируется при запросе ресурса B. P освобождает ресурсы A и B. Когда процесс Q продолжает выполнение, он может получить оба ресурса.
6. P получает ресурс A, затем — ресурс B, затем освобождает ресурсы A и B. Когда процесс Q продолжает выполнение, он может получить оба ресурса.

На рис. 6.2 имеется область, которую можно назвать фатальной, — если путь выполнения входит в эту область, взаимоблокировка неизбежна. Обратите внимание, что существование фатальной области зависит от логики этих двух процессов. Однако взаимоблокировка неизбежна только когда совместное выполнение этих двух процессов создает путь, который входит в фатальную область.

Произойдет взаимоблокировка или нет, зависит как от динамики выполнения процессов, так и от подробностей построения приложения. Предположим, например, что процесс P не требует получения обоих ресурсов одновременно и имеет следующий вид:

Процесс P	Процесс Q
...	...
Получение A	Получение B
...	...
Освобождение A	Получение A
...	...
Получение B	Освобождение B
...	...
Освобождение B	Освобождение A
...	...

Эта ситуация изображена на рис. 6.3. Немного поразмыслив, вы можете убедиться, что независимо от того, каким образом два процесса выполняются друг относительно друга, взаимоблокировка невозможна.

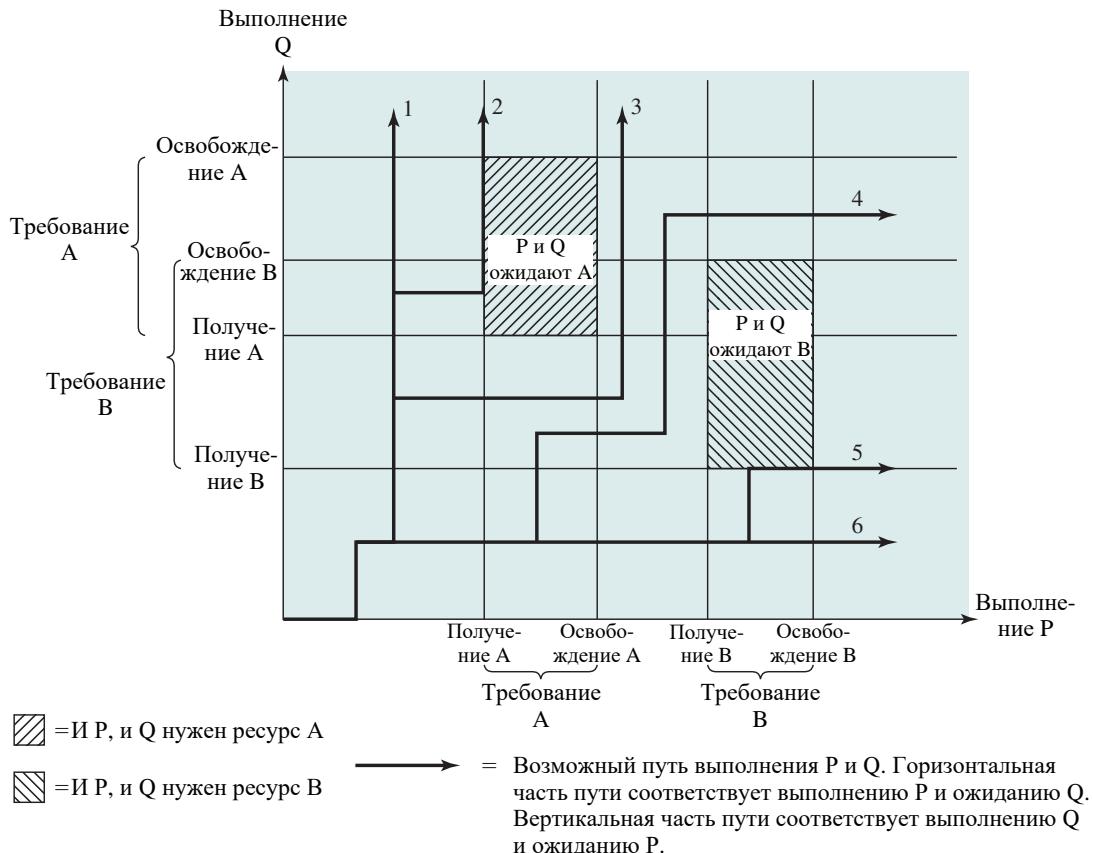


Рис. 6.3. Пример отсутствия взаимоблокировки [15]

Как видите, диаграмма совместного выполнения может использоваться для записи истории выполнения двух процессов, которые совместно используют ресурсы. В тех случаях, когда более двух процессов могут конкурировать за один и тот же ресурс, требуется диаграмма более высокой размерности. Принципы же, касающиеся фатальных областей и взаимоблокировок, остаются при этом теми же.

## Повторно используемые ресурсы

Ресурсы можно разделить на две основные категории: повторно используемые (reusable) и расходуемые (consumable). Повторно используемые ресурсы могут безопасно использоваться одновременно только одним процессом и при этом не истощаться. Процесс получает ресурс, который позже освобождается для повторного использования другими процессами. Примерами повторно используемых ресурсов могут служить процессор, каналы ввода-вывода, основная и вторичная память, периферийные устройства, а также структуры данных, такие как файлы, базы данных и семафоры.

В качестве примера взаимоблокировки с повторно используемым ресурсом рассмотрим два процесса, которые конкурируют за исключительный доступ к дисковому файлу D и примеру T. Программа выполняет показанные на рис. 6.4 операции.

<u>Процесс P</u>		<u>Процесс Q</u>	
Шаг	Действие	Шаг	Действие
p <sub>0</sub>	Запрос (D)	q <sub>0</sub>	Запрос (T)
p <sub>1</sub>	Блокировка (D)	q <sub>1</sub>	Блокировка (T)
p <sub>2</sub>	Запрос (T)	q <sub>2</sub>	Запрос (D)
p <sub>3</sub>	Блокировка (T)	q <sub>3</sub>	Блокировка (D)
p <sub>4</sub>	Выполнение функции	q <sub>4</sub>	Выполнение функции
p <sub>5</sub>	Деблокирование (D)	q <sub>5</sub>	Деблокирование (T)
p <sub>6</sub>	Деблокирование (T)	q <sub>6</sub>	Деблокирование (D)

**Рис. 6.4.** Пример конкуренции двух процессов в борьбе за повторно используемый ресурс

Взаимоблокировка осуществляется в том случае, когда каждый процесс захватывает один ресурс и запрашивает другой. Например, взаимоблокировка произойдет при следующем чередовании двух процессов:

P<sub>0</sub> P<sub>1</sub> Q<sub>0</sub> Q<sub>1</sub> P<sub>2</sub> Q<sub>2</sub>

Может показаться, что это ошибка программиста, не имеющая отношения к разработчику операционной системы. Однако, как мы знаем, разработка программ для параллельных вычислений — весьма сложная задача, и выявление источника взаимоблокировки в сложной программе — дело очень непростое. Одна из стратегий при работе с такими взаимоблокировками состоит в наложении системных ограничений на порядок запроса ресурсов.

Другим примером взаимоблокировки с повторно используемыми ресурсами могут быть запросы к основной памяти. Предположим, что для распределения доступно 200 Кбайт памяти и выполняется такая последовательность запросов.

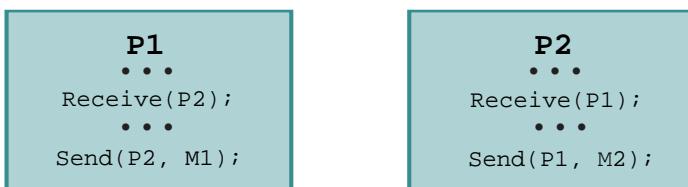
P1	P2
• • •	• • •
Запрос 80 Кбайт;	Запрос 70 Кбайт;
• • •	• • •
Запрос 60 Кбайт;	Запрос 80 Кбайт;

Если оба процесса дойдут до своего второго запроса, возникнет взаимоблокировка. Если количество требуемой памяти заранее неизвестно, работать с таким типом взаимоблокировок на уровне системных ограничений (в том числе и операционной системы) очень сложно. Наилучший способ справиться с этой конкретной проблемой заключается в использовании виртуальной памяти, о которой рассказывается в главе 8, “Виртуальная память”.

## Расходуемые ресурсы

Расходуемыми являются те ресурсы, которые могут быть созданы (произведены) и уничтожены (потреблены). Обычно ограничений на количество расходуемых ресурсов определенного типа нет. Незаблокированный процесс-производитель может выпустить любое количество таких ресурсов; когда процесс запрашивает некоторый ресурс, последний прекращает свое существование. Примерами расходуемых ресурсов могут служить прерывания, сигналы, сообщения и информация в буферах ввода-вывода.

В качестве примера взаимоблокировки с расходуемыми ресурсами рассмотрим следующую пару процессов, каждый из которых которой пытается получить сообщение от другого процесса, а затем отправить сообщение своему визави.



Взаимоблокировка осуществляется, если операция `Receive` является блокирующей (т.е. получающий процесс блокируется до тех пор, пока сообщение не будет получено). И вновь причиной взаимоблокировки является ошибка при разработке программы. Такие ошибки обычно довольно таинственны и трудноуловимы. Кроме того, может оказаться, что взаимоблокировку вызывает только достаточно редкая комбинация событий, и тогда до того, как ошибка проявит себя, программа может быть в эксплуатации многие годы.

Единой эффективной стратегии для работы со всеми типами взаимоблокировок нет. Основные подходы к решению этой проблемы следующие.

- **Предотвращение взаимоблокировок.** Не позволять осуществляться одному из трех необходимых условий возникновения взаимоблокировки или предотвратить возникновение условия циклического ожидания.
- **Устранение взаимоблокировок.** Не выполнять запрос ресурса, если его выделение может привести к взаимоблокировке.
- **Обнаружение взаимоблокировок.** По возможности выполнять запросы ресурсов, но периодически проверять наличие взаимоблокировки и принимать меры для восстановления.

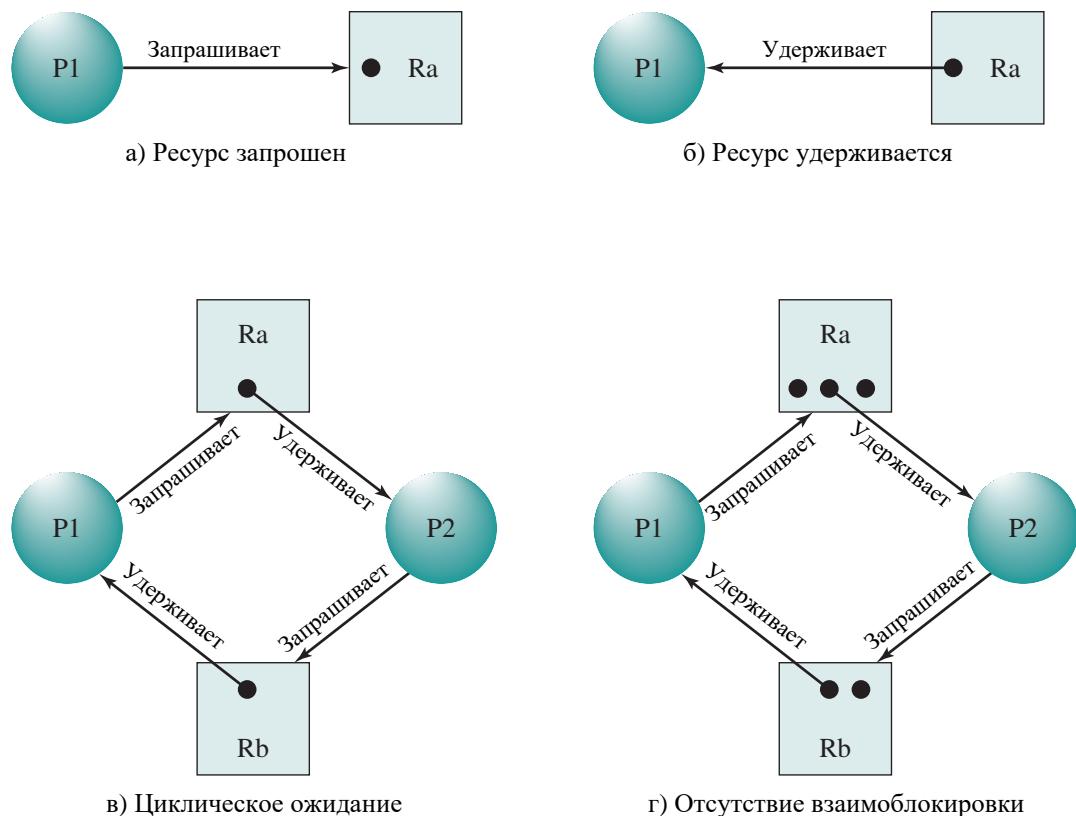
Мы изучим все перечисленные подходы к решению проблемы, но сначала рассмотрим условия возникновения взаимоблокировок.

## Графы распределения ресурсов

Полезным инструментом для характеристики распределения ресурсов между процессами является **граф распределения ресурсов**, предложенный Холтом (Holt) [109]. Граф распределения ресурсов представляет собой ориентированный граф, который изображает состояние системы ресурсов и процессов, в котором каждый процесс и каждый ресурс представлены узлами. Дуги графа, направленные от процесса к ресурсу, указывают ресурс, запрошенный процессом, но еще не предоставленный ему (рис. 6.5, а).

В узле ресурса каждый экземпляр ресурса указан точкой. Примерами типов ресурсов, которые могут иметь несколько экземпляров, являются устройства ввода-вывода, которые выделяются модулем управления ресурсами в операционной системе. Дуга графа, направленная из узла повторно используемого ресурса к процессу, указывает запрос, который был выполнен (рис. 6.5, б), т.е. этому процессу была назначена одна единица ресурса. Дуга графа, направленная из узла расходного ресурса к процессу, указывает, что процесс является производителем данного ресурса.

На рис. 6.5, в показан пример взаимоблокировки. Имеется только по одной единице ресурсов Ra и Rb. Процесс P1 удерживает Rb и запрашивает Ra, а P2 удерживает Ra и запрашивает Rb. Граф на рис. 6.5, г имеет ту же топологию, что и на рис. 6.5, в, но здесь нет взаимоблокировки, потому что доступно несколько единиц каждого ресурса.



**Рис. 6.5.** Примеры графов распределения ресурсов

Граф распределения ресурсов на рис. 6.6 соответствует взаимоблокировке на рис. 6.1, б. Обратите внимание, что в этом случае у нас нет простой ситуации, в которой каждый из двух процессов удерживает ресурс, необходимый другому процессу. В этом случае скорее имеется круговая цепь процессов и ресурсов, приводящая к взаимоблокировке.

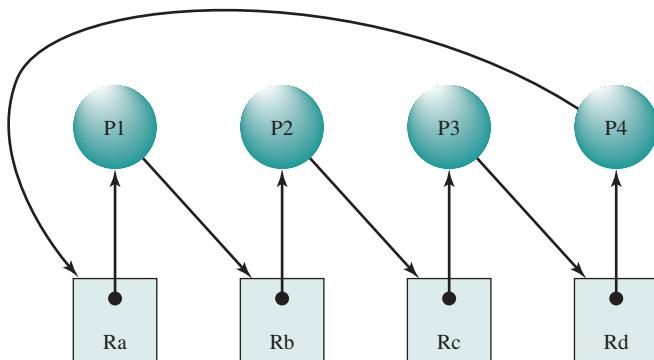


Рис. 6.6. Граф распределения ресурсов для рис. 6.1, б

## УСЛОВИЯ ВОЗНИКНОВЕНИЯ ВЗАИМОБЛОКИРОВОК

Для того чтобы взаимоблокировка стала возможной, требуется три условия.

- Взаимные исключения.** Одновременно использовать ресурс может только один процесс.
- Удержание и ожидание.** Процесс может удерживать выделенные ресурсы во время ожидания других ресурсов.
- Отсутствие перераспределения.** Ресурс не может быть принудительно отобран у удерживающего его процесса.

Эти условия выполняются довольно часто. Например, взаимоисключения необходимы для гарантии согласованности результатов и целостности базы данных. Аналогично невозможно произвольное применение перераспределения, в особенности при работе с данными, когда требуется обеспечить механизм отката.

Кроме перечисленных трех условий, необходимых, но не достаточных, для реального осуществления взаимоблокировки, требуется выполнение четвертого условия.

- Циклическое ожидание.** Существует замкнутая цепь процессов, каждый из которых удерживает как минимум один ресурс, необходимый процессу, следующему в цепи после данного (см. рис. 6.5, в и 6.6).

Четвертое условие в действительности представляет собой потенциальное следствие первых трех, т.е. при наличии первых трех условий может осуществиться такая последовательность событий, которая приведет к неразрешимому циклическому ожиданию (что, по сути, и является определением взаимоблокировки). Неразрешимость циклического ожидания из условия 4 обеспечивается выполнением предыдущих трех условий. Таким образом, совокупность четырех перечисленных выше условий является необходимым и достаточным условием взаимоблокировки.<sup>2</sup>

<sup>2</sup> Обычно в литературе все четыре условия перечисляются как необходимые для осуществления взаимоблокировки, однако такое изложение скрывает некоторые тонкости данного вопроса. Условие циклического ожидания кардинально отличается от остальных трех условий. Первые три условия представляют собой, по сути, незыблемые правила, в то время как условие 4 представляет собой ситуацию, которая может осуществляться при определенной последовательности запросов и освобождений ресурсов процессом. Объединение всех четырех условий в единый блок методологически приводит к стиранию различий между предотвращением и устранением взаимоблокировок. Подробное обсуждение этих вопросов приведено в [229] и [230].

Чтобы понять это, полезно вернуться к концепции диаграммы совместного выполнения, показанной на рис. 6.2. Напомним, что мы определили фатальную область как область, обладающую тем свойством, что вошедшие в нее процессы будут обязательно взаимно блокированы. Фатальная область существует, только если соблюдены все три первые перечисленные выше условия. Если одно или несколько из этих условий не выполняются, фатальной области не существует, и взаимоблокировка невозможна. Таким образом, эти условия являются необходимыми для существования взаимоблокировки. Но чтобы она произошла реально, должна иметься не только фатальная область, но и такая последовательность запросов ресурсов, которая приведет в эту область. Если возникает условие циклического ожидания, значит, произошел вход в фатальную область. Таким образом, всех четырех перечисленных выше условий достаточно для взаимоблокировки.

#### Взаимоблокировка возможна

1. Взаимное исключение
2. Нет перераспределения
3. Удержание и ожидание

#### Взаимоблокировка имеет место

1. Взаимное исключение
2. Нет перераспределения
3. Удержание и ожидание
4. Циклическое ожидание

Для решения проблем взаимоблокировки имеется три основных подхода. Во-первых, взаимоблокировки можно **предотвратить** путем принятия стратегии, которая устраниет одно из условий (условий 1–4). Во-вторых, можно **устранить** взаимоблокировку, сделав соответствующие динамические выборы на основе текущего состояния распределения ресурсов. В-третьих, можно попытаться **обнаружить** взаимоблокировку (условия 1–4) и принять меры для восстановления работоспособности. Мы обсудим каждый из этих подходов ниже.

## 6.2. ПРЕДОТВРАЩЕНИЕ ВЗАИМОБЛОКИРОВОК

Стратегия предотвращения, по сути, представляет собой такую разработку системы, которая позволит исключить саму возможность взаимоблокировок. Методы предотвращения взаимоблокировок можно разбить на два класса. Косвенный метод состоит в предотвращении одного из первых трех условий возникновения взаимоблокировки; прямой метод предотвращает циклическое ожидание (условие 4). Рассмотрим приемы, связанные с каждым из условий, в отдельности.

### Взаимоисключений

В общем случае избежать использования взаимоисключений невозможно. Если доступ к ресурсу должен быть исключительным, то операционная система обязана поддерживать взаимоисключения. Некоторые ресурсы, такие как файлы, могут позволять множественный доступ для чтения и исключительный доступ для записи. Но даже в этом случае возможно возникновение взаимоблокировки, если право записи в файл требуется нескольким процессам одновременно.

## Удержание и ожидание

Этого условия можно избежать, потребовав, чтобы процесс запрашивал все необходимые ресурсы одновременно, и блокировать процесс до тех пор, пока такой запрос не сможет быть выполнен полностью в один и тот же момент времени. Такой подход неэффективен по двум причинам. Во-первых, процесс может длительное время ожидать одновременной доступности всех затребованных ресурсов, в то время как реально он мог бы работать и только с частью из них. Во-вторых, затребованные процессом ресурсы могут оставаться неиспользуемыми значительное время, в течение которого они оказываются недоступными другим процессам. Еще одна проблема состоит в том, что процессу может не быть известно заранее, какие именно ресурсы ему потребуются.

Имеется также практическая проблема, возникающая при использовании парадигмы модульности или многопоточности в программировании приложения. Использующее описанную технологию приложение для одновременного запроса должно знать обо всех необходимых ресурсах на всех уровнях или во всех модулях, что противоречит упомянутой парадигме.

## Отсутствие перераспределения

Этого условия можно избежать несколькими путями. Например, можно поступить следующим образом: если процесс удерживает некоторые ресурсы и ему отказано в очередном запросе, то он должен освободить захваченные ресурсы и при необходимости запросить их вновь вместе с тем ресурсом, в доступе к которому ему было отказано. С другой стороны, если процесс затребовал некий ресурс, в настоящий момент захваченный другим процессом, то операционная система может вытеснить этот процесс и потребовать от него освободить захваченные им ресурсы. Этот метод может предотвратить взаимоблокировку лишь в том случае, когда все процессы имеют разные приоритеты.

Такой подход на практике применим только к тем ресурсам, состояние которых можно легко сохранить, а позже восстановить, как, например, в случае, когда ресурс представляет собой процессор.

## Циклическое ожидание

Условия циклического ожидания можно избежать путем упорядочения типов ресурсов. При этом, если процесс запросил ресурс типа  $R_i$ , далее он может запросить только ресурсы, следующие согласно указанному упорядочению за  $R_i$ .

Чтобы убедиться в эффективности данной стратегии, свяжем с каждым типом ресурса свой индекс. Тогда ресурс  $R_i$  предшествует ресурсу  $R_j$ , если  $i < j$ . Теперь предположим, что два процесса, А и В, взаимно заблокированы, поскольку процесс А захватил ресурс  $R_i$  и запрашивает ресурс  $R_j$ , а процесс В захватил ресурс  $R_j$  и запрашивает ресурс  $R_i$ . Однако такая ситуация невозможна, в силу того что из нее следует одновременное выполнение условий  $i < j$  и  $j < i$ .

Как и в случае предотвращения удержания и ожидания, технология предотвращения циклического ожидания может оказаться неэффективной, снижающей скорость работы процесса и закрывающей доступ к ресурсам без особой на то необходимости.

## 6.3. УСТРАНЕНИЕ ВЗАЙМОБЛОКИРОВОК

Другим подходом к решению проблемы взаимоблокировок является устранение взаимоблокировок.<sup>3</sup> В случае **предотвращения взаимоблокировок** мы накладывали определенные ограничения на запросы к ресурсам, с тем чтобы сделать невозможным осуществление по крайней мере одного из необходимых условий существования взаимоблокировок и тем самым предотвратить саму возможность их возникновения. К сожалению, этот метод приводит к неэффективному использованию ресурсов и снижению скорости работы процесса. **Устранение взаимоблокировок** допускает наличие трех необходимых условий возникновения взаимоблокировок, но мы принимаем меры к тому, чтобы ситуация взаимного блокирования процессов не могла быть достигнута. Соответственно, устранение взаимоблокировок обеспечивает большую параллельность вычислений, чем предотвращение. Решение о том, способен ли текущий запрос ресурса в случае его удовлетворения привести к возникновению взаимоблокировки, принимается в этом случае динамически (и следовательно, при использовании данной технологии необходимо знать о том, какие ресурсы потребуются процессу в дальнейшем).

В этом разделе мы познакомимся с двумя подходами к устранению взаимоблокировок.

1. Не запускать процесс, если его запросы могут привести к взаимоблокировке.
2. Не удовлетворять запросы процесса, если их выполнение способно привести к взаимоблокировке.

### Запрещение запуска процесса

Рассмотрим систему из  $n$  процессов и  $m$  различных типов ресурсов. Определим следующие векторы и матрицы.

Ресурс: $\mathbf{R} = (R_1, R_2, \dots, R_m)$	Общее количество каждого ресурса в системе
Доступность: $\mathbf{V} = (V_1, V_2, \dots, V_m)$	Общее количество каждого ресурса, не выделенного процессам
Требования: $\mathbf{C} = \begin{pmatrix} C_{11} & C_{12} & \cdots & C_{1m} \\ C_{21} & C_{22} & \cdots & C_{2m} \\ \vdots & & & \\ C_{n1} & C_{n2} & \cdots & C_{nm} \end{pmatrix}$	$C_{ij}$ — запрос процессом $i$ ресурса $j$
Распределение: $\mathbf{A} = \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \vdots & & & \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{pmatrix}$	$A_{ij}$ — текущее распределение процессу $i$ ресурса $j$

<sup>3</sup> Термин “устранение” (avoidance) несколько запутывает ситуацию. В некотором смысле рассматриваемые в данном разделе стратегии можно считать примерами предотвращения взаимоблокировок, так как их использование в действительности предотвращает появление последних.

Матрица требований, в которой каждая строка описывает один из процессов, указывает максимальные требования каждого процесса к разным ресурсам, т.е.  $C_{ij}$  — это требования процессом  $i$  ресурса  $j$ . Для работоспособности метода устранения взаимоблокировок эта информация должна быть объявлена процессом заранее. Аналогично  $A_{ij}$  — текущее количество ресурса  $j$ , выделенное процессу  $i$ .

Должны выполняться следующие соотношения.

1.  $R_j = V_j + \sum_{i=1}^n A_{ij}$  для всех  $j$ : все ресурсы либо свободны, либо выделены.
2.  $C_{ij} \leq R_j$  для всех  $i$  и  $j$ : ни один процесс не может потребовать ресурса, превышающий его общее количество в системе.
3.  $A_{ij} \leq C_{ij}$  для всех  $i$  и  $j$ : ни один процесс не может получить больше ресурсов, чем было им затребовано.

Когда все указанные величины определены, мы в состоянии создать стратегию устранения взаимоблокировок, которая запрещает запуск нового процесса, если его требования ресурсов могут привести к взаимоблокировке. Новый процесс  $P_{n+1}$  запускается, только если

$$R_j \geq C_{(n+1)j} + \sum_{i=1}^n C_{ij} \text{ для всех } j.$$

Это означает, что запуск нового процесса произойдет только в том случае, если могут быть удовлетворены максимальные требования всех текущих процессов плюс требования запускаемого процесса. Эта стратегия ни в коей мере не является оптимальной, поскольку предполагает худшее: что все процессы предъявляют максимальные требования одновременно.

## Запрет выделения ресурса

Стратегия запрета выделения ресурса, известная также как **алгоритм банкира**,<sup>4</sup> впервые была предложена в работе [66]. Мы начнем рассмотрение с концепций состояния (state) и безопасного состояния (safe state). Остановимся на системе с фиксированным количеством процессов и фиксированным количеством ресурсов. В каждый момент времени процесс может иметь несколько выделенных ему ресурсов (или не иметь ни одного). Состояние системы представляет собой просто текущее распределение ресурсов по процессам. Следовательно, состояние можно представить как два ранее определенных вектора (ресурсов и доступности) и две матрицы (требований и распределения).

---

<sup>4</sup> Дейкстра воспользовался таким названием, сравнивая эту задачу с деятельностью банка, в котором клиент, желающий осуществить заем денег, отождествляется с процессом, а деньги — с ресурсом. Банк имеет ограниченное количество денег для займов и список клиентов, у каждого из которых имеется своя предельная величина кредита. Клиент может неоднократно осуществлять заем в пределах этой величины, причем нет никакой гарантии, что он погасит хотя бы часть задолженности до того момента, пока не займет всю возможную сумму. Таким образом, если несколько клиентов берут частичные займы, то гарантировать возврат этих денег банку может только некоторый резервный фонд. В худшем случае, когда все клиенты погашают займы только по достижении ими предельной суммы кредита, этот фонд должен обеспечить возможность выдать им эти максимальные суммы. Следовательно, при исчерпании денежных запасов до размеров резервного фонда банк будет вынужден отказать очередному клиенту в предоставлении займа, иначе деньги в банк могут никогда не вернуться.

**Безопасное состояние** — это такое состояние, в котором имеется по крайней мере одна последовательность, которая не приводит к взаимоблокировке (т.е. все процессы могут быть выполнены до завершения). Состояние, не являющееся безопасным, называется, соответственно, **опасным состоянием**.

Описанную концепцию иллюстрирует приведенный далее пример. На рис. 6.7, а показано состояние системы, образованной четырьмя процессами и тремя ресурсами. Общее количество ресурсов R1, R2 и R3 составляет соответственно 9, 3 и 6 единиц. В результате сделанного к этому моменту распределения ресурсов по процессам доступными остались по одной единице ресурсов R2 и R3. Безопасно ли данное состояние? Для ответа на этот вопрос зададимся другим вопросом, а именно — может ли какой-нибудь из четырех процессов быть выполнен при данных доступных ресурсах до завершения? Или, говоря иначе, могут ли при данном распределении ресурсов быть удовлетворены максимальные требования какого-то из процессов за счет оставшихся ресурсов? В терминах введенных ранее матриц и векторов для процесса  $i$  должно выполняться условие

$$C_{ij} - A_{ij} \leq V_j \text{ для всех } j.$$

Ясно, что для процесса P1 это невозможно, так как у него имеется только одна единица ресурса R1 и для полного удовлетворения ему требуется еще по две единицы ресурсов R1, R2 и R3. Однако если выделить процессу P2 одну единицу ресурса R3, то будут удовлетворены максимальные требования этого процесса и он сможет быть завершен. Когда процесс P2 оказывается завершенным, распределенные ему ресурсы могут быть возвращены в пул доступных. Это состояние системы показано на рис. 6.7, б. Вернувшись к вопросу о том, какой из процессов может быть завершен в данной ситуации, мы находим, что могут быть удовлетворены максимальные требования как процесса P1, так и процесса P3. Предположим, что мы выбрали процесс P1. По его завершении и возвращении захваченных им ресурсов в пул доступных ситуация будет выглядеть так, как показано на рис. 6.7, в. И наконец, по завершении процесса P3 мы придем к состоянию, изображенному на рис. 6.7, г. Наконец, мы можем завершить процесс P4. В этот момент все процессы завершены, и следовательно, исходное состояние (рис. 6.7, а) является безопасным.

Описанная концепция автоматически приводит к стратегии устранения взаимоблокировок, которая гарантирует, что система процессов и ресурсов всегда находится в безопасном состоянии. Когда процесс делает запрос к некоторому множеству ресурсов, предполагается, что запрос удовлетворен, после чего определяется, является ли обновленное состояние системы безопасным. Если это так, то запрос удовлетворяется; в противном случае процесс блокируется до тех пор, пока удовлетворение его запроса не станет безопасным.

Рассмотрим состояние, определяемое матрицей на рис. 6.8, а. Предположим, что процесс P2 делает запрос на одну дополнительную единицу ресурса R1 и одну — ресурса R3. Если предположить, что запрос будет выполнен, то в результате состояние системы станет таким, как показано на рис. 6.7, а. Мы уже видели, что это состояние является безопасным. Таким образом, запрос можно удовлетворить. Теперь вернемся к состоянию, представленному на рис. 6.8, а, и предположим, что такой же запрос на одну дополнительную единицу ресурса R1 и одну — ресурса R3 делает процесс P1. Если этот запрос будет удовлетворен, состояние системы станет таким, как показано на рис. 6.8, б. Является ли это состояние безопасным? В данном случае ответ — “нет”, поскольку каждому из процессов требуется по одной дополнительной единице ресурса R1, а в наличии свободных единиц этого ресурса уже нет. Следовательно, запрос процесса P1 должен быть отклонен, а сам процесс — блокирован.

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Матрица требований С

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Матрица распределения А

	R1	R2	R3
P1	2	2	2
P2	0	0	1
P3	1	0	3
P4	4	2	0

C – A

	R1	R2	R3
	9	3	6

Вектор ресурсов R

	R1	R2	R3
	0	1	1

Вектор доступности V

а) Исходное состояние

	R1	R2	R3
P1	3	2	2
P2	0	0	0
P3	3	1	4
P4	4	2	2

Матрица требований С

	R1	R2	R3
P1	1	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Матрица распределения А

	R1	R2	R3
P1	2	2	2
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

	R1	R2	R3
	9	3	6

Вектор ресурсов R

	R1	R2	R3
	6	2	3

Вектор доступности V

б) Процесс P2 выполнен до завершения

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	3	1	4
P4	4	2	2

Матрица требований С

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Матрица распределения А

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

	R1	R2	R3
	9	3	6

Вектор ресурсов R

	R1	R2	R3
	7	2	3

Вектор доступности V

в) Процесс P1 выполнен до завершения

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	2

Матрица требований С

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	2

Матрица распределения А

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	0

C – A

	R1	R2	R3
	9	3	6

Вектор ресурсов R

	R1	R2	R3
	9	3	4

Вектор доступности V

г) Процесс P3 выполнен до завершения

Рис. 6.7. Определение безопасного состояния

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Матрица требований С

	R1	R2	R3
P1	1	0	0
P2	5	1	1
P3	2	1	1
P4	0	0	2

Матрица распределения А

	R1	R2	R3
P1	2	2	2
P2	1	0	2
P3	1	0	3
P4	4	2	0

C – A

	R1	R2	R3
	9	3	6

Вектор ресурсов R

	R1	R2	R3
	1	1	2

Вектор доступности V

a) Исходное состояние

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Матрица требований С

	R1	R2	R3
P1	2	0	1
P2	5	1	1
P3	2	1	1
P4	0	0	2

Матрица распределения А

	R1	R2	R3
P1	1	2	1
P2	1	0	2
P3	1	0	3
P4	4	2	0

C – A

	R1	R2	R3
	9	3	6

Вектор ресурсов R

	R1	R2	R3
	0	1	1

Вектор доступности V

б) Процесс P1 запрашивает по одной единице ресурсов R1 и R3

Рис. 6.8. Определение опасного состояния

Важно отметить, что состояние, представленное на рис. 6.8, б, не является взаимоблокированной. Оно всего лишь может привести к ней. Например, при работе процесс P1 может освободить по одной единице ресурсов R1 и R3, и система вновь вернется в безопасное состояние. Следовательно, стратегия устранения взаимоблокировок не занимается точным предсказанием взаимоблокировок; она лишь предвидит их возможность и устраняет ее.

На рис. 6.9 приведена логика работы абстрактного алгоритма устранения взаимоблокировок. Основной алгоритм показан в части б. Состояние системы описывается структурой `state`, а вектор `request[*]` определяет ресурсы, затребованные процессом `i`. Сначала выполняется проверка того, что запрос не превышает исходные требования процесса. Если запрос корректен, то следующий шаг алгоритма состоит в определении возможности удовлетворения запроса (т.е. в выяснении, достаточно ли для этого свободных ресурсов). Если нет, процесс приостанавливается; если же ресурсов достаточно, выполняется последняя проверка — безопасно ли выполнение запроса. Для этого процессу гипотетически выделяются требуемые ресурсы, в результате чего получается новое состояние системы `newstate`, которое и проверяется на безопасность с использованием алгоритма из рис. 6.9, в.

Устранение взаимоблокировок обладает тем преимуществом, что при его использовании не нужны ни перераспределение, ни откат процессов, как в случае обнаружения взаимоблокировок; кроме того, этот метод накладывает меньше ограничений по сравнению с предотвращением взаимоблокировок.

```
struct state {
    int resource[m];
    int available[m];
    int claim[n][m];
    int alloc[n][m];
}
```

## а) Глобальная структура данных

```
if (alloc [i,*] + request [*] > claim [i,*])
    <Ошибка>; /* Суммарный запрос превышает требования */
else if (request [*] > available [*])
    <Приостановка процесса>;
else { /* Моделируем выполнение запроса */
    <Определение нового состояния>
    alloc [i,*] = alloc [i,*] + request [*];
    available [*] = available [*] - request [*];
}
if (safe (newstate))
    <Выполнение распределения>;
else {
    <Восстановление исходного состояния>;
    <Приостановка процесса>;
}
```

## б) Алгоритм выделения ресурсов

```
boolean safe (state S) {
    int currentavail[m];
    process rest[<Количество процессов>];
    currentavail = available;
    rest = {Все процессы};
    possible = true;
    while (possible) {
        <Найти процесс Pk в rest такой, что
        claim [k,*] - alloc [k,*] <= currentavail;
        if (found) { /* Моделирование выполнения Pk */
            currentavail = currentavail + alloc [k,*];
            rest = rest - {Pk};
        }
        else possible = false;
    }
    return (rest == null);
}
```

## в) Проверка безопасности алгоритма (алгоритм банкира)

**Рис. 6.9.** Алгоритм устранения взаимоблокировок

Однако использование этого метода требует выполнения определенных условий.

- Должны быть заранее указаны максимальные требования каждого процесса к ресурсам.
- Рассматриваемые процессы должны быть независимы, т.е. порядок их выполнения не должен ограничиваться никакими требованиями синхронизации.
- Должно иметься фиксированное количество распределяемых ресурсов.
- Ни один процесс не должен завершаться в состоянии захвата ресурсов.

## 6.4. ОБНАРУЖЕНИЕ ВЗАИМОБЛОКИРОВОК

Стратегии предотвращения взаимоблокировок весьма консервативны; они решают проблему взаимоблокировок путем ограничения доступа процессов к ресурсам и наложения ограничений на процессы. Их противоположность — стратегии обнаружения взаимоблокировок, которые не ограничивают доступ к ресурсам и не налагают никаких ограничений на действия процессов. При обнаружении взаимоблокировок запрошенные ресурсы выделяются процессам при первой возможности. Периодически операционная система выполняет алгоритм, который позволяет обнаружить условия циклического ожидания (см. рис. 6.6).

### Алгоритм обнаружения взаимоблокировки

Проверка наличия взаимоблокировки может выполняться как при каждом запросе ресурса, так и менее часто, в зависимости от того, насколько вероятно возникновение взаимоблокировки. С одной стороны, проверка при каждом запросе ресурса имеет два основных преимущества: раннее обнаружение и упрощение алгоритма, поскольку он основан на инкрементных изменениях состояния системы. С другой стороны, столь частая проверка приводит к заметному потреблению времени процессора.

Обобщенный алгоритм обнаружения взаимоблокировок описан в [48]. В нем используются матрица распределения и вектор доступности, описанные в предыдущем разделе. Кроме того, определена матрица запросов  $\mathbf{Q}$ , такая, что  $Q_{ij}$  представляет собой количество ресурсов типа  $j$ , затребованное процессом  $i$ . Алгоритм работает, помечая незаблокированные процессы. Изначально все процессы не помечены. После этого выполняются следующие шаги.

1. Помечаем все процессы, строки в матрице распределения которых состоят из одних нулей.
2. Временный вектор  $\mathbf{W}$  инициализируем значениями вектора доступности.
3. Находим индекс  $i$ , такой, что процесс  $i$  в настоящий момент не помечен и  $i$ -я строка матрицы  $\mathbf{Q}$  не превышает  $\mathbf{W}$ , т.е. для всех  $1 \leq k \leq m$  выполняется  $Q_{ik} \leq W_k$ . Если такой строки нет, алгоритм прекращает свою работу.
4. Если такая строка имеется, помечаем процесс  $i$  и добавляем соответствующую строку матрицы распределения к  $\mathbf{W}$ , т.е. выполняем присвоение  $W_k = W_k + A_{ik}$  для всех  $1 \leq k \leq m$ . Возвращаемся к шагу 3.

Взаимоблокировка имеется тогда и только тогда, когда после выполнения алгоритма есть непомеченные процессы. Множество непомеченных процессов в точности соот-

вествует множеству заблокированных процессов. Стратегия этого алгоритма состоит в поиске процесса, запросы которого могут быть удовлетворены доступными ресурсами, а затем предполагается, что эти ресурсы ему выделены и процесс, завершив свою работу, освобождает их. После этого алгоритм приступает к поиску другого процесса, который может успешно завершить свою работу. Заметим, что данный алгоритм не гарантирует предотвращения взаимоблокировок — это зависит от порядка удовлетворения запросов процессов. Все, что делает данный алгоритм, — это определяет, имеется ли взаимоблокировка в настоящий момент.

Для иллюстрации алгоритма обнаружения взаимоблокировок рассмотрим рис. 6.10. Алгоритм работает следующим образом.

1. Помечаем P4, поскольку этот процесс не имеет распределенных ему ресурсов.
2. Устанавливаем  $\mathbf{W} = (0 \ 0 \ 0 \ 1)$ .
3. Запрос процесса P3 не превышает  $\mathbf{W}$ , так что помечаем P3 и устанавливаем  $\mathbf{W} = \mathbf{W} + (0 \ 0 \ 0 \ 1 \ 0) = (0 \ 0 \ 0 \ 1 \ 1)$ .
4. Других непомеченных процессов, строки матрицы  $\mathbf{Q}$  которых не превышают  $\mathbf{W}$ , нет. Алгоритм прекращает свою работу.

Таким образом, алгоритм позволяет заключить, что процессы P1 и P2 взаимно заблокированы.

	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	0	0	0	0	1
P4	1	0	1	0	1

Матрица запросов Q

	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

Матрица распределения A

	R1	R2	R3	R4	R5
	2	1	1	2	1

Вектор ресурсов

	R1	R2	R3	R4	R5
	0	0	0	0	1

Вектор доступности

Рис. 6.10. Пример обнаружения взаимоблокировки

## Восстановление

После того как взаимоблокировка обнаружена, требуется некоторая стратегия для восстановления нормальной работоспособности системы. Вот несколько возможных подходов к решению этой проблемы, перечисленные в порядке возрастания сложности.

1. Прекратить выполнение всех заблокированных процессов. Хотите — верьте, хотите — нет, но это один из самых распространенных, если не самый распространенный, подходов, принятых в операционных системах.
2. Вернуть каждый из заблокированных процессов в некоторую ранее определенную точку и перезапустить все процессы. Для этого в систему должны быть встроены механизмы отката и перезапуска. Самый большой риск при таком подходе заключается в том, что взаимоблокировка может проявиться вновь. Однако неопределенность относительных скоростей выполнения параллельных вычислений обычно позволяет этого избежать.

3. Последовательно прекращать выполнение заблокированных процессов по одному до тех пор, пока взаимоблокировка не прекратится. Порядок выбора уничтожаемых процессов должен базироваться на некотором критерии минимальной стоимости. После каждого уничтожения процесса должен быть вызван алгоритм обнаружения взаимоблокировок для проверки, не устраниены ли они.
4. Последовательно перераспределять ресурсы до тех пор, пока взаимоблокировка не прекратится. Как и в предыдущем случае, выбор процесса должен осуществляться в соответствии с некоторым критерием минимальной стоимости, а после осуществления перераспределения должен вызываться алгоритм обнаружения взаимоблокировок. Процесс, ресурсы которого перераспределяются, должен быть возвращен к состоянию, в котором он находился до получения этого ресурса.

В случае использования вариантов 3 и 4 критерий выбора процесса может быть, например, одним из следующих:

- процесс, потребляющий минимальное время процессора;
- процесс с минимальным выводом информации;
- процесс с наибольшим временем ожидания;
- процесс с минимальным количеством захваченных ресурсов;
- процесс с минимальным приоритетом.

При выборе критерия следует учитывать затраты времени на вычисление той или иной стоимости, а также отдавать себе отчет в том, что в данной ситуации понятие “стоимости” имеет смысл только для операционной системы в целом.

## 6.5. ИНТЕГРИРОВАННЫЕ СТРАТЕГИИ РАЗРЕШЕНИЯ ВЗАИМОБЛОКИРОВОК

У каждой стратегии разрешения взаимоблокировок есть свои преимущества и недостатки, а потому наиболее эффективным путем может оказаться применение разных подходов в различных ситуациях. В работе [110] предлагается следующий подход к данной проблеме.

- Сгруппировать ресурсы в несколько различных классов.
- Для предотвращения циклического ожидания во избежание взаимоблокировок между классами ресурсов использовать описанный ранее метод линейного упорядочения типов ресурсов.
- В пределах одного класса ресурсов использовать наиболее подходящий для данного типа ресурсов алгоритм.

В качестве примера такой методики рассмотрим следующие классы ресурсов.

- **Пространство подкачки.** Блоки памяти на вторичных устройствах хранения информации, используемые при свопинге процессов.
- **Ресурсы процесса.** Назначаемые устройства, такие как стримеры или файлы.
- **Основная память.** Страницы или сегменты, назначаемые процессу.
- **Внутренние ресурсы.** Такие ресурсы, как, например, каналы ввода-вывода.

Порядок перечисления ресурсов в приведенном списке представляет собой порядок их выделения. Этот порядок обосновывается обычной последовательностью действий процесса. В пределах каждого класса ресурсов могут использоваться следующие стратегии.

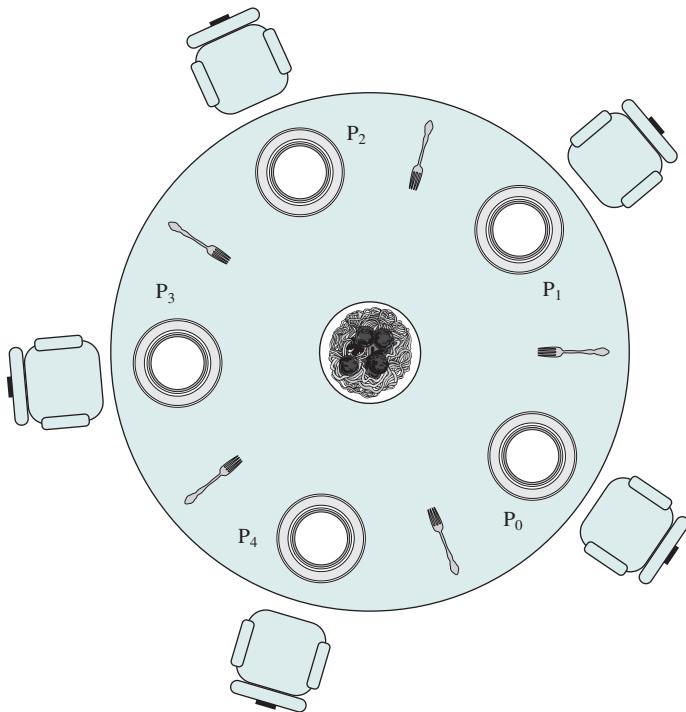
- **Пространство подкачки.** Предотвращение взаимоблокировок с помощью требования, чтобы все ресурсы распределялись одновременно. Такая стратегия вполне применима, если известны максимальные требования (что зачастую выполняется на практике). Можно также использовать стратегию устранения взаимоблокировок.
- **Ресурсы процесса.** В этой категории ресурсов зачастую наиболее эффективным является использование стратегии устранения взаимоблокировок, поскольку можно ожидать, что процесс заранее объявит о требуемых ему ресурсах этого типа. Кроме того, можно использовать предотвращение взаимоблокировок путем упорядочения ресурсов в пределах данного класса.
- **Основная память.** Пожалуй, наиболее подходящим методом предотвращения взаимоблокировок в этом случае может оказаться перераспределение ресурсов. Процесс, ресурсы которого перераспределяются, просто сбрасывается на вторичные устройства хранения информации, освобождая основную память для разрешения взаимоблокировки.
- **Внутренние ресурсы.** Можно использовать предотвращение взаимоблокировок путем упорядочения ресурсов в пределах данного класса.

## 6.6. ЗАДАЧА ОБ ОБЕДАЮЩИХ ФИЛОСОФАХ

Теперь рассмотрим задачу об обедающих философах, представленную Дейкстрой в [65]. Итак, в некотором царстве, в некотором государстве жили вместе пять философов. Жизнь каждого из них проходила в основном в размышлениях, прерываемых приемом пищи. Философы давно сошлись во мнении, что только спагетти в состоянии восстанавливать их подточенные непрерывными размышлениями силы.

Питались они за одним круглым столом (рис. 6.11), на который помещались большое блюдо со спагетти, пять тарелок, по одной для каждого философа, и пять вилок. Проголодавшийся философ садится на свое место за столом и, пользуясь двумя вилками, приступает к еде. Задача состоит в том, чтобы разработать ритуал (читай — алгоритм) обеда, который обеспечивает взаимоисключения (два философа не могут одновременно пользоваться одной вилкой) и не допускает взаимоблокировок и голодания (обратите внимание, насколько уместным оказался этот термин в данной задаче!).

Эта задача Дейкстры может показаться не очень важной, но она очень хорошо иллюстрирует проблемы взаимоблокировок и голодания. Кроме того, при решении данной задачи приходится сталкиваться со многими трудностями в организации параллельных вычислений (см., например, [89]). Задача об обедающих философах может рассматриваться как типичная задача, возникающая в многопоточных приложениях при работе с совместно используемыми ресурсами и, соответственно, может выступать в качестве тестовой при разработке новых подходов к проблеме синхронизации.



**Рис. 6.11.** Обеденный стол философов

## Решение с использованием семафоров

На рис. 6.12 предложено решение этой задачи с использованием семафоров. Каждый философ, сядясь за стол, сначала берет левую вилку, а затем правую. После того как философ пообедает, использованные им вилки заменяются. Увы, такое решение может привести к взаимоблокировке, если философы, одновременно проголодавшись, все вместе сядут за стол и одновременно возьмут лежащие слева вилки. В этой неприятной ситуации им придется голодать.

Чтобы избежать риска взаимоблокировки, можно купить еще пять вилок (кстати, самое подходящее решение задачи с точки зрения гигиены!) или научить философов есть спагетти одной вилкой. Еще один подход состоит в том, чтобы нанять вышибалу, который не позволит пяти философам садиться за стол одновременно. Если же за столом сберутся не более четырех философов, то по крайней мере один из них сможет воспользоваться двумя вилками. На рис. 6.13 приведено соответствующее решение задачи (вновь с использованием семафоров). Ни взаимоблокировок, ни голодания при таком решении просто не может быть.

## Решение с использованием монитора

На рис. 6.14 показано решение задачи об обедающих философах с помощью монитора. В программе определен вектор из пяти условных переменных, по одной условной переменной на вилку. Эти условные переменные используются для того, чтобы обеспечить ожидание философом доступности вилки.

```

/* program diningphilosophers */
semaphore fork [5] = {1};
int i;
void philosopher (int i)
{
    while (true) {
        think();
        wait (fork[i]);
        wait (fork [(i+1) mod 5]);
        eat();
        signal(fork [(i+1) mod 5]);
        signal(fork[i]);
    }
}
void main()
{
    parbegin (philosopher (0), philosopher (1),
              philosopher (2), philosopher (3),
              philosopher (4));
}

```

**Рис. 6.12.** Первое решение задачи об обедающих философах

```

/* program diningphilosophers */
semaphore fork[5] = {1};
semaphore room = {4};
int i;
void philosopher (int i)
{
    while (true) {
        think();
        wait (room);
        wait (fork[i]);
        wait (fork [(i+1) mod 5]);
        eat();
        signal (fork [(i+1) mod 5]);
        signal (fork[i]);
        signal (room);
    }
}
void main()
{
    parbegin (philosopher (0), philosopher (1),
              philosopher (2), philosopher (3),
              philosopher (4));
}

```

**Рис. 6.13.** Второе решение задачи об обедающих философах

```

monitor dining_controller;
cond ForkReady[5]; /* Условная переменная для синхронизации */
boolean fork[5] = {true}; /* Состояние доступности каждой вилки */

void get_forks(int pid) /* pid - идентификатор философа */
{
    int left = pid;
    int right = (++pid) % 5;
    /* Предоставление левой вилки */
    if (!fork[left])
        cwait(ForkReady[left]); /* Очередь условной переменной */
    fork[left] = false;
    /* Предоставление правой вилки */
    if (!fork[right])
        cwait(ForkReady[right]);/* Очередь условной переменной */
    fork[right] = false;
}

void release_forks(int pid)
{
    int left = pid;
    int right = (++pid) % 5;
    /* Освобождение левой вилки */
    if (empty(ForkReady[left])) /* Этую вилку никто не ждет */
        fork[left] = true;
    else /* Пробуждение процесса, ожидающего эту вилку */
        csignal(ForkReady[left]);
    /* Освобождение правой вилки */
    if (empty(ForkReady[right]))/* Этую вилку никто не ждет */
        fork[right] = true;
    else /* Пробуждение процесса, ожидающего эту вилку */
        csignal(ForkReady[right]);
}

void philosopher[k=0 to 4]      /* Пять философов-клиентов */
{
    while (true) {
        <Размышления>;
        get_forks(k); /* Запрос вилок с помощью монитора */
        <Поедание спагетти>;
        release_forks(k); /* Освобождение вилок с помощью монитора */
    }
}

```

**Рис. 6.14.** Решение задачи об обедающих философах с использованием монитора

Кроме того, имеется вектор логических значений, который записывает состояние доступности каждой вилки (`true` означает, что вилка доступна). Монитор состоит из двух процедур. Процедура `get_forks` используется философом для захвата его левой и правой вилок. Если любая из вилок недоступна, процесс философа помещается в очередь соответствующей условной переменной. Это позволяет другим процессам философов войти в монитор. Процедура `release_forks` используется для освобождения двух вилок. Обратите внимание, что структура этого решения похожа на структуру решения с семафорами, предложенного на рис. 6.12. В обоих случаях философ сначала завладевает левой вилкой, а затем — правой. В отличие от решения с использованием семафора рассматриваемое решение с использованием монитора не страдает от взаимоблокировки, потому что в мониторе одновременно может находиться только один процесс. Например, первый вошедший в монитор процесс философа гарантированно сможет забрать вилку справа (после того, как выберет левую вилку) до того, как следующий философ справа получит шанс взять свою левую вилку (являющуюся правой вилкой для первого философа).

## 6.7. МЕХАНИЗМЫ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ В UNIX

UNIX предоставляет различные механизмы для синхронизации и связи между процессами. В этом разделе мы рассмотрим важнейшие из них.

- Каналы
- Сообщения
- Совместно используемая память
- Семафоры
- Сигналы

Каналы, сообщения и совместно используемая память обеспечивают обмен данными между процессами, в то время как семафоры и сигналы используются для инициации некоторых действий другого процесса.

### Каналы

Каналы (pipes) являются одним из наиболее значительных вкладов UNIX в развитие операционных систем. Разработанные на основе концепции сопрограмм [204], каналы представляют собой циклические буферы, которые позволяют двум процессам связываться друг с другом в соответствии с моделью “производитель/потребитель”. Следовательно, канал — не что иное, как очередь, работающая по принципу “первым вошел — первым вышел”, запись в которую осуществляется одним процессом, а чтение — другим.

При создании канала он получает буфер определенного размера. При записи в канал при наличии свободного места соответствующий запрос удовлетворяется немедленно; в противном случае процесс блокируется. Аналогично блокируется процесс, пытающийся прочесть из канала большее количество информации, чем имеющееся в нем; в противном случае запрос на чтение выполняется немедленно. Операционная система обеспечивает взаимоисключения — одновременно доступ к каналу имеет только один процесс.

Существует два типа каналов: именованные и неименованные. Совместно использовать неименованные каналы могут только связанные друг с другом процессы; не связанные друг с другом процессы могут совместно использовать только именованные каналы.

## Сообщения

Сообщение представляет собой блок байтов с определенным типом данных. UNIX для работы с системой передачи сообщений предоставляет процессам системные вызовы `msgsnd` и `msgrcv`. С каждым процессом связана очередь сообщений, функционирующая подобно почтовому ящику.

Указанный отправителем тип сообщения может быть использован получателем как критерий отбора сообщений. Он может получать сообщения либо в соответствии с принципом “первым вошел — первым вышел”, либо в соответствии с их типом. При попытке отправить сообщение в заполненную очередь выполнение процесса приостанавливается, так же как и при попытке прочесть сообщение из пустой очереди. Если же процесс пытается прочесть сообщение определенного типа, но такого сообщения в очереди нет, процесс не приостанавливается.

## Совместно используемая память

Наиболее быстрым видом связи между процессами, обеспечиваемым операционной системой UNIX, является совместно используемая память. Это общий блок виртуальной памяти, совместно используемый многими процессами. Процессы читают информацию и записывают ее в эту общую память с помощью тех же инструкций чтения и записи, что и при работе с другими частями своего виртуального пространства памяти. Права доступа (чтение и запись или только чтение) к общей памяти предоставляются каждому из процессов в отдельности. Взаимоисключения не являются частью механизма совместно используемой памяти и должны обеспечиваться процессами, использующими общую память.

## Семафоры

Система вызовов семафоров в UNIX System V представляет собой обобщение примитивов `semWait` и `semSignal`, определенных в главе 5, “Параллельные вычисления: взаимоисключения и многозадачность”. Одновременно могут выполняться несколько операций, а операции инкремента и декремента могут давать значения, большие 1. Все требуемые при работе с семафорами операции выполняются ядром автоматически; ни один процесс не может получить доступ к семафору, пока с ним выполняется операция, вызванная другим процессом.

Семафор состоит из следующих элементов.

- Текущее значение семафора.
- Идентификатор последнего процесса, работавшего с семафором.
- Количество процессов, ожидающих, пока значение семафора превысит текущее.
- Количество процессов, ожидающих, пока значение семафора станет равным нулю.

С семафором связаны очереди приостановленных процессов.

При создании семафоры принадлежат множествам. Множество может содержать как один, так и несколько создаваемых семафоров. Системный вызов `semctl` позволяет установить значения всех семафоров множества одновременно. Кроме того, имеется системный вызов `sem_op`, в качестве аргумента которому передается список операций с семафорами (по одной для каждого семафора из множества). При этом вызове ядро выполняет указанные операции одновременно. Каждая из операций определяется значением `sem_op`.

- Если `sem_op` положительно, ядро увеличивает значение семафора и активизирует все процессы, ожидающие увеличения значения семафора.
- Если `sem_op` равно 0, ядро проверяет значение семафора. Если оно нулевое, то ядро переходит к выполнению следующей операции из списка. В противном случае ядро увеличивает количество процессов, ожидающих обнуления семафора, и приостанавливает процесс до тех пор, пока значение семафора не станет равным нулю.
- Если `sem_op` отрицательно, а его абсолютное значение не превышает значение семафора, ядро добавляет `sem_op` (отрицательное число!) к значению семафора. Если полученный результат равен нулю, ядро активизирует все процессы, ожидающие обнуления значения семафора.
- Если `sem_op` отрицательно, а его абсолютное значение больше значения семафора, ядро приостанавливает процесс до тех пор, пока значение семафора не увеличится.

Такое обобщение семафоров обеспечивает значительную гибкость при выполнении синхронизации и координации процессов.

## Сигналы

Сигнал представляет собой программный механизм, информирующий процесс о наступлении асинхронного события. Сигнал подобен аппаратному прерыванию, но не использует систему приоритетов, т.е. все сигналы обрабатываются одинаково. Процесс получает сигналы по одному, без специального упорядочения.

Процессы могут посыпать сигналы друг другу; в обмене сигналами может принимать участие и ядро. Доставка сигнала выполняется путем обновления поля в таблице процесса, которому послан данный сигнал. Поскольку каждый сигнал соответствуетциальному биту, сигналы одного типа не могут накапливаться в виде очереди. Сигнал обрабатывается сразу же после активизации процесса или возврата его из системного вызова. Процесс может ответить на сигнал выполнением некоторых действий по умолчанию (например, завершением работы), выполнить функцию обработки сигнала или проигнорировать его.

В табл. 6.1 перечислены сигналы UNIX SVR4.

**ТАБЛИЦА 6.1. Сигналы UNIX**

<b>Значение</b>	<b>Имя</b>	<b>Описание</b>
01	SIGHUP	Завесить; посыпается процессу, когда ядро полагает, что пользователь этого процесса выполняет бесполезную работу
02	SIGINT	Прерывание
03	SIGQUIT	Выход; посыпается пользователем, для того чтобы вызвать остановку процесса и сброс дампа памяти
04	SIGILL	Некорректная инструкция
05	SIGTRAP	Запуск кода трассировки процесса
06	SIGIOT	Команда IOT
07	SIGEMT	Команда EMT
08	SIGFPT	Исключение при работе с числами с плавающей точкой
09	SIGKILL	Прекращение работы процесса
10	SIGBUS	Ошибка шины
11	SIGSEGV	Нарушение сегментации; процесс пытается обратиться к ячейке памяти вне своего виртуального адресного пространства
12	SIGSYS	Неверный аргумент системного вызова
13	SIGPIPE	Запись в канал, к которому не присоединены процессы, считающие из него информацию
14	SIGALARM	Сигнал от часов; используется, когда процесс должен получить сигнал после определенного периода времени
15	SIGTERM	Завершение работы программы
16	SIGUSR1	Пользовательский сигнал 1
17	SIGUSR2	Пользовательский сигнал 2
18	SIGCLD	Завершение дочернего процесса
19	SIGPWR	Сбой питания

## 6.8. МЕХАНИЗМЫ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ ЯДРА LINUX

Linux включает все механизмы параллелизма, используемые в других системах UNIX, таких как SVR4, в том числе каналы, сообщения, совместно используемую память и сигналы. Linux также поддерживает специальный тип сигнализации, известный как сигналы реального времени (RT). Они являются частью расширений реального времени POSIX.1b. RT-сигналы отличаются от стандартных сигналов UNIX (или POSIX.1) следующим:

- поддерживается доставка сигналов с учетом приоритета;
- несколько сигналов могут быть организованы в очередь;
- в случае стандартных сигналов целевому процессу нельзя отправить никакое значение или сообщение. RT-сигналы могут, кроме самого сигнала, пересыпать значения (такие, как целое число или указатель).

Linux также включает в себя богатый набор механизмов параллелизма, специально предназначенных для использования при выполнении потоков в режиме ядра, т.е. это механизмы, используемые в ядре для обеспечения параллелизма при выполнении кода ядра. В этом разделе рассматриваются механизмы параллелизма ядра Linux.

## Атомарные операции

Linux предоставляет набор операций, которые гарантируют атомарность операций над переменными. Эти операции могут использоваться, чтобы избежать простых состояний гонки. Атомарная операция выполняется без прерывания и без вмешательства извне. В однопроцессорной системе поток, выполняющий атомарную операцию, не может быть прерван после начала операции и до ее завершения. В многопроцессорной системе, кроме того, переменная, над которой выполняется операция, блокируется от доступа со стороны других потоков до завершения этой операции.

В Linux определены два типа атомарных операций: операции с целыми числами, которые работают с целочисленными переменными, и операции над битовыми массивами (bitmap), которые действуют на один бит в массиве (табл. 6.2). Эти операции должны быть реализованы в любой архитектуре, где реализуется Linux. Для одних архитектур существуют соответствующие команды языка ассемблера для атомарных операций. В других архитектурах для гарантии атомарности операция блокирует шину памяти.

**Таблица 6.2. Атомарные операции Linux**

<b>Атомарные целочисленные операции</b>	
ATOMIC_INIT(int i)	Объявление: инициализация atomic_t значением i
int atomic_read atomic_t *v)	Чтение целочисленного значения v
void atomic_set atomic_t *v, int i)	Установка значения v равным целочисленному значению i
void atomic_add(int i, atomic_t *v)	Прибавление i к v
void atomic_sub(int i, atomic_t *v)	Вычитание i из v
void atomic_inc(atomic_t *v)	Прибавление 1 к v
void atomic_dec(atomic_t *v)	Вычитание 1 из v
int atomic_sub_and_test(int i, atomic_t *v)	Вычитание i из v; возвращает 1, если результат равен 0, и 0 в противном случае
int atomic_add_negative(int i, atomic_t *v)	Прибавление i к v; возвращает 1, если результат отрицательный, и 0 в противном случае (используется для реализации семафоров)
int atomic_dec_and_test(atomic_t *v)	Вычитание 1 из v; возвращает 1, если результат равен 0, и 0 в противном случае
int atomic_inc_and_test(atomic_t *v)	Прибавление 1 к v; возвращает 1, если результат равен 0, и 0 в противном случае

### Атомарные операции с битовыми массивами

<code>void set_bit(int nr, void *addr)</code>	Установка бита <code>nr</code> в битовом массиве, на который указывает <code>addr</code>
<code>void clear_bit(int nr, void *addr)</code>	Сброс бита <code>nr</code> в битовом массиве, на который указывает <code>addr</code>
<code>void change_bit(int nr, void *addr)</code>	Инверсия бита <code>nr</code> в битовом массиве, на который указывает <code>addr</code>
<code>int test_and_set_bit(int nr, void *addr)</code>	Установка бита <code>nr</code> в битовом массиве, на который указывает <code>addr</code> ; возвращает старое значение бита
<code>int test_and_clear_bit(int nr, void *addr)</code>	Сброс бита <code>nr</code> в битовом массиве, на который указывает <code>addr</code> ; возвращает старое значение бита
<code>int test_and_change_bit(int nr, void *addr)</code>	Инверсия бита <code>nr</code> в битовом массиве, на который указывает <code>addr</code> ; возвращает старое значение бита
<code>int test_bit(int nr, void *addr)</code>	Возврат значения бита <code>nr</code> в битовом массиве, на который указывает <code>addr</code>

Для атомарных целочисленных операций используется специальный тип данных `atomic_t`. Атомарные целочисленные операции могут использоваться только с этим типом данных, и никакие другие операции с этим типом данных не разрешены. В [158] перечисляются следующие преимущества этих ограничений.

1. Атомарные операции никогда не используются для переменных, которые могут в некоторых обстоятельствах оказаться незащищенными от состояния гонки.
2. Переменные этого типа данных защищены от некорректного использования неатомарными операциями.
3. Компилятор не может ошибочно оптимизировать доступ к такому значению (например, используя псевдоним, а не корректный адрес памяти).
4. Этот тип данных служит для скрытия архитектурно-зависимых различий при реализации.

Типичное применение атомарных целочисленных данных — при реализации счетчиков.

**Атомарные операции над битовыми массивами** работают с одной из последовательностей битов в произвольном месте памяти, на которое указывает соответствующий указатель. Таким образом, здесь не имеется эквивалента типу данных `atomic_t`, необходимому для атомарных целочисленных операций.

Атомарные операции являются простейшим из средств синхронизации на уровне ядра. Поверх них могут быть построены более сложные механизмы блокировки.

## Циклические блокировки

Наиболее распространенным методом, используемым для защиты критических участков в Linux, является циклическая блокировка, или спин-блокировка (spinlock). Захватить циклическую блокировку одновременно может только один поток. Любой другой поток, пытающийся захватить ту же блокировку, будет предпринимать (циклические) попытки, пока не сможет ее захватить. По сути, циклическая блокировка построена на целочисленной ячейке памяти, которая проверяется каждым потоком перед тем как войти в соответствующий критический участок. Если ее значение равно 0, поток устанавливает значение равным 1 и входит в критический участок. Если значение ненулевое, поток постоянно проверяет это значение до тех пор, пока оно не станет равным 0. Циклическая блокировка легко реализуется, но имеет тот недостаток, что блокированные потоки продолжают выполняться в режиме пережидания занятости. Таким образом, данная разновидность блокировок является наиболее эффективной в тех случаях, когда время ожидания блокировки должно быть очень коротким, по порядку величины не более двух переключений контекста.

Основной формой использования циклической блокировки является следующая:

```
spin_lock(&lock)
/* Критический участок */
spin_unlock(&lock)
```

### Базовые циклические блокировки

Базовые циклические блокировки (в отличие от циклических блокировок читателя-писателя, о которых речь пойдет далее) имеют четыре разновидности (табл. 6.3).

- **Простая.** Если критический участок кода не выполняется обработчиками прерываний или если прерывания во время выполнения критического участка отключены, может использоваться простая циклическая блокировка. Она не влияет на состояние прерываний процессора, на котором выполняется.
- **\_irq.** Если прерывания всегда включены, должна использоваться данная циклическая блокировка.
- **\_irqsave.** Если не известно, какие прерывания (если таковые имеются) будут включены или отключены во время выполнения, то должна быть использована эта версия. Когда блокировка захвачена, сохраняется текущее состояние прерываний локального процессора, которое затем восстанавливается, когда блокировка освобождается.
- **\_bh.** Когда происходит прерывание, соответствующим обработчиком прерывания выполняется минимальный объем необходимой работы. Фрагмент кода, именуемый *нижней половиной* (bottom half), выполняет остальную часть связанной с обработкой прерывания работы, позволяя текущему прерыванию быть включенным как можно скорее. Такая циклическая блокировка используется для того, чтобы отключить, а затем включить нижние половины во избежание конфликта с защищенным критическим участком.

ТАБЛИЦА 6.3. ЦИКЛИЧЕСКИЕ БЛОКИРОВКИ LINUX

<code>void spin_lock(spinlock_t *lock)</code>	Захват указанной блокировки; при необходимости циклическое ожидание доступности
<code>void spin_lock_irq(spinlock_t *lock)</code>	Аналогично <code>spin_lock</code> , но, кроме того, отключает прерывания на локальном процессоре
<code>void spin_lock_irqsave(spinlock_t *lock, unsigned long flags)</code>	Аналогично <code>spin_lock_irq</code> , но, кроме того, сохраняет текущее состояние прерываний в флагах
<code>void spin_lock_bh(spinlock_t *lock)</code>	Аналогично <code>spin_lock</code> , но, кроме того, отключает выполнения всех нижних половин
<code>void spin_unlock(spinlock_t *lock)</code>	Освобождает указанную блокировку
<code>void spin_unlock_irq(spinlock_t *lock)</code>	Освобождает указанную блокировку и включает локальные прерывания
<code>void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags)</code>	Освобождает указанную блокировку и восстанавливает переданное предыдущее состояние локальных прерываний
<code>void spin_unlock_bh(spinlock_t *lock)</code>	Освобождает указанную блокировку и включает нижние половины
<code>void spin_lock_init(spinlock_t *lock)</code>	Инициализирует данную циклическую блокировку
<code>int spin_trylock(spinlock_t *lock)</code>	Пытается захватить определенную блокировку; возвращает ненулевое значение, если блокировка в настоящее время захвачена, и нуль в противном случае
<code>int spin_is_locked(spinlock_t *lock)</code>	Возвращает ненулевое значение, если блокировка в настоящее время захвачена, и нуль в противном случае

Если программист знает, что к защищенным данным нет обращений со стороны обработчика прерываний или нижней половины, используется простая циклическая блокировка. В противном случае используется соответствующая циклическая блокировка, не являющаяся простой.

Циклические блокировки реализуются по-разному в однопроцессорной и в многопроцессорной системах. В случае однопроцессорной системы применяются следующие соображения. Если вытеснение ядра отключено, так что поток в режиме ядра не может быть прерван, блокировки удаляются во время компиляции — они не нужны. Если вытеснение ядра включено, что допускает прерывания, то циклические блокировки вновь при компиляции убираются (т.е. тест ячейки памяти циклической блокировки не происходит) и реализуются просто как код, который включает/отключает прерывания. В многопроцессорной системе циклическая блокировка компилируется в код, который на самом деле проверяет содержимое памяти. Использование механизма циклических блокировок в программе позволяет ей быть независимой от того, выполняется ли она в однопроцессорной или многопроцессорной системе.

## Циклические блокировки читателя/писателя

Циклические блокировки читателя/писателя представляют собой механизм, который обеспечивает более высокую степень параллелизма в ядре, чем базовые циклические блокировки. Циклические блокировки читателя/писателя позволяют нескольким потокам иметь одновременный доступ к одной и той же структуре данных только для чтения, но дают эксклюзивный доступ к циклической блокировке для потока, который намерен обновить структуру данных. Каждая такая блокировка состоит из 24-битового счетчика читателей и флага разблокирования, со следующей интерпретацией.

Счетчик	Флаг	Интерпретация
0	1	Блокировка освобождена для последующего использования
0	0	Блокировка захвачена для записи единственным потоком
$n$ ( $n > 0$ )	0	Блокировка захвачена для чтения $n$ потоками
$n$ ( $n > 0$ )	1	Некорректное сочетание

Как и для базовых циклических блокировок, имеются простая версия данной разновидности блокировок, а также `_irq-` и `_irqsave-`версии циклических блокировок читателей/писателей.

Обратите внимание, что циклическая блокировка читателя/писателя отдает предпочтение читателям перед писателями. Если циклическая блокировка захвачена читателями, то до тех пор, пока имеется по крайней мере один читатель, циклическая блокировка не может быть вытеснена писателем. Кроме того, к циклической блокировке новые читатели могут быть добавлены даже во время ожидания писателя.

## Семафоры

На уровне пользователей Linux предоставляет интерфейс семафоров, соответствующий интерфейсу в UNIX SVR4. Внутренне Linux предоставляет реализацию семафоров для собственного использования. То есть код, который является частью ядра, может вызывать семафоры ядра. Эти семафоры ядра недоступны пользовательским программам непосредственно через системные вызовы. Они реализованы как функции внутри ядра и, таким образом, более эффективны, чем семафоры, видимые пользователям.

Linux предоставляет три типа семафоров ядра: бинарные семафоры, семафоры со счетчиками и семафоры читателей/писателей.

### Бинарные семафоры и семафоры со счетчиками

Бинарные семафоры и семафоры со счетчиками, определенные в Linux 2.6 (см. табл. 6.4), имеют функциональность, описанную для таких семафоров в главе 5, “Параллельные вычисления: взаимоисключения и многозадачность”. Приводимые здесь функции с именами `down` и `up` используются для получения функциональности, описанной в указанной главе как `semWait` и `semSignal` соответственно.

Семафор со счетчиком инициализируется с помощью функции `sema_init`, которая дает семафору имя и устанавливает его начальное значение. Бинарные семафоры, именуемые в Linux мьютексами, инициализируются с помощью функций `init_MUTEX` и `init_MUTEX_LOCKED`, которые инициализируют семафор значениями 1 и 0 соответственно.

Linux предоставляет три версии операции `down` (`semWait`).

- Функция `down` соответствует традиционной операции `semWait`, т.е. поток проверяет семафор и блокируется, если семафор недоступен. Поток будет пробужден при соответствующей операции над этим семафором. Обратите внимание, что данное имя функции используется для операции как над семафорами со счетчиками, так и над бинарными семафорами.
- Функция `down_interruptible` позволяет семафору получать сигналы ядра и отвечать на них, будучи в заблокированном операцией `down` состоянии. Если поток разбужен сигналом, функция `down_interruptible` увеличивает значение счетчика семафора и возвращает код ошибки, известный в Linux как `EINTR`. Это сообщает потоку, что вызванная функция семафора была прервана. По сути, поток вынужден “отказаться” от семафора. Эта функция полезна для драйверов устройств и других служб, в которых удобно иметь возможность переопределения операции семафора.
- Функция `down_trylock` делает возможной попытку захвата семафора без блокировки. Если семафор доступен, он захватывается. В противном случае эта функция возвращает ненулевое значение без блокировки потока.

**Таблица 6.4. Семафоры Linux**

---

#### Традиционные семафоры

---

<code>void sema_init(struct semaphore *sem, int count)</code>	Инициализация динамически созданного семафора данным значением <code>count</code>
<code>void init_MUTEX(struct semaphore *sem)</code>	Инициализация динамически созданного семафора значением счетчика 1 (изначально незаблокирован)
<code>void init_MUTEX_LOCKED(struct semaphore *sem)</code>	Инициализация динамически созданного семафора значением счетчика 0 (изначально заблокирован)
<code>void down(struct semaphore *sem)</code>	Попытка захвата данного семафора; если семафор недоступен, вход в непрерываемый спящий режим
<code>int down_interruptible(struct semaphore *sem)</code>	Попытка захвата данного семафора; если семафор недоступен, вход в непрерываемый спящий режим. Возвращает значение <code>EINTR</code> , если получен сигнал, отличающийся от результата операции
<code>int down_trylock(struct semaphore *sem)</code>	Попытка захвата данного семафора; если семафор недоступен, возвращает ненулевое значение
<code>void up(struct semaphore *sem)</code>	Освобождение указанного семафора

---

Окончание табл. 6.4

### Семафоры читателя/писателя

void init_rwsem(struct rw_semaphore *rwsem)	Инициализация динамически созданного семафора значением счетчика 1
void down_read(struct rw_semaphore *rwsem)	Операция down для читателей
void up_read(struct rw_semaphore *rwsem)	Операция up для читателей
void down_write(struct rw_semaphore *rwsem)	Операция down для писателей
void up_write(struct rw_semaphore *rwsem)	Операция up для писателей

### Семафоры читателя/писателя

Семафор читателя/писателя делит пользователей на читателей и писателей; он разрешает одновременно работать нескольким читателям (но не одновременно с писателем), но только одному писателю (но не одновременно с читателями). По сути, семафор функционирует как семафор со счетчиком для читателей и как бинарный семафор (мьютекс) — для писателей. В табл. 6.4 показаны основные операции семафора читателя/писателя. Такой семафор использует непрерывяемый сон, поэтому имеется только одна версия каждой из операций down.

## Барьеры

В некоторых архитектурах компиляторы и/или аппаратное обеспечение процессоров может переупорядочить доступы к памяти в исходном тексте программы для оптимизации производительности. Эти переупорядочения делаются для того, чтобы оптимизировать использование конвейера команд процессора. Алгоритмы переупорядочения содержат проверки, гарантирующие, что зависимости данных не нарушаются. Например, код

```
a = 1;
b = 1;
```

может быть переупорядочен так, что ячейка памяти b оказывается обновленной до ячейки памяти a. Однако код

```
a = 1;
b = a;
```

не будет переупорядочиваться. Тем не менее бывают случаи, когда важно, чтобы код чтения или записи выполнялся в указанном порядке из-за использования информации, которая создается другим потоком или устройством.

Чтобы обеспечить порядок выполнения команд, Linux предоставляет барьеры памяти. В табл. 6.5 перечислены наиболее важные функции, которые определены для этой функциональной возможности. Операция rmb() гарантирует, что никакие операции чтения не проходят через барьер, определяемый местом размещения rmb() в коде. Аналогично операция wmb() гарантирует, что никакие операции записи не проходят через барьер, определяемый местом размещения wmb() в коде. Операция mb() обеспечивает загрузку и сохранение барьера.

Следует отметить два важных момента, касающихся операций с барьерами.

- Барьеры относятся к машинным командам, а именно — к загрузке и сохранению информации в памяти. Таким образом, команда языка программирования высокого уровня `a = b` включает в себя как загрузку (чтение) из ячейки `b`, так и ее сохранение (запись) в ячейку `a`.
- Операции `rmb`, `wmb` и `mb` определяют поведение как компилятора, так и процессора. Что касается компилятора, то команды барьера указывают, что компилятор не должен переупорядочивать команды во время компиляции. Процессору команды барьера указывают, что любые команды в конвейере до барьера должны быть завершены до выполнения любой команды, находящейся после такого.

Операция `barrier()` представляет собой облегченную версию `mb()`, — облегченную в том отношении, что она управляет только поведением компилятора. Это может быть полезным, если известно, что процессор не будет выполнять нежелательные переупорядочения. Например, процессоры Intel x86 не переупорядочивают команды записи.

**Таблица 6.5. ОПЕРАЦИИ БАРЬЕРОВ ПАМЯТИ В LINUX**

<code>rmb()</code>	Предотвращает переупорядочение загрузки от пересечения барьера
<code>wmb()</code>	Предотвращает переупорядочение сохранения от пересечения барьера
<code>mb()</code>	Предотвращает переупорядочение загрузки и сохранения от пересечения барьера
<code>barrier()</code>	Предотвращает переупорядочение загрузки и сохранения компилятором от пересечения барьера
<code>smp_rmb()</code>	В SMP действует, как <code>rmb()</code> , а в UP — как <code>barrier()</code>
<code>smp_wmb()</code>	В SMP действует, как <code>wmb()</code> , а в UP — как <code>barrier()</code>
<code>smp_mb()</code>	В SMP действует, как <code>mb()</code> , а в UP — как <code>barrier()</code>

*Примечание.* SMP — симметричная многопроцессорная система;  
UP — однопроцессорная система.

Операции `smp_rmb`, `smp_wmb` и `smp_mb` обеспечивают оптимизацию кода, который может быть скомпилирован в однопроцессорной (UP) или симметричной многопроцессорной (SMP) системе. Эти команды определяются как обычные барьеры памяти для SMP, но для UP они являются барьерами только для компилятора. Операции `smp_` полезны в ситуациях, когда проблемы зависимости данных возникают только в контексте SMP.

### RCU (Read-Copy-Update)

Механизм RCU (read-copy-update — чтение-копирование-обновление) представляет собой расширенный упрощенный механизм синхронизации, который был интегрирован в ядро Linux в 2002 году. RCU широко используется в ядре Linux, например в подсистемах сети, памяти, в виртуальной файловой системе и многих других. RCU используется и другими операционными системами; DragonFly BSD использует механизм, который напоминает Linux Sleepable RCU (SRCU). Имеется также библиотека RCU для пользовательского пространства, именуемая `librcu`.

В отличие от распространенных механизмов синхронизации Linux RCU-читатели не блокируются. Совместно используемые ресурсы, которые защищает механизм RCU, должны быть доступны через указатели. Базовый API RCU довольно небольшой и состоит только из шести следующих методов.

- `rcu_read_lock()`
- `rcu_read_unlock()`
- `call_rcu()`
- `synchronize_rcu()`
- `rcu_assign_pointer()`
- `rcu_dereference()`

Помимо этих методов, есть около 20 незначительных методов API RCU.

RCU-механизм обеспечивает доступ к общему ресурсу для нескольких читателей и писателей; когда писатель хочет обновить этот ресурс, он создает его копию, обновляет ее и присваивает указатель на новую копию. Впоследствии, когда она больше не нужна, старая версия ресурса освобождается. Обновление указателя является атомарной операцией. Таким образом, читатель может получить доступ к этому ресурсу до или после завершения обновления, но не во время самой операции обновления. С точки зрения производительности механизм синхронизации RCU лучше всего подходит при частом чтении и редкой записи.

Доступ читателей к общему ресурсу должен инкапсулироваться в пределах блока `rcu_read_lock()/rcu_read_unlock()`. Кроме того, доступ к указателю (`ptr`) общего ресурса в пределах этого блока должен осуществляться путем вызова `rcu_dereference(ptr)`, а не путем прямого доступа к нему.

Метод `rcu_dereference()` не должен вызываться за пределами такого блока.

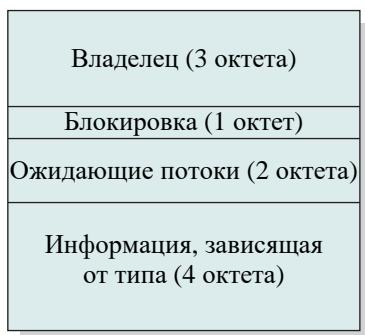
После того как писатель создал копию и изменил ее значение, писатель не может освободить старую версию до тех пор, пока не будет уверен, что она больше не требуется читателям. Это можно сделать с помощью вызова `synchronize_rcu()` или путем вызова неблокирующего метода `call_rcu()`. Второй параметр метода `call_rcu()` ссылается на функцию обратного вызова, которая будет вызвана, когда механизм RCU будет знать, что ресурс может быть освобожден.

## 6.9. ПРИМИТИВЫ СИНХРОНИЗАЦИИ ПОТОКОВ SOLARIS

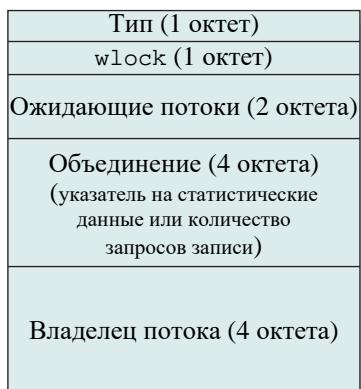
Solaris в дополнение к механизмам параллельных вычислений UNIX SVR4 поддерживает четыре примитива синхронизации потоков.

- Блокировки взаимоисключений (мьютексы)
- Семафоры
- Блокировки читатели/писатель (несколько читателей, один писатель)
- Условные переменные

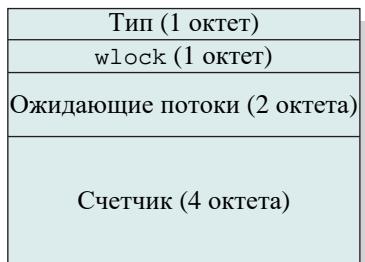
Примитивы для потоков ядра в Solaris реализованы в самом ядре; для работы с пользовательскими потоками имеется соответствующая библиотека. Работа с примитивами осуществляется через структуры данных, содержащие определяемые создавшим эти структуры потоком параметры (см. рис. 6.15). Функции инициализации для этих примитивов заполняют некоторые из их полей необходимыми значениями. После создания синхронизирующего объекта с ним, по сути, могут выполняться только две операции: войти (захватить, заблокировать) и освободить (деблокировать). Ни в ядре, ни в библиотеке потоков нет механизмов, обеспечивающих взаимоисключения или отсутствие взаимоблокировок. Если поток попытается получить доступ к фрагменту данных или кода, который должен быть защищен, но не использует соответствующий примитив синхронизации, то такой доступ будет потоку предоставлен. Если поток блокирует объект, а затем не разблокирует его, не предпринимаются никакие действия ядра.



а) Мьютекс



б) Блокировка читателя/писателя



б) Семафор



г) Условная переменная

Рис. 6.15. Структуры данных синхронизации Solaris

Все примитивы синхронизации требуют наличия машинных команд, которые позволяют атомарно проверить и установить значение объекта.

## БЛОКИРОВКИ ВЗАИМОИСКЛЮЧЕНИЙ

Блокировки взаимоисключений (мьютексы) предотвращают одновременную работу нескольких потоков при захвате блокировки. Поток, блокировавший работу остальных потоков (посредством вызова примитива `mutex_enter`), должен и деблокировать их

(посредством вызова примитива `mutex_exit`). Если примитив `mutex_enter` не в состоянии установить блокировку (поскольку она уже установлена другим потоком), то его дальнейшие действия зависят от информации из структуры данных блокировки. По умолчанию поток циклически опрашивает состояние блокировки, однако может применяться и механизм прерываний с очередью заблокированных потоков.

Для работы с блокировками используются следующие примитивы.

- `mutex_enter()`. Захват блокировки; если уже захвачена, блокирование потока.
- `mutex_exit()`. Освобождение блокировки с возможным деблокированием ожидающего потока.
- `mutex_tryenter()`. Захват блокировки, если она в настоящее время не захвачена

Неблокирующий примитив `mutex_tryenter()` обеспечивает программисту возможность применять на уровне пользовательских потоков технологию пережидания занятости, что позволяет избежать блокирования всего процесса в целом из-за блокирования одного из потоков.

## Семафоры

Solaris поддерживает классические семафоры-счетчики, предоставляя для работы с ними следующие примитивы.

- `sema_p()`. Уменьшает значение семафора (с возможным блокированием потока).
- `sema_v()`. Увеличивает значение семафора (с возможным деблокированием ожидающего потока).
- `sema_tryp()`. Уменьшает значение семафора (если не требуется блокирование).

Примитив `sema_tryp()` обеспечивает программисту возможность использовать на уровне пользовательских потоков технологию пережидания занятости.

## Блокировки “читатели/писатель”

Данная блокировка обеспечивает возможность одновременного доступа только для чтения к защищенному ею объекту нескольким потокам, а также исключительный доступ для записи объекта одному потоку (“исключительный” означает, что, когда такая блокировка оказывается захваченной потоком для записи, все остальные потоки, как читающие, так и записывающие, переходят в состояние ожидания). Для работы с блокировками этого вида используются следующие примитивы.

- `rw_enter()`. Попытка захвата блокировки для чтения или записи.
- `rw_exit()`. Освобождение блокировки.
- `rw_tryenter()`. Неблокирующий захват.
- `rw downgrade()`. Поток, захвативший блокировку для записи, превращает ее в блокировку для чтения. Все потоки записи в состоянии ожидания ждут освобождения блокировки. Если таких нет, то примитив активизирует все потоки чтения.
- `rw_tryupgrade()`. Пытается преобразовать блокировку для чтения в блокировку для записи.

## Условные переменные

Эти переменные используются для ожидания выполнения некоторого условия и должны применяться вместе с мьютексами (тем самым реализуются мониторы, показанные на рис. 6.14). Для работы с ними имеются следующие примитивы.

- `cv_wait()`. Блокирование потока до тех пор, пока условие не будет выполнено.
- `cv_signal()`. Активизация одного из потоков, заблокированных примитивом `cv_wait()`.
- `cv_broadcast()`. Активизация всех потоков, заблокированных примитивом `cv_wait()`.

Примитив `cv_wait()` освобождает связанный с ним мьютекс перед блокированием потока и вновь захватывает его перед завершением работы. Поскольку повторный захват мьютекса может быть заблокирован другим ожидающим потоком, следует повторно проверить выполнение условия, вызвавшего ожидание. Таким образом, типичный фрагмент кода, работающего с переменными условий, выглядит следующим образом:

```
mutex_enter(&m);
    ...
while(some_condition) {
    cv_wait(&cv, &m);
}
    ...
mutex_exit(&m);
```

Поскольку при таком подходе условие защищено мьютексом, в качестве него может использоваться сложное выражение.

## 6.10. МЕХАНИЗМЫ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ В WINDOWS

Windows обеспечивает синхронизацию потоков как часть объектной архитектуры. Наиболее важными средствами синхронизации являются объекты исполнительного диспетчера (Executive dispatcher object), критические участки в режиме пользователя, блокировки читателя/писателя, условные переменные и операции, свободные от блокировок. Объекты диспетчера используют функции ожидания. Мы сначала опишем эти функции ожидания, а затем рассмотрим методы синхронизации.

### ФУНКЦИИ ОЖИДАНИЯ

Функции ожидания позволяют потоку блокировать собственное выполнение. Возврат из функции ожидания не происходит до тех пор, пока указанные критерии не будут удовлетворены. Тип функции ожидания определяет набор использованных критериев. При вызове функция ожидания проверяет, были ли выполнены критерии ожидания. Если критерии не выполнены, вызывающий поток переходит в состояние ожидания. Во время ожидания критериев он не использует процессорное время.

Наиболее простой тип функции ожидания — функция, ожидающая единственный объект. Функция `WaitForSingleObject` требует дескриптора одного объекта синхронизации. Возврат из функции осуществляется, когда происходит одно из следующих действий.

- Указанный объект находится в сигнализирующем состоянии.
- Истек разрешенный интервал времени ожидания. Этот интервал может быть установлен как `INFINITE`, указывая, что ожидание не может завершиться по тайм-ауту.

## Объекты диспетчера

Механизм, используемый исполнительной системой Windows для реализации возможностей синхронизации, представляет собой семейство объектов диспетчера, перечисленных в табл. 6.6.

Первые пять типов объектов в таблице разработаны специально для поддержки синхронизации. Остальные типы объектов имеют другие применения, но могут использоваться и для синхронизации.

**Таблица 6.6. Объекты синхронизации Windows<sup>5</sup>**

Тип объекта	Определение	Переход в сигнализирующее состояние	Влияние на ожидающие потоки
Событие уведомления	Извещение о наступлении некоторого события в системе	Поток осуществляет событие	Освобождаются все потоки
Событие синхронизации	Извещение о наступлении некоторого события в системе	Поток осуществляет событие	Освобождается один поток
Мьютекс	Механизм обеспечения взаимоисключений; эквивалентен бинарному семафору	Поток освобождает мьютекс	Освобождается один поток
Семафор	Счетчик, регулирующий количество потоков, которые могут использовать ресурс	Счетчик семафора обнуляется	Освобождаются все потоки
Таймер ожидания	Счетчик, учитывающий прошедшее время	Наступил указанный момент времени или истек указанный интервал времени	Освобождаются все потоки
Файл	Открытый файл или устройство ввода-вывода	Завершение операции ввода-вывода	Освобождаются все потоки
Процесс	Программа, включая адресное пространство и ресурсы, требуемые для ее выполнения	Завершение последнего потока	Освобождаются все потоки
Поток	Выполнимая единица внутри процесса	Завершение работы потока	Освобождаются все потоки

<sup>5</sup> Выделенные строки соответствуют объектам, созданным специально для целей синхронизации.

Каждый экземпляр объекта диспетчера может быть либо в сигнализирующем, либо в несигнализирующем состоянии. Поток может быть заблокирован объектом в несигнализирующем состоянии; когда объект переходит в сигнализирующее состояние, поток освобождается. Механизм прост: поток выдает запрос на ожидание исполнительной системе Windows, используя дескриптор объекта синхронизации. Когда объект переходит в сигнализирующее состояние, исполнительная система Windows освобождает один или все объекты потоков, ожидающих данный объект диспетчера.

**Объект события** (*event object*) используется для отправления потоку сигнала, указывающего, что произошло определенное событие. Например, при перекрывающемся вводе и выводе система устанавливает определенный объект события в сигнализирующее состояние при завершении перекрывающейся операции. **Объект мьютекса** (*mutex object*) используется для обеспечения взаимно исключающего доступа к ресурсу, позволяя одновременно получить доступ к нему только одному объекту потока. Таким образом, он функционирует как бинарный семафор. Когда объект мьютекса переходит в сигнализирующее состояние, освобождается только один из потоков, ожидающих мьютекс. Мьютексы могут использоваться для синхронизации потоков в различных процессах. Подобно мьютексам, **объекты семафоров** (*semaphore object*) могут совместно использоваться потоками в нескольких процессах. Семафор Windows является подсчитывающим семафором. Объект таймера ожидания (*waitable timer object*) сигнализирует в определенное время и/или через определенный интервал времени.

## Критические участки

Критические участки предоставляют механизм синхронизации, подобный предоставляемому объектами мьютексов, с тем отличием, что критические участки могут использоваться потоками только одного процесса. Объекты событий, мьютексов и семафоров могут использоваться также в однопроцессорном приложении, однако критические участки обеспечивают гораздо более быстрый и эффективный механизм взаимоисключающей синхронизации.

Процесс отвечает за выделение памяти, используемой критическим участком. Как правило, это делается путем простого объявления переменной, имеющей тип CRITICAL\_SECTION. Прежде чем потоки процесса смогут ее использовать, критический участок должен быть инициализирован с помощью функции InitializeCriticalSection.

Для запроса на владение критическим участком поток использует функцию EnterCriticalSection или TryEnterCriticalSection. Для освобождения критического участка используется функция LeaveCriticalSection. Если критический участок в настоящее время принадлежит другому потоку, функция EnterCriticalSection ожидает получения участка во владение в течение неопределенного времени (в отличие от нее функция ожидания объекта мьютекса при использовании для взаимного исключения получает в качестве параметра интервал времени ожидания). Функция TryEnterCriticalSection выполняет попытку входа в критический участок без блокирования вызывающего потока.

Критические участки используют сложный алгоритм при попытке захвата мьютекса. Если система многопроцессорная, код будет пытаться захватить циклическую блокировку (spinlock). Это хорошо работает в ситуациях, когда критический участок захватывается лишь на короткое время. Циклическая блокировка эффективно оптимизируется в ситуации, когда поток, который в настоящее время владеет критическим участком, выпол-

няется на другом процессоре. Если циклическая блокировка не может быть захвачена в течение разумного количества итераций, для блокирования потока используется объект диспетчера, так что ядро может передать процессору для выполнения еще один поток.

Объект диспетчера выделяется только в качестве последнего средства. Большинство критических участков необходимы для корректности работы, но на практике борьба за них происходит очень редко. Отложенное выделение объекта диспетчера позволяет системе сохранить значительное количество виртуальной памяти ядра.

## Гибкие блокировки читателя/писателя и условные переменные

В Windows Vista добавлена блокировка читателя/писателя для пользовательского режима. Подобно критическим участкам, блокировка читателя/писателя (reader-writer lock) использует ядро для блокировки только после попыток использовать циклическую блокировку. Эта *гибкая* блокировка обычно требует выделения блока памяти, достаточного для размещения только одного указателя.

Чтобы использовать блокировку SRW (slim reader-writer), процесс объявляет переменную типа SRWLOCK и вызывает функцию InitializeSRWLock для ее инициализации. Потоки вызывают AcquireSRWLockExclusive или AcquireSRWLockShared для захвата блокировки и ReleaseSRWLockExclusive или ReleaseSRWLockShared — для ее освобождения.

В Windows имеются также условные переменные. Процесс должен объявить объект типа CONDITION\_VARIABLE и инициализировать его в некотором потоке с помощью вызова функции InitializeConditionVariable. Условные переменные могут использоваться либо с критическими участками, либо с SRW-блокировками, так что существуют две функции, SleepConditionVariableCS и SleepConditionVariableSRW, которые атомарно переводят поток в состояние сна для заданной условной переменной и освобождают указанную блокировку.

Имеются также два метода пробуждения, WakeConditionVariable и WakeAllConditionVariable, которые пробуждают соответственно один или все спящие потоки. Условные переменные используются следующим образом.

1. Захват исключительной блокировки.
2. Цикл `while(predicate() == FALSE) SleepConditionVariable();`
3. Выполнение защищенной операции.
4. Освобождение блокировки.

## Синхронизация без участия блокировок

Windows активно использует блокировки для синхронизации. Операции блокировки для гарантии атомарности чтения, изменения и записи ячеек памяти как единой операции используют аппаратные средства. Примерами могут служить функции InterlockedIncrement и InterlockedCompareExchange; последняя позволяет обновлять ячейку памяти, если ее значение при чтении равно указанному.

Реализации многих примитивов синхронизации не используют блокирующие операции, но эти операции также доступны для программистов в ситуациях, когда желательно выполнить синхронизацию без использования программных блокировок. Эти так назы-

ваемые примитивы синхронизации, свободные от блокировок (lock-free synchronization primitives), обладают тем преимуществом, что поток не может быть вытеснен с процессора (скажем, в конце кванта времени), если блокировка все еще сохраняется. Таким образом, они не могут блокировать выполнение других потоков.

Более сложные примитивы без блокировок могут быть созданы на основе операций блокировки. Здесь особенно следует отметить односвязные списки Windows SLLists, которые обеспечивают очереди без использования блокировок. SLLists управляются с помощью таких функций, как `InterlockedPushEntrySList` и `InterlockedPopEntrySList`.

## 6.11. МЕЖПРОЦЕССНОЕ ВЗАИМОДЕЙСТВИЕ В ANDROID

Ядро Linux включает ряд функций, которые могут быть использованы для межпроцессного взаимодействия (interprocess communication — IPC), включая каналы, совместно используемую память, сообщения, сокеты, семафоры и сигналы. Android не использует эти возможности для межпроцессного взаимодействия, добавляя к ядру вместо этого новую функциональную возможность, известную как Binder. Binder обеспечивает облегченный удаленный вызов процедур (remote procedure call — RPC), эффективный с точки зрения как требуемого количества памяти, так и требований к обработке, и хорошо отвечает потребностям встроенной системы.

Binder используется в качестве посредника во всех взаимодействиях между двумя процессами. Компонент в одном процессе (клиент) выполняет вызов. Этот вызов направляется Binder в ядре, который передает вызов компоненту назначения в целевом процессе (служба). Возврат от компонента назначения проходит через Binder и доставляется вызывающему компоненту в вызывающем процессе.

Традиционно термин *RPC* означает взаимодействие типа вызов/возврат между клиентским процессом на одной машине и процессом сервера — на другой. В случае Android механизм RPC работает между двумя процессами в одной и той же системе, но в различных виртуальных машинах.

Метод, используемый для взаимодействия с Binder, представляет собой системный вызов `ioctl`. Вызов `ioctl` — это универсальный системный вызов для операций ввода-вывода конкретного устройства. Он может использоваться для обращения к драйверам устройств и к так называемым драйверам псевдоустройств, примером которых является Binder. Драйвер псевдоустройства использует тот же общий интерфейс, что и драйвер устройства, но применяется для управления некоторыми функциями ядра. Вызов `ioctl` включает в качестве параметров выполняемые команды и соответствующие аргументы.

На рис. 6.16 показано типичное использование Binder. Вертикальные пунктирные линии представляют собой потоки в процессе. Перед тем как процесс сможет использовать службу, она должна стать ему известной. Процесс, в котором размещена служба, порождает несколько потоков, так что он может обрабатывать несколько запросов одновременно. Каждый поток уведомляет о себе Binder с помощью блокирующего вызова `ioctl`.

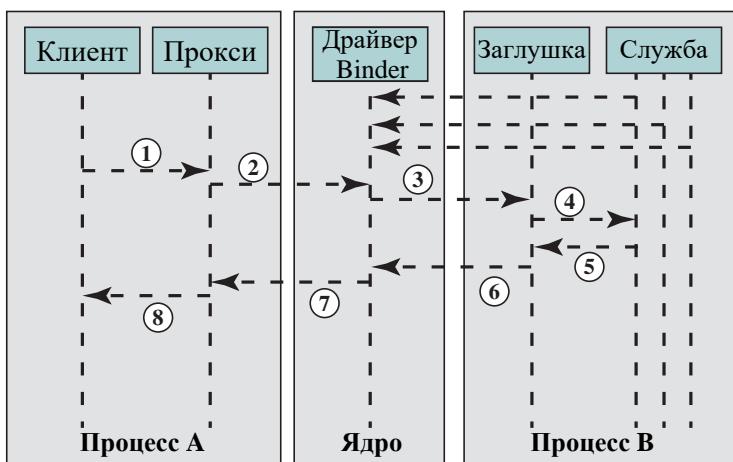


Рис. 6.16. Операции Binder

Взаимодействие осуществляется следующим образом.

1. Компонент клиента, такой как операция, вызывает службу — в виде вызова с данными в аргументах.
2. Вызов обращается к прокси, который транслирует вызов в транзакцию с драйвером Binder. Прокси выполняет процедуру, именуемую **маршаллингом** (marshalling), которая преобразует структуры данных приложений (например, параметры запросов) в **пакет** (parcel). Этот пакет представляет собой контейнер для сообщения (ссылки на данные и объекты), которое пересыпается посредством драйвера Binder. Затем прокси передает транзакцию Binder с помощью блокирующего вызова ioctl.
3. Binder отправляет сигнал целевому потоку, который пробуждает нужный поток из его блокирующего вызова ioctl. Пакет передается компоненту-заглушке в целевом процессе.
4. Заглушка выполняет процедуру под названием **демаршаллинг** (unmarshalling), которая восстанавливает структуры данных приложений из пакетов, полученных с помощью транзакций. Затем прокси вызывает компонент службы с помощью вызова, идентичного вызову, выполненному компонентом клиента.
5. Вызванный компонент службы возвращает результат заглушке.
6. Заглушка маршаллирует возвращаемые данные в ответный пакет и передает его Binder с помощью ioctl.
7. Binder пробуждает вызывающий ioctl в клиентском прокси, который получает возвращенные транзакцией данные.
8. Прокси демаршаллирует результат из ответного пакета и возвращает результат компоненту клиента, который выполнил вызов службы.

## 6.12. Резюме

Взаимоблокировка представляет собой блокирование множества процессов, которые либо конкурируют в борьбе за обладание системными ресурсами, либо поддерживают связь друг с другом. Такая блокировка оказывается постоянной, если только операционная система не предпримет экстраординарных действий типа прекращения или отката одного или нескольких процессов. Взаимоблокировка может происходить при работе как с повторно используемыми, так и с расходуемыми ресурсами. Повторно используемый ресурс, например канал ввода-вывода или область памяти, не истощается и не уничтожается при работе с ним. Расходуемые ресурсы уничтожаются при захвате их процессом; примером такого ресурса может служить информация в буфере ввода-вывода.

При работе с взаимоблокировками имеется три основных подхода: предотвращение, обнаружение и устранение. Предотвращение путем устранения одной из необходимых причин взаимоблокировки гарантирует, что взаимоблокировка возникнуть не может. Обнаружение требуется, если операционная система всегда готова удовлетворить запрос на ресурс; в таком случае операционная система должна регулярно проверять наличие взаимоблокировок и предпринимать действия по их разрешению, если таковые возникают. Устранение взаимоблокировок включает анализ каждого нового запроса для выяснения, не может ли его выполнение привести к взаимоблокировке, и удовлетворение запроса, только если взаимоблокировка невозможна.

## 6.13. КЛЮЧЕВЫЕ ТЕРМИНЫ, КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАЧИ

### Ключевые термины

Алгоритм банкира	Канал	Фатальная область
Граф распределения ресурсов	Удержание и ожидание	Вытеснение
Расходуемый ресурс	Взаимное исключение	Повторно используемый ресурс
Барьер памяти	Небезопасное состояние	Циклическая блокировка
Диаграмма совместного выполнения	Устранение взаимоблокировок	Голодание
Сообщение	Взаимоблокировка	Предотвращение взаимоблокировки
Безопасное состояние	Обнаружение взаимоблокировок	Циклическое ожидание

## Контрольные вопросы

- 6.1. Приведите примеры расходуемых и повторно используемых ресурсов.
- 6.2. Какие три условия являются необходимыми для осуществления взаимоблокировки?
- 6.3. Выполнение каких четырех условий приводит к возникновению взаимоблокировки?
- 6.4. Каким образом можно предотвратить условие удержания и ожидания?
- 6.5. Перечислите два способа предотвращения условия отсутствия перераспределения.
- 6.6. Каким образом можно предотвратить циклическое ожидание?
- 6.7. В чем заключается разница между предотвращением, обнаружением и устранением блокировок?

## Задачи

- 6.1. Покажите, что четыре условия взаимоблокировки применимы к рис. 6.1, а.
- 6.2. Покажите, как каждый из методов предотвращения, обнаружения и устранения взаимоблокировок может быть применен к рис. 6.1.
- 6.3. Опишите каждый из шести приведенных на рис. 6.3 путей так же, как это сделано для рис. 6.2 из раздела 6.1.
- 6.4. Как указывалось, в ситуации, изображенной на рис. 6.3, взаимоблокировка возникнуть не может. Обоснуйте это утверждение.
- 6.5. Пусть задано следующее состояние алгоритма банкира:

6 процессов P0–P5

4 типа ресурсов: A (15 экземпляров); B (6 экземпляров); C (9 экземпляров); D (10 экземпляров)

Вот снимок в момент времени T0.

Доступно

	A	B	C	D
	6	3	5	4

Процесс	Текущее распределение				Максимальная потребность			
	A	B	C	D	A	B	C	D
P0	2	0	2	1	9	5	5	5
P1	0	1	1	1	2	2	3	3
P2	4	1	0	2	7	5	4	4
P3	1	0	0	1	3	3	3	2
P4	1	1	0	0	5	2	2	1
P5	1	0	1	1	4	4	4	4

Первые четыре столбца матрицы  $6 \times 8$  составляют матрицу распределения. Последние четыре столбца составляют матрицу запросов. Матрица потребностей, упомянутая в п. б), представляет собой не что иное, как матрицу  $C - A$ , описанную в основном тексте.

- Проверьте корректность вычисления массива доступности.
- Вычислите матрицу потребностей.
- Покажите, что текущее состояние является безопасным, т.е. покажите безопасную последовательность процессов. Кроме того, для этой последовательности покажите, как при каждом завершении процесса будет меняться массив доступности (рабочий массив).
- Имеется запрос (3,2,3,3) от процесса P5. Должен ли этот запрос быть удовлетворен? Почему?

**6.6.** В приведенном ниже тексте три процесса соперничают за шесть ресурсов, обозначенных буквами от A до F.

- Используя граф распределения ресурсов (см. рис. 6.5 и 6.6), покажите возможность взаимоблокировки в данной реализации.
- Измените порядок некоторых из этих запросов, чтобы предотвратить возможность любой взаимоблокировки. Вы не можете перемещать запросы между процедурами, а можете только изменять их порядок внутри каждой процедуры. Используйте граф распределения ресурсов для обоснования своего ответа.

<pre>void P0() {     while (true) {         get(A);         get(B);         get(C);         // Критический участок:         // Использование A, B, C         release(A);         release(B);         release(C);     } }</pre>	<pre>void P1() {     while (true) {         get(D);         get(E);         get(B);         // Критический участок:         // Использование D, E, B         release(D);         release(E);         release(B);     } }</pre>	<pre>void P2() {     while (true) {         get(C);         get(F);         get(D);         // Критический участок:         // Использование C, F, D         release(C);         release(F);         release(D);     } }</pre>
--	--	--

**6.7.** Система буферизации, представленная на рис. 6.17, состоит из процесса ввода I, пользовательского процесса Р и процесса вывода О, соединенных с двумя буферами. Процессы обмениваются блоками данных одинакового размера, которые буферизуются на диске; граница между входным и выходным буферами динамическая, зависящая от скорости работы процессов. Используемые процессами примитивы гарантируют выполнение условия  $i + o \leq max$ , где

$max$  — максимальное количество блоков на диске;

$i$  — количество входных блоков на диске;

$o$  — количество выходных блоков на диске.

О работе процессов известно следующее.

1. Пока система производит данные, процесс I записывает их на диск (если на диске имеется свободное место).
2. Пока на диске имеются входные данные, процесс P считывает их и для каждого считанного блока выводит некоторое конечное количество данных (если на диске имеется свободное место).
3. Пока на диске имеются выходные данные, процесс O считывает их.

Покажите, что в этой системе может возникнуть взаимоблокировка.



**Рис. 6.17.** Система буферизации

- 6.8. Предложите дополнительное условие использования ресурсов, которое предотвращает взаимоблокировку в предыдущей задаче, но при этом разрешает перемещение границы между входным и выходным буферами в зависимости от текущих нужд процессов.
- 6.9. Во многозадачной системе ТНЕ барабан (предшественник магнитных дисков) разделен на входные буфера, рабочую область и выходные буфера с подвижными границами, зависящими от скорости работы процессов. Текущее состояние барабана можно охарактеризовать следующими параметрами.

*max* — максимальное количество страниц на барабане;

*i* — количество входных страниц на барабане;

*p* — количество страниц рабочей области на барабане;

*o* — количество выходных страниц на барабане;

*reso* — минимальное количество страниц, зарезервированных для выхода;

*resp* — минимальное количество страниц, зарезервированных для работы.

Сформулируйте ограничения, которые должны быть наложены на ресурсы, с тем чтобы гарантировать, что не будет превышена емкость барабана и что для выходных данных и рабочей области постоянно зарезервировано минимальное количество страниц.

- 6.10. В многозадачной системе ТНЕ возможны следующие переходы между состояниями страницы.

1. Пустая → входной буфер (ввод данных)
2. Входной буфер → рабочая область (использование ввода)
3. Рабочая область → выходной буфер (вывод данных)
4. Выходной буфер → пустая (использование вывода)
5. Пустая → рабочая область (вызов процедуры)
6. Рабочая область → пустая (возврат из процедуры)

- а. Опишите действие этих переходов в терминах величин  $i$ ,  $o$  и  $r$ .
- б. Пусть процесс ввода, пользовательский и процесс вывода подчиняются условиям задачи 6.6. Могут ли в таком случае эти переходы привести к взаимной блокировке?
- 6.11. Рассмотрим систему с общим количеством памяти, равным 150 единицам, распределенным между процессами следующим образом.

Процесс	Максимум	Получено
1	70	45
2	60	40
3	60	15

Примените алгоритм банкира для определения того, безопасно ли удовлетворение описанных ниже запросов, и, если безопасно, укажите соответствующую последовательность завершения работы процессов. Если нет, покажите результирующую таблицу распределения.

- а. Начинает работу четвертый процесс с максимальными требованиями к памяти, равными 60 единицам, и начальным запросом на 25 единиц памяти.
- б. Начинает работу четвертый процесс с максимальными требованиями к памяти, равными 60 единицам, и начальным запросом на 35 единиц памяти.
- 6.12. Оцените применимость алгоритма банкира в реальной жизни.
- 6.13. Алгоритм конвейера реализован таким образом, что поток элементов данных типа  $T$ , производимых процессом  $P_0$ , проходит через последовательность процессов  $P_1, P_2, \dots, P_{n-1}$ , обрабатывающих проходящие данные в указанном порядке.
- а. Определите обобщенный буфер сообщений, который содержит все частично потребленные элементы данных, и разработайте алгоритм работы процесса  $P_i$  ( $0 \leq i \leq n-1$ ). Алгоритм должен иметь следующий вид:

**repeat**

получение информации от предшественника;  
потребление полученного элемента;  
послать информацию преемнику;

**forever**

Для простоты считаем, что процесс  $P_0$  получает пустые элементы данных, посылаемые ему процессом  $P_{n-1}$ . Разработанный алгоритм должен позволять процессу работать непосредственно с сообщениями, хранящимися в буфере, не выполняя копирования элементов данных.

- б. Покажите, что, несмотря на работу с общим буфером, взаимоблокировка процессов в этой ситуации невозможна.

- 6.14.** Предположим, что два процесса, `foo` и `bar`, выполняются параллельно и совместно используют переменные семафоров `S` и `R` (каждый из которых инициализирован значением 1) и целочисленную переменную `x` (инициализированную значением 0).

<pre>void foo( ) {     do     {         semWait(S);         semWait(R);         x++;         semSignal(S);         SemSignal(R);     }     while (1); }</pre>	<pre>void bar( ) {     do     {         semWait(R);         semWait(S);         x--;         semSignal(S);         SemSignal(R);     }     while (1); }</pre>
---	---

- Может ли параллельное выполнение этих двух процессов привести к бесконечной блокировке одного или обоих процессов? Если может, приведите последовательность выполнения процессов, при которой один или оба процесса будут заблокированы навсегда.
  - Может ли параллельное выполнение этих двух процессов привести к бесконечному откладыванию одного из них? Если может, приведите последовательность выполнения процессов, при которой один процесс бесконечно откладывается.
- 6.15.** Рассмотрим систему из четырех процессов и одного ресурса. Текущее состояние матриц требований и распределения следующее:

$$\mathbf{C} = \begin{pmatrix} 3 \\ 2 \\ 9 \\ 7 \end{pmatrix} \quad \mathbf{A} = \begin{pmatrix} 1 \\ 1 \\ 3 \\ 2 \end{pmatrix}$$

Каково минимальное количество единиц ресурса должно быть доступно, чтобы это состояние было безопасным?

- 6.16.** Рассмотрим следующие способы работы с взаимоблокировками: 1) алгоритм банкира; 2) обнаружение взаимоблокировки и прекращение работы потока с освобождением всех ресурсов; 3) резервирование всех ресурсов заранее; 4) перезапуск потока и освобождение всех ресурсов, если поток должен находиться в состоянии ожидания; 5) упорядочение ресурсов; 6) обнаружение взаимоблокировки и откат потока.
- Одним из критериев оценки различных подходов к проблеме блокировки является оценка максимальной допустимой параллельности вычислений. Иными словами, какой из методов позволяет работать наибольшему количеству потоков без ожидания и возникновения взаимоблокировок? Упорядочьте перечисленные ранее методы борьбы с взаимоблокировками в порядке уменьшения обеспечиваемой ими степени параллельности и прокомментируйте свое решение.

- б. Еще одним критерием может служить эффективность. Другими словами, какой из методов требует меньших накладных расходов процессорного времени? Считая, что взаимоблокировка — весьма редкое событие, упорядочьте перечисленные ранее методы борьбы с взаимоблокировками в порядке уменьшения их эффективности и прокомментируйте свое решение. Изменится ли порядок, если взаимоблокировки будут происходить часто?

- 6.17. Прокомментируйте следующее решение задачи об обедающих философах. Проголодавшийся философ сначала берет вилку слева от себя. Затем, если вилка справа свободна, он берет ее и приступает к еде; в противном случае он кладет левую вилку на стол и повторяет цикл.
- 6.18. Предположим, что имеется два типа философов — “левые”, которые всегда берут первой вилку слева, и “правые”, которые начинают с правой вилки. Поведение левых философов показано на рис. 6.12; поведение правых — в приведенном фрагменте кода.

```

begin
repeat
    think;
    wait ( fork[ (i+1) mod 5] );
    wait ( fork[i] );
    eat;
    signal ( fork[i] );
    signal ( fork[ (i+1) mod 5]
);
forever
end;

```

Докажите следующие утверждения.

- а. Любое размещение за столом левых и правых философов с присутствием как минимум по одному философу каждого типа позволит устраниТЬ взаимоблокировку.
- б. Любое размещение за столом левых и правых философов с присутствием как минимум по одному философу каждого типа позволит предотвратить голодаНИЕ.
- 6.19. На рис. 6.18 показано еще одно решение задачи об обедающих философах с применением мониторов. Сравните это решение с решением на рис. 6.14 и расскажите, какие выводы вы сделали.
- 6.20. В табл. 6.2 некоторые из атомарных операций Linux, такие как `atomic_read(atomic_t *v)`, не связаны с двумя обращениями к переменной. Простое чтение, очевидно, является атомарной операцией в любой архитектуре. Тогда почему эта операция добавлена в набор атомарных?
- 6.21. Рассмотрим следующий фрагмент кода в Linux.

```

read_lock(&mr_rwlock);
write_lock(&mr_rwlock);

```

Здесь `mr_rwlock` является блокировкой читателя/писателя. Что делает данный код?

```
monitor dining_controller;
enum states {thinking, hungry, eating} state[5];
cond needFork[5] /* Условная переменная */

void get_forks(int pid) /* pid-идентификатор философа */
{
    state[pid] = hungry; /* Сообщение, что я голоден */

    if (state[(pid+1) % 5] == eating || (state[(pid-1) % 5] == eating))
        cwait(needFork[pid]); /* wait if either neighbor is eating */

    state[pid] = eating; /* Ожидание, если сосед ест */
}

void release_forks(int pid)
{
    state[pid] = thinking;

    /* Даем правому (с большим номером) соседу шанс */
    if (state[(pid+1)%5]==hungry) && (state[(pid+2)%5]) != eating)
        csignal(needFork[pid+1]);

    /* Даем левому (с меньшим номером) соседу шанс */
    else if (state[(pid-1)%5]==hungry) && (state[(pid-2)%5]) != eating)
        csignal(needFork[pid-1]);
}

void philosopher[k=0 to 4] /* Пять философов-клиентов */
{
    while (true) {
        <Думает>;
        /* Клиент запрашивает две вилки через монитор */
        get_forks(k);
        <Ест спагетти>;
        /* Клиент освобождает две вилки через монитор */
        release_forks(k);
    }
}
```

**Рис. 6.18.** Еще одно решение задачи об обедающих философах с применением мониторов

6.22. Две переменные, *a* и *b*, имеют начальные значения соответственно 1 и 2. Приведенный далее код работает в Linux.

Поток 1	Поток 2
<i>a</i> = 3;	—
<i>mb</i> ();	—
<i>b</i> = 4;	<i>c</i> = <i>b</i> ;
—	<i>rmb</i> ();
—	<i>d</i> = <i>a</i> ;

Каких возможных ошибок удается избежать с помощью барьеров памяти?

Часть



---

# ПАМЯТЬ



## ГЛАВА

# УПРАВЛЕНИЕ ПАМЯТЬЮ

В ЭТОЙ ГЛАВЕ...

### **7.1. Требования к управлению памятью**

- Перемещение
- Захист
- Совместное использование
- Логическая организация
- Физическая организация

### **7.2. Распределение памяти**

- Фиксированное распределение
- Размеры разделов
- Алгоритм размещения
- Динамическое распределение
- Алгоритм размещения
- Алгоритм замещения
- Система двойников
- Перемещение

### **7.3. Страницчная организация памяти**

### **7.4. Сегментация**

### **7.5. Резюме**

### **7.6. Ключевые термины, контрольные вопросы и задачи**

- Ключевые термины
- Контрольные вопросы
- Задачи

### **Приложение 7.А. Загрузка и связывание**

- Загрузка
  - Абсолютная загрузка
  - Перемещаемая загрузка
  - Динамическая загрузка во время выполнения
- Компоновка
  - Редактор связей
  - Динамический компоновщик

## УЧЕБНЫЕ ЦЕЛИ

- Обсудить основные требования к управлению памятью.
- Понимать причины распределения памяти и пояснить работу различных используемых технологий.
- Понимать концепцию страничной организации памяти.
- Понимать концепцию сегментации.
- Оценить относительные преимущества страничной организации памяти и сегментации.
- Описать концепции загрузки и связывания.

В однозадачных системах основная память разделяется на две части: одна часть — для операционной системы (резидентный монитор, ядро), а вторая — для выполняющейся в текущий момент времени программы. В многозадачных системах “пользовательская” часть памяти должна быть распределена для размещения нескольких процессов. Эта задача распределения выполняется операционной системой динамически и известна под названием **управление памятью** (*memory management*).

Эффективное управление памятью жизненно важно для многозадачных систем. Если в памяти располагается только небольшое число процессов, то большую часть времени все эти процессы будут находиться в состоянии ожидания выполнения операций ввода-вывода, и загрузка процессора будет низкой. Таким образом, желательно эффективное распределение памяти, позволяющее разместить в ней как можно больше процессов.

Глава начинается с рассмотрения требований, которым должны удовлетворять разрабатываемые системы управления памятью. Позже мы приступим к рассмотрению различных технологий управления памятью, начав с применения нескольких простых схем.

В табл. 7.1 приведены некоторые ключевые термины по данной теме.

**Таблица 7.1. Основные термины, связанные с управлением памятью**

<b>Кадр (frame)</b>	Блок основной памяти фиксированной длины
<b>Страница (page)</b>	Блок данных фиксированной длины, находящийся во вторичной памяти (такой, как диск). Страница данных может быть временно скопирована в кадр основной памяти
<b>Сегмент (segment)</b>	Блок данных переменной длины, находящийся во вторичной памяти. В доступную область основной памяти может быть скопирован сегмент полностью (сегментация); возможно также разделение сегмента на страницы, которые копируются в основную память по отдельности (комбинированная сегментация, или страничная организация памяти)

## 7.1. ТРЕБОВАНИЯ К УПРАВЛЕНИЮ ПАМЯТЬЮ

При рассмотрении различных механизмов и стратегий, связанных с управлением памятью, полезно помнить требования, которым они должны удовлетворять. Эти требования включают следующее.

- Перемещение
- Защита
- Совместное использование
- Логическая организация
- Физическая организация

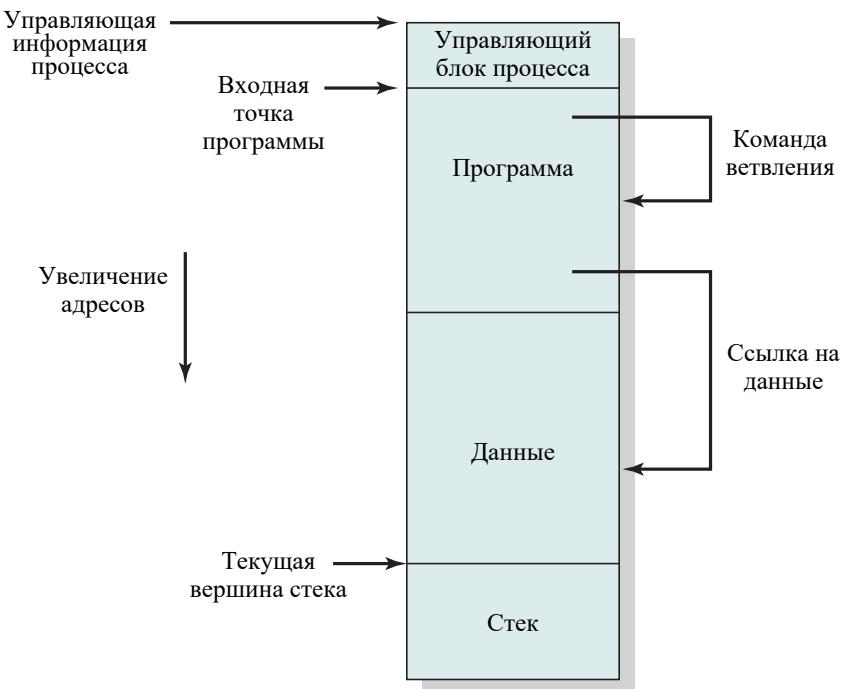
### Перемещение

В многозадачной системе доступная основная память в общем случае разделяется среди множества процессов. Обычно программист не знает заранее, какие программы будут резидентно находиться в основной памяти во время работы разрабатываемой им программы. Кроме того, для максимизации загрузки процессора желательно иметь большой пул процессов, готовых к выполнению, для чего требуется возможность загрузки и выгрузки активных процессов из основной памяти. Требование, чтобы выгруженная из памяти программа была вновь загружена в то же самое место, где находилась и ранее, было бы слишкоменным ограничением. Крайне желательно, чтобы программа могла быть **перемещена** (*relocate*) в другую область памяти.

Таким образом, заранее неизвестно, где именно будет размещена программа, а кроме того, программа может быть перемещена из одной области памяти в другую при свопинге. Эти обстоятельства обусловливают наличие определенных технических требований к адресации, проиллюстрированных на рис. 7.1. На рисунке представлен образ процесса. Для простоты предположим, что образ процесса занимает одну непрерывную область основной памяти. Очевидно, что операционной системе необходимо знать местоположение управляющей информации процесса и стека выполнения, а также точки входа для начала выполнения процесса. Поскольку управлением памятью занимается операционная система и она же размещает процесс в основной памяти, соответствующие адреса она получает автоматически. Однако, помимо получения операционной системой указанной информации, процесс должен иметь возможность обращаться к памяти в самой программе. Так, команды ветвления содержат адреса, указывающие на команды, которые должны быть выполнены после них; команды обращения к данным — адреса байтов или слов, с которыми они работают. Так или иначе, но процессор и программное обеспечение операционной системы должны быть способны перевести ссылки в коде программы в реальные физические адреса, соответствующие текущему расположению программы в основной памяти.

### Защита

Каждый процесс должен быть защищен от нежелательного воздействия других процессов, случайного или преднамеренного. Следовательно, код других процессов не должен иметь возможности без разрешения обращаться к памяти данного процесса для чтения или записи. Однако удовлетворение требованию перемещаемости усложняет задачу защиты.



**Рис. 7.1.** Требования к адресации процесса

Поскольку расположение программы в основной памяти непредсказуемо, проверка абсолютных адресов во время компиляции невозможна. Кроме того, в большинстве языков программирования возможно динамическое вычисление адресов во время выполнения (например, вычисление адреса элемента массива или указателя на поле структуры данных). Следовательно, во время работы программы необходимо выполнять проверку всех обращений к памяти, генерируемых процессом, чтобы удостовериться, что все они — только к памяти, выделенной данному процессу. К счастью, как вы увидите позже, механизмы поддержки перемещений обеспечивают и поддержку защиты.

Обычно пользовательский процесс не может получить доступ ни к какой части операционной системы — ни к коду, ни к данным. Код одного процесса не может выполнить команду ветвления, целевой код которой находится в другом процессе. Если не приняты специальные меры, код одного процесса не может получить доступ к данным другого процесса. Процессор должен быть способен прервать выполнение таких команд.

Заметим, что требования защиты памяти должны быть удовлетворены на уровне процессора (аппаратного обеспечения), а не на уровне операционной системы (программного обеспечения), поскольку операционная система не в состоянии предвидеть все обращения к памяти, которые будут выполнены программой. Даже если бы такое было возможно, сканирование каждой программы в поиске предлагаемых нарушений защиты было бы слишком расточительным с точки зрения использования процессорного времени. Следовательно, соответствующие возможности аппаратного обеспечения — единственное средство определения допустимости обращения к памяти (данным или коду) во время работы программы.

## Совместное использование

Любой механизм защиты должен иметь достаточную гибкость для того, чтобы обеспечить возможность нескольким процессам обращаться к одной и той же области основной памяти. Например, если несколько процессов выполняют один и тот же машинный код, то будет выгодно позволить каждому процессу работать с одной и той же копией этого кода, а не создавать собственную. Процессам, сотрудничающим в работе над некоторой задачей, может потребоваться совместный доступ к одним и тем же структурам данных. Система управления памятью должна, таким образом, обеспечивать управляемый доступ к разделяемым областям памяти, при этом никоим образом не ослабляя защиту памяти. Как мы увидим позже, механизмы поддержки перемещений обеспечивают и поддержку совместного использования памяти.

## Логическая организация

Практически всегда основная память в компьютерной системе организована как линейное (одномерное) адресное пространство, состоящее из последовательности байтов или слов. Аналогично организована и вторичная память на своем физическом уровне. Хотя такая организация и отражает особенности используемого аппаратного обеспечения, она не соответствует способу, которым обычно создаются программы. Большинство программ организованы в виде модулей, одни из которых неизменны (только для чтения, только для выполнения), а другие содержат данные, которые могут быть изменены. Если операционная система и аппаратное обеспечение компьютера могут эффективно работать с пользовательскими программами и данными, представленными модулями, то это обеспечивает ряд преимуществ.

1. Модули могут быть созданы и скомпилированы независимо один от другого, при этом все ссылки из одного модуля во второй разрешаются системой во время работы программы.
2. Разные модули могут получить разные степени защиты (только для чтения, только для выполнения) за счет весьма умеренных накладных расходов.
3. Возможно применение механизма, обеспечивающего совместное использование модулей разными процессами. Основное достоинство обеспечения совместного использования на уровне модулей заключается в том, что они соответствуют взгляду программиста на задачу и, следовательно, ему проще определить, требуется ли совместное использование того или иного модуля.

Инструментом, наилучшим образом удовлетворяющим данным требованиям, является сегментация, которая будет рассмотрена в данной главе среди прочих методов управления памятью.

## Физическая организация

Как указывалось в разделе 1.5, память компьютера разделяется как минимум на два уровня: основная и вторичная. Основная память обеспечивает быстрый доступ по относительно высокой цене; кроме того, она энергозависима, т.е. не обеспечивает долговременное хранение. Вторичная память медленнее и дешевле основной и обычно энергонезависима. Следовательно, вторичная память большой емкости может служить для дол-

говременного хранения программ и данных, а основная память меньшей емкости — для хранения программ и данных, использующихся в текущий момент.

В такой двухуровневой структуре основной заботой системы становится организация потоков информации между основной и вторичной памятью. Ответственность за эти потоки может быть возложена и на отдельного программиста, но это непрактично и нежелательно по следующим причинам.

1. Основной памяти может быть недостаточно для программы и ее данных. В этом случае программист вынужден прибегнуть к практике, известной как структуры с перекрытием — **оверлеи** (overlay), когда программа и данные организованы таким образом, что различные модули могут быть назначены одной и той же области памяти; основная программа при этом ответственна за перезагрузку модулей при необходимости. Даже при помощи соответствующего инструментария компиляции оверлеев разработка таких программ приводит к дополнительным затратам времени программиста.
2. Во многозадачной среде программист при разработке программы не знает, какой объем памяти будет доступен программе и где эта память будет располагаться.

Таким образом, очевидно, что задача перемещения информации между двумя уровнями памяти должна возлагаться на операционную систему. Эта задача является сущностью управления памятью.

## 7.2. РАСПРЕДЕЛЕНИЕ ПАМЯТИ

Главной операцией управления памятью является размещение программы в основной памяти для ее выполнения процессором. Практически во всех современных многозадачных системах эта задача предполагает использование сложной схемы, известной как виртуальная память. Виртуальная память, в свою очередь, основана на использовании одной или обеих базовых технологий — сегментации (segmentation) и страничной организации памяти (paging). Перед тем как перейти к рассмотрению этих методов организации виртуальной памяти, мы должны познакомиться с более простыми методами, не связанными с виртуальной памятью (табл. 7.2). Одна из приведенных в таблице технологий — распределение (partitioning) памяти — использовалась в различных вариациях в некоторых уже подзабытых к настоящему времени операционных системах. Две другие технологии — простые страничная организация памяти и сегментация — сами по себе не используются, однако их рассмотрение в отрыве от виртуальной памяти упростит дальнейшее понимание предлагаемого материала.

### Фиксированное распределение

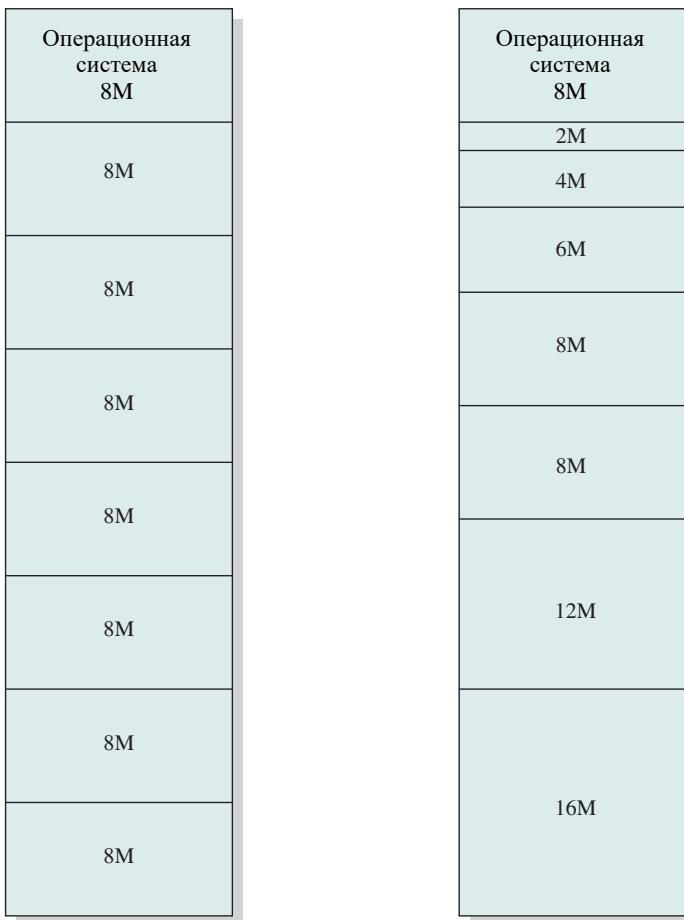
В большинстве схем управления памятью мы будем полагать, что операционная система занимает некоторую фиксированную часть основной памяти и что остальная часть основной памяти доступна для использования многочисленным процессам. Простейшая схема управления этой доступной памятью — ее распределение на области с фиксированными границами.

**ТАБЛИЦА 7.2. ТЕХНОЛОГИИ УПРАВЛЕНИЯ ПАМЯТЬЮ**

Технология	Описание	Сильные стороны	Слабые стороны
<b>Фиксированное распределение</b>	Основная память разделяется на ряд статических разделов во время генерации системы. Процесс может быть загружен в раздел равного или большего размера	Простота реализации, малые системные накладные расходы	Неэффективное использование памяти из-за внутренней фрагментации, фиксированное максимальное количество активных процессов
<b>Динамическое распределение</b>	Разделы создаются динамически; каждый процесс загружается в раздел строго необходимого размера	Отсутствует внутренняя фрагментация, более эффективное использование основной памяти	Неэффективное использование процессора из-за необходимости уплотнения для противодействия внешней фрагментации
<b>Простая страницчная организация</b>	Основная память разделена на ряд кадров равного размера. Каждый процесс разделен на некоторое количество страниц равного размера и той же длины, что и кадры. Процесс загружается путем загрузки всех его страниц в доступные, но не обязательно последовательные кадры	Отсутствует внешняя фрагментация	Наличие небольшой внутренней фрагментации
<b>Простая сегментация</b>	Каждый процесс распределен на ряд сегментов. Процесс загружается путем загрузки всех своих сегментов в динамические (не обязательно смежные) разделы	Отсутствует внутренняя фрагментация; по сравнению с динамическим распределением повышенная эффективность использования памяти и сниженные накладные расходы	Внешняя фрагментация
<b>Страницчная организация виртуальной памяти</b>	Все, как при простой страницочной организации, с тем исключением, что не требуется одновременно загружать все страницы процесса. Необходимые нерезидентные страницы автоматически загружаются в память	Нет внешней фрагментации; более высокая степень многозадачности; большое виртуальное адресное пространство	Накладные расходы из-за сложности системы управления памятью
<b>Сегментация виртуальной памяти</b>	Все, как при простой сегментации, с тем исключением, что не требуется одновременно загружать все сегменты процесса. Необходимые нерезидентные сегменты автоматически загружаются в память	Нет внутренней фрагментации; более высокая степень многозадачности; большое виртуальное адресное пространство; поддержка защиты и совместного использования	Накладные расходы из-за сложности системы управления памятью

## Размеры разделов

На рис. 7.2 показаны два варианта фиксированного распределения. Одна возможность состоит в использовании разделов одинакового размера. В этом случае любой процесс, размер которого не превышает размер раздела, может быть загружен в любой доступный раздел. Если все разделы заняты и нет ни одного процесса в состоянии готовности или работы, операционная система может выгрузить процесс из любого раздела и загрузить другой процесс, обеспечивая тем самым процессор работой.



**Рис. 7.2.** Пример фиксированного распределения 64-мегабайтовой памяти

При использовании разделов с одинаковым размером имеются две трудности.

- Программа может быть слишком велика для размещения в разделе. В этом случае программист должен разрабатывать программу, использующую оверлеи, с тем чтобы в любой момент времени ей требовался только один раздел основной памяти. Когда требуется модуль, который в настоящий момент отсутствует в основной памяти, пользовательская программа должна сама загрузить этот модуль в раздел памяти программы (независимо от того, является ли этот модуль кодом или данными).

- Использование основной памяти при этом крайне неэффективно. Любая программа, независимо от ее размера, занимает раздел целиком. Так, в нашем примере программа размером менее мегабайта все равно будет занимать целиком раздел в 8 Мбайт; при этом остаются неиспользованными 7 Мбайт блока. Этот феномен появления неиспользованной памяти из-за того, что загружаемый блок по размеру меньше раздела, называется **внутренней фрагментацией** (internal fragmentation).

Бороться с этими трудностями (хотя и не устраниТЬ полностью) можно посредством использования разделов разных размеров (см. рис. 7.2, б). В этом случае программа размером 16 Мбайт может обойтись без оверлеев, а разделы малого размера позволяют уменьшить внутреннюю фрагментацию при загрузке программ малого размера.

### Алгоритм размещения

В том случае, когда разделы имеют одинаковый размер, размещение процессов в памяти представляет собой тривиальную задачу. Не имеет значения, в каком из свободных разделов будет размещен процесс. Если все разделы заняты процессами, которые не готовы к немедленной работе, любой из них может быть выгружен для освобождения памяти для нового процесса. Принятие решения о том, какой именно процесс следует выгрузить, — задача планировщика (об этом мы поговорим в части IV, “Планирование”).

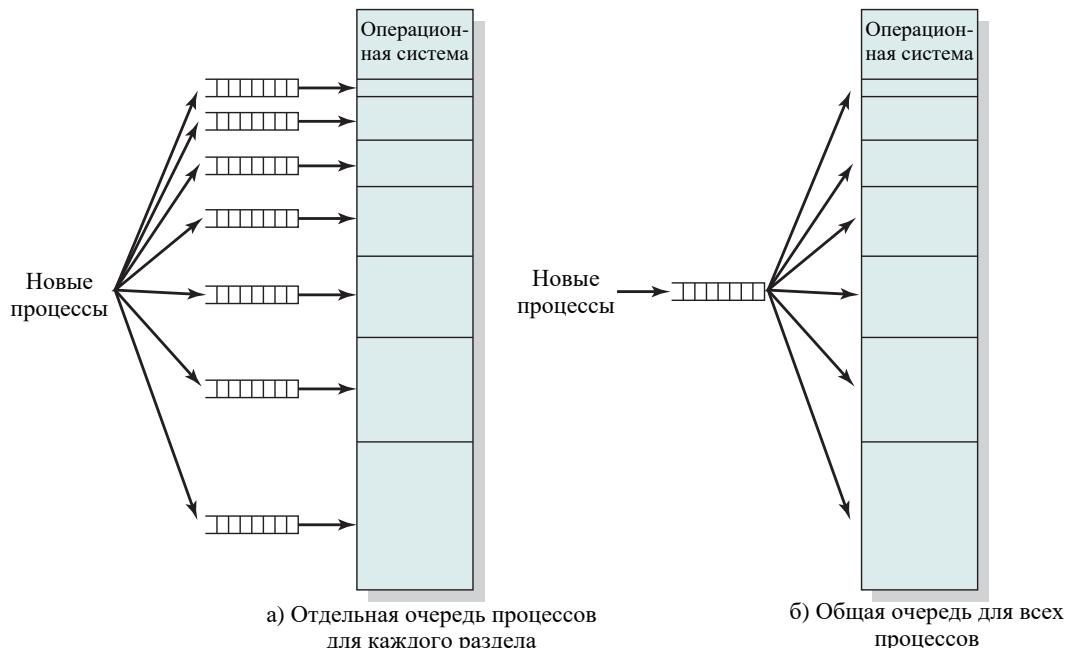
Когда разделы имеют разные размеры, есть два возможных подхода к назначению процессов разделам памяти. Простейший путь состоит в том, чтобы каждый процесс размещался в наименьшем разделе, способном полностью вместить данный процесс.<sup>1</sup> В таком случае для каждого раздела требуется очередь планировщика, в которой хранятся выгруженные из памяти процессы, предназначенные для данного раздела памяти (рис. 7.3, а). Преимущество такого подхода заключается в том, что процессы могут быть распределены между разделами памяти так, чтобы минимизировать внутреннюю фрагментацию.

Хотя этот метод представляется оптимальным с точки зрения отдельного раздела, он не оптимален с точки зрения системы в целом. Представим, что в системе, изображенной на рис. 7.2, б, в некоторый момент времени нет ни одного процесса размером от 12 до 16 Мбайт. В результате раздел размером 16 Мбайт будет пустовать, в то время как он мог бы с успехом использоваться меньшими процессами. Таким образом, более предпочтительным подходом является использование одной очереди для всех процессов (см. рис. 7.3, б). В момент, когда требуется загрузить процесс в основную память, для этого выбирается наименьший доступный раздел, способный вместить данный процесс. Если все разделы заняты, следует принять решение об освобождении одного из них. По-видимому, следует отдать предпочтение процессу, занимающему наименьший раздел, способный вместить загружаемый процесс. Можно учесть и другие факторы, такие как приоритет процесса или его состояние (заблокирован он или активен).

Использование разделов разного размера по сравнению с использованием разделов одинакового размера придает дополнительную гибкость данному методу. Кроме того, схемы с фиксированными разделами относительно просты, предъявляют минимальные требования к операционной системе; накладные расходы работы процессора невелики.

---

<sup>1</sup> При этом предполагается, что заранее известно, какое максимальное количество памяти может потребоваться процессу, однако это далеко не всегда так. Если максимальное количество необходимой процессу памяти неизвестно, следует использовать оверлеи или виртуальную память.



**Рис. 7.3.** Распределение памяти при фиксированном распределении

Однако у этих схем имеются серьезные недостатки.

- Количество разделов, определенное в момент генерации системы, ограничивает количество активных (не приостановленных) процессов.
- Поскольку размеры разделов устанавливаются заранее, в момент генерации системы, небольшие процессы приводят к неэффективному использованию памяти. В средах, в которых заранее известны потребности в памяти всех задач, применение описанной схемы может быть оправдано, но в большинстве случаев эффективность этой технологии крайне низка.

Фиксированное распределение в настоящее время практически не используется. Примером успешной операционной системы с использованием данной технологии может служить ранняя операционная система IBM для мейнфреймов OS/MFT (мультипрограммная с фиксированным количеством задач — Multiprogramming with a Fixed number of Tasks).

## Динамическое распределение

Для преодоления сложностей, связанных с фиксированным распределением, был разработан альтернативный подход, известный как динамическое распределение. Этот подход в настоящее время также вытеснен более сложными и эффективными технологиями управления памятью. В свое время динамическое распределение использовала операционная система IBM для мейнфреймов OS/MVT (мультипрограммная с переменным количеством задач — Multiprogramming with a Variable number of Tasks).

При динамическом распределении образуется переменное количество разделов переменной длины. При размещении процесса в основной памяти для него выделяется строго необходимое количество памяти, и не более того. В качестве примера рассмотрим

рим использование 64 Мбайт основной памяти (рис. 7.4). Изначально вся память пуста, за исключением области, используемой операционной системой (рис. 7.4, а). Первые три процесса загружаются в память, начиная с адреса, которым заканчивается операционная система, и используя ровно столько памяти, сколько требуется данному процессу (рис. 7.4, б-г). После этого в конце основной памяти остается “дыра”, слишком малая для размещения четвертого процесса. В некоторый момент все процессы в памяти оказываются неактивными, и операционная система выгружает второй процесс (рис. 7.4, д), после которого остается достаточно памяти для загрузки нового, четвертого процесса (рис. 7.4, е). Поскольку процесс 4 меньше процесса 2, создается еще одна небольшая “дыра” в памяти. После того как в некоторый момент времени все процессы в памяти оказываются неактивными, но зато готов к работе процесс 2, свободного места в памяти для него не находится, и операционная система вынуждена выгрузить процесс 1, чтобы освободить необходимое место (рис. 7.4, ж) и разместить процесс 2 в основной памяти (рис. 7.4, з).

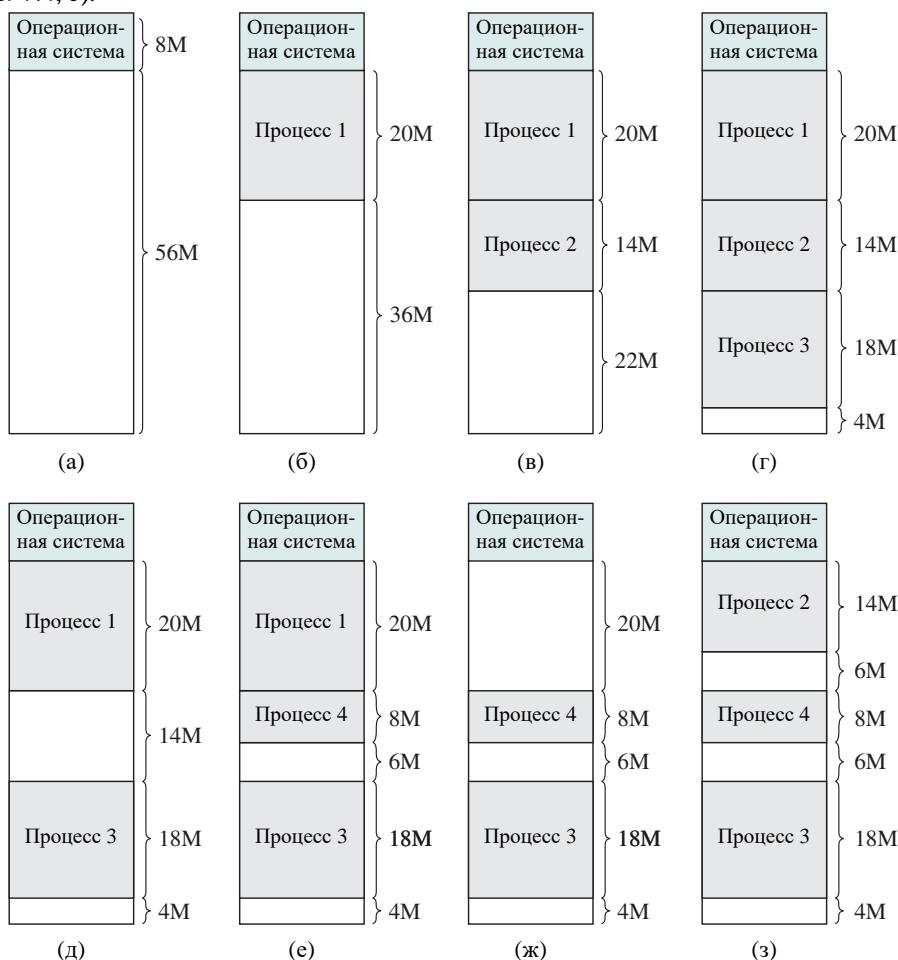


Рис. 7.4. Пример динамического распределения

Как показано в данном примере, этот метод хорошо начинает работу, но плохо продолжает — в конечном счете приводит к наличию множества мелких дыр в памяти. Со временем память становится все более и более фрагментированной и снижается эффективность ее использования. Это явление называется **внешней фрагментацией** (external fragmentation), что отражает тот факт, что сильно фрагментированной становится память, внешняя по отношению ко всем разделам (в отличие от рассмотренной ранее внутренней фрагментации).

Один из методов преодоления этого явления состоит в **уплотнении** (compaction): время от времени операционная система перемещает процессы в памяти так, чтобы они занимали смежные области памяти; свободная память при этом собирается в один блок. Например, на рис. 7.4,з после уплотнения памяти мы получим блок свободной памяти размером 16 Мбайт, чего может оказаться вполне достаточно для загрузки нового процесса. Сложность применения уплотнения состоит в том, что при этом расходуется дополнительное время; кроме того, уплотнение требует динамического перемещения процессов в памяти, т.е. должна быть обеспечена возможность перемещения программы из одной области основной памяти в другую без потери корректности ее обращений к памяти (см. приложение к данной главе).

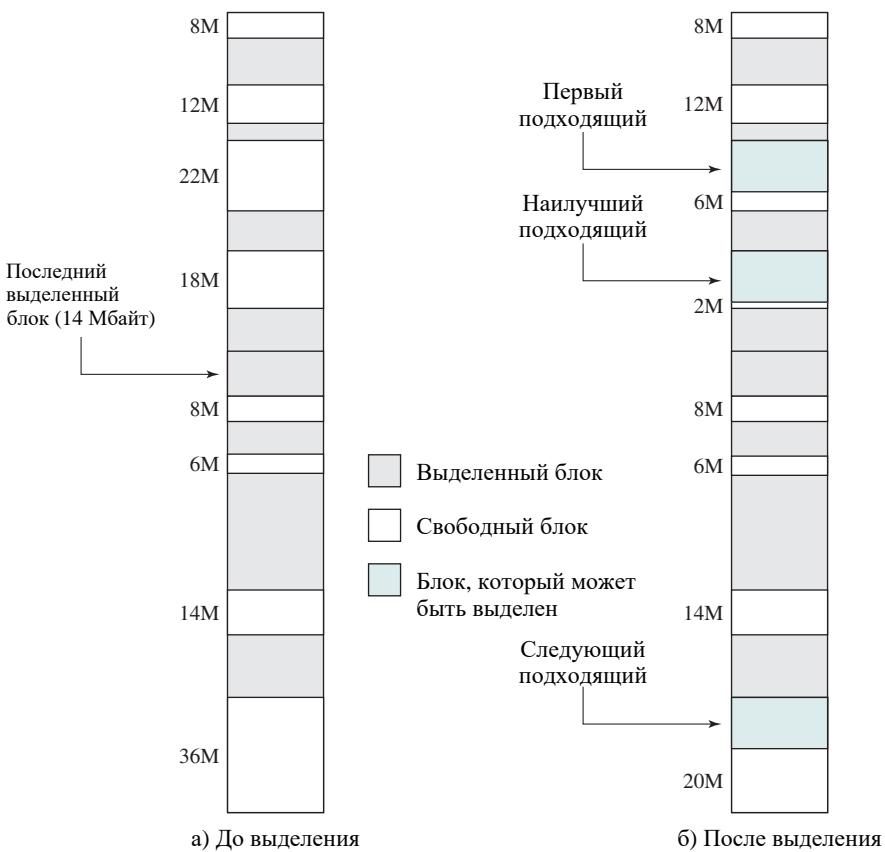
### Алгоритм размещения

Поскольку уплотнение памяти вызывает дополнительные расходы времени процессора, разработчик операционной системы должен принять разумное решение о том, каким образом размещать процессы в памяти (образно говоря, каким образом затыкать дыры). Когда наступает момент загрузки процесса в основную память и имеется несколько блоков свободной памяти достаточного размера, операционная система должна принять решение о том, какой именно свободный блок использовать.<sup>2</sup>

Можно рассматривать три основных алгоритма — наилучший подходящий, первый подходящий, следующий подходящий. Все они, само собой разумеется, ограничены выбором среди свободных блоков размера, достаточно большого для размещения процесса. Метод **наилучшего подходящего** выбирает блок, размер которого наиболее близок к требуемому; метод **первого подходящего** проверяет все свободные блоки с начала памяти и выбирает первый достаточный по размеру для размещения процесса. Метод **следующего подходящего** работает так же, как и метод первого подходящего, однако начинает проверку с того места, где был выделен блок в последний раз (по достижении конца памяти он продолжает работу с ее начала).

На рис. 7.5, а показан пример конфигурации памяти после ряда размещений и выгрузки процессов из памяти. Последним использованным блоком был блок размером 22 Мбайт, в котором был создан раздел в 14 Мбайт. На рис. 7.5, б показано различие в технологии наилучшего, первого и следующего подходящего при выполнении запроса на выделение блока размером 16 Мбайт. Метод наилучшего подходящего просматривает все свободные блоки и выбирает наиболее близкий по размеру блок в 18 Мбайт, оставляя фрагмент размером 2 Мбайт. Метод первого подходящего в данной ситуации оставляет фрагмент свободной памяти размером 6 Мбайт, а метод следующего подходящего — 20 Мбайт.

<sup>2</sup> Дополнительный материал по вопросам динамического распределения памяти читатель может найти в разделе 2.5 (с. 488) книги Клут Д. Э. *Искусство программирования. Том 1. Основные алгоритмы*, 3-е изд. — М.: Издательский дом “Вильямс”, 2000. — Примеч. ред.



**Рис. 7.5.** Пример конфигурации памяти до и после выделения блока размером 16 Мбайт

Какой из этих методов окажется наилучшим, будет зависеть от точной последовательности загрузки и выгрузки процессов и их размеров. Однако можно говорить о некоторых обобщенных выводах (см. [19], [28], [228]). Обычно алгоритм первого подходящего не только проще, но и быстрее и дает лучшие результаты. Алгоритм следующего подходящего, как правило, дает немного худшие результаты. Это связано с тем, что алгоритм следующего подходящего проявляет склонность к более частому выделению памяти из свободных блоков в конце памяти. В результате самые большие блоки свободной памяти (которые обычно располагаются в конце памяти) быстро разбиваются на меньшие фрагменты и, следовательно, при использовании метода следующего подходящего уплотнение должно выполняться чаще. С другой стороны, алгоритм первого подходящего обычно засоряет начало памяти небольшими свободными блоками, что приводит к увеличению времени поиска подходящего блока в последующем. Метод наилучшего подходящего, вопреки своему названию, оказывается, как правило, наихудшим. Так как он ищет блоки, наиболее близкие по размеру к требуемому, он оставляет после себя множество очень маленьких блоков. В результате, хотя при каждом выделении впустую тратится наименьшее возможное количество памяти, основная память очень быстро засоряется множеством мелких блоков, неспособных удовлетворить ни один запрос (так что при этом алгоритме уплотнение памяти должно выполняться значительно чаще).

## Алгоритм замещения

В многозадачной системе с использованием динамического распределения наступает момент, когда все процессы в основной памяти находятся в заблокированном состоянии, а памяти для дополнительного процесса недостаточно даже после уплотнения. Чтобы избежать потерь процессорного времени на ожидание деблокирования активного процесса, операционная система может выгрузить один из процессов из основной памяти и, таким образом, освободить место для нового процесса или процесса в состоянии готовности. Задача операционной системы — определить, какой именно процесс должен быть выгружен из памяти. Поскольку тема алгоритма замещения будет детально рассматриваться в связи с различными схемами виртуальной памяти, пока что мы отложим обсуждение этого вопроса.

## Система двойников

Как фиксированное, так и динамическое распределение памяти имеют свои недостатки. Фиксированное распределение ограничивает количество активных процессов и неэффективно использует память при несоответствии между размерами разделов и процессов. Динамическое распределение реализуется более сложно и включает накладные расходы по уплотнению памяти. Интересным компромиссом в этом плане является система двойников ([136], [191]).

В системе двойников память распределяется блоками размером  $2^K$ ,  $L \leq K \leq U$ , где

$2^L$  — минимальный размер выделяемого блока памяти;

$2^U$  — наибольший распределяемый блок; вообще говоря,  $2^U$  представляет собой размер всей доступной для распределения памяти.

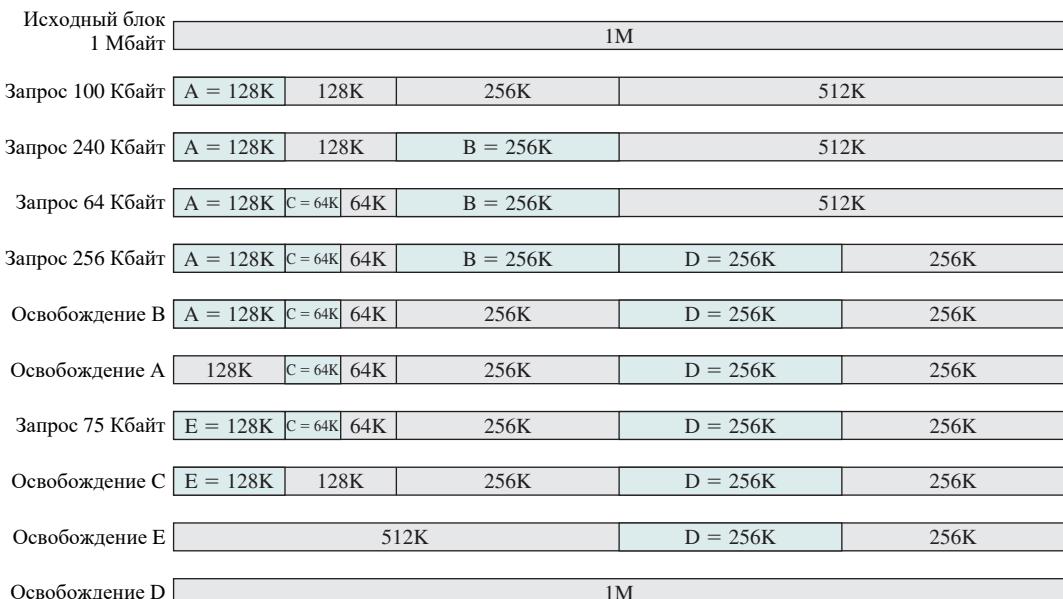
Вначале все доступное для распределения пространство рассматривается как единый блок размером  $2^U$ . При запросе размером  $s$ , таким, что  $2^{U-1} < s \leq 2^U$ , выделяется весь блок. В противном случае блок разделяется на два эквивалентных двойника с размерами  $2^{U-1}$ . Если  $2^{U-2} < s \leq 2^{U-1}$ , то по запросу выделяется один из двух двойников; в противном случае один из двойников вновь делится пополам. Этот процесс продолжается до тех пор, пока не будет сгенерирован наименьший блок, размер которого не меньше  $s$ . Система двойников постоянно ведет список “дыр” (доступных блоков) для каждого размера  $2^i$ . Дыра может быть удалена из списка ( $i+1$ ) разделением ее пополам и внесением двух новых дыр размером  $2^i$  в список  $i$ . Когда пара двойников в списке  $i$  оказывается освобожденной, они удаляются из списка и объединяются в единый блок в списке ( $i+1$ ). Ниже приведен рекурсивный алгоритм для удовлетворения запроса размером  $2^{i-1} < k \leq 2^i$ , в котором осуществляется поиск дыры размером  $2^i$ .

```
void get_hole(int i)
{
    if (i == (U+1)) < Ошибка >;
    if (< Список i пуст >)
    {
        get_hole(i+1);
        < Разделить дыру на двойники >;
        < Поместить двойники в список i >;
    }
    < Взять первую дыру из списка i >;
}
```

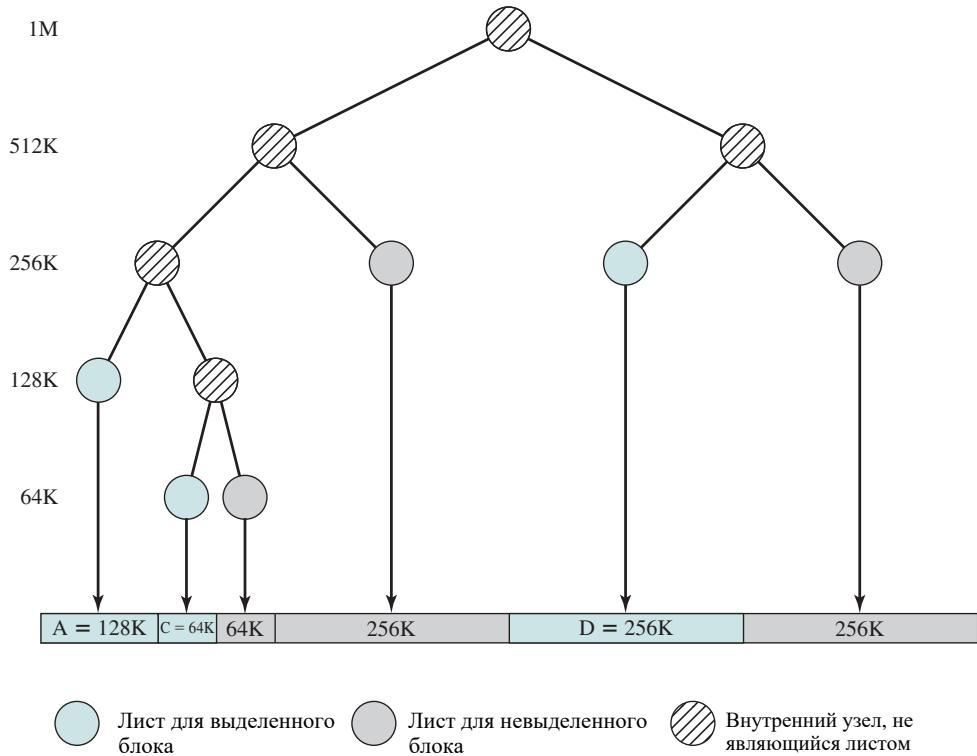
На рис. 7.6 приведен пример использования блока с начальным размером 1 Мбайт. Первый запрос А — на 100 Кбайт (для него требуется блок размером 128 Кбайт). Для этого начальный блок делится на два двойника по 512 Кбайт. Первый из них делится на двойники размером 256 Кбайт, и, в свою очередь, первый из получившихся при этом разделении двойников также делится пополам. Один из получившихся двойников размером 128 Кбайт выделяется запросу А. Следующий запрос В требует 256 Кбайт. Такой блок имеется в наличии и выделяется. Процесс продолжается с разделением и слиянием двойников при необходимости. Обратите внимание, что после освобождения блока Е происходит слияние двойников по 128 Кбайт в один блок размером 256 Кбайт, который, в свою очередь, тут же сливается со своим двойником.

На рис. 7.7 показано представление системы двойников в виде бинарного дерева, непосредственно после освобождения блока В. Листья представляют текущее распределение памяти. Если два двойника являются листьями, то по крайней мере один из них занят; в противном случае они должны слиться в блок большего размера.

Система двойников представляет собой разумный компромисс для преодоления недостатков схем фиксированного и динамического распределения, но в современных операционных системах ее превосходит виртуальная память, основанная на страничной организации и сегментации. Однако система двойников нашла применение в параллельных системах как эффективное средство распределения и освобождения параллельных программ (см., например, [118]). Модифицированная версия системы двойников используется для распределения памяти ядром UNIX (подробнее об этом вы узнаете в главе 8, “Виртуальная память”).



**Рис. 7.6.** Пример системы двойников



**Рис. 7.7.** Представление системы двойников в виде бинарного дерева

## Перемещение

Перед тем как мы рассмотрим способы, с помощью которых можно избежать недостатков распределения, следует до конца разобраться в вопросах, связанных с размещением процессов в памяти. При использовании фиксированной схемы распределения, показанной на рис. 7.3, *а*, можно ожидать, что процесс всегда будет назначаться одному и тому же разделу памяти. Это означает, что какой бы раздел ни был выбран для нового процесса, для размещения этого процесса после выгрузки и последующей загрузки в память всегда будет использоваться именно этот раздел. В данном случае можно использовать простейший загрузчик, описанный в приложении к данной главе: при загрузке процесса все относительные ссылки в коде замещаются абсолютными адресами памяти, определенными на основе базового адреса загруженного процесса.

Если размеры разделов равны (см. рис. 7.2) и существует единая очередь процессов для разделов разного размера (см. рис. 7.3, *б*), процесс по ходу работы может занимать разные разделы. При первом создании образа процесса он загружается в некоторый раздел памяти; позже, после того как он был выгружен из памяти и вновь загружен, процесс может оказаться в другом разделе (не в том, в котором размещался в последний раз). Та же ситуация возможна и при динамическом распределении. Так, на рис. 7.4, *в* и 7.4, *з* процесс 2 занимает при размещении в памяти различные места. Кроме того, при выполнении уплотнения процессы также перемещаются в основной памяти. Таким образом, расположение команд и данных, к которым обращается процесс, не является

фиксированным и изменяется всякий раз при выгрузке и загрузке (или перемещении) процесса. Для решения этой проблемы следует различать типы адресов. **Логический адрес** представляет собой ссылку на ячейку памяти, не зависящую от текущего расположения данных в памяти; перед тем как получить доступ к этой ячейке памяти, необходимо транслировать логический адрес в физический. **Относительный адрес** представляет собой частный случай логического адреса, когда адрес определяется положением относительно некоторой известной точки (обычно — начала программы). **Физический адрес** (известный также как абсолютный) представляет собой действительное расположение интересующей нас ячейки основной памяти.

Программа, которая использует относительные адреса в памяти, загружается с помощью динамической загрузки времени выполнения (см. приложение к данной главе). Обычно все ссылки на память в загруженном процессе даны относительно начала этой программы. Таким образом, для корректной работы программы требуется аппаратный механизм, который транслировал бы относительные адреса в физические в процессе выполнения команды, которая обращается к памяти.

На рис. 7.8 показан обычно используемый способ трансляции адреса. Когда процесс переходит в состояние выполнения, в специальный регистр процессора, иногда называемый базовым, загружается начальный адрес процесса в основной памяти. Кроме того, используется “границный” (bounds) регистр, в котором содержится адрес последней ячейки памяти программы. Эти значения заносятся в регистры при загрузке программы в основную память.

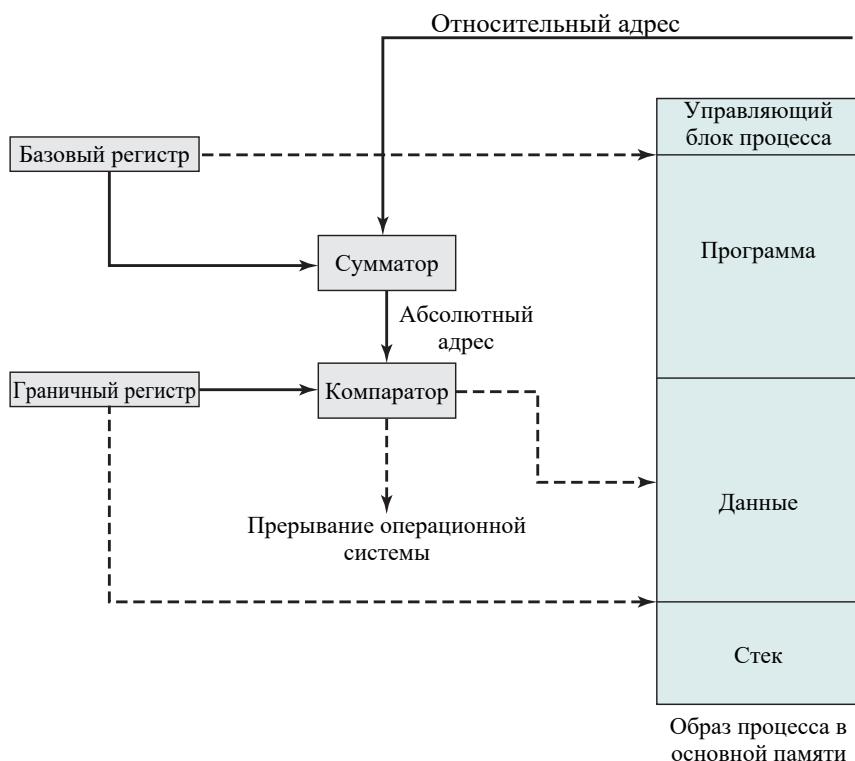


Рис. 7.8. Аппаратная поддержка перемещения

При выполнении процесса встречающиеся в командах относительные адреса обрабатываются процессором в два этапа. Сначала к относительному адресу прибавляется значение базового регистра для получения абсолютного адреса. Затем полученный абсолютный адрес сравнивается со значением в граничном регистре. Если полученный абсолютный адрес принадлежит данному процессу, команда может быть выполнена; в противном случае генерируется соответствующее данной ошибке прерывание операционной системы.

Схема, представленная на рис. 7.8, обеспечивает возможность выгрузки и загрузки программ в основную память в процессе их выполнения; кроме того, образ каждого процесса ограничен адресами, содержащимися в базовом и граничном регистрах, и, таким образом, защищен от нежелательного доступа со стороны других процессов.

## 7.3. СТРАНИЧНАЯ ОРГАНИЗАЦИЯ ПАМЯТИ

Как разделы с разными фиксированными размерами, так и разделы переменного размера недостаточно эффективно используют память. Результатом работы первых становится внутренняя фрагментация, результатом работы последних — внешняя. Предположим, однако, что основная память разделена на одинаковые блоки относительно небольшого фиксированного размера. Тогда блоки процесса, известные как **страницы** (pages), могут быть связаны со свободными блоками памяти, известными как **кадры** (frames) или **фреймы**. Каждый кадр может содержать одну страницу данных. При такой организации памяти, о которой вы узнаете из этого раздела, внешняя фрагментация отсутствует вовсе, а потери из-за внутренней фрагментации ограничены частью последней страницы процесса.

На рис. 7.9 показано использование страниц и кадров. В любой момент времени некоторые из кадров памяти используются, а некоторые свободны. Операционная система поддерживает список свободных кадров. Процесс A, хранящийся на диске, состоит из четырех страниц. Когда приходит время загрузить этот процесс в память, операционная система находит четыре свободных кадра и загружает страницы процесса A в эти кадры (рис. 7.9, б). Затем загружаются процесс B, состоящий из трех страниц, и процесс C, состоящий из четырех страниц. После этого процесс B приостанавливается и выгружается из основной памяти. Позже наступает момент, когда все процессы в памяти оказываются заблокированными, и операционная система загружает в память новый процесс D, состоящий из пяти страниц.

Теперь предположим, что, как в только что рассмотренном выше примере, не имеется одной непрерывной области кадров, достаточной для размещения процесса целиком. Помешает ли это операционной системе загрузить процесс D? Нет, поскольку в этой ситуации можно воспользоваться концепцией логических адресов. Однако одного регистра базового адреса в этой ситуации недостаточно, и для каждого процесса операционная система должна поддерживать **таблицу страниц**. Таблица страниц указывает расположение кадров каждой страницы процесса. Внутри программы логический адрес состоит из номера страницы и смещения внутри нее. Вспомним, что в случае простого распределения логический адрес представляет собой расположение слова относительно начала программы, которое процессор транслирует в физический адрес. При страничной организации преобразование логических адресов в физические также остается задачей аппаратного уровня, решаемой процессором.

Номер кадра	Основная память
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	

а) 15 доступных кадров

Основная память
A.0
A.1
A.2
A.3

б) Загрузка процесса А

Основная память
A.0
A.1
A.2
A.3

в) Загрузка процесса В

Основная память
A.0
A.1
A.2
A.3

г) Загрузка процесса С

Основная память
A.0
A.1
A.2
A.3

д) Выгрузка процесса В

Основная память
A.0
A.1
A.2
A.3

е) Загрузка процесса D

**Рис. 7.9. Распределение страниц по свободным кадрам**

Теперь процессор должен иметь информацию о том, где находится таблица страниц текущего процесса. Представленный логический адрес (номер страницы и смещение) процессор превращает с использованием таблицы страниц в физический адрес (номер кадра, смещение).

На рис. 7.10 показаны различные таблицы страниц, после того как процесс D оказывается загруженным в страницы 4, 5, 6, 11 и 12. Таблица страниц содержит по одной записи для каждой страницы процесса, так что таблицу легко проиндексировать номером страницы, начиная с 0. Каждая запись содержит номер фрейма в основной памяти (если таковой имеется), в котором хранится соответствующая страница. Кроме того, операционная система поддерживает единый список свободных (т.е. не занятых никаким процессом и доступных для размещения в них страниц) кадров.

0	0
1	1
2	2
3	3

### Таблица страниц процесса А

0	—
1	—
2	—

## Таблица страниц процесса В

0	7
1	8
2	9
3	10

## Таблица страниц процесса С

0	4
1	5
2	6
3	11
4	12

## Таблица страниц процесса D

14

## Список свободных кадров

**Рис. 7.10.** Структуры данных, соответствующие примеру на рис. 7.9, е

Таким образом, описанная здесь простая страничная организация подобна фиксированному распределению. Отличия заключаются в достаточно малом размере разделов, которые к тому же могут не быть смежными.

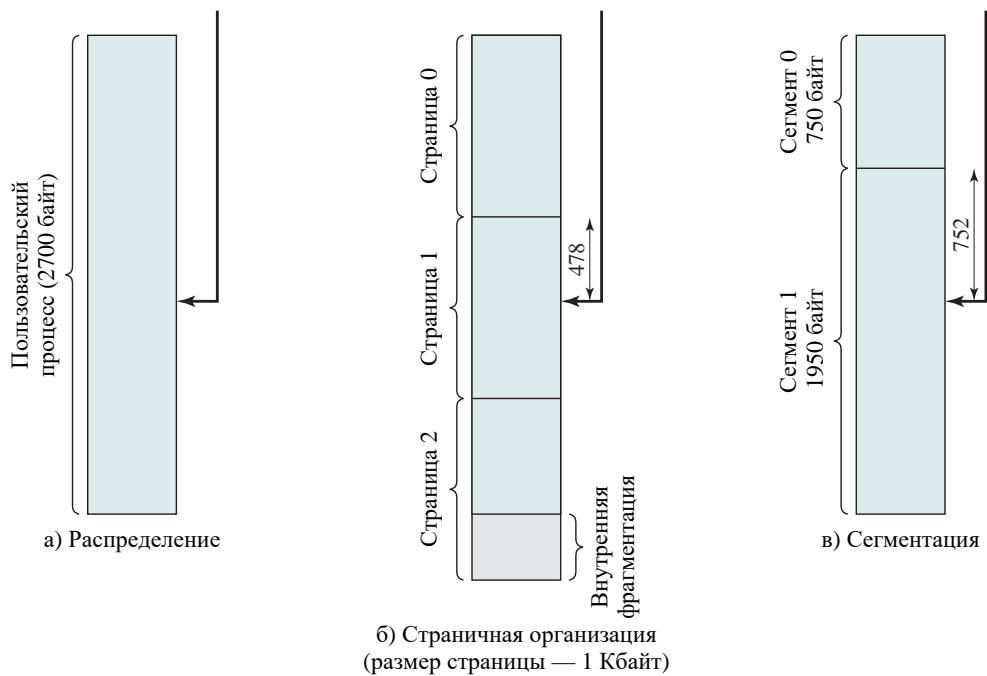
Для удобства работы с такой схемой добавим правило, в соответствии с которым размер страницы (а следовательно, и размер кадра) должен представлять собой степень 2. При использовании такого размера страниц легко показать, что относительный адрес, который определяется относительно начала программы, и логический адрес, представляющий собой номер кадра и смещение, идентичны. Соответствующий пример приведен на рис. 7.11. Здесь используются 16-битный адрес и страницы размером 1 Кбайт = 1024 байт. Относительный адрес 1502 в бинарном виде записывается как 0000010111011110. При размере страницы в 1 Кбайт поле смещения требует 10 бит, оставляя 6 бит для номера страницы.

Относительный  
адрес = 1502  
000010111011110

Логический адрес =  
Страница 1, Смещение = 478  

0	0	0	0	0	1	0	1	1	0	1	1	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---

Логический адрес =  
Сегмент 1, Смещение = 752  
 $\boxed{000100101110000}$



**Рис. 7.11.** Логические адреса

Таким образом, программа может состоять максимум из  $2^6 = 64$  страниц по 1 Кбайт каждая. Как показано на рис. 7.11, б, относительный адрес 1502 соответствует смещению 478 (0111011110) на странице 1 (000001), что дает то же 16-битное число 0000010111011110.

Использование страниц с размером, равным степени двойки, приводит к таким следствиям. Во-первых, схема логической адресации прозрачна для программиста, ассемблера и компоновщика. Каждый логический адрес (номер страницы и смещение) программы идентичен относительному адресу. Во-вторых, при этом относительно просто реализуется аппаратная функция преобразования адресов во время работы. Рассмотрим адрес из  $n+m$  бит, где крайние слева  $n$  бит представляют собой номер страницы, а крайние справа  $m$  бит — смещение. В нашем примере (рис. 7.11, б)  $n = 6$  и  $m = 10$ . Для преобразования адреса необходимо выполнить следующие шаги.

1. Выделить номер страницы, который представлен  $n$  левыми битами логического адреса.
2. Используя номер страницы в качестве индекса в таблице страниц процесса, найти номер кадра  $k$ .
3. Начальный физический адрес кадра —  $k \cdot 2^m$ , и интересующий нас физический адрес представляет собой это число плюс смещение. Такой адрес не надо вычислять — он получается в результате простого добавления номера кадра к смещению.

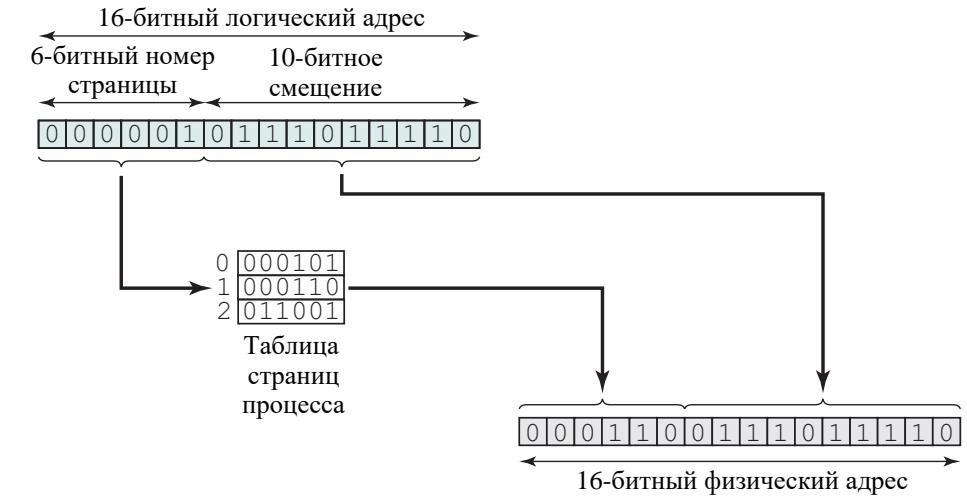
В нашем примере имеется логический адрес 0000010111011110, представляющий страницу номер 1 и смещение 478. Предположим, что эта страница размещена в кадре основной памяти номер 6 (бинарное представление — 000110). В таком случае физический адрес представляет собой кадр 6, смещение 478, т.е. 0001100111011110 (рис. 7.12, а).

Итак, в случае простой страничной организации основная память разделяется на множество небольших кадров одинакового размера. Каждый процесс разделяется на страницы того же размера, что и кадры; малые процессы требуют меньшего количества кадров, большие — большего. При загрузке процесса в память все его страницы загружаются в свободные кадры, и информация о размещении страниц заносится в соответствующую таблицу. Такой подход позволяет избежать множества присущих распределению памяти проблем.

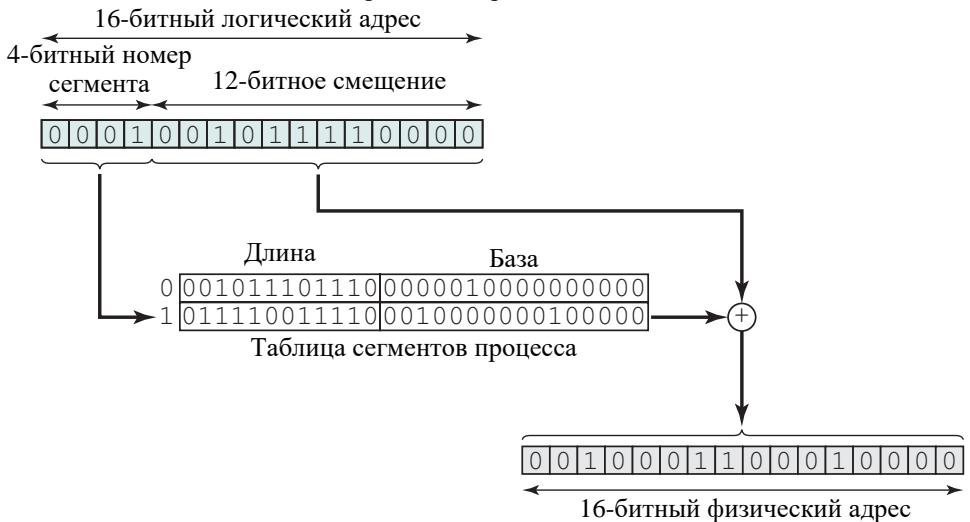
## 7.4. СЕГМЕНТАЦИЯ

Альтернативным способом распределения пользовательской программы является сегментация. В этом случае программа и связанные с ней данные разделяются на ряд **сегментов**. Хотя и существует максимальный размер сегмента, на сегменты не накладывается условие равенства размеров. Как и при страничной организации, логический адрес состоит из двух частей, в данном случае — номера сегмента и смещения.

Использование сегментов разного размера делает этот способ похожим на динамическое распределение памяти. Если не используются оверлеи и виртуальная память, то для выполнения программы все ее сегменты должны быть загружены в память; однако в отличие от динамического распределения в этом случае сегменты могут занимать несколько разделов, которые к тому же могут не быть смежными.



а) Страницчная организация



б) Сегментация

Рис. 7.12. Примеры трансляции логических адресов в физические

При сегментации устраняется внутренняя фрагментация, однако, как и при динамическом распределении, наблюдается внешняя фрагментация. Тем не менее ее степень снижается, в силу того что процесс разбивается на ряд небольших частей.

В то время как страницчная организация невидима для программиста, сегментация, как правило, видима и обычно используется при размещении кода и данных в разных сегментах. При использовании принципов модульного программирования как код, так и данные могут быть дополнительно разбиты на сегменты. Главным недостатком при работе с сегментами является необходимость заботиться о том, чтобы размер сегмента не превысил максимальный.

Еще одно следствие того, что сегменты имеют разные размеры, состоит в отсутствии простой связи между логическими и физическими адресами. Аналогично страничной организации схема простой сегментации использует таблицу сегментов для каждого процесса и список свободных блоков основной памяти. Каждая запись таблицы сегментов должна содержать стартовый адрес сегмента в основной памяти и его длину, чтобы обезопасить систему от использования некорректных адресов. При работе процесса адрес его таблицы сегментов заносится в специальный регистр, используемый аппаратным обеспечением управления памятью. Рассмотрим адрес из  $n+m$  бит, где крайние слева  $n$  бит являются номером сегмента, а правые  $m$  бит — смещением. В примере, показанном на рис. 7.11,  $b$ ,  $n = 4$  и  $m = 12$ . Таким образом, максимальный размер сегмента составляет  $2^{12} = 4096$  байт. Для трансляции адреса необходимо выполнить следующие действия.

1. Выделить из логического адреса  $n$  крайних слева битов, получив таким образом номер сегмента.
2. Используя номер сегмента в качестве индекса в таблице сегментов процесса, найти физический адрес начала сегмента.
3. Сравнить смещение, представляющее собой крайние справа  $m$  бит, с длиной сегмента. Если смещение больше длины, адрес некорректен.
4. Требуемый физический адрес представляет собой сумму физического адреса начала сегмента и смещения.

В нашем примере имеется логический адрес 0001001011110000, представляющий собой сегмент номер 1, смещение 752. Предположим, что этот сегмент располагается в основной памяти начиная с физического адреса 001000000100000. Тогда интересующий нас физический адрес равен  $001000000100000 + 001011110000 = 0010001100010000$  (см. рис. 7.12,  $b$ ).

Итак, в случае простой сегментации процесс разделяется на ряд сегментов, размер которых может быть разным. При загрузке процесса все его сегменты размещаются в свободных областях памяти и соответствующая информация вносится в таблицу сегментов.

## 7.5. Резюме

Одной из наиболее важных и сложных задач операционной системы является управление памятью. Основная память может рассматриваться как ресурс, который распределяется и совместно используется рядом активных процессов. Для эффективного использования процессора и устройств ввода-вывода желательно размещение в основной памяти максимально возможного количества процессов. Кроме того, желательно дать программисту возможность разрабатывать программы без ограничений их размера.

Основными инструментами управления памятью являются страничная организация и сегментация. При страничной организации каждый процесс разделяется на относительно малые страницы фиксированного размера; сегментация позволяет использовать части разного размера. Кроме того, возможно комбинирование сегментации и страничной организации в единой схеме управления памятью.

## 7.6. КЛЮЧЕВЫЕ ТЕРМИНЫ, КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАЧИ

### Ключевые термины

Абсолютная загрузка	Кадр	Совместное использование
Внешняя фрагментация	Логическая организация	Страница
Внутренняя фрагментация	Логический адрес	Страницчная организация
Динамическая загрузка времени выполнения	Относительный адрес	Таблица страниц
Динамическое распределение	Перемещаемая загрузка	Уплотнение
Динамическое связывание	Редактор связей	Управление памятью
Загрузка	Связывание	Физическая организация
Защита	Сегмент	Физический адрес
	Сегментация	Фиксированное распределение
	Система двойников	

### Контрольные вопросы

- 7.1. Каким требованиям должно удовлетворять управление памятью?
- 7.2. Почему желательно обеспечить возможность перемещения процессов?
- 7.3. Почему невозможно обеспечить защиту памяти во время компиляции программы?
- 7.4. По каким причинам может потребоваться обеспечение доступа к одной области памяти нескольким процессам?
- 7.5. В чем состоит преимущество использования разделов разного размера при использовании схемы фиксированного распределения?
- 7.6. В чем состоит различие между внутренней и внешней фрагментацией?
- 7.7. В чем заключается различие между логическим, относительным и физическим адресами?
- 7.8. В чем разница между страницей и кадром?
- 7.9. В чем разница между страницей и сегментом?

## Задачи

- 7.1. В разделе 2.3 были перечислены пять целей управления памятью, а в разделе 7.1 — пять требований. Обоснуйте взаимосвязанность этих списков.
- 7.2. Рассмотрим схему фиксированного распределения с разделами равного размера, равного  $2^{16}$  байт, и общим количеством основной памяти  $2^{24}$  байт. Поддерживается таблица процессов, включающая указатель на раздел для каждого резидентного процесса. Сколько битов требуется для этого указателя?
- 7.3. Рассмотрите схему динамического распределения. Покажите, что в среднем количество свободных блоков памяти (“дыр”) в два раза меньше количества выделенных процессам разделов.
- 7.4. Для реализации различных алгоритмов распределения, обсуждавшихся при рассмотрении динамического распределения (раздел 7.2), необходима поддержка списка свободных блоков памяти. Какова средняя продолжительность поиска для каждого из рассмотренных методов (наилучшего, первого и следующего подходящего)?
- 7.5. Рассмотрите еще один алгоритм размещения при динамическом распределении — метод наихудшего подходящего, при котором для размещения процесса используется наибольший свободный блок памяти.
- Каковы его преимущества и недостатки по сравнению с другими рассмотренными методами (наилучшего подходящего, первого подходящего, следующего подходящего)?
  - Какова средняя длина поиска при этом методе?
- 7.6. Ниже показан пример конфигурации памяти при динамическом распределении после того, как был проведен ряд операций размещения и выгрузки. Адреса идут слева направо; серые области указывают блоки, занятые процессами; белые области указывают свободные блоки памяти. Последним был размещен процесс, занявший 2 Мбайт памяти и помеченный как X. После этого был выгружен только один процесс.



- Каким был максимальный размер выгруженного процесса?
  - Каким был размер свободного блока непосредственно перед тем, как он был распределен процессом X?
  - Далее должен быть удовлетворен новый запрос на выделение 3 Мбайт памяти. Укажите интервал памяти, в котором будет создан раздел для нового процесса при использовании каждого из четырех алгоритмов размещения: наилучшего подходящего, первого подходящего, следующего подходящего и наихудшего подходящего. Для каждого алгоритма подчеркните соответствующий отрезок под полосой памяти и пометьте его.
- 7.7. Система двойников используется для распределения блока размером 1 Мбайт.
- Изобразите в виде, подобном приведенному на рис. 7.6, результат выполнения такой последовательности запросов: запрос A — 70 Кбайт, запрос B — 35 Кбайт, запрос C — 80 Кбайт, освобождение A, запрос D — 60 Кбайт, освобождение B, освобождение D, освобождение C.

6. Покажите представление системы двойников в виде бинарного дерева после освобождения В.
- 7.8. Рассмотрим систему двойников, в которой некий только что выделенный блок имеет адрес 011011110000.
- Если размер блока равен 4, то каков бинарный адрес его двойника?
  - Если размер блока равен 16, то каков бинарный адрес его двойника?
- 7.9. Пусть  $buddy_k(x)$  — адрес того блока размером  $2^k$ , который является двойником блока по адресу  $x$ . Запишите выражение для  $buddy_k(x)$ .
- 7.10. Последовательность Фибоначчи определяется следующим образом:
- $$F_0 = 0, F_1 = 1, \dots, F_{n+2} = F_{n+1} + F_n, n \geq 0$$
- Можно ли использовать эту последовательность для разработки системы двойников?
  - Каким преимуществом должна обладать такая система двойников по сравнению с описанной в данной главе?
- 7.11. Во время выполнения программы процессор увеличивает содержимое регистра команд (счетчика кода) на одно слово после выборки каждой команды. Однако если встречаются команды ветвления или вызова подпрограммы, то содержимое данного регистра вызывает продолжение выполнения в некотором другом месте программы. Рассмотрим рис. 7.8. Имеются две альтернативы.
- В регистре команд использовать относительный адрес и выполнять трансляцию адресов динамически, с содержимым регистра команд в качестве входных данных. Команды ветвления или вызова подпрограммы при успешном выполнении генерируют относительный адрес, который и заносится в регистр команд.
  - В регистре команд содержать абсолютный адрес. В этом случае команды ветвления или вызова подпрограммы используется динамическая трансляция адреса и результат трансляции заносится в регистр команд.
- Какой подход предпочтительнее?
- 7.12. Рассмотрим простую страницочную систему со следующими параметрами:  $2^{32}$  байт физической памяти; размер страниц —  $2^{10}$  байт;  $2^{16}$  страниц логического адресного пространства.
- Сколько битов в логическом адресе?
  - Сколько байт в кадре?
  - Сколько битов физического адреса определяет кадр?
  - Сколько записей в таблице страниц?
  - Сколько битов в каждой записи таблицы страниц? Предполагаем, что каждая запись таблицы страниц содержит бит корректности страницы.
- 7.13. Запишите бинарную трансляцию логического адреса 0001010010111010 при следующих гипотетических схемах управления памятью и объясните свой ответ.
- Страницочная организация памяти с размером страницы, равным 256 адресам, с использованием таблицы страниц, в которой номер кадра в четыре раза меньше номера страницы.

б. Система сегментации с максимальным размером сегмента 1К адресов, с использованием таблицы сегментов, в которой базы располагаются регулярно по реальным адресам  $22 + 4096 \times$  номер сегмента..

**7.14.** Рассмотрим простую систему сегментации, при которой используется следующая таблица сегментов.

Начальный адрес	Длина (байты)
660	248
1752	422
222	198
996	604

Для каждого из следующих логических адресов определите физический адрес или укажите, что заданный адрес ошибочен.

- а. 0, 198
- б. 2, 156
- в. 1, 530
- г. 3, 444
- д. 0, 222

**7.15.** Рассмотрим память, в которой смежные сегменты  $S_1, S_2, \dots, S_n$  размещаются в порядке их создания от одного конца памяти к другому, как показано ниже.



При создании сегмента  $S_{n+1}$  он размещается непосредственно после сегмента  $S_n$ , даже если некоторые из сегментов  $S_1, S_2, \dots, S_n$  были к этому моменту удалены. Когда граница между сегментами (используемыми или удаленными) и свободной памятью достигает конца памяти, используемые сегменты уплотняются.

- а. Покажите, что доля времени  $F$ , затрачиваемая на уплотнение, удовлетворяет следующему неравенству:

$$F \geq \frac{1-f}{1+kf}, \text{ где } k = \frac{t}{2s} - 1$$

Здесь

$s$  — средняя длина сегмента в словах,

$t$  — среднее время жизни сегмента (в обращениях к памяти),

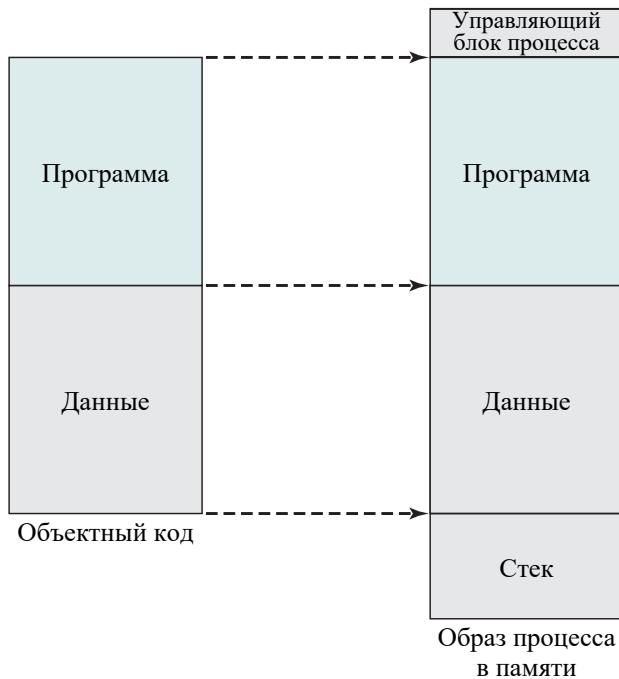
$f$  — доля памяти, не используемая в установленныхся условиях.

**Указание:** найдите среднюю скорость, с которой рассматриваемая граница движется по памяти, и предположите, что копирование одного слова требует как минимум двух обращений к памяти.

- б. Определите  $F$  для  $f = 0,2$ ,  $t = 1000$  и  $s = 50$ .

## Приложение 7.А. Загрузка и связывание

Первым шагом в создании активного процесса является загрузка программы в основную память и создание образа процесса (рис. 7.13). На рис. 7.14 показан типичный для большинства систем сценарий. Приложение состоит из ряда скомпилированных или ассемблированных модулей в виде объектного кода. Эти модули связываются для разрешения всех ссылок между ними, а также обращений к библиотечным подпрограммам (которые могут быть внедрены в программу или быть совместно используемым кодом, представляемым операционной системой). В этом приложении мы познакомимся с ключевыми свойствами компоновщиков и загрузчиков. Для ясности изложения мы начнем с описания задачи загрузки программы, состоящей из одного модуля, когда связывание не требуется.



**Рис. 7.13.** Функция загрузки

## Загрузка

На рис. 7.14 загрузчик размещает загружаемый модуль в основной памяти, начиная с адреса  $x$ . При этом должны удовлетворяться требования к адресации процессса, приведенные на рис. 7.1. В общем случае могут использоваться три подхода.

- Абсолютная загрузка
- Перемещаемая загрузка
- Динамическая загрузка времени выполнения

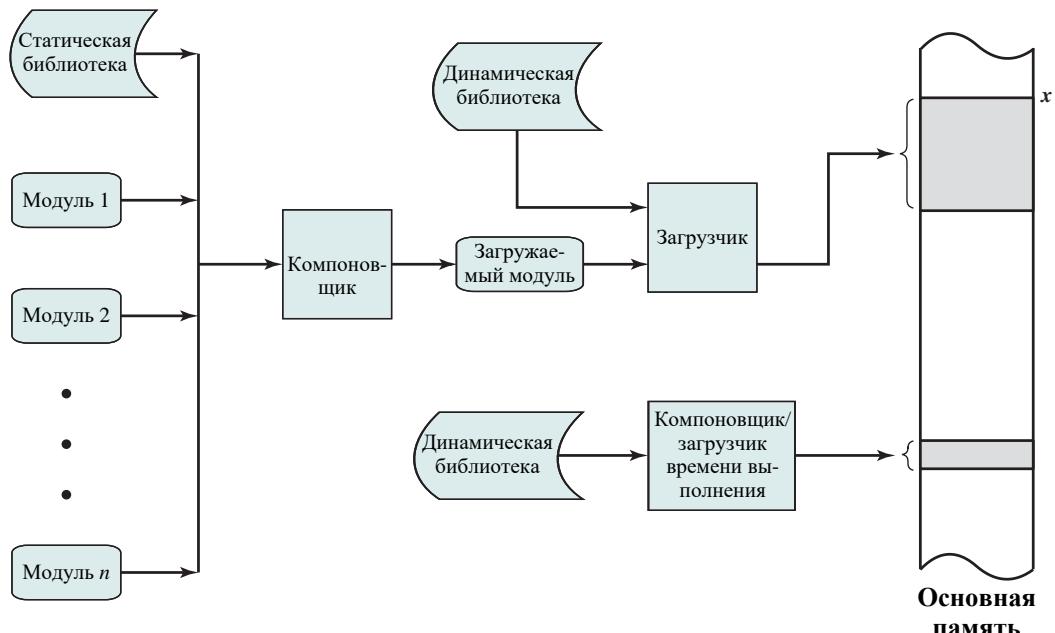


Рис. 7.14. Сценарий загрузки и связывания

### Абсолютная загрузка

Абсолютный загрузчик требует, чтобы данный загружаемый модуль всегда располагался в одном и том же месте в памяти. Следовательно, в модуле, передаваемом для загрузки, все обращения должны быть к конкретным, или абсолютным, адресам основной памяти. Например, если *x* на рис. 7.14 представляет собой ячейку памяти 1024, то первое слово в загружаемом модуле, предназначенном для данной области памяти, имеет адрес 1024.

Назначение определенных значений адресов ссылкам на память в программе может быть выполнено либо программистом, либо автоматически в процессе компиляции или ассемблирования (табл. 7.3, *a*). Такой подход имеет ряд серьезных недостатков. Во-первых, каждый программист должен знать стратегию размещения модулей в основной памяти. Во-вторых, при любых изменениях в программе, которые включают вставку или удаление кода или данных, требуется соответствующим образом изменить все адреса. Поэтому желательно, чтобы все адреса в памяти были выражены символьно, с тем чтобы в процессе компиляции или ассемблирования разрешить эти символьные ссылки (рис. 7.15). Каждая ссылка на команду или элемент данных изначально представлена символом. При подготовке модуля к абсолютной загрузке ассемблер или компилятор преобразует все эти ссылки в конкретные адреса, как показано на рис. 7.15, *b*.

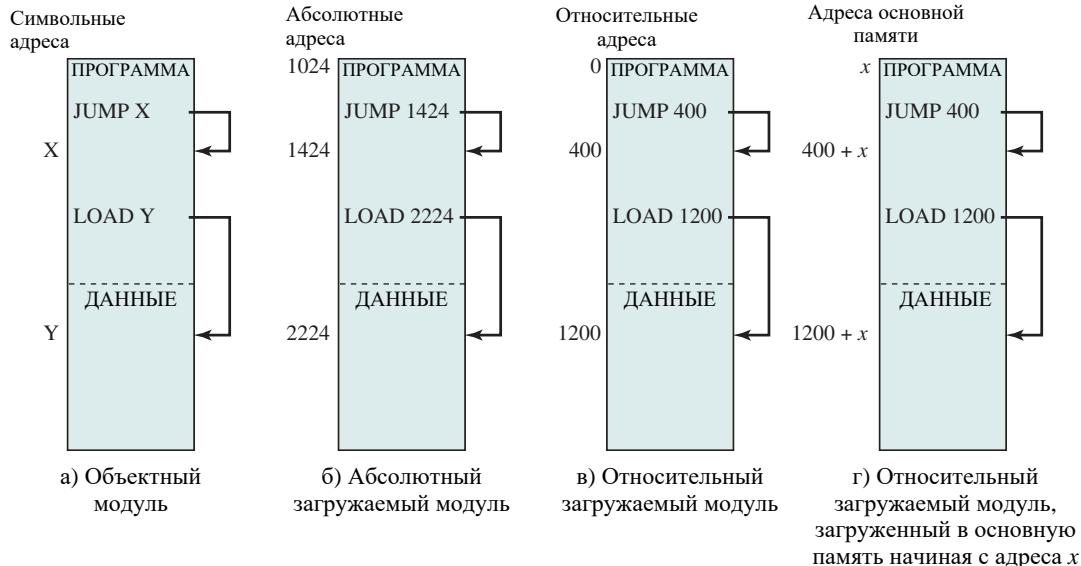


Рис. 7.15. Аbsoluteные и перемещаемые загружаемые модули

Таблица 7.3. Связывание адресов

**а) Загрузчик**

Этап связывания	Действия
Разработка программы	Программист использует конкретные физические адреса непосредственно в программе
Компиляция	Программа содержит ссылки на символьные адреса, которые преобразуются в реальные физические адреса компилятором или ассемблером
Загрузка	Компилятор или ассемблер генерирует относительные адреса, которые транслируются в абсолютные в процессе загрузки программы
Выполнение программы	Загруженная программа использует относительные адреса, которые динамически конвертируются процессором в абсолютные

**б) Компоновщик**

Этап компоновки	Действия
Разработка программы	Не разрешены никакие ссылки на внешний код или данные. Программист должен разместить в программе исходный код всех подпрограмм, на которые имеются ссылки
Компиляция	Код каждой подпрограммы, к которой имеется обращение, должен быть ассемблирован в качестве модуля
Создание загружаемого модуля	Все объектные модули ассемблируются с использованием относительных адресов. Эти модули связываются в одно целое, и все ссылки пересчитываются относительно начала образующегося единого модуля

Окончание табл. 7.3

Этап компоновки	Действия
Загрузка	Внешние ссылки не разрешаются до тех пор, пока модуль не будет размещен в основной памяти. В это время динамически связываемые модули, к которым имеются обращения, добавляются к основному модулю и в основной или виртуальной памяти размещается весь комплект целиком
Выполнение программы	Внешние ссылки не разрешаются до тех пор, пока внешний вызов не будет выполнен процессором. В этот момент процесс прерывается и необходимый модуль связывается с вызывающей программой

### Перемещаемая загрузка

Недостаток привязки обращений к памяти к конкретным адресам до загрузки заключается в том, что полученный модуль может быть загружен только в одну область основной памяти. Однако, когда память совместно используется несколькими программами, нежелательно заранее определять, в какую область памяти должен быть загружен тот или иной модуль. Такое решение лучше принимать в момент загрузки. Следовательно, нам требуется, чтобы загружаемый модуль мог быть размещен в произвольном месте памяти.

Для удовлетворения этого нового требования ассемблер или компилятор генерирует не абсолютные адреса, а адреса относительно некоторой известной точки, такой как начало программы. Этот метод продемонстрирован на рис. 7.15, в. Началу загружаемого модуля назначается относительный адрес 0, и все прочие ссылки внутри модуля выражаются относительно его начала.

Когда все ссылки выражены в относительном виде, размещение модуля в любом месте памяти становится достаточно простой задачей. Если модуль загружается в память, начиная с адреса  $x$ , то загрузчик при размещении модуля в памяти просто добавляет  $x$  к каждой ссылке. Для этого загружаемый модуль должен включать информацию, которая сообщает загрузчику, где именно располагаются обращения к памяти и как их следует трактовать (обычно от начала программы, однако могут быть и другие способы отсчета, например от текущей позиции). Эта информация подготавливается компилятором и обычно известна под названием **словарь перемещения** (relocation dictionary).

### Динамическая загрузка во время выполнения

Перемещаемая загрузка — обычное, широко распространенное явление с очевидными преимуществами перед абсолютной загрузкой. Однако в многозадачной среде (даже в независящей от виртуальной памяти) схема перемещаемой загрузки становится неадекватной. Мы уже говорили о необходимости выгрузки и загрузки процессов в основную память для максимального использования процессора. Для максимального использования основной памяти нам нужна возможность загрузки процесса, после того как он был выгружен, всякий раз в другое место в памяти. Следовательно, будучи однажды загруженной, программа может быть сброшена на диск, а затем загружена в другое место в памяти. Это невозможно, если обращения к памяти в момент начальной загрузки привязаны к абсолютным адресам.

Альтернативное решение состоит в том, чтобы отложить вычисление абсолютного адреса до того момента, пока он не потребуется реально во время выполнения. С этой целью модуль загружается в основную память со ссылками в относительном виде (рис. 7.15, 2), и только при реальном выполнении команды вычисляется абсолютный адрес. Для того чтобы такой метод не снижал общую производительность, требуется аппаратная поддержка вычисления абсолютного адреса (о чем говорилось в разделе 7.2).

Динамическое вычисление адресов обеспечивает высокую гибкость: программа может быть загружена в любую область основной памяти, причем программа может быть выгружена из основной памяти, а позднее — загружена в другое место в памяти.

## Компоновка

Компоновщик получает в качестве входных данных набор объектных модулей и генерирует на их основе загружаемый модуль путем объединения множества модулей кодов и данных для последующей передачи его загрузчику. В каждом объектном модуле могут иметься ссылки на память в других модулях. Каждая такая ссылка в нескомпонованном модуле может быть выражена только символично. Компоновщик создает единый загружаемый модуль, который объединяет все отдельные объектные модули. Каждая ссылка из одного модуля в другой должна быть разрешена и транслирована из символьного адреса в ссылку на ячейку памяти общего загружаемого модуля. Например, модуль А на рис. 7.16, а содержит вызов процедуры из модуля В. Когда эти модули комбинируются в единый загрузочный модуль, символическая ссылка на модуль В заменяется конкретной ссылкой на адрес точки входа В в загрузочном модуле.

### Редактор связей

Природа компоновки адресов зависит от типа создаваемого модуля и времени компоновки (см. табл. 7.3, б). Если, как это обычно и происходит, требуется создание перемещаемого модуля, то компоновка обычно выполняется следующим образом. Каждый скомпилированный объектный модуль создается со ссылками относительно начала объектного модуля. Все эти модули объединяются в единый перемещаемый загружаемый модуль, в котором все ссылки даны относительно начала единого модуля. Такой модуль можно использовать для перемещаемой загрузки или динамической загрузки времени выполнения.

Компоновщик, который создает перемещаемый загрузочный модуль, часто называется редактором связей (на рис. 7.16 проиллюстрирована его работа).

### Динамический компоновщик

Как и в случае загрузки, некоторые функции компоновки могут быть отложены. Метод, при котором компоновка отдельных внешних модулей откладывается на время после создания загружаемого модуля, называется **динамической компоновкой**, или **динамическим связыванием** (dynamic linking). Таким образом, загружаемый модуль содержит неразрешенные обращения к другим программам, которые могут быть разрешены либо в процессе загрузки, либо во время работы программы.

При **динамическом связывании во время загрузки** (верхняя динамическая библиотека на рис. 7.14) выполняются следующие действия. Загрузочный модуль (модуль приложения) считывается в память.

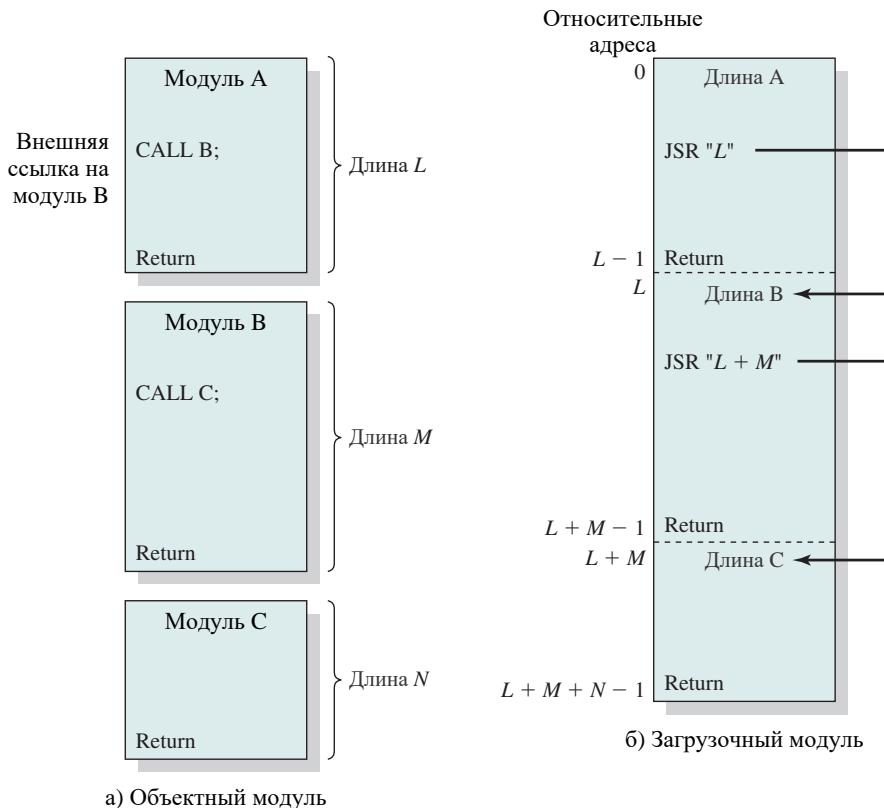


Рис. 7.16. Компоновка

Любые обращения ко внешнему (целевому) модулю приводят к поиску этого модуля загрузчиком, его загрузке и преобразованию ссылки в относительный адрес, отсчитываемый от начала модуля приложения. У такого динамического способа связывания имеется ряд преимуществ по сравнению со статической загрузкой.

- В этом случае облегчается возможность внесения изменений в целевой модуль, который, например, может представлять собой утилиту операционной системы. В случае статической компоновки изменения в таком модуле поддержки могут потребовать полной сборки всего модуля приложения заново, а это не только неэффективно, но зачастую и невозможно. Так, например, большинство коммерческих программ выпускаются в виде загрузочных модулей, и их исходные тексты и объектные модули попросту недоступны.
- Наличие целевого кода в динамически связываемом файле естественным путем приводит к совместному использованию этого кода различными программами. Поскольку операционная система загружает и связывает целевой код с приложением, она может распознать, что один и тот же целевой код используется несколькими приложениями одновременно. Операционная система может использовать имеющуюся в ее распоряжении информацию, для того чтобы загрузить в память только одну копию целевого кода и связать ее с несколькими приложениями, вместо того чтобы загружать для каждого приложения отдельную копию.

- Независимым производителям программного обеспечения становится легче расширять функциональность таких распространенных операционных систем, как, например, Linux. Разработчик может предоставить новую полезную функцию для ряда приложений в виде динамически компонуемого модуля.

При **динамическом связывании во время выполнения** часть связей остается неразрешенной до времени выполнения. При выполнении вызова подпрограммы из отсутствующего модуля операционная система находит этот модуль, загружает и связывает его с вызывающим модулем. Такие модули обычно являются совместно используемыми. В Windows они называются динамически подключаемыми библиотеками (*dynamic link library* — DLL). Таким образом, если один процесс уже использует совместно используемый модуль динамически подключаемой библиотеки, то этот модуль уже находится в основной памяти, и новый процесс может просто обращаться к уже загруженному модулю.

Использование DLL может привести к проблеме, которую обычно называют **адом DLL** (*DLL hell*). Она возникает, когда два или более процессов совместно используют один DLL-модуль, но ожидают различных его версий. Например, приложение или системная функция может быть переустановлена и принести с собой DLL старой версии.

Как мы уже видели, динамическая загрузка позволяет перемещать в память весь модуль целиком; однако структура модуля является статической, оставаясь неизменной как во время работы процесса, так и при различных запусках процесса. Однако в некоторых случаях до начала выполнения невозможно указать, какие именно объектные модули будут необходимы. Подобные ситуации типичны в приложениях обработки транзакций, таких как система резервирования авиабилетов или банковские приложения. Природа транзакций определяет, какие именно модули требуются для выполнения программы; именно эти модули затем загружаются в память и связываются с основной программой. Главное преимущество такого динамического связывания состоит в том, что для модулей программы не требуется выделять память до тех пор, пока эти модули не становятся действительно необходимыми. Это свойство динамического связывания используется при поддержке системы сегментации.

Кроме того, приложению не надо знать ни имен всех модулей, ни точек входа подпрограмм, которые могут быть им вызваны. Например, у нас имеется программа для построения диаграмм, которая может выводить результаты своей работы на различных плоттерах, каждый из которых управляет своим набором драйверов. Приложение может узнать о том, какой плоттер используется в данной системе, от другого процесса или из конфигурационного файла и использовать соответствующие динамические компонуемые драйверы. Это позволяет приложению использовать новые плоттеры, которых попросту могло не существовать в тот момент, когда выполнялась разработка самого приложения.

# 8

## ГЛАВА

# ВИРТУАЛЬНАЯ ПАМЯТЬ

В ЭТОЙ ГЛАВЕ...

## 8.1. Аппаратное обеспечение и управляющие структуры

Локальность и виртуальная память

Страницчная организация

Структура таблицы страниц

Инвертированная таблица страниц

Буфер быстрой переадресации (ББП)

Размер страницы

Сегментация

Значение виртуальной памяти

Организация

Комбинация сегментации и страницочной организации

Защита и совместное использование

## 8.2. Программное обеспечение операционной системы

Стратегия выборки

Стратегия размещения

Стратегия замещения

Блокировка кадров

Основные алгоритмы

Буферизация страниц

Стратегия замещения и размер кеша

Управление резидентным множеством

Размер резидентного множества

Область видимости замещения

Фиксированное распределение, локальная область видимости

Переменное распределение, глобальная область видимости

Переменное распределение, локальная область видимости

Стратегия очистки

Управление загрузкой

Уровень многозадачности

Приостановка процессов

### **8.3. Управление памятью в UNIX и Solaris**

Страницчная система

Структуры данных

Замещение страниц

Распределение памяти ядра

### **8.4. Управление памятью в Linux**

Виртуальная память Linux

Адресация виртуальной памяти

Распределение страниц

Алгоритм замещения страниц

Распределение памяти ядра

### **8.5. Управление памятью в Windows**

Карта виртуальных адресов Windows

Страницчная организация Windows

Свопинг в Windows

### **8.6. Управление памятью в Android**

#### **8.7. Резюме**

#### **8.8. Ключевые термины, контрольные вопросы и задачи**

Ключевые термины

Контрольные вопросы

Задачи

## УЧЕБНЫЕ ЦЕЛИ

- Понимать концепцию виртуальной памяти.
- Описать аппаратное обеспечение и управляющие структуры для поддержки виртуальной памяти.
- Описать различные механизмы операционной системы, используемые для реализации виртуальной памяти.
- Описать механизмы управления виртуальной памятью в UNIX, Linux и Windows.

В главе 7, “Управление памятью”, вы познакомились с концепциями страничной организации памяти и сегментации и с анализом их недостатков. Теперь мы перейдем к обсуждению виртуальной памяти. Трудность изучения этой темы заключается в том, что управление памятью представляет собой сложную связь между аппаратным обеспечением (процессором) и программным обеспечением операционной системы. Сначала мы остановимся на рассмотрении аппаратного аспекта виртуальной памяти, познакомимся с использованием страничной организации, сегментации и их комбинации, а затем обратимся к вопросам, возникающим при разработке средств виртуальной памяти в операционных системах.

В табл. 8.1 определены некоторые ключевые термины, связанные с виртуальной памятью.

**Таблица 8.1. Терминология, связанная с виртуальной памятью**

<b>Виртуальная память</b>	Схема распределения памяти, в которой вторичная память может адресоваться так, как если бы она была частью основной памяти. Адреса, которые программа может использовать для обращения к памяти, отличаются от адресов, используемых системой управления памятью для идентификации физической памяти, и генерируемые программой адреса автоматически транслируются в соответствующие машинные адреса. Размер виртуальной памяти ограничен схемой адресации компьютерной системы и количеством доступной вторичной памяти, но не фактическим количеством ячеек основной памяти
<b>Виртуальный адрес</b>	Адрес, присвоенный местоположению в виртуальной памяти, который позволяет обращаться к данному местоположению так, как если бы это была часть основной памяти
<b>Виртуальное адресное пространство</b>	Виртуальная память, назначенная процессу
<b>Адресное пространство</b>	Диапазон адресов памяти, доступный процессу
<b>Реальный адрес</b>	Адрес ячейки основной памяти

## 8.1. АППАРАТНОЕ ОБЕСПЕЧЕНИЕ И УПРАВЛЯЮЩИЕ СТРУКТУРЫ

Сравнивая простую страничную организацию и сегментацию с одной стороны и фиксированное и динамическое распределение памяти — с другой, мы видим основание для фундаментального прорыва в технологии управления памятью. Ключевыми для него являются следующие характеристики страничной организации и сегментации.

1. Все ссылки на память в рамках процесса представляют собой логические адреса, которые динамически транслируются в физические адреса во время выполнения. Это означает, что процесс может быть выгружен на диск и вновь загружен в основную память, так что в разные моменты времени выполнения он может находиться в разных местах основной памяти.
2. Процесс может быть разбит на ряд частей (страниц или сегментов), которые не обязательно должны располагаться в основной памяти единым непрерывным блоком. Это обеспечивается за счет динамической трансляции адресов и использования таблицы страниц или сегментов.

Теперь перейдем к нашему прорыву в технологии управления памятью. *Если в системе выполняются указанные характеристики, то наличие всех страниц или сегментов процесса в основной памяти одновременно не является обязательным условием.* Если фрагмент (сегмент или страница), в котором находится следующая выбираемая команда, и фрагмент, в котором находится ячейка памяти, к которой обращается программа, располагаются в основной памяти, то как минимум некоторое время выполнение программы может продолжаться.

Рассмотрим, каким образом это может осуществляться. Пока что мы говорим об этой технологии в общем, так что используем термин **блок** для обозначения страницы или сегмента — в зависимости от того, имеем ли мы дело со страничной организацией или с сегментацией. Предположим, что наступило время загрузки нового процесса в память. Операционная система начинает ее с размещения в памяти только одного или нескольких блоков, включая блок, содержащий начало программы. Часть процесса, располагающаяся в некоторый момент времени в основной памяти, называется **резидентным множеством** (resident set) процесса. Во время выполнения процесса все происходит так, как если бы все ссылки были только на резидентное множество процесса. При помощи таблицы сегментов или страниц процессор всегда может определить, располагается ли блок, к которому требуется обращение, в основной памяти. Если процессор сталкивается с логическим адресом, который не находится в основной памяти, он генерирует прерывание, свидетельствующее об ошибке доступа к памяти. Операционная система переводит прерванный процесс в заблокированное состояние и получает управление. Чтобы продолжить выполнение прерванного процесса, операционной системе необходимо загрузить в основную память блок, содержащий вызвавший проблемы логический адрес. Для этого операционная система использует запрос на чтение с диска (во время выполнения которого может продолжаться выполнение других процессов). После того как необходимый блок загружен в основную память, выполняется прерывание ввода-вывода, передающее управление операционной системе, которая, в свою очередь, переводит заблокированный процесс в состояние готовности.

Естественно, тут же возникает вопрос об эффективности использования такой технологии, когда выполнение процесса постоянно прерывается только из-за того, что в основной памяти размещены не все его блоки. Отложим пока рассмотрение вопроса эффективности, полагая, что эффективная работа все же возможна, и обратимся к следствиям применения нашей новой стратегии. Их два (причем второе потрясает воображение), и оба приводят к повышению эффективности использования системы.

- 1. В основной памяти может поддерживаться большее количество процессов.** Поскольку в основную память загружаются только некоторые из блоков каждого процесса, мы можем разместить в ней больше процессов. Это, в свою очередь, приводит к более эффективному использованию процессора, поскольку повышается вероятность наличия активных процессов в любой момент времени.
- 2. Процесс может быть больше, чем вся основная память.** Преодолено одно из наиболее существенных ограничений в программировании. Обычно программист должен изначально рассчитывать, какое количество памяти потребуется его программе. Если разработанная программа окажется слишком большой, программист должен принять соответствующие меры по разделению программы на части, которые могли бы быть загружены в память в отдельности, с использованием той или иной оверлейной стратегии. В случае использования виртуальной памяти на основе страничной организации или сегментации эта функция передается операционной системе и аппаратному обеспечению. Программе (и программисту) при этом доступен огромный объем памяти, по сути, представляющий собой размер дискового пространства. Операционная система при необходимости автоматически загружает нужные блоки процесса в память.

Поскольку процесс выполняется только в основной памяти, эта память называется также **реальной** (real memory). Однако программист или пользователь имеет дело с потенциально гораздо большей памятью — выделенной на диске. Эта память известна как **виртуальная** (virtual memory). Виртуальная память обеспечивает очень эффективную многозадачность и облегчает работу пользователя, снимая жесткие ограничения относительно объема основной памяти. В табл. 8.2 приведены основные характеристики страничной организации и сегментации с использованием виртуальной памяти и без нее.

## Локальность и виртуальная память

Преимущества виртуальной памяти весьма привлекательны, но насколько практична данная схема? Одно время на эту тему шли оживленные дебаты, однако опыт ее использования многими операционными системами продемонстрировал бесспорную жизнеспособность этой схемы — виртуальная память стала неотъемлемым компонентом большинства современных операционных систем.

Чтобы понять, в чем заключается ключевое свойство виртуальной памяти и почему она вызвала такие бурные дебаты, обратимся вновь к задачам операционной системы, связанным с виртуальной памятью. Рассмотрим большой процесс, состоящий из длинного кода и ряда массивов данных. В каждый небольшой промежуток времени выполнение программы сосредоточивается в малой части кода (например, в одной из подпрограмм), и обращается эта часть кода, как правило, только к одному или двум массивам данных.

**Таблица 8.2. ХАРАКТЕРИСТИКИ СТРАНИЧНОЙ ОРГАНИЗАЦИИ И СЕГМЕНТАЦИИ**

Простая страницчная организация	Страницчная организация с виртуальной памятью	Простая сегментация	Сегментация с виртуальной памятью
Основная память разделена на небольшие блоки фиксированного размера, именуемые кадрами		Основная память не разделена	
Программа разбита на страницы компилятором или системой управления памятью		Сегменты программы определены программистом при компиляции (решение о разбиже на сегменты принимается программистом)	
Внутренняя фрагментация в кадрах		Внутренняя фрагментация отсутствует	
Внешняя фрагментация отсутствует		Внешняя фрагментация	
Операционная система должна поддерживать таблицу страниц для каждого процесса, указывающую, какой кадр занят данной страницей процесса		Операционная система должна поддерживать таблицу сегментов для каждого процесса, указывающую адрес загрузки и длину каждого сегмента	
Операционная система должна поддерживать список свободных кадров		Операционная система должна поддерживать список свободных блоков памяти	
Для вычисления абсолютного адреса процессор использует номер страницы и смещение		Для вычисления абсолютного адреса процессор использует номер сегмента и смещение	
Для работы процесса все его страницы должны находиться в основной памяти (кроме случая использования оверлеев)	Для работы процесса не все его страницы должны находиться в основной памяти; они могут загружаться при необходимости	Для работы процесса все его сегменты должны находиться в основной памяти (кроме случая использования оверлеев)	Для работы процесса не все его сегменты должны находиться в основной памяти; они могут загружаться при необходимости
	Считывание страницы в основную память может требовать записи страницы на диск		Считывание сегмента в основную память может требовать записи одного или нескольких сегментов на диск

В таком случае загружать в память все данные и код, в то время как перед приостановкой и выгрузкой процесса из памяти будут использоваться только небольшие их части, — расточительство. Память будет использоваться гораздо эффективнее, если загружать в нее только необходимые части программы. Соответственно, в этом случае при обращении к данным (или к коду), которых в настоящий момент нет в основной памяти, происходит прерывание выполнения программы, которое говорит операционной системе о необходимости загрузки в основную память затребованной части программы.

Таким образом, в основной памяти в каждый момент времени находится только некоторая, как правило, небольшая, часть данного процесса, и следовательно, в основной памяти может быть одновременно размещено большее количество процессов. К тому

же при этом получается определенная экономия времени, так как неиспользуемые части процессов не приходится постоянно выгружать из памяти и загружать вновь. Однако операционная система должна очень разумно работать с такой схемой управления памятью. В установившемся состоянии практически вся основная память занята фрагментами процессов, чтобы процессор и операционная система могли работать с как можно большим количеством процессов одновременно. Следовательно, при загрузке в основную память некоторого блока другой блок должен быть выгружен оттуда. Если из памяти выгрузить блок, который тут же потребуется вновь, операционная система будет заниматься постоянным перемещением одних и тех же блоков в основную память и на диск. Большое количество таких перебросок приводит к ситуации, известной как **снижение пропускной способности (thrashing)**: процессор в основном занимается не выполнением процессов, а выгрузкой и загрузкой в основную память. Задаче устранения этого нежелательного эффекта посвящен ряд исследовательских работ, выполнявшихся в 1970-х годах и приведших к появлению различных сложных, но эффективных алгоритмов. По сути, они сводятся к попыткам определить на основании последних событий в системе, какие блоки памяти потребуются в ближайшем будущем.

Эти методы базируются на принципе локальности (см. главу 1, “Обзор компьютерной системы”, в особенности приложение 1.А), который гласит, что обращения к коду и данным в процессе имеют тенденцию к кластеризации. Следовательно, логично допустить, что в течение некоторого небольшого времени для работы будет требоваться только небольшая часть процесса; кроме того, можно сделать правильные предположения о том, какие именно части процесса потребуются для работы в ближайшем будущем, и тем самым избежать снижения пропускной способности.

Принцип локальности позволяет надеяться на эффективность работы схемы виртуальной памяти. Для этого требуется, чтобы, во-первых, имелась аппаратная поддержка страничной организации и/или сегментации, а во-вторых, операционная система должна включать программное обеспечение для управления перемещением страниц и/или сегментов между вторичной и основной памятью. В этом разделе мы рассмотрим аппаратный аспект проблемы и познакомимся с необходимыми управляющими структурами, создаваемыми и поддерживаемыми операционной системой, но используемыми аппаратным обеспечением; вопросы же работы операционной системы будут рассмотрены в следующем разделе.

## Страницчная организация

Термин *виртуальная память* обычно ассоциируется с системами, использующими страницочную организацию, хотя используется и виртуальная память на основе сегментации (которую мы рассмотрим чуть позже). Впервые сообщение о виртуальной памяти на основе страницочной организации появилось в работе [131], и вскоре после этого виртуальная память стала широко использоваться в коммерческих системах. Вспомните из главы 7, “Управление памятью”, что в случае простой страницочной организации основная память делится на ряд кадров одинакового размера. Каждый процесс делится на ряд страниц того же одинакового размера, что и размер кадров. Процесс загружается путем загрузки всех его страниц в доступные, хотя и не всегда смежные, кадры в памяти. В случае страницочной организации виртуальной памяти мы снова имеем страницы одинакового размера, равного размеру кадров, однако для выполнения не все страницы обязаны быть загружены в кадры основной памяти.

При рассмотрении простой страничной организации мы указывали, что каждый процесс имеет собственную таблицу страниц, которая создается при загрузке всех страниц процесса в основную память. Каждая запись в таблице страниц содержит номер кадра соответствующей страницы в памяти. Такая же таблица страниц, связанная с каждым из процессов, требуется и при организации виртуальной памяти на базе страничной организации — однако в этом случае структура записей таблицы становится несколько более сложной (рис. 8.1, а). Поскольку в основной памяти могут находиться только некоторые из страниц процесса, в каждой записи таблицы должен иметься бит  $P$ , указывающий на присутствие соответствующей страницы в основной памяти. Если данная страница располагается в основной памяти, то в записи таблицы содержится номер ее кадра.

Виртуальный адрес

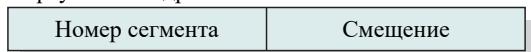


Запись таблицы страниц

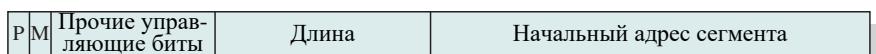


а) Страницчная организация

Виртуальный адрес



Запись таблицы сегментов

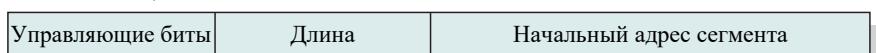


б) Сегментация

Виртуальный адрес



Запись таблицы сегментов



Запись таблицы страниц



P — бит присутствия  
M — бит модификации

в) Комбинация страницочной организации и сегментации

**Рис. 8.1.** Типичные форматы систем управления памятью

Другим управляющим битом в записи таблицы страниц является бит модификации,  $M$ , который указывает, было ли изменено содержимое данной страницы со времени последней загрузки в основную память. Если изменений не было, то, когда наступит время замены страницы в занимаемом ею в данный момент кадре, записывать эту страницу на диск не понадобится, так как на диске уже есть ее точная копия. В записи таблицы страниц могут быть и другие управляющие биты, например, служащие для целей защиты или совместного использования памяти на уровне страниц.

### Структура таблицы страниц

Базовый механизм чтения слова из памяти включает в себя трансляцию виртуального, или логического, адреса, состоящего из номера страницы и смещения, в физический адрес, который представляет собой номер кадра и смещение, с использованием таблицы страниц. Поскольку таблица страниц имеет переменную длину, зависящую от размера процесса, разместить ее в регистрах не представляется возможным, и таблица страниц должна располагаться в основной памяти. На рис. 8.2 показана аппаратная реализация этого механизма. При выполнении некоторого процесса стартовый адрес его таблицы страниц хранится в регистре, а номер страницы из виртуального адреса используется в качестве индекса элемента, в котором ищется соответствующий номер кадра. Затем этот номер объединяется со смещением из виртуального адреса для получения реального физического адреса интересующей нас ячейки памяти. Как правило, поле номера страницы длиннее, чем поле номера кадра ( $n > m$ ). Это неравенство вытекает из того факта, что количество страниц в процессе может превышать количество кадров в основной памяти.

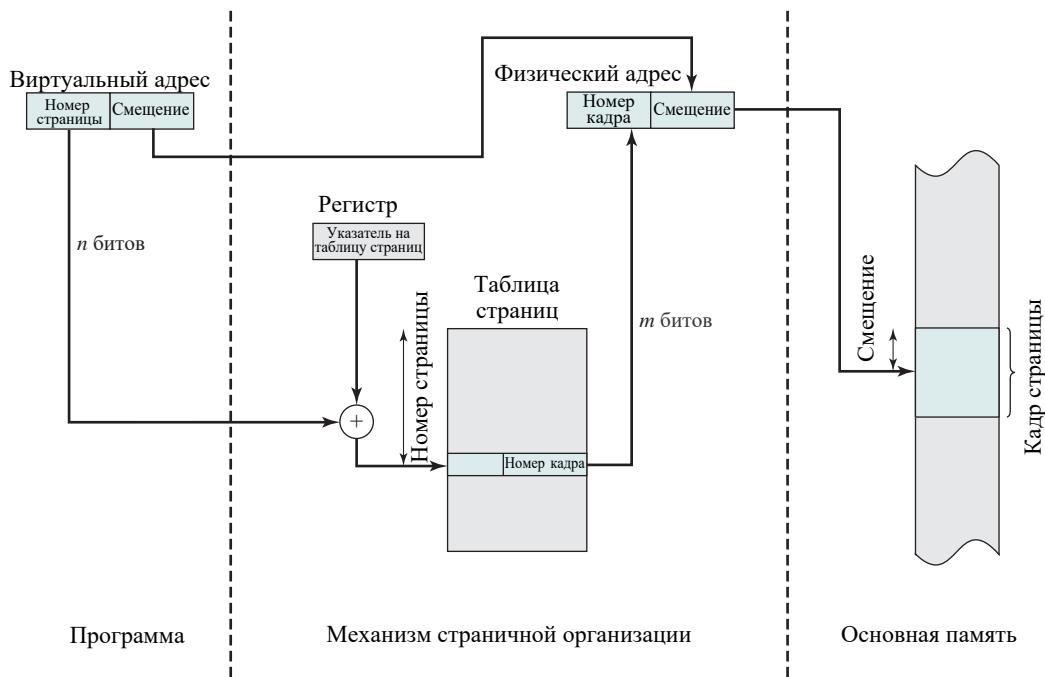
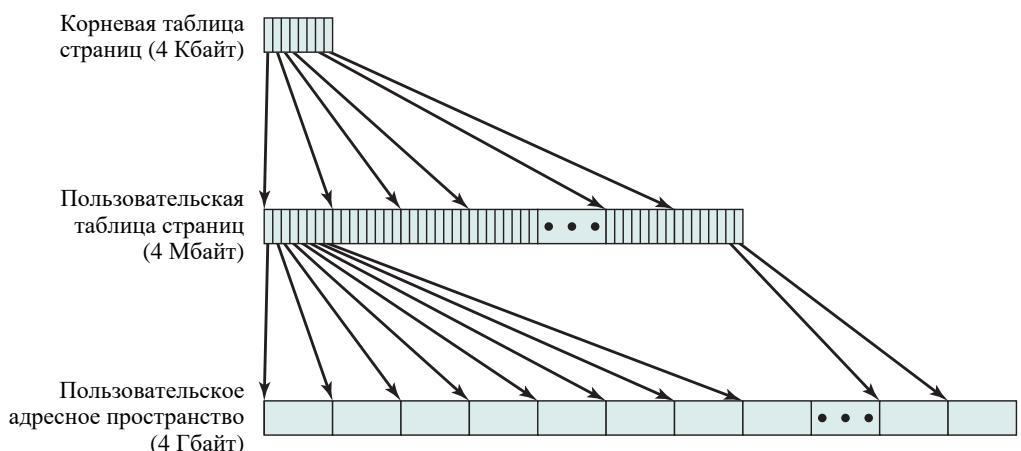


Рис. 8.2. Трансляция адреса в системе со страничной организацией

В большинстве систем для каждого процесса имеется одна таблица страниц. Однако каждый процесс может использовать большой объем виртуальной памяти. Так, например, в архитектуре VAX (Virtual Address Extension) каждый процесс может иметь до  $2^{31} = 2$  Гбайт виртуальной памяти. При использовании страниц размером  $2^9 = 512$  байт оказывается, что нам требуется до  $2^{22}$  записей в таблице страниц для каждого процесса. Понятно, что такое количество памяти, отводимое таблицам страниц, неприемлемо. Для преодоления этой проблемы большинство схем виртуальной памяти хранят таблицы страниц не в реальной, а в виртуальной памяти. Это означает, что сами таблицы страниц становятся объектами страничной организации, как и любые другие страницы.

При работе процесса как минимум часть его таблицы страниц должна располагаться в основной памяти, в том числе запись о странице, выполняющейся в настоящий момент. Некоторые процессоры используют двухуровневую схему для больших таблиц страниц. При такой схеме имеется каталог таблиц страниц, в котором каждая запись указывает на таблицу страниц. Таким образом, если размер каталога —  $X$ , а максимальное количество записей в таблице страниц —  $Y$ , то процесс может состоять максимум из  $X \times Y$  страниц. Обычно максимальный размер таблицы страниц определяется условием ее размещения в одной странице (такой подход используется, например, в процессоре Pentium).

На рис. 8.3 приведен пример двухуровневой схемы, типичной для 32-битовой адресации.



**Рис. 8.3. Двухуровневая иерархическая таблица страниц**

Принимая условие адресации байтов и 4-килобайтовые ( $2^{12}$ ) страницы, мы получим 4-гигабайтовое ( $2^{32}$ ) виртуальное адресное пространство, составленное из  $2^{20}$  страниц. Если каждая из этих страниц отображается посредством одной 4-байтовой записи в таблице страниц, то мы можем создать пользовательскую таблицу страниц, состоящую из  $2^{20}$  записей, общим объемом 4 Мбайт ( $2^{22}$  байт). Такая огромная таблица может быть размещена в  $2^{10}$  страницах виртуальной памяти, которые отображаются корневой таблицей страниц, состоящей из  $2^{10}$  записей, которые занимают 4 Кбайт ( $2^{12}$  байт) основной памяти.

На рис. 8.4 показаны действия, выполняемые при трансляции адреса в двухуровневой системе. Корневая страница всегда остается в основной памяти. Первые 10 бит виртуального адреса используются для индекса корневой таблицы для поиска записи о стра-

нице пользовательской таблицы. Если нужная страница отсутствует в основной памяти, генерируется ошибка доступа к странице. Если же необходимая страница находится в основной памяти, то следующие 10 бит виртуального адреса используются как индекс для поиска записи о странице, на которую ссылается исходный виртуальный адрес.

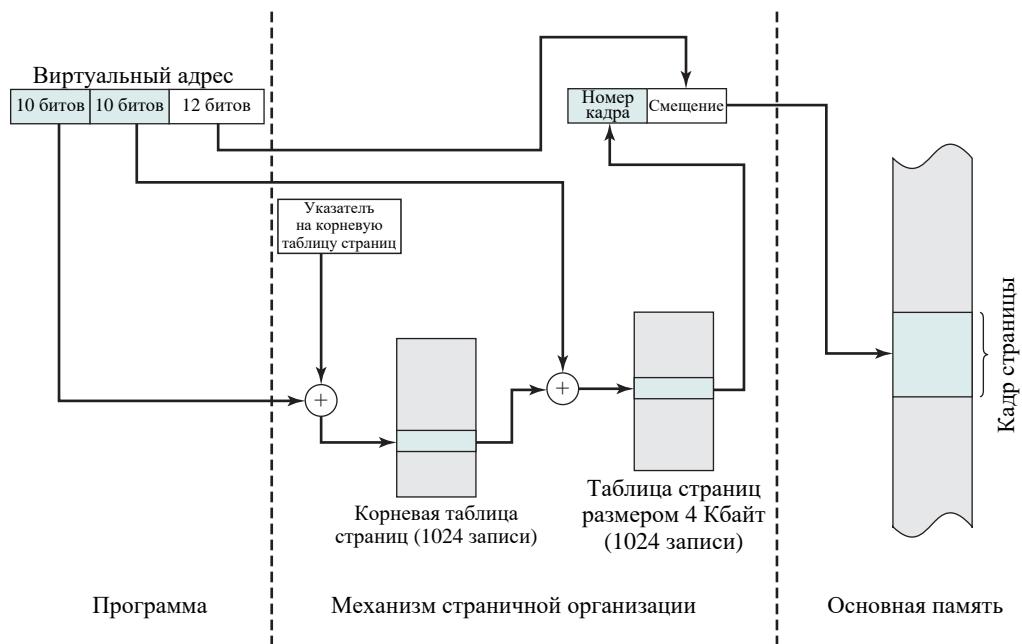


Рис. 8.4. Трансляция адреса в системе с двухуровневой страничной организацией

### Инвертированная таблица страниц

Недостатком таблиц страницирования рассматриваемого типа является то, что их размер пропорционален размеру виртуального адресного пространства.

Альтернативным подходом к использованию одно- или двухуровневых таблиц страниц является применение **инвертированной таблицы страниц** (inverted page table). Варианты этого подхода применялись на машинах Power PC, UltraSPARC и в архитектуре IA-64. Этот же подход использован и в операционной системе Mach на RT-PC.

При таком подходе часть виртуального адреса, представляющая собой номер страницы, отображается в хеш-таблицу с использованием простой функции хеширования.<sup>1</sup> Хеш-таблица содержит указатель на инвертированную таблицу страниц. Каждому кадру страницы реальной памяти при этом соответствует одна запись в инвертированной таблице страниц. Таким образом, для хранения таблиц требуется фиксированная часть основной памяти, независимо от размера и количества процессов и поддерживаемых виртуальных страниц. Поскольку на одну и ту же запись хеш-таблицы могут отображаться несколько виртуальных адресов, для обработки переполнения используется технология цепочек (которые на практике обычно достаточно коротки — как правило, от одной до двух записей).

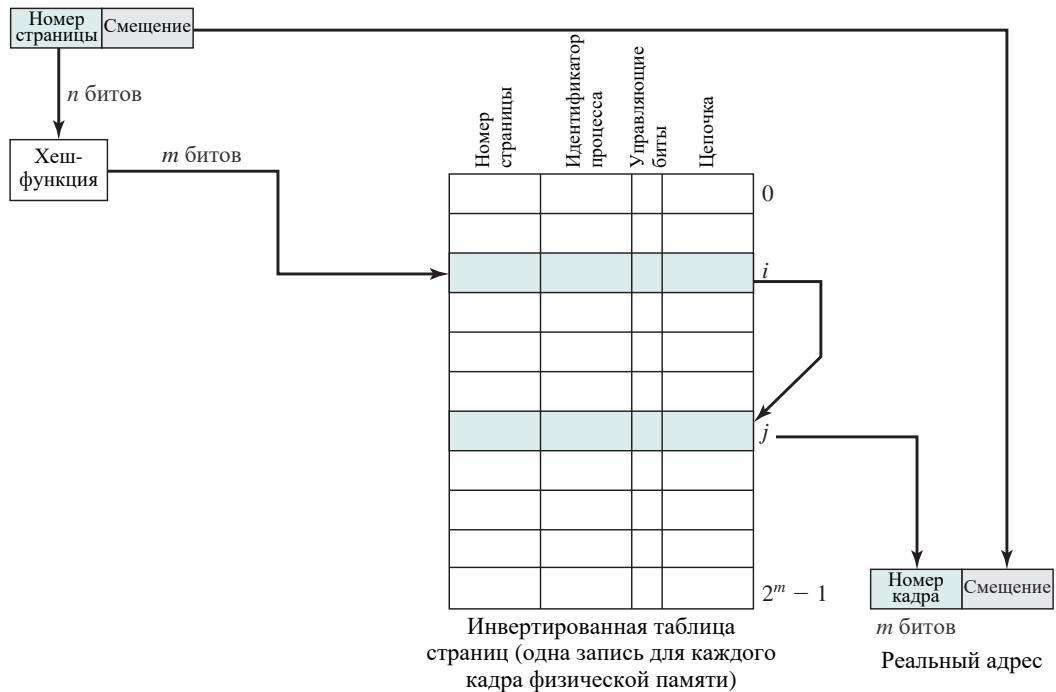
<sup>1</sup> Вопросы хеширования рассматриваются в приложении Е, “Хеш-таблицы”.

Структура таблицы страниц называется **инвертированной**, потому что она индексирует записи страницы таблицы номерами кадров, а не номерами виртуальных страниц.

На рис. 8.5 показана типичная реализация подхода с инвертированной таблицей страниц.

Виртуальный адрес

$n$  битов



**Рис. 8.5.** Структура инвертированной таблицы страниц

Для физической памяти размером  $2^m$  кадров инвертированная таблица страниц содержит  $2^m$  записей, так что  $i$ -я запись относится к кадру  $i$ . Каждая запись в таблице страниц включает следующую информацию.

- **Номер страницы.** Часть виртуального адреса, относящаяся к номеру страницы.
- **Идентификатор процесса.** Процесс, владеющий этой страницей. Комбинация номера страницы и идентификатора процесса идентифицирует страницу в виртуальном пространстве определенного процесса.
- **Управляющие биты.** Это поле включает флаги, такие как корректность, модифицированность и иные, а также информацию о защите и блокировках.
- **Указатель цепочки.** Это поле нулевое (вероятно, указывается отдельным битом), если для этой записи нет других связанных цепочкой записей. В противном случае поле содержит значение индекса (число от 0 до  $2^m - 1$ ) следующей записи в цепочке.

В этом примере виртуальный адрес включает  $n$ -битный номер страницы, где  $n > m$ . Хеш-функция отображает  $n$ -битный номер в  $m$ -битное число, которое используется в качестве индекса инвертированной таблицы страниц.

## Буфер быстрой переадресации

В принципе, каждый виртуальный адрес может привести к обращению к двум физическим адресам: к одному — для выборки соответствующей записи из таблицы страниц и еще к одному — для обращения к адресуемым данным. Таким образом, простая схема виртуальной памяти, по сути, удваивает время обращения к памяти. Для преодоления этой проблемы большинство реально использующихся схем виртуальной памяти использует специальный высокоскоростной кеш для записей таблицы страниц, который обычно называют буфером быстрого преобразования адреса или просто буфером быстрой переадресации (translation lookaside buffer — TLB). Этот кеш функционирует так же, как и обычный кеш памяти (см. главу 1, “Обзор компьютерной системы”), и содержит те записи таблицы страниц, которые использовались последними. Организация аппаратной поддержки использования TLB показана на рис. 8.6. Получив виртуальный адрес, процессор сначала просматривает TLB. Если требуемая запись найдена (*попадание*), процессор получает номер кадра и формирует реальный адрес. Если запись в TLB не найдена (*промах*), то процессор использует номер страницы в качестве индекса для таблицы страниц процесса и просматривает соответствующую запись. Если бит присутствия в ней установлен, значит, искомая страница находится в основной памяти, и процессор просто получает номер кадра из записи таблицы страниц и формирует реальный адрес, одновременно внося использованную запись таблицы страниц в TLB. И наконец, если бит присутствия не установлен, значит, искомой страницы в основной памяти нет, и процессор генерирует ошибку обращения к странице. В этот момент подключается операционная система, которая загружает требуемую страницу в основную память и обновляет таблицу страниц.

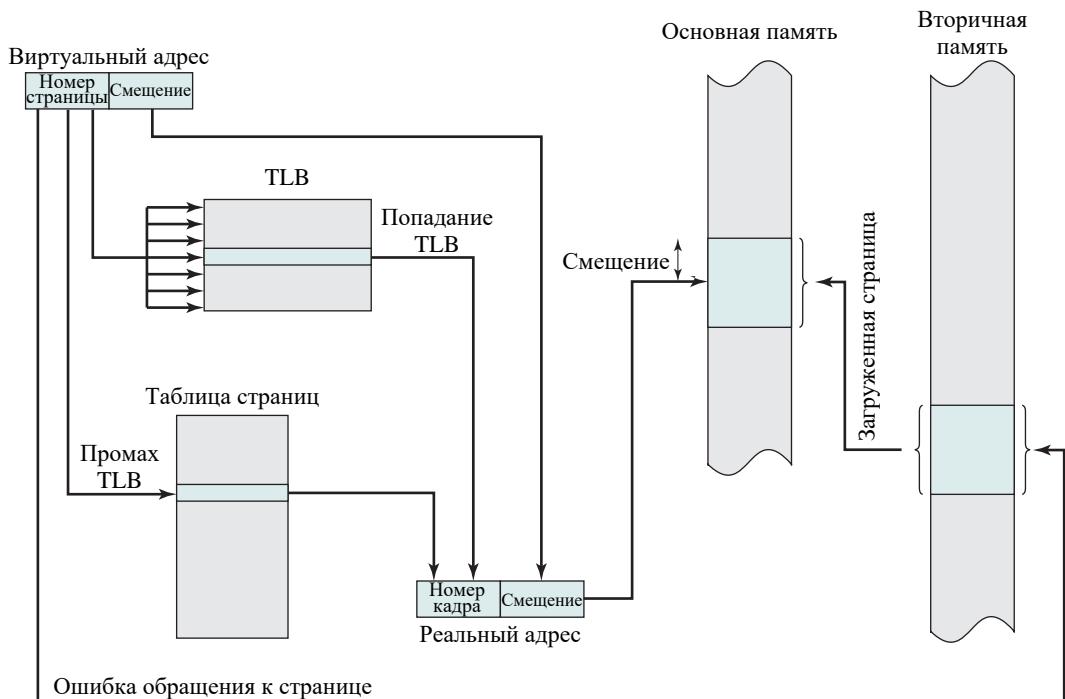


Рис. 8.6. Использование TLB

На рис. 8.7 приведена диаграмма использования TLB. На ней показано, что если требующаяся страница отсутствует в основной памяти, то прерывание ошибки обращения к странице вызывает соответствующую программу обработки. Для упрощения диаграммы в ней не отражен тот факт, что в процессе выполнения операций ввода-вывода операционная система может параллельно с медленными операциями дискового ввода-вывода выполнять другой процесс.

Исходя из принципа локальности большинство обращений к виртуальной памяти будут сосредоточены в недавно использованных страницах, и соответствующие записи будут находиться в кеше, так что с помощью TLB существенно повышается эффективность работы виртуальной памяти, что, например, показано при изучении VAX TLB [46, 214].

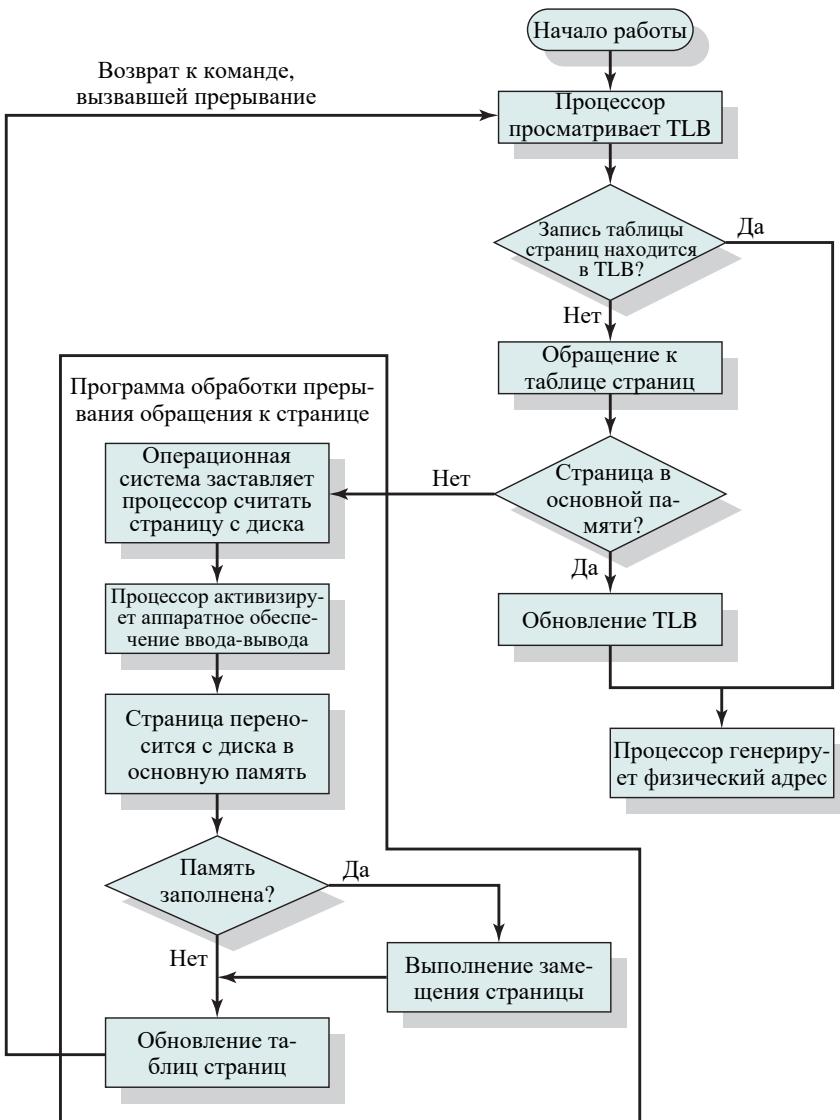
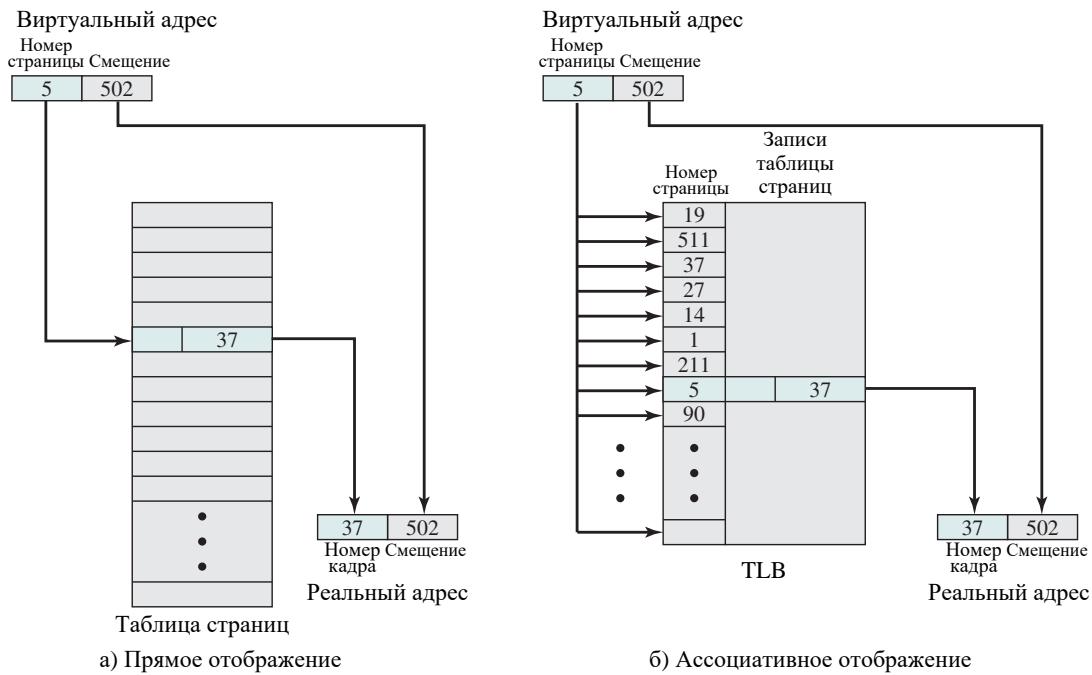


Рис. 8.7. Диаграмма трансляции адреса с использованием TLB

В реальной организации TLB имеется ряд дополнительных деталей. Так, поскольку TLB содержит только некоторые из записей таблицы страниц, индексация записей в TLB на основе номера страницы не представляется возможной; вместо этого каждая запись TLB должна наряду с полной информацией из записи таблицы страниц включать номер страницы. Процессор аппаратно способен опрашивать ряд записей TLB одновременно для определения того, какая из них соответствует заданному номеру страницы. Такая методика известна как **ассоциативное отображение** (associative mapping), в отличие от прямого отображения, или индексирования, применяемого для поиска в таблице страниц, как показано на рис. 8.8. Дизайн TLB должен также предусматривать способ организации записей в кеше и принятия решения о том, какая из старых записей должна быть удалена при внесении в кеш новой записи. Впрочем, этим вопросам следует уделить внимание при разработке любого аппаратного кеша. Однако они не входят в круг вопросов, рассматриваемых в нашей книге; заинтересованному читателю можно порекомендовать обратиться к специализированной литературе на эту тему (например, [242]).



**Рис. 8.8.** Прямой и ассоциативный поиск записи таблицы страниц

Наконец, механизм виртуальной памяти должен взаимодействовать с кешем (кешем основной памяти, не TLB). Это взаимодействие продемонстрировано на рис. 8.9. Виртуальный адрес, вообще говоря, представляет собой пару “номер страницы — смещение”. Сначала происходит обращение к TLB для выяснения, имеется ли в нем соответствующая запись таблицы страниц. При положительном результате путем объединения номера кадра и смещения генерируется реальный (физический) адрес (если требуемой записи в TLB нет, ее получают из таблицы страниц).

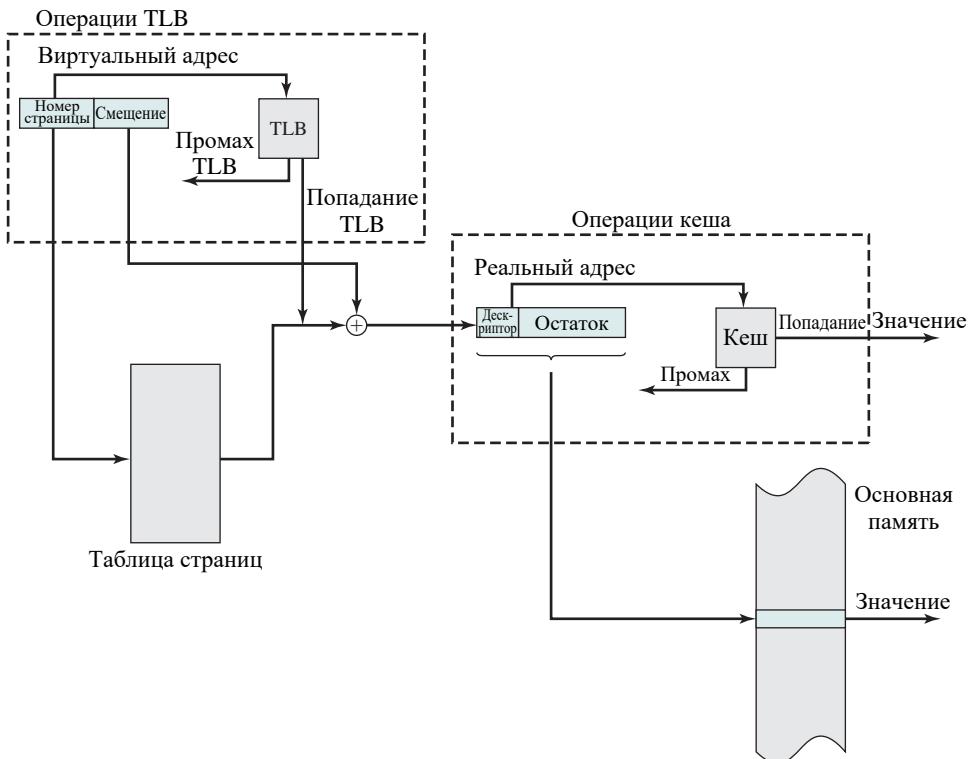


Рис. 8.9. Работа кеша и TLB

После того как сгенерирован реальный адрес, представляющий собой дескриптор<sup>2</sup> и остаток адреса, выполняется обращение к кешу для выяснения, не содержится ли в нем блок с интересующим нас словом. Если ответ положительный, то требуемое значение передается процессору; в противном случае происходит выборка слова из основной памяти.

Читатель должен оценить сложность аппаратного обеспечения процессора, вовлеченного в единственное обращение к памяти, когда виртуальный адрес преобразуется в реальный. Это приводит к обращению к записи таблицы страниц, которая может оказаться в TLB, в основной памяти или на диске. Само слово, к которому осуществляется обращение, тоже может оказаться в разных местах — в кеше, в основной памяти или на диске. Если слово находится только на диске, страница, содержащая это слово, должна быть загружена в основную память, а блок, содержащий слово, — в кеш. Кроме того, должна быть обновлена запись для данной страницы в таблице страниц.

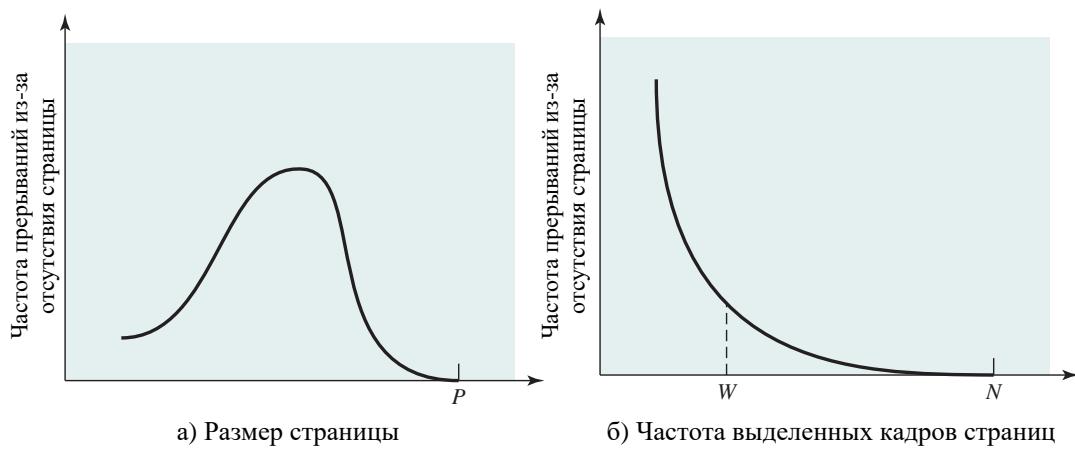
### Размер страницы

Весьма важным вопросом при разработке является выбор размера страниц. Здесь следует учесть сразу несколько факторов. Один из них — внутренняя фрагментация. Понятно, что внутренняя фрагментация, которую желательно уменьшить для оптимизации использования основной памяти, находится в прямой зависимости от размера страницы.

<sup>2</sup> См. рис. 1.17. Обычно дескриптор представляет собой несколько крайних слева битов реального адреса. Более детально о кешировании можно узнать в [241].

С другой стороны, чем меньше размер страниц, тем больше их требуется для процесса, что означает увеличение таблицы страниц. Для больших программ в загруженной многозадачной среде это может означать, что часть таблиц страниц активных процессов будет находиться в виртуальной памяти и при обращении к памяти будет возникать двойное прерывание из-за отсутствия страницы: сначала — при получении требуемой записи из таблицы страниц, а затем — при обращении к странице процесса. Еще одним фактором, который следует учесть, являются физические характеристики большинства устройств вторичной памяти, приводящие к тому, что передача больших блоков осуществляется более эффективно.

Вопрос усложняется еще и тем, что на частоту возникновения прерывания из-за отсутствия страницы в основной памяти влияет размер страницы. На рис. 8.10, а показано обычное поведение частоты возникновения прерываний из-за отсутствия страницы с учетом принципа локальности. Если размер страницы очень мал, то в памяти размещается относительно большое количество страниц процесса.



$P$  — размер процесса

$W$  — размер рабочего множества

$N$  — общее количество страниц процесса

**Рис. 8.10.** Типичное поведение страницной организации

Через некоторое время страницы в памяти будут содержать части процесса, сосредоточенные вблизи последних обращений, и частота возникновения прерывания из-за отсутствия страницы должна быть невелика. По мере увеличения размера страницы каждая отдельная страница будет содержать данные, которые располагаются все дальше и дальше от последних выполненных обращений к памяти. Соответственно, действие принципа локальности ослабевает, и наблюдается рост количества прерываний из-за отсутствия страницы. В конце концов, когда размер страницы начинает становиться сравнимым с размером процесса (точка  $P$  на графике), прерывания из-за отсутствия страницы становятся все более и более редкими, а по достижении размера этого процесса прекращаются вовсе.

Следует учитывать также влияние количества кадров, выделенных процессу. На рис. 8.10, б показано, что для фиксированного размера страниц частота возникновения прерываний из-за отсутствия страницы уменьшается с ростом числа страниц, находя-

щихся в основной памяти.<sup>3</sup> Таким образом, на программную стратегию (объем памяти, выделяемой процессу) влияет аппаратное решение (размер страницы).

В табл. 8.3 приведены размеры страниц на некоторых машинах.

**Таблица 8.3. Примеры размеров страниц**

Компьютер	Размер страницы
Atlas	512 48-битовых слов
Honeywell-Multics	1024 36-битовых слова
IBM 370/XA и 370/ESA	4 Кбайт
Семейство VAX	512 байт
IBM AS/400	512 байт
DEC Alpha	8 Кбайт
MIPS	От 4 Кбайт до 16 Мбайт
UltraSPARC	От 8 Кбайт до 4 Мбайт
Pentium	От 4 Кбайт до 4 Мбайт
Intel Itanium	От 4 Кбайт до 256 Мбайт
Intel core i7	От 4 Кбайт до 1 Гбайт

И наконец, решение об используемом размере страниц связано с размером физической основной памяти и размером программы. Ведь растет не только объем основной памяти в компьютерах, но и адресное пространство, используемое приложениями. Эта тенденция наиболее заметна в персональных компьютерах и рабочих станциях, в которых особенно резко проявляется увеличение размеров и возрастание сложности используемых приложений. Кроме того, современные технологии программирования, используемые в больших программах, приводят к снижению локальности ссылок процесса [111]. В качестве примеров можно привести следующие.

- Объектно-ориентированные технологии, стимулирующие использование множества мелких модулей кода и данных с обращениями к большому количеству объектов за относительно короткое время.
- Многопоточные приложения, приводящие к внезапным изменениям в потоке команд и обращениям к памяти, разбросанным по разным адресам.

Результативность поиска в TLB определенного размера с ростом размера процессов и уменьшением локальности снижается. При таком положении дел TLB может стать узким местом, ограничивающим производительность [43].

Один из способов повышения производительности TLB — использование большого TLB с большим количеством записей. Однако увеличение размера TLB связано с другими аспектами аппаратного решения вопросов обращения к памяти, с такими как размер кеша основной памяти и количество обращений к памяти при выполнении одной ко-

<sup>3</sup> Параметр W представляет размер рабочего множества, которое рассматривается в разделе 8.2.

манды [251], что заставляет сделать вывод о невозможности роста размера TLB такими же темпами, как и увеличение размера основной памяти. Альтернативой может быть использование больших размеров страниц, с тем чтобы каждая запись в TLB ссылалась на большой блок памяти. Однако мы уже видели, что использование больших размеров страниц может привести к потере производительности.

Учитывая обстоятельства, рассмотренные нами ранее, ряд разработчиков пришли к использованию множественных размеров страниц [129, 251], и некоторые из микропроцессоров, включая такие, как MIPS R4000, Alpha, UltraSPARC, x86 и IA-64, поддерживают эту методику. Множественные размеры страниц обеспечивают необходимую для эффективного использования TLB гибкость. Большие непрерывные области адресного пространства процесса, например программный код, могут отображаться с использованием небольшого количества больших страниц, в то время как для отображения стеков потоков могут использоваться страницы малого размера. Однако большинство коммерческих операционных систем все еще поддерживают только один размер страниц, независимо от способности аппаратного обеспечения работать со страницами разного размера. Причина этого отставания в том, что с размером страниц связано большое количество разнообразных аспектов операционных систем, и переход на множественный размер страниц оказывается очень сложным [85].

## Сегментация

### Значение виртуальной памяти

Сегментация позволяет программисту рассматривать память как область, состоящую из множества адресных пространств, или сегментов. Сегменты могут иметь разные (на самом деле динамические) размеры. Обращения к памяти используют адреса, представляющие собой пары (*номер сегмента, смещение*).

Такая организация имеет ряд преимуществ по сравнению с несегментированным адресным пространством.

1. Упрощается обработка растущих структур данных. Если программисту заранее не известен размер структур данных, с которыми предстоит работать, и есть возможность использовать сегментацию, структуре данных может быть назначен ее собственный сегмент, размер которого операционная система будет увеличивать или уменьшать по мере необходимости. Если сегмент, размер которого следует увеличить, находится в основной памяти и для его увеличения нет свободного места, операционная система может переместить его в большую область или выгрузить на диск (в этом случае увеличенный сегмент будет загружен вновь при первой возможности).
2. Программы могут изменяться и перекомпилироваться независимо от компиляции или компоновки всего множества программ (что осуществляется при использовании множественных сегментов).
3. Упрощается совместное использование кода и данных разными процессами. Программист может поместить код утилиты или необходимые данные в отдельный сегмент, к которому будут обращаться другие процессы.
4. Улучшается защита. Так как сегмент представляет собой точно определенные множества программ или данных, программист или системный администратор может назначать права доступа просто и удобно.

## Организация

При рассмотрении простой сегментации мы отмечали, что каждый процесс имеет собственную таблицу сегментов, и при загрузке всех сегментов процесса в основную память создается таблица сегментов процесса, которая также загружается в основную память. В каждой записи таблицы сегментов указаны начальный адрес соответствующего сегмента в основной памяти и его длина. Такая же таблица сегментов нужна и при схеме виртуальной памяти, основанной на сегментации. Типичным приемом является использование отдельной таблицы сегментов для каждого процесса. Записи таблицы сегментов в этом случае усложняются (рис. 8.1, б). Поскольку в основной памяти могут находиться не все сегменты процесса, в каждой записи требуется наличие бита присутствия, указывающего, располагается ли данный сегмент в основной памяти. Если сегмент расположен в основной памяти, то запись включает его начальный адрес и длину.

Еще один бит, необходимый в данной схеме, — бит модификации, указывающий, было ли изменено содержимое сегмента со времени его последней загрузки в основную память. Если изменений не было, то при выгрузке сегмента нет необходимости в его записи на диск. Могут иметься и другие управляющие биты, например при организации защиты или совместного использования на уровне сегментов.

Основной механизм чтения слова из памяти включает трансляцию виртуального, или логического, адреса, состоящего из номера сегмента и смещения, в физический адрес с использованием таблицы сегментов. Поскольку таблица сегментов имеет переменную длину, зависящую от размера процесса, мы не можем рассчитывать на ее хранение в регистрах, и для хранения таблицы сегментов используется основная память.

На рис. 8.11 предложена аппаратная реализация описываемой схемы (обратите внимание на подобие с рис. 8.2).

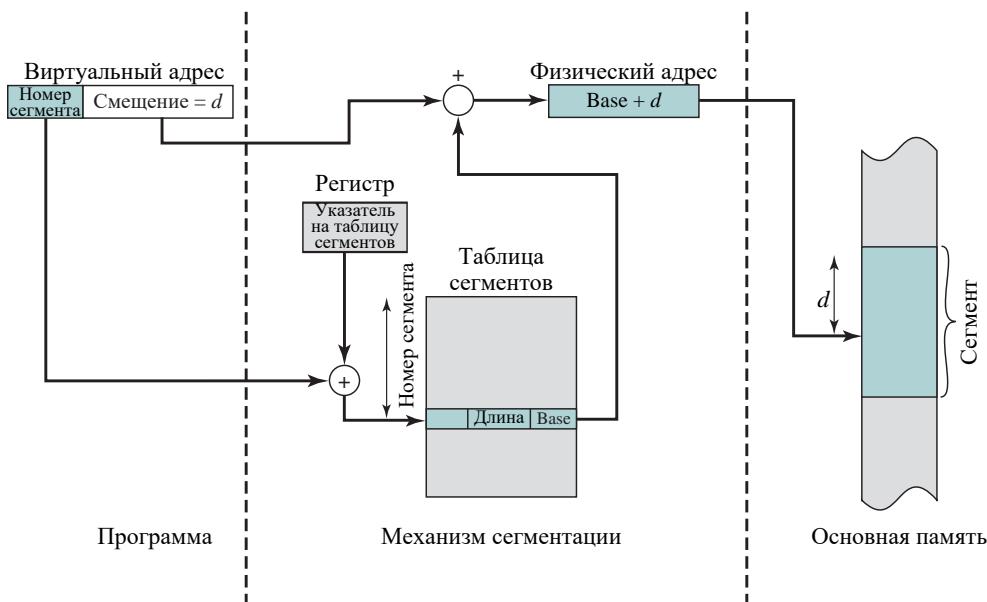


Рис. 8.11. Трансляция адреса в системе с сегментацией

Когда запускается определенный процесс, в регистре хранится стартовый адрес его таблицы сегментов. Номер сегмента из виртуального адреса используется в качестве индекса таблицы, позволяющего определить начальный адрес сегмента. Для получения физического адреса к начальному адресу сегмента добавляется смещение из виртуального адреса.

## Комбинация сегментации и страничной организации

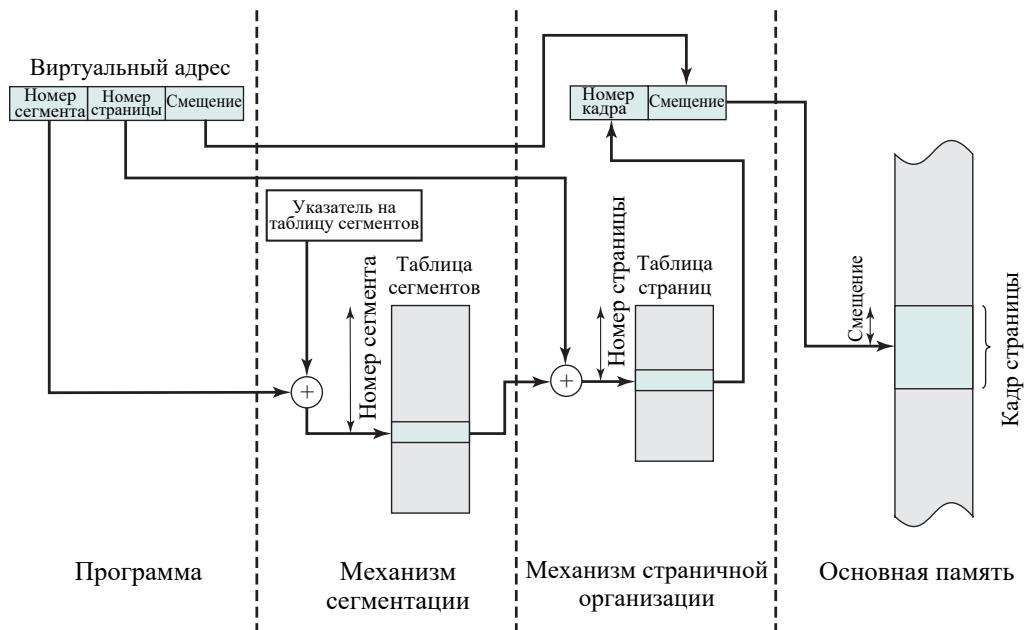
И страничная организация, и сегментация имеют свои достоинства. Страницная организация, прозрачная для программиста, устраниет внешнюю фрагментацию и таким образом обеспечивает эффективное использование основной памяти. Кроме того, поскольку перемещаемые в основную память и из нее блоки имеют фиксированный, одинаковый размер, облегчается создание эффективных алгоритмов управления памятью. Сегментация, являясь видимой для программиста, имеет перечисленные в предыдущем разделе достоинства, включающие модульность, возможность обработки растущих структур данных, а также поддержку совместного использования и защиты памяти. Некоторые вычислительные системы, будучи оснащенными соответствующим аппаратным обеспечением и операционной системой, используют достоинства обоих методов.

В такой комбинированной системе адресное пространство пользователя разбивается на ряд сегментов по усмотрению программиста. Каждый сегмент, в свою очередь, разбивается на ряд страниц фиксированного размера, соответствующего размеру кадра основной памяти. Если размер сегмента меньше размера страницы, он занимает страницу целиком. С точки зрения программиста, логический адрес в этом случае состоит из номера сегмента и смещения в нем. С позиции операционной системы смещение в сегменте следует рассматривать как номер страницы определенного сегмента и смещение в ней.

На рис. 8.12 предложена структура для поддержки комбинации сегментации и страничной организации (обратите внимание на схожесть с рис. 8.4, а). С каждым процессом связаны одна таблица сегментов и несколько (по одной на сегмент) таблиц страниц. При работе определенного процесса в регистре процессора хранится начальный адрес соответствующей таблицы сегментов.

Получив виртуальный адрес, процессор использует его часть, представляющую номер сегмента, в качестве индекса в таблице сегментов для поиска таблицы страниц данного сегмента. После этого часть адреса, представляющая собой номер страницы, используется для поиска соответствующего кадра основной памяти в таблице страниц; затем часть адреса, представляющая смещение, используется для получения искомого физического адреса путем добавления к начальному адресу кадра.

На рис. 8.1, в предложены форматы записей таблицы сегментов и таблицы страниц. Как и ранее, запись таблицы сегментов содержит значение длины сегмента, а также поле с начальным адресом сегмента, которое теперь указывает на таблицу страниц. Биты присутствия и модификации в записи таблицы сегментов в данном случае не нужны, так как эти вопросы решаются на уровне страниц. Использование других управляющих битов может продолжаться, например, в упомянутых ранее целях совместного использования и защиты. Запись таблицы страниц, по сути, та же, что и использованная в системе с “чистой” страничной организацией. При наличии страницы в основной памяти (на что указывает бит присутствия) ее номер отображается в номер соответствующего кадра; бит модификации указывает, требуется ли перезапись страницы на диск при ее выгрузке из памяти. Как и ранее, могут использоваться и другие управляющие биты.



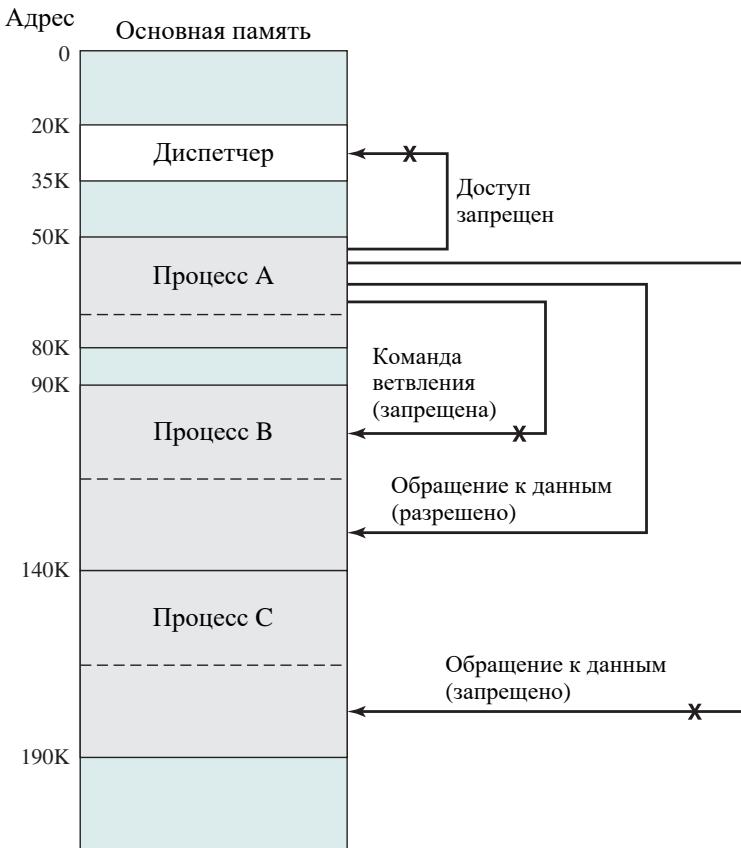
**Рис. 8.12.** Трансляция адреса при совместном использовании сегментации и страничной организации

## ЗАЩИТА И СОВМЕСТНОЕ ИСПОЛЬЗОВАНИЕ

Сегментация вполне пригодна для реализации стратегии защиты и совместного использования. Поскольку каждая запись таблицы сегментов включает начальный адрес и значение длины, программа не в состоянии непреднамеренно обратиться к основной памяти за границами сегмента. Для осуществления совместного использования ссылки на один и тот же сегмент могут быть в таблицах сегментов нескольких процессов. Тот же механизм, естественно, осуществим и на уровне страниц. Однако в случае использования страниц они невидимы для программиста и делают определение правил защиты и совместного использования неудобным. На рис. 8.13 показаны типы защиты, которые могут быть реализованы в такой системе.

Можно обеспечить и более интеллектуальный механизм защиты. Обычная схема использует кольцевую структуру защиты, с которой мы встречались в главе 3, “Описание процессов и управление ими” (см. рис. 3.18). В этой схеме внутренние кольца (с меньшими номерами) имеют большие привилегии по сравнению с внешними кольцами. Обычно кольцо 0 зарезервировано для функций ядра операционной системы, а приложения находятся во внешнем кольце. Некоторые утилиты или операционная система могут занимать промежуточные кольца. Основными принципами системы колец являются следующие.

- Программа может получить доступ только к данным, расположенным в том же или менее привилегированном кольце.
- Программа может вызвать сервис из того же или более привилегированного кольца.



**Рис. 8.13.** Отношения защиты между сегментами

## 8.2. ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ОПЕРАЦИОННОЙ СИСТЕМЫ

Особенности разработки части программного обеспечения операционной системы, управляющей памятью, зависят от ответа на три основных вопроса.

1. Будет ли использоваться виртуальная память.
2. Будет ли использоваться сегментация, страничная адресация или обе указанные технологии.
3. Какие алгоритмы будут использованы для различных аспектов управления памятью.

Ответы на первые два вопроса тесно связаны с используемой аппаратной платформой. Так, ранние реализации UNIX не использовали виртуальную память, поскольку процессоры, на которых работали эти операционные системы, не поддерживали ни сегментации, ни страничной организации, а без аппаратной поддержки преобразования адресов ни сегментация, ни страничная организация никакой практической ценности не представляют.

Приведем два небольших замечания по поводу первых двух вопросов из представленного списка. Во-первых, за исключением операционных систем типа MS-DOS для

некоторых старых персональных компьютеров и специализированных систем, все важные операционные системы используют виртуальную память. Во-вторых, “чистая” сегментация становится очень редким явлением. При комбинации сегментации и страничной организации основная масса вопросов управления памятью, с которыми сталкивается разработчик операционной системы, лежит в области страничной организации.<sup>4</sup> Таким образом, мы сосредоточим свое внимание на вопросах, связанных со страничной организацией памяти.

Ответ на третий из поставленных в этом разделе вопросов связан с программным обеспечением операционной системы, и именно ему посвящен данный раздел. В табл. 8.4 перечислены основные рассматриваемые в этом разделе стратегии. В каждом случае ключевым вопросом становится производительность: требуется сократить количество прерываний из-за отсутствия страницы в памяти, поскольку их обработка приводит к существенным накладным расходам, которые включают как минимум принятие решения о том, какие резидентные страницы должны быть замещены, и операции ввода-вывода по замене страниц в основной памяти. Кроме того, операционная система должна активировать на время выполнения медленных операций ввода-вывода другой готовый к работе процесс. Ни одна из перечисленных в табл. 8.4 стратегий не является “наилучшей”.

Задача управления памятью чрезвычайно сложна. Кроме того, производительность каждого определенного набора стратегий зависит от размера основной памяти, относительной скорости основной и вторичной памяти, размера и количества конкурирующих за ресурсы процессов и поведения отдельных программ. Последняя характеристика зависит от природы приложения, использованных языка программирования и компилятора, стиля программиста, а для интерактивных программ — от динамичности пользователя. Так что читатель не должен ожидать окончательного ответа на поставленные вопросы.

**Таблица 8.4. Стратегии операционной системы для виртуальной памяти**

<b>Стратегия выборки</b>	<b>Управление резидентным множеством</b>
По требованию	Размер резидентного множества
Предварительная выборка	Фиксированный
<b>Стратегия размещения</b>	Переменный
<b>Стратегия замещения</b>	Область видимости замещения
Основные алгоритмы	Глобальная
Оптимальный	Локальная
Дольше всех неиспользовавшиеся	<b>Стратегия очистки</b>
Первым вошел — первым вышел	По требованию
Часовой	Предварительная очистка
Буферизация страниц	<b>Управление загрузкой</b>
	Степень многозадачности

<sup>4</sup> В комбинированной системе защита и совместное использование обычно реализованы на уровне сегментов. Эти вопросы будут рассматриваться в последующих главах.

Для малых вычислительных систем разработчик операционной системы должен попытаться выбрать такой набор стратегий, который представляется ему наиболее подходящим; в случае использования больших систем, в частности мейнфреймов, операционная система должна быть оснащена инструментами для мониторинга и управления, которые позволяют администратору настроить ее таким образом, чтобы получить наивысшую производительность для данных физических параметров и решаемых на ней задач.

## Стратегия выборки

Стратегия выборки определяет, когда страница должна быть передана в основную память. Два основных варианта — по требованию и предварительно. При **выборке по требованию** страница передается в основную память только тогда, когда выполняется обращение к ячейке памяти, расположенной на этой странице. Если все прочие элементы системы управления памятью работают хорошо, то должно произойти следующее. Когда процесс только запускается, возникает поток прерываний обращений к странице, но далее срабатывает принцип локальности, и все большее количество обращений выполняется к недавно загруженным страницам. Соответственно, количество прерываний из-за отсутствия страницы снижается до весьма низкого уровня.

В случае **предварительной выборки** загружается не только страница, вызвавшая прерывание обращения. Предварительная выборка использует характеристики большинства устройств вторичной памяти, таких как диски, у которых имеются время поиска и задержка, связанная с вращением диска. Если страницы процесса расположены во вторичной памяти последовательно, то гораздо более эффективной будет загрузка в основную память нескольких последовательных страниц за один раз, чем загрузка этих же страниц по одной в течение некоторого промежутка времени. Естественно, эта стратегия не дает никакого выигрыша, если обращения к дополнительно загруженным страницам не происходит.

Предварительная выборка может применяться либо при первом запуске процесса к страницам, тем или иным способом указываемым программистом, либо каждый раз при каждом прерывании обращения к странице. Последний случай кажется более предпочтительным, поскольку он прозрачен для программиста.

Не следует путать предварительную выборку и свопинг. При выгрузке процесса из памяти и переводе его в приостановленное состояние из основной памяти удаляются все его резидентные страницы. При возобновлении выполнения процесса все его страницы, которые ранее находились в основной памяти, вновь возвращаются в нее.

## Стратегия размещения

Стратегия размещения определяет, где именно в физической памяти будут располагаться части процесса. В случае “чистой” сегментации стратегия размещения является весьма важным вопросом, решения которого в виде стратегий первого подходящего, очередного подходящего и других рассматривались в главе 7, “Управление памятью”. Однако для систем, использующих только страницочную организацию или страницочную организацию в сочетании с сегментацией, стратегия размещения обычно не так важна, поскольку аппаратная трансляция адреса и аппаратное обращение к памяти одинаково результативны при любых сочетаниях “страница–кадр”.

В так называемых многопроцессорных системах с неоднородным доступом к памяти (nonuniform memory access — NUMA) размещение является довольно важным вопросом, требующим всестороннего исследования. Обратиться к распределенной совместно используемой памяти может любой процессор, однако на время доступа к определенному физическому адресу влияет расстояние между процессором и модулем памяти. Таким образом, суммарная производительность в огромной степени зависит от того, насколько близко к процессору размещены обрабатываемые им данные [24, 55, 145]. В системах с неоднородным доступом к памяти в соответствии со стратегией автоматического размещения страницы должны размещаться в модулях памяти, обеспечивающих наибольшую производительность.

## Стратегия замещения

В большинстве публикаций, посвященных операционным системам, рассмотрение управления памятью включает в себя раздел, озаглавленный “Стратегия замещения”, в котором говорится о выборе страниц в основной памяти для их замещения загружаемыми из вторичной памяти страницами. Эта тема достаточно сложна методологически, поскольку включает ряд взаимосвязанных вопросов.

- Какое количество кадров должно быть выделено каждому активному процессу.
- Должно ли множество страниц, которые потенциально могут быть замещены загружаемыми страницами, ограничиваться одним процессом или в качестве кандидатов на замещение могут рассматриваться все кадры страниц основной памяти.
- Какие именно страницы из рассматриваемого множества следует выбрать для замещения.

Первые два вопроса — из области *управления резидентным множеством*, о чем мы поговорим в следующем подразделе; термин же *стратегия замещения* мы будем использовать для обозначения третьего вопроса.

Вопросы стратегии замещения представляют собой, пожалуй, наиболее полно изученный аспект управления памятью. Когда все кадры основной памяти заняты и требуется разместить новую страницу в процессе обработки прерывания из-за отсутствия страницы, стратегия замещения определяет, какая из находящихся в настоящее время в основной памяти страниц должна быть выгружена, чтобы освободить кадр для загружаемой страницы. Все стратегии направлены на то, чтобы выгрузить страницу, обращений к которой в ближайшем будущем не последует. В соответствии с принципом локальности часто наблюдается сильная корреляция между множеством страниц, к которым в последнее время были обращения, и множеством страниц, к которым будут обращения в ближайшее время. Таким образом, большинство стратегий пытаются определить будущее поведение программы на основе ее прошлого поведения. При рассмотрении разных стратегий следует учитывать, что чем более совершенный и интеллектуальный алгоритм использует стратегия, тем выше будут накладные расходы при его реализации.

## Блокировка кадров

Перед рассмотрением различных алгоритмов следует упомянуть об одном ограничении: некоторые кадры основной памяти могут быть заблокированы. Блокировка кадра означает, что страница, хранящаяся в данный момент в этом кадре, не может быть замещена. Большинство ядер операционных систем хранятся в заблокированных кадрах,

так же как и основные управляющие структуры. Кроме того, в заблокированных кадрах основной памяти могут располагаться буферы ввода-вывода и другие критические по отношению ко времени доступа данные и код. Блокировка осуществляется путем установки соответствующего бита у каждого кадра. Этот бит может находиться как в таблице кадров, так и быть включенным в текущую таблицу страниц.

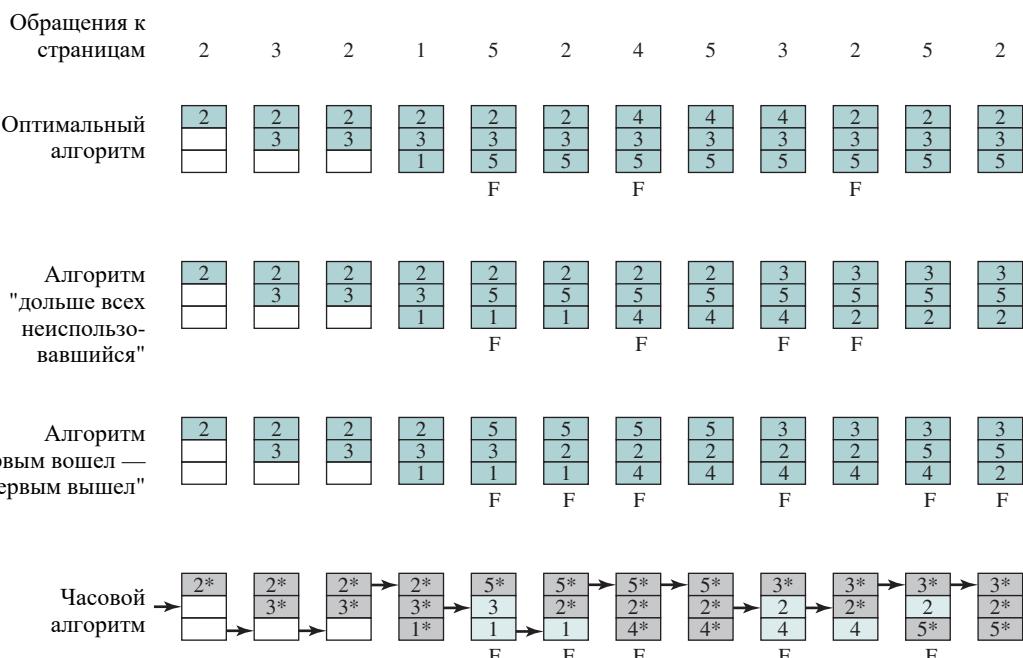
### Основные алгоритмы

Независимо от стратегии управления резидентным множеством имеется ряд основных алгоритмов, используемых для выбора замещаемой страницы.

- Оптимальный алгоритм
- Алгоритм дольше всех неиспользовавшегося элемента
- Алгоритм “первым вошел — первым вышел”
- Часовой алгоритм

**Оптимальная** стратегия состоит в выборе для замещения той страницы, обращение к которой будет через наибольший промежуток времени по сравнению со всеми остальными страницами. Можно показать, что этот алгоритм приводит к минимальному количеству прерываний из-за отсутствия страницы [20]. Понятно, что реализовать такой алгоритм невозможно, поскольку для этого системе требуется знать все будущие события. Однако этот алгоритм является стандартом, с которым сравниваются реальные алгоритмы.

На рис. 8.14 приведен пример оптимальной стратегии.



F — прерывания обращения к странице после первоначального заполнения кадров

Рис. 8.14. Поведение четырех алгоритмов замещения страниц

Предполагается, что для данного процесса используется фиксированное распределение кадров (фиксированный размер резидентного множества, состоящего из трех кадров). Выполнение процесса приводит к обращениям к пяти различным страницам. В процессе работы обращения к страницам выполняются в следующем порядке:

2 3 2 1 5 2 4 5 3 2 5 2

Это означает, что сначала выполняется обращение к странице 2, затем — к странице 3 и т.д. Оптимальная стратегия приводит после заполнения всего множества кадров к трем прерываниям обращения к странице (обозначенным на рисунке буквами “F”).

Стратегия дольше всех неиспользовавшегося элемента замещает в памяти ту страницу, обращений к которой не было дольше, чем к другим. Согласно принципу локальности можно ожидать, что эта страница не будет использоваться и в ближайшем будущем. Эта стратегия и в самом деле недалека от оптимальной. Основная проблема заключается в сложности ее реализации. Один из вариантов реализации предполагает отмечать время последнего обращения к странице; это должно делаться при каждом обращении к памяти, независимо от того, к чему выполняется обращение — к коду или к данным. Даже в случае аппаратной поддержки этой схемы накладные расходы слишком велики. Еще один вариант предполагает поддержание стека обращений к страницам, что тоже обходится недешево для производительности системы.

На рис. 8.14 приведен пример выполнения алгоритма дольше всех неиспользовавшегося элемента с тем же потоком данных, что и для оптимального алгоритма. В этом примере возникают четыре прерывания обращения к странице.

Стратегия “первым вошел — первым вышел” рассматривает кадры страниц процесса как циклический буфер с циклическим же удалением страниц из него. Все, что требуется для реализации этой стратегии, — это указатель, циклически проходящий по кадрам страниц процесса. Таким образом, это одна из простейших в реализации стратегий замещения. Логика ее работы заключается в том, что замещается страница, находящаяся в основной памяти дольше других. Однако далеко не всегда эта страница редко используется; очень часто некоторая область данных или кода интенсивно используется программой, и страницы из этой области при использовании описанной стратегии будут загружаться и выгружаться вновь и вновь.

На рис. 8.14 описанная стратегия приводит к шести прерываниям отсутствия страницы. Заметим, что предыдущая стратегия распознает, что чаще других используются страницы 2 и 5, в то время как стратегия “первым вошел — первым вышел” на это не способна.

Хотя стратегия дольше всех неиспользовавшегося элемента и близка к оптимальной, она трудна в реализации и приводит к значительным накладным расходам. Стратегия “первым вошел — первым вышел” реализуется очень просто, но относительно редко приводит к хорошим результатам. В течение долгого времени разработчики операционных систем испытывали различные алгоритмы, пытаясь достичь увеличения производительности стратегии дольше всех неиспользовавшегося элемента при значительном снижении накладных расходов. Многие из этих алгоритмов представляют собой варианты схемы, известной как **часовая стратегия** (*clock policy*).

В простейшей схеме часовой стратегии с каждым кадром связывается один дополнительный бит, известный как бит использования. Когда страница впервые загружается в кадр, бит использования устанавливается равным 1. При последующих обращениях к странице, вызвавших прерывание из-за отсутствия страницы, этот бит также устанавливается равным 1. Если бит использования равен 1, то страница считается активной. Если же бит использования равен 0, то страница считается неактивной. Время от времени (когда страница не используется достаточно долго) страница считается вытесненной из кадра, если в нем уже находится другая страница, которая считается активной. Время от времени (когда страница не используется достаточно долго) страница считается вытесненной из кадра, если в нем уже находится другая страница, которая считается активной.

вается равным 1. При работе алгоритма замещения множество кадров, являющихся кандидатами на замещение (текущий процесс, локальная область видимости, вся основная память или глобальная область видимости<sup>5</sup>), рассматривается как циклический буфер, с которым связан указатель. При замещении страницы указатель перемещается к следующему кадру в буфере. Когда наступает время замещения страницы, операционная система сканирует буфер для поиска кадра, бит использования которого равен 0. Всякий раз, когда в процессе поиска встречается кадр с битом использования, равным 1, он сбрасывается в 0. Первый же встреченный кадр с нулевым битом использования выбирается для замещения. Если все кадры имеют бит использования, равный 1, указатель совершает полный круг и возвращается к начальному положению, заменяя страницу в этом кадре. Как видим, эта стратегия схожа со стратегией “первым вошел — первым вышел”, но отличается тем, что кадры, имеющие установленный бит использования, пропускаются алгоритмом. Буфер кадров страниц представлен в виде круга, откуда и произошло название стратегии. Ряд операционных систем используют различные варианты часовой стратегии (например, Multics [51]).

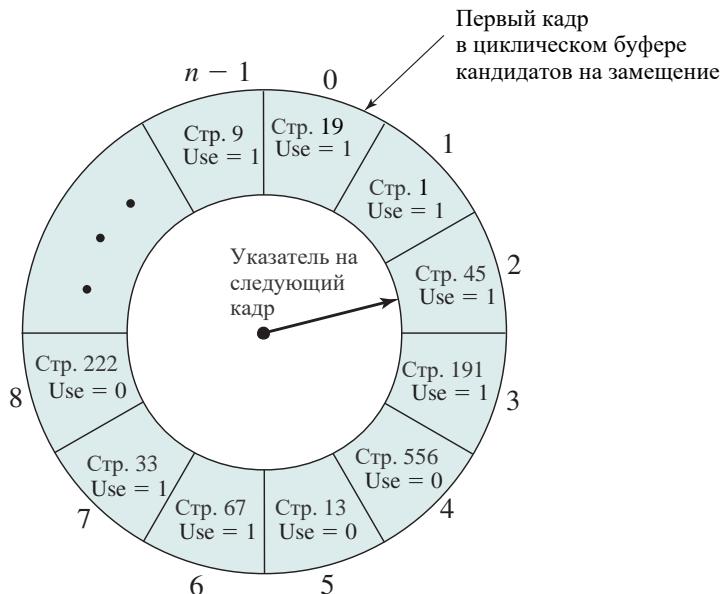
На рис. 8.15 приведен простейший пример использования часовой стратегии. Для замещения доступны *и* кадров основной памяти, представленные в виде циклического буфера. Непосредственно перед тем, как заместить страницу в буфере загружаемой из вторичной памяти страницей 727, указатель буфера указывает на кадр 2, содержащий страницу 45.

Теперь приступим к выполнению часового алгоритма. Поскольку бит использования страницы 45 в кадре 2 равен 1, эта страница не замещается; вместо этого ее бит использования сбрасывается, а указатель перемещается к следующему кадру. Не замещается также страница 191 из кадра 3; в соответствии с алгоритмом сбрасывается ее бит использования. В следующем кадре (номер 4) бит использования страницы равен 0. Таким образом, страница 556 замещается загружаемой в основную память страницей 727, бит использования которой устанавливается равным 1. Далее указатель буфера переходит к кадру 5, и на этом выполнение алгоритма завершается.

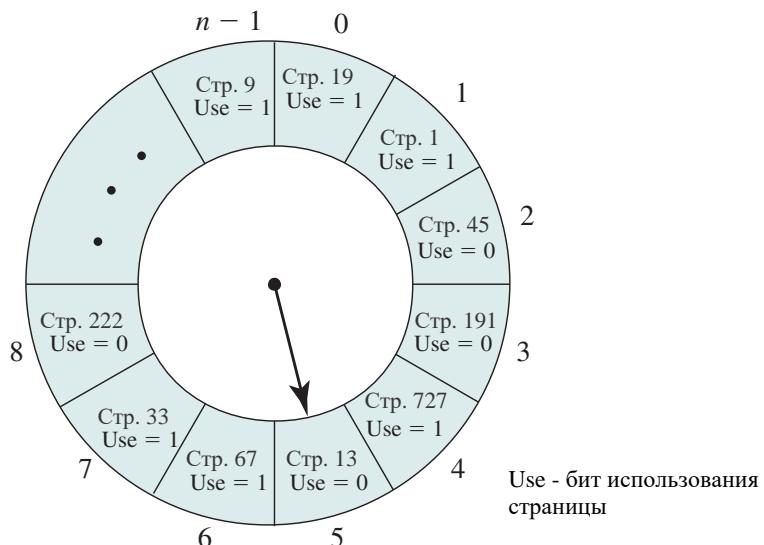
Поведение часового алгоритма проиллюстрировано на рис. 8.14. Звездочка означает, что бит использования соответствующей страницы равен 1, а стрелочка указывает текущее положение указателя. Заметим, что данный алгоритм пытается защитить страницы 2 и 5 от замещения.

На рис. 8.16 показаны результаты эксперимента [16], в котором сравнивались четыре рассмотренных в этом разделе алгоритма; предполагается, что количество отводимых процессу кадров постоянно. Результат основан на выполнении  $0,25 \cdot 10^6$  обращений к памяти в программе на языке FORTRAN с использованием страниц размером 256 слов. Эксперимент проводился с выделением процессу 6, 8, 10, 12 и 14 кадров. Различия в используемых алгоритмах наиболее четко видны при малом количестве кадров (при этом алгоритм “первым вошел — первым вышел” более чем в два раза хуже оптимального). Все четыре кривые на графике имеют тот же вид, что и идеализированное поведение, показанное на рис. 8.10, б. Для эффективной работы желательно находиться справа от перегиба кривой (с небольшим количеством прерываний отсутствия страницы) при сохранении небольшого количества выделенных кадров (слева от перегиба). Эти два ограничения указывают, что желательный режим работы находится в области перегиба кривой.

<sup>5</sup> Концепция области видимости рассматривается в подразделе “Область видимости замещения”.

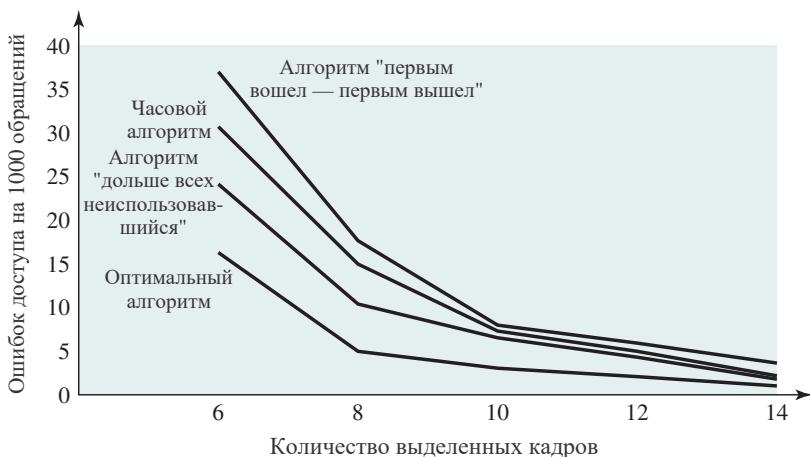


а) Состояние буфера непосредственно перед замещением страницы



б) Состояние буфера непосредственно после замещения страницы

**Рис. 8.15.** Пример работы часового алгоритма



**Рис. 8.16.** Сравнение различных стратегий замещения страниц

Практически такие же результаты представлены в [81], где максимальное отклонение также оказывалось больше чем в 2 раза. В этой работе моделировалось влияние различных стратегий на сгенерированной последовательности обращений к страницам длиной 10 000 обращений к виртуальному пространству из 100 страниц. Для достижения эффекта локальности использовалось экспоненциальное распределение вероятности ссылок к конкретной странице. Отмечалось, что ряд исследователей может прийти к заключению, что в разработке алгоритмов замещения страниц нет особого смысла, если выигрыш оказывается порядка 2 раз. Но на самом деле эта разница будет иметь заметное влияние на требования и к размеру основной памяти (чтобы избежать снижения производительности операционной системы), и к производительности операционной системы (чтобы избежать необходимости увеличения основной памяти).

Проводилось также сравнение часового алгоритма с прочими при выделении процессу переменного количества кадров в глобальной или локальной области видимости [36]. Как выяснилось, по производительности часовой алгоритм наиболее близок к алгоритму дольше всех неиспользовавшегося элемента.

Повысить эффективность часового алгоритма можно путем увеличения количества используемых при его работе битов<sup>6</sup>. Во всех поддерживающих страницу организацию процессорах с каждой страницей в основной памяти (а следовательно, с каждым кадром) связан бит модификации. Этот бит используется для указания того, что данная страница не может быть замещена до тех пор, пока ее содержимое не будет записано обратно во вторичную память. Этот бит может использоваться часовым алгоритмом следующим образом. Принимая во внимание биты использования и модификации, все кадры можно разделить на четыре категории ( $u$  — бит использования,  $m$  — бит модификации):

1. использован давно, не модифицирован ( $u = 0, m = 0$ );
2. использован недавно, не модифицирован ( $u = 1, m = 0$ );
3. использован давно, модифицирован ( $u = 0, m = 1$ );
4. использован недавно, модифицирован ( $u = 1, m = 1$ ).

<sup>6</sup> С другой стороны, уменьшение количества битов до нуля даст нам алгоритм "первым вошел — первым вышел".

Используя эту классификацию, изменим часовой алгоритм, который теперь описывается следующим образом.

1. Сканируем буфер кадров, начиная с текущего положения. В процессе сканирования бит использования не изменяется. Первая же страница с состоянием ( $u = 0$ ,  $m = 0$ ) замещается.
2. Если выполнение первого шага алгоритма не увенчалось успехом, ищем страницу с параметрами ( $u = 0$ ,  $m = 1$ ). Если таковая найдена, она замещается. В процессе выполнения данного шага у всех просмотренных страниц сбрасывается бит использования.
3. Если выполнение предыдущего шага не дало результата, указатель возвращается в исходное положение, но у всех страниц значение бита использованияброшено в 0. Повторим шаг 1 и при необходимости — шаг 2. Очевидно, на этот раз требуемая страница будет найдена.

Итак, часовой алгоритм циклически проходит по всем страницам буфера в поисках страницы, которая не была модифицирована со времени загрузки и давно не использовалась. Такая страница — хороший кандидат на замещение, особенно с учетом того, что ее не надо записывать на диск. Если при первом проходе кандидатов на замещение не нашлось, алгоритм снова проверяет буфер, теперь уже в поисках модифицированной, давно не использовавшейся страницы. Хотя такая страница и должна быть записана перед замещением, в соответствии с принципом локальности она вряд ли понадобится в ближайшем будущем. Если и этот проход окажется неудачным, все страницы помечаются как давно не использованные, и выполняется третий проход.

Такой алгоритм использован в схеме виртуальной памяти Macintosh [90]. Преимущество этого алгоритма состоит, в отличие от простого часового алгоритма, в замене не изменявшихся страниц по сравнению с заменой модифицированных страниц, что дает непосредственную экономию времени.

### **Буферизация страниц**

Хотя алгоритмы дольше всех неиспользовавшегося и часовой и превосходят алгоритм “первым вошел — первым вышел”, они оба сложны и имеют высокие накладные расходы по сравнению с последним. Кроме того, следует учитывать, что стоимость замещения модифицированной страницы выше стоимости замещения немодифицированной, которую не надо записывать во вторичную память.

Есть еще одна интересная стратегия, которая может повысить производительность страничной организации при использовании простейшего алгоритма замещения. Это буферизация страниц, использованная в VAX VMS. В качестве алгоритма замещения страниц используется простейший алгоритм “первым вошел — первым вышел”. Для повышения его производительности замещаемая страница не теряется, а вносится в один из двух списков: в список свободных страниц, если страница не модифицировалась, или в список модифицированных страниц. Заметим, что физически страница не перемещается — вместо этого ее запись удаляется из таблицы страниц и переносится в список свободных или модифицированных страниц.

Список свободных страниц представляет собой список кадров страниц, доступных для чтения. VMS пытается постоянно поддерживать некоторое небольшое количество

свободных кадров. Когда страница считывается в кадр, используется кадр, расположенный в начале списка; при этом страница, находившаяся в нем ранее, уничтожается. При замещении немодифицированной страницы она остается в памяти, а ее кадр добавляется к концу списка свободных страниц; аналогично, когда модифицированная страница записывается во вторичную память и замещается, ее кадр страницы добавляется к списку модифицированных страниц.

Важным аспектом этих перемещений является то, что замещаемые страницы остаются в памяти. Следовательно, если процесс обращается к такой странице, она возвращается в резидентное множество процесса без значительных затрат. По сути, списки свободных и модифицированных страниц работают в качестве кеша страниц. Список модифицированных страниц позволяет записывать их не по одной, а кластерами, что существенно снижает количество операций ввода-вывода, а следовательно, и время обращения к диску.

Более простая версия буферизации страниц реализована в операционной системе Mach [197]. В этой операционной системе не делается различий между модифицированными и немодифицированными страницами.

### **Стратегия замещения и размер кеша**

Как отмечалось ранее, размер основной памяти со временем становится все больше, как, впрочем, и размер приложений, так что локальность снижается. Утешением может служить то, что размеры кешей также увеличиваются. Большие — в несколько мегабайтов — кэши в настоящее время вполне доступны [26]. При использовании кешей большого размера замещение страниц виртуальной памяти может оказывать влияние на производительность. Если кадр страницы, выбранный для замещения, располагается в кеше, то вместе со страницей из блока кеша теряется весь блок.

В системах с использованием буферизации страниц того или иного вида производительность кеша можно увеличить путем добавления к стратегии замещения стратегии размещения страниц в буфере. Большинство операционных систем размещают страницы в буфере в произвольных кадрах, как правило, с использованием алгоритма “первым вошел — первым вышел”. Исследования в [128] показали, что правильный выбор стратегии размещения может привести к уменьшению неудачных поисков в кеше на 10–20%.

В [128] описаны исследования ряда алгоритмов размещения. Подробное их изложение выходит за рамки данной книги, так как они зависят от деталей структуры кеша и используемых стратегий. Суть этих стратегий состоит в размещении последовательных страниц в основной памяти таким образом, чтобы минимизировать количество кадров страниц, отображаемых в одни и те же слоты кеша.

## **Управление резидентным множеством**

Как указывалось ранее в этой главе, часть процесса, которая в действительности располагается в основной памяти в любой момент времени, является резидентным множеством процесса.

### **Размер резидентного множества**

При использовании страничной виртуальной памяти для подготовки процесса к выполнению нет необходимости (да это может быть и невозможно) размещать в основной

памяти все его страницы. Следовательно, операционная система должна принять решение о том, какое количество страниц следует загрузить, т.е. какое количество памяти выделяется конкретному процессу. Здесь играет роль ряд факторов.

- Чем меньше памяти выделяется процессу, тем больше процессов может одновременно находиться в основной памяти. Это увеличивает вероятность того, что операционная система в любой момент времени найдет как минимум один готовый к выполнению процесс и, следовательно, снижаются затраты времени на свопинг процессов.
- При относительно небольшом количестве страниц процесса, размещенных в основной памяти, несмотря на принцип локальности частота возникновения прерываний из-за отсутствия страницы будет достаточно велика (см. рис. 8.10, б).
- После определенного предела дополнительное выделение основной памяти некоторому процессу в соответствии с принципом локальности не будет приводить к сколь-нибудь значительному снижению частоты возникновения прерываний из-за отсутствия страницы.

С учетом этих факторов в современных операционных системах используются два типа стратегий. Стратегия **фиксированного распределения** выделяет процессу фиксированное количество кадров основной памяти, в пределах которого он выполняется. Это количество определяется в момент начальной загрузки (при создании процесса) и может быть определено на основании типа процесса (интерактивный, пакетный и т.п.) либо на основании указаний программиста или системного администратора. При использовании стратегии фиксированного распределения прерывание из-за отсутствия страницы приводит к замещению требуемой страницей одной из страниц процесса.

Стратегия **переменного распределения** позволяет количеству выделенных процессу кадров страниц изменяться во время работы процесса. В идеале процессу, который страдает от большого количества прерываний из-за отсутствия страницы (принцип локальности для данного процесса выполняется слабо), выделяются дополнительные кадры страниц; и напротив, у процесса, при работе которого таких прерываний относительно мало (что указывает на то, что поведение процесса с точки зрения локальности достаточно хорошее), могут быть изъяты кадры в расчете на то, что это не намного увеличит частоту возникновения прерываний. Использование стратегии переменного распределения связано с концепцией области видимости замещения, речь о которой пойдет в следующем подразделе.

Стратегия переменного распределения представляется более мощной, однако трудности данного подхода состоят в том, что операционная система при этом должна отслеживать поведение процессов. Это приводит к очень высоким накладным расходам, зависящим от возможностей аппаратного обеспечения конкретной платформы.

### Область видимости замещения

Область видимости (scope) стратегии замещения можно классифицировать как локальную или глобальную. Стратегии обоих типов активируются прерыванием обращения к странице при отсутствии свободных кадров. **Локальная стратегия замещения** выбирает страницу только среди резидентных страниц того процесса, который стал причиной прерывания. **Глобальная стратегия замещения** рассматривает в качестве кандидатов на замещение все незаблокированные страницы в основной памяти, независимо от принадлежности конкретной страницы тому или иному процессу. Как упоминалось

ранее, когда кадр заблокирован, страница, хранящаяся в нем в настоящее время, не может быть заменена. Разблокированная страница является просто страницей в незаблокированном кадре основной памяти. Хотя локальная стратегия и проще для анализа, нет убедительных доказательств того, что она дает лучшие результаты по сравнению с глобальной стратегией, которая привлекает своей простотой реализации и минимальными накладными расходами [36, 161].

Имеется связь между областью видимости замещения и размером резидентного множества (табл. 8.5). Фиксированное резидентное множество приводит к локальной стратегии замещения — для поддержания фиксированного размера резидентного множества удаляемая из основной памяти страница должна быть замещена другой страницей того же процесса. Стратегия переменного распределения, естественно, совместима с глобальным замещением: замена страницы одного процесса в основной памяти страницей другого процесса приводит к перераспределению размеров содержащихся в основной памяти частей процессов. Мы также узнаем, что переменное распределение может работать и с локальным замещением. А теперь рассмотрим все три возможных сочетания в отдельности.

**Таблица 8.5. УПРАВЛЕНИЕ РЕЗИДЕНТНЫМ МНОЖЕСТВОМ**

	<b>Локальное замещение</b>	<b>Глобальное замещение</b>
<b>Фиксированное распределение</b>	<ul style="list-style-type: none"> <li>• Количество кадров процесса фиксировано</li> <li>• Страница для замещения выбирается среди выделенных процессу кадров</li> </ul>	<ul style="list-style-type: none"> <li>• Невозможно</li> </ul>
<b>Переменное распределение</b>	<ul style="list-style-type: none"> <li>• Количество выделенных процессу кадров может время от времени изменяться</li> <li>• Страница для замещения выбирается среди выделенных процессу кадров</li> </ul>	<ul style="list-style-type: none"> <li>• Страница для замещения выбирается среди всех доступных кадров в основной памяти; это приводит к изменению размера резидентного множества процесса</li> </ul>

### **Фиксированное распределение, локальная область видимости**

Имеется работающий процесс, количество кадров основной памяти которого фиксировано. При прерывании обращения к странице операционная система должна выбрать для замещения страницу среди резидентных страниц данного процесса. Для этого может использоваться один из рассмотренных нами алгоритмов.

При фиксированном распределении необходимо заранее решить вопрос о количестве выделяемых процессу кадров. Это решение может быть принято в зависимости от типа приложения и количества памяти, запрашиваемого программой. У такого подхода основной недостаток — двойной: если выделить процессу слишком малую память, получим высокую частоту возникновения прерываний обращения к памяти, что, в свою очередь, приведет к снижению производительности многозадачной системы; если же выделить процессу неоправданно много памяти, то в основной памяти удастся разместить слишком мало программ, и производительность системы будет снижена за счет необходимости частого выполнения свопинга.

## *Переменное распределение, глобальная область видимости*

Эта комбинация, вероятно, наиболее проста в реализации и принята во многих операционных системах. В любой момент времени в основной памяти имеется несколько процессов, каждому из которых выделено некоторое количество кадров. Обычно операционная система поддерживает также список свободных кадров. При возникновении прерывания обращения к странице к резидентному множеству процесса добавляется свободный кадр и затребованная страница загружается в него. Таким образом, размер процесса постепенно растет, что должно снижать общее количество прерываний из-за отсутствия страницы в системе.

Сложность при таком подходе заключается в выборе страницы для замещения. Когда свободные кадры оказываются израсходованными, операционная система должна выбрать для замещения страницу, находящуюся в данный момент в основной памяти. Этот выбор производится из всех незаблокированных страниц в памяти. При использовании любой из рассмотренных ранее стратегий выбираемая страница может принадлежать любому из резидентных процессов; не существует способа определения того, какой из процессов должен потерять страницу из своего резидентного множества. Таким образом, снижение размера резидентного множества процесса может оказаться не оптимальным.

Одним из способов учета потенциальных проблем с производительностью при переменном распределении с глобальной областью видимости является использование буферизации страниц. В этом случае выбор замещаемой страницы играет меньшую роль, так как страница может быть восстановлена, если обращение к ней будет выполнено до очередного перемещения на диск.

## *Переменное распределение, локальная область видимости*

С помощью данной стратегии делается попытка преодолеть проблемы, возникающие при использовании стратегии глобальной области видимости. Вкратце ее можно описать следующим образом.

1. При загрузке нового процесса в основную память ему в качестве резидентного множества выделяется некоторое количество кадров страниц; количество кадров определяется исходя из типа приложения, запроса программы или на основе других критериев. Для заполнения резидентного множества используется стратегия выборки по требованию либо предварительная выборка.
2. При возникновении прерывания из-за отсутствия страницы страница для замещения выбирается среди резидентного множества процесса, генерировавшего прерывание.
3. Время от времени выполняется переоценка распределения памяти процессам, которая приводит к увеличению или уменьшению размера выделяемой процессу памяти для повышения общей производительности системы.

При использовании данной стратегии решение об увеличении или уменьшении размера резидентного множества принимается на основе оценки ожидаемых требований активных процессов. Такая оценка делает эту стратегию более сложной, чем простая стратегия глобального замещения, но приводит к повышению производительности системы.

Ключевыми элементами стратегии переменного распределения с локальной областью видимости являются критерии, используемые для определения размера резидентного множества и момента внесения изменений. Одна из стратегий, чаще других упомина-

емая в литературе, известна как **стратегия рабочего множества** (working set strategy). Хотя ее реализация очень сложна, следует изучить данную стратегию хотя бы как критерий для оценки других.

Рабочее множество представляет собой концепцию, введенную Деннингом (Denning) и популяризованную в работах [60, 62, 63]; эта концепция оказала большое влияние на разработку систем управления виртуальной памятью. Рабочее множество  $W(t, \Delta)$  с параметром  $\Delta$  процесса в виртуальный момент времени  $t$  представляет собой множество страниц, к которым процесс обращался за последние  $\Delta$  единиц виртуального времени.

Здесь мы используем концепцию виртуального времени, которое определяется следующим образом. Рассмотрим последовательность обращений к памяти  $r(1), r(2), \dots$ , где  $r(i)$  представляет собой страницу, которая содержит  $i$ -й виртуальный адрес, сгенерированный данным процессом. Время измеряется в обращениях к памяти; таким образом,  $t = 1, 2, 3, \dots$  представляет собой внутреннее виртуальное время процесса.

Рассмотрим каждую из двух переменных  $W$ . Переменная  $\Delta$  — это “окно” виртуального времени, сквозь которое мы наблюдаем за процессом. Размер рабочего множества представляет собой неубывающую функцию от размера окна. На рис. 8.17 (взятом из [14]) показаны последовательности обращений процесса к страницам.

Последовательность обращений  
к страницам  $W$

Размер окна,  $\Delta$

	2	3	4	5
24	24	24	24	24
15	24 15	24 15	24 15	24 15
18	15 18	24 15 18	24 15 18	24 15 18
23	18 23	15 18 23	24 15 18 23	24 15 18 23
24	23 24	18 23 24	•	•
17	24 17	23 24 17	18 23 24 17	15 18 23 24 17
18	17 18	24 17 18	•	18 23 24 17
24	18 24	•	24 17 18	•
18	•	18 24	•	24 17 18
17	18 17	24 18 17	•	•
17	17	18 17	•	•
15	17 15	17 15	18 17 15	24 18 17 15
24	15 24	17 15 24	17 15 24	•
17	24 17	•	•	17 15 24
24	•	24 17	•	•
18	24 18	17 24 18	17 24 18	15 17 24 18

Рис. 8.17. Зависимость размера рабочего множества процесса от размера окна

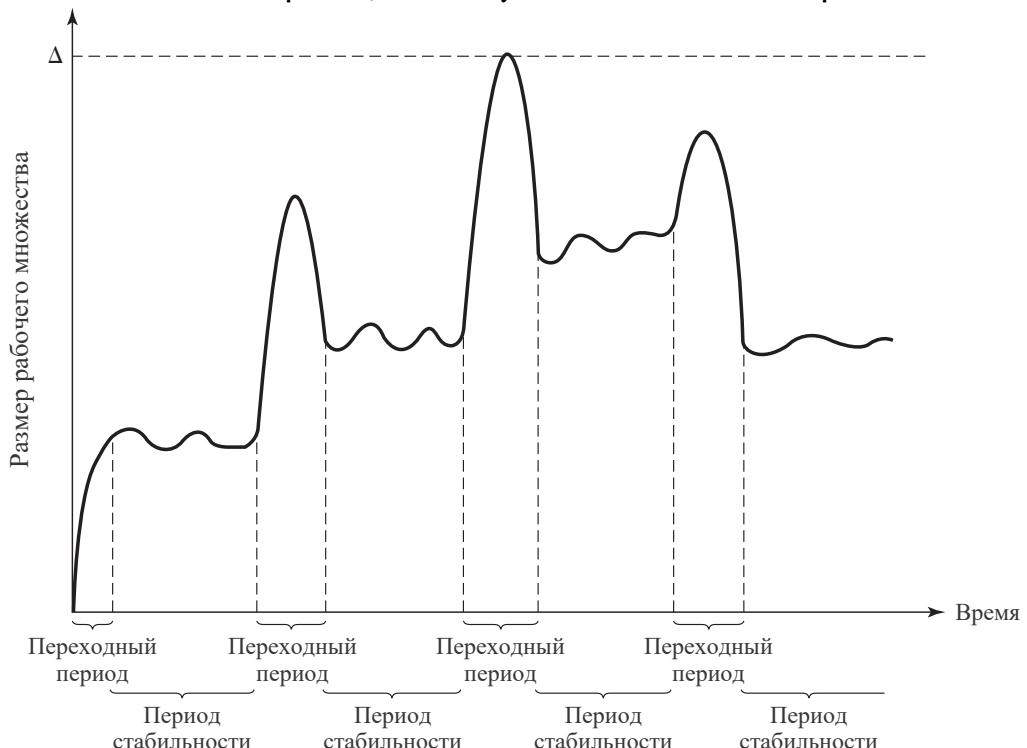
Точки обозначают моменты времени, когда рабочее множество не изменялось. Обратите внимание — чем больше размер окна, тем больше и рабочее множество. Это можно выразить следующим соотношением:

$$W(t, \Delta + 1) \supseteq W(t, \Delta)$$

Рабочее множество является функцией и от времени. Если продолжительность процесса более  $\Delta$  единиц времени и он использует только одну страницу, то  $|W(t, \Delta)| = 1$ . Рабочее множество процесса может расти до размера количества страниц процесса  $N$ , если при его выполнении происходят обращения к различным страницам и если это позволяет выбранный размер окна, т.е.

$$1 \leq |W(t, \Delta)| \leq \min(\Delta, N)$$

На рис. 8.18 показан один из вариантов изменения во времени размера рабочего множества при фиксированном значении  $\Delta$ . Для многих программ периоды относительно стабильного рабочего множества чередуются с периодами быстрых изменений. В начале выполнения процесса рабочее множество быстро растет за счет обращения к новым страницам. В конечном итоге в соответствии с принципом локальности процесс должен стабилизироваться с определенным множеством страниц. Последующие переходные периоды отражают изменение стабильного состояния. В это время некоторые старые страницы все еще остаются в пределах окна  $\Delta$ , вызывая всплеск размера рабочего множества при обращении процесса к новым страницам. Постепенно старые страницы уходят из окна, и в окне остаются только страницы, соответствующие новой локальности процесса.



**Рис. 8.18.** Типичная диаграмма изменения размера рабочего множества [161]

Концепция рабочего множества может использоваться стратегией определения размера резидентного множества.

1. Отслеживаем рабочее множество каждого процесса.
2. Периодически удаляем из резидентного множества страницы, не входящие в рабочее множество. По сути, это стратегия последнего использовавшегося.
3. Процесс может выполняться только тогда, когда его рабочее множество находится в основной памяти (т.е. его резидентное множество включает рабочее множество).

Эта стратегия, использующая принцип локальности, должна минимизировать количество прерываний из-за отсутствия страниц, но, к сожалению, при этом возникает ряд проблем.

1. По прошлому не всегда можно судить о будущем. Как размер рабочего множества, так и его состав время от времени изменяются (см., например, рис. 8.18).
2. Определение истинного рабочего множества каждого процесса непрактично. Для этого необходимо помечать время обращения каждого процесса к каждой странице с использованием виртуального времени процесса, а также поддерживать упорядоченную по времени обращения очередь страниц для каждого процесса.
3. Оптимальное значение  $\Delta$  неизвестно и для разных ситуаций может быть различным.

Тем не менее сама идея данной стратегии вполне корректна, и ряд операционных систем пытаются к ней приблизиться. Один из способов сделать это заключается не в работе с конкретными обращениями к страницам, а в работе с уровнем генерации данным процессом прерываний из-за отсутствия страницы. Как показано на рис. 8.10, б, с ростом резидентного множества процесса уровень генерации прерываний падает. Таким образом, вместо непосредственного отслеживания рабочего множества мы можем получить сравнимые результаты путем отслеживания уровня генерации прерываний. Если уровень генерации прерываний у какого-то процесса ниже некоторого минимального порога, система может выиграть, назначив данному процессу резидентное множество меньшего размера (и освободив кадры основной памяти для других процессов) без ущерба для этого процесса. Если же для некоторого процесса уровень генерации прерываний превысил некоторое максимальное пороговое значение, то следует по возможности увеличить размер его резидентного множества.

Соответствующий этой стратегии алгоритм называется алгоритмом **частоты прерываний обращения к странице** (page fault frequency — PFF) [45, 96]. Этот алгоритм требует наличия у каждой страницы в основной памяти бита использования, устанавливаемого равным 1 при обращении к странице. Когда возникает прерывание обращения к странице, операционная система замечает виртуальное время с момента последней генерации прерывания из-за отсутствия страницы данным процессом; это осуществляется посредством счетчика обращений к страницам. Если прошедшее с момента последнего прерывания время меньше некоторого определенного порога  $F$ , то страница добавляется к резидентному множеству процесса. В противном случае все страницы с битом использования, равным 0, сбрасываются, и соответственно, уменьшается резидентное множество процесса. В этот же момент битам использования всех оставшихся страниц присваивается нулевое значение. Стратегия может быть усовершенствована с помощью двух пороговых значений — верхнего порога, используемого для роста резидентного множества, и нижнего, по достижении которого резидентное множество уменьшается.

Промежутки времени между прерываниями обращения к странице соответствуют частоте генерации прерываний. Хотя лучшим методом представляется измерение средней частоты генерации прерываний обращения к странице, измерение промежутков времени между ними представляет собой вполне разумный компромисс, позволяющий принимать решения о размере резидентного множества. При использовании такой стратегии совместно с буферизацией страниц должны получаться неплохие результаты.

Тем не менее этот подход имеет один существенный недостаток, заключающийся в его неработоспособности в момент перехода процесса из одного состояния локальности в другое. В этот момент частота генерации прерываний обращения к страницам резко возрастает, что, в соответствии с рассмотренным алгоритмом, вызывает резкое увеличение размера резидентного множества и может привести к таким нежелательным результатам, как деактивация и свопинг других процессов.

Обойтись небольшими накладными расходами в переходные периоды призвана стратегия **рабочего множества с переменным пробным интервалом** (variable-interval sampled working set — VSWS) [79]. Эта стратегия вычисляет размер рабочего множества процесса по выборкам, основанным на истекшем виртуальном времени. В начале интервала выборки бит использования всех резидентных страниц процесса сбрасывается; в конце интервала бит использования установлен только у тех страниц, к которым на этом интервале было обращение. Эти страницы остаются в резидентном множестве в течение следующего интервала времени; остальные страницы сбрасываются. Таким образом, размер резидентного множества может уменьшаться только в конце каждого интервала. В течение интервала страницы, вызвавшие ошибку обращения, добавляются к резидентному множеству (таким образом, в это время размер резидентного множества не убывает).

Стратегия VSWS управляет тремя параметрами:

- $M$  — минимальная продолжительность интервала выборки;
- $L$  — максимальная продолжительность интервала выборки;
- $Q$  — допустимое количество прерываний обращения к странице, которые могут произойти между интервалами выборки.

Стратегия VSWS работает следующим образом.

1. Если виртуальный промежуток времени с момента последнего пробного интервала достиг  $L$ , процесс приостанавливается и выполняется сканирование битов использования.
2. Если до истечения виртуального времени  $L$  произошло  $Q$  прерываний обращения, то:
  - а) если виртуальное время с момента последнего пробного интервала меньше  $M$ , мы ожидаем, пока пройдет этот промежуток времени, чтобы приступить к сканированию битов использования;
  - б) если же это время не меньше  $M$ , процесс приостанавливается и начинается сканирование битов использования.

Значения параметров выбираются такими, чтобы обычно процесс сканирования запускался по достижении  $Q$ -го прерывания обращения к странице (случай 2, б); остальные два параметра ( $M$  и  $L$ ) служат граничными значениями для исключительных условий. С помощью стратегии VSWS предпринимается попытка снизить пиковые тре-

бования к памяти, вызываемые переходами процесса от одной локальности к другой путем сброса неиспользуемых страниц при учащении генерации прерываний обращения. Опыт использования этой стратегии в операционной системе для мейнфреймов GCOS 8 показывает, что стратегия VSWS столь же проста в реализации, как и стратегия PFF, но при этом более эффективна [192].

## Стратегия очистки

Стратегия очистки является противоположностью стратегии выборки. Ее задача состоит в определении момента, когда измененная страница должна быть записана во вторичную память. Два основных ее метода — очистка по требованию и предварительная очистка. При очистке по требованию страница записывается во вторичную память только тогда, когда она выбирается для замещения. Предварительная очистка записывает модифицированные страницы до того, как потребуются занимаемые ими кадры, так что эти страницы могут записываться целыми пакетами.

Прямолинейное следование любой из стратегий чревато неприятностями. При предварительной очистке записанная страница остается в основной памяти до тех пор, пока ее не удалит оттуда алгоритм замещения. Предварительная очистка позволяет записывать страницы пакетами, но не имеет смысла записывать сотни или тысячи страниц только для того, чтобы убедиться, что до замещения они вновь успели модифицироваться. Пропускная способность вторичной памяти ограничена и не должна засоряться излишними операциями очистки.

С другой стороны, при очистке по требованию запись модифицированной страницы сопровождается чтением новой страницы, предшествуя ей, так что несмотря на минимизацию записей страниц прерывание обращения может вызывать пересылку двух страниц между основной и вторичной памятью и тем самым снижать эффективность использования процессора.

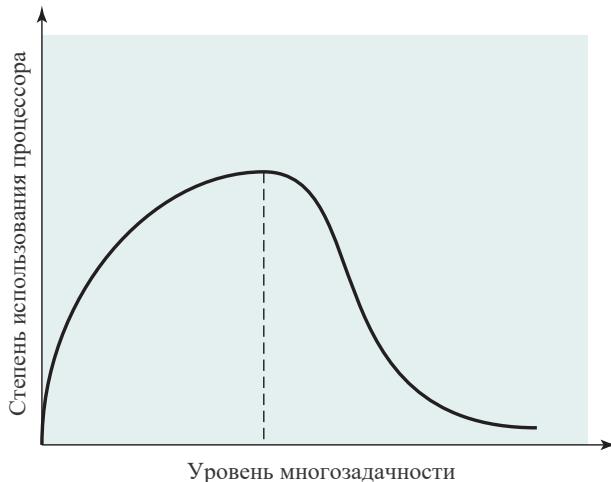
Улучшенный подход включает буферизацию страниц, что позволяет принять следующую стратегию: очищать только замещаемые страницы, но при этом разделять операции очистки и замещения. При использовании буферизации страниц замещаемые страницы могут находиться в двух списках: модифицированных и немодифицированных страниц. Страницы из списка модифицированных могут периодически записываться пакетами и переноситься в список немодифицированных. Страница из списка немодифицированных страниц может либо быть удалена из него при обращении к ней, либо потеряна при загрузке в ее кадр новой страницы.

## Управление загрузкой

Управление загрузкой — это определение количества процессов, которые будут резидентными в основной памяти. Стратегия управления загрузкой представляет собой критическую часть эффективно работающей системы управления памятью. Если одновременно резидентны только несколько процессов, то очень часто будет возникать ситуация, в которой все процессы будут оказываться заблокированными, и системе придется тратить излишнее время на осуществление свопинга. С другой стороны, если разместить в основной памяти очень много процессов, то в среднем размер резидентного множества каждого процесса окажется довольно малым, что приведет к слишком частой генерации ошибки обращения и снижению пропускной способности системы.

## Уровень многозадачности

Снижение пропускной способности проиллюстрировано на рис. 8.19. При возрастании уровня многозадачности от малых значений эффективность использования процессора возрастает в связи с уменьшением вероятности одновременного блокирования всех процессов. Однако через некоторое время достигается состояние, когда средний размер резидентного множества становится неадекватным. Это приводит к существенному росту количества прерываний обращений к странице и как следствие к снижению эффективности использования процессора.



**Рис. 8.19.** Влияние уровня многозадачности на степень использования процессора

Имеется ряд вариантов решения этой проблемы. Алгоритмы рабочего множества или частоты генерации прерываний неявным образом включают управление загрузкой. Выполняться могут только те процессы, резидентные множества которых достаточно велики. Обеспечивая для каждого активного процесса требуемый размер резидентного множества, используемая стратегия автоматически (и динамически) определяет количество активных программ.

Еще один подход, известный под названием *критерий L = S*, был предложен Деннингом и его коллегами [63]. При этом подходе уровень многозадачности настраивается таким образом, чтобы среднее время между прерываниями равнялось среднему времени, требующемуся для обработки прерывания. В результате исследований сделан вывод, что этот уровень многозадачности обеспечивает максимальную производительность процессора. Использование стратегии с аналогичным эффектом, предложенной в [149] и известной как *критерий 50%*, способствует поддержке степени использования устройства хранения страниц на уровне 50%. Исследования показывают, что и при этом методе также достигается максимальная степень использования процессора.

Еще один подход состоит в адаптации часового алгоритма замещения, описанного ранее (см. рис. 8.15). В [36] описана методика с использованием глобальной области видимости, которая включает отслеживание частоты сканирования циклического буфера кадров. Если частота меньше предопределенного нижнего порога, это указывает на то, что возникла одна из следующих ситуаций.

1. Генерируется малое количество прерываний из-за отсутствия страницы, что приводит к малому количеству перемещений указателя.
2. Среднее количество сканируемых для каждого запроса кадров мало; это говорит о том, что в системе много резидентных страниц, к которым не было обращений и которые могут быть немедленно замещены.

В обоих случаях уровень многозадачности может быть безопасно увеличен. С другой стороны, если частота сканирования превышает верхний порог, это указывает либо на большое количество прерываний из-за отсутствия страницы, либо на сложность обнаружения страниц для замещения, что свидетельствует о слишком высоком уровне многозадачности.

### Приостановка процессов

При необходимости снижения степени многозадачности один или несколько резидентных в настоящее время процессов должны быть приостановлены (выгружены во вторичную память). В [36] перечислены шесть возможностей.

- **Процесс с наименьшим приоритетом.** Так реализована стратегия планировщика, не имеющая отношения к вопросам производительности.
- **Процесс, вызывающий прерывания.** Данный выбор основан на большой вероятности того, что у процесса, генерирующего прерывания из-за отсутствия страницы, рабочее множество не резидентно, и суммарная производительность системы не пострадает при приостановке данного процесса. Кроме того, при таком выборе блокируется процесс, который и так практически все время находится в заблокированном состоянии, так что его приостановка приводит к снижению накладных расходов, связанных с замещением страниц и операциями ввода-вывода.
- **Последний активированный процесс.** Маловероятно, что у этого процесса рабочее множество резидентно.
- **Процесс с минимальным резидентным множеством.** Этот выбор минимизирует будущие затраты на загрузку данного процесса. К сожалению, таковыми являются процессы с высокой степенью локальности.
- **Наибольший процесс.** При этом выборе мы освобождаем большое количество кадров перегруженной процессами памяти, снижая тем самым количество процессов, которые должны быть деактивированы.
- **Процесс с максимальным остаточным окном выполнения.** В большинстве схем планирования процесс может выполняться только определенное количество квантов времени до прерывания и перемещения его в конец очереди активных процессов. Данный выбор приближается к стратегии планирования, предоставляющей преимущество процессам с наименьшим временем работы.

Как и во многих других областях разработки операционных систем, выбор стратегии основан на здравом смысле и зависит от множества факторов, например от характеристик выполняемых в системе программ.

## 8.3. УПРАВЛЕНИЕ ПАМЯТЬЮ В UNIX И SOLARIS

Поскольку операционная система UNIX разрабатывалась как машинно-независимая, система управления памятью в UNIX варьируется от одной операционной системы к другой. Ранние версии UNIX использовали переменное распределение памяти без применения виртуальной памяти. Текущие реализации UNIX и Solaris используют страничную виртуальную память.

В SVR4 и Solaris, по сути, имеются две раздельные схемы управления памятью. **Страницчная система** обеспечивает реализацию возможностей виртуальной памяти, распределяя кадры основной памяти среди процессов, а также среди буферов диска. Хотя описанная схема эффективно работает с пользовательскими процессами и дисковым вводом-выводом, страницчная виртуальная память мало приспособлена для управления памятью ядра. Для этой цели используется **распределение памяти ядра**. Рассмотрим оба механизма.

### Страницчная система

#### Структуры данных

Для страницочной виртуальной памяти UNIX использует ряд структур данных, которые (с минимальной коррекцией) являются машинно-независимыми (см. рис. 8.20 и табл. 8.6).

- **Страница таблиц.** Обычно для каждого процесса используется одна таблица страниц, в которой каждой странице виртуальной памяти процесса соответствует одна запись.
- **Дескриптор дискового блока.** В этой таблице каждой странице процесса соответствует запись, описывающая дисковую копию этой страницы.
- **Таблица кадров страниц.** Описывает каждый кадр реальной памяти; таблица проиндексирована номерами кадров. Эта таблица используется алгоритмом замещения.
- **Таблица использования свопинга.** Для каждого устройства свопинга имеется своя таблица, в которой для каждой страницы на этом устройстве имеется своя запись.

Большинство полей, определенных в табл. 8.6, не требуют пояснений. Добавим только несколько комментариев. Поле *возраста* в записи таблицы страниц указывает, как давно программа не обращалась к этому кадру. Размер и частота обновления этого поля зависят от конкретной реализации. Таким образом, нет универсального использования операционной системой UNIX этого поля при реализации стратегии замещения страниц.

Наличие поля типа памяти в дескрипторе дискового блока необходимо по следующей причине: когда выполнимый файл используется для создания нового процесса, в реальную память может быть загружена только часть кода и данных.

Номер кадра страницы	Возраст	Копирование при записи	Модифицирована	Обращения	В памяти	Защищенность			
а) Запись таблицы страниц									
Номер устройства свопинга	Номер блока устройства			Тип памяти					
б) Дескриптор дискового блока									
Состояние страницы	Количество ссылок	Логическое устройство	Номер блока	Указатель на данные кадра					
в) Запись таблицы кадров									
Количество ссылок	Номер страницы/единицы памяти								
г) Запись таблицы свопинга									

**Рис. 8.20.** Структуры данных системы управления памятью UNIX SVR4**Таблица 8.6. ПАРАМЕТРЫ УПРАВЛЕНИЯ ПАМЯТЬЮ UNIX SVR4****Запись таблицы страниц****Номер кадра страницы**

Указывает кадр в реальной памяти.

**Возраст**

Указывает, как долго страница находится в памяти без обращения к ней. Длина и содержимое данного поля зависят от используемого процессора.

**Копирование при записи**

Устанавливается, когда страница совместно используется несколькими процессами. Если один из процессов производит запись в страницу, сначала должны быть сделаны отдельные копии страницы для каждого из совместно использующих ее процессов. Эта возможность позволяет отложить операцию копирования до тех пор, пока она не станет необходима, и избежать ее в тех случаях, когда она не является таковой.

**Модифицирована**

Указывает, изменено ли содержимое страницы.

**Обращения**

Указывает, что к странице были обращения. Этот бит устанавливается равным нулю при первой загрузке страницы и может периодически сбрасываться алгоритмом замещения страниц.

**В памяти**

Указывает, что страница находится в основной памяти.

**Зашщищенность**

Указывает, что разрешена операция записи.

**Дескриптор дискового блока****Номер устройства свопинга**

Номер логического устройства вторичной памяти, хранящей соответствующую страницу. Позволяет использовать для свопинга больше одного устройства.

**Номер блока устройства**

Расположение блока страницы на устройстве вторичной памяти.

**Тип памяти**

Вторичная память может представлять собой модуль свопинга или выполнимый файл. В последнем случае имеется признак, указывающий, должна ли распределяемая виртуальная память быть предварительно очищенной.

**Запись таблицы кадров****Состояние страницы**

Указывает, свободен ли кадр или содержит страницу. В этом случае указывает статус страницы: на устройстве свопинга, в выполнимом файле или выполняется прямое обращение к памяти.

**Количество ссылок**

Количество процессов, обращающихся к странице.

**Логическое устройство**

Логическое устройство, содержащее копию страницы.

**Номер блока**

Расположение блока копии страницы на логическом устройстве.

**Указатель на данные кадра**

Указатель на другие записи таблицы в списке свободных страниц и в хеш-очереди страниц.

**Таблица свопинга****Количество ссылок**

Количество записей таблицы страниц, указывающих на страницы на устройстве свопинга.

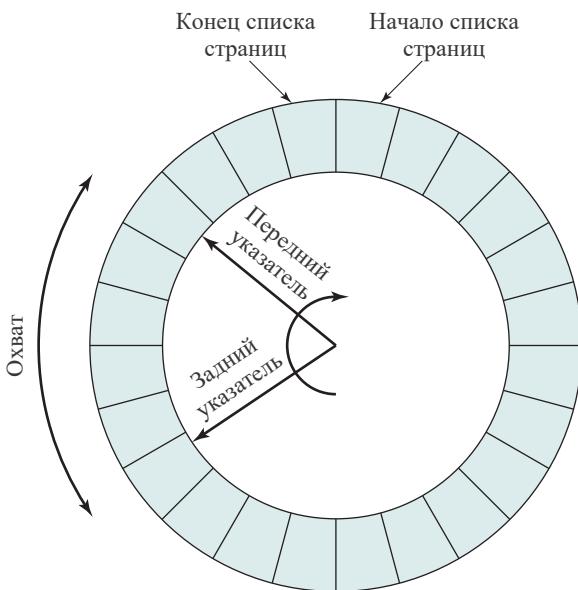
**Номер страницы/единицы памяти**

Идентификатор страницы в модуле вторичной памяти.

Позже, при возникновении прерывания из-за отсутствия страницы, в память загружаются новые порции кода или данных. Страницы виртуальной памяти создаются и связываются с определенными положениями на устройстве свопинга только в момент первоначальной загрузки. В этот момент операционная система решает, следует ли очистить (установить равными 0) ячейки кадра страницы перед первой загрузкой блока кода или данных.

### Замещение страниц

Для замещения страниц используется таблица кадров. Для создания списков в этой таблице применяются несколько указателей. Все доступные кадры объединены в один список свободных кадров, в которых могут размещаться страницы. Когда количество доступных страниц становится ниже некоторого порогового значения, ядро в качестве компенсации отдает ряд страниц загруженных процессов.



**Рис. 8.21.** Часовой алгоритм замещения с двумя стрелками

Алгоритм замещения страниц, использованный в SVR4, представляет собой усовершенствованный часовой алгоритм (см. рис. 8.15), известный под названием “часовой алгоритм с двумя стрелками” (рис. 8.21). Этот алгоритм использует бит обращений из записи таблицы страниц для каждой из страниц памяти, которая может быть выгружена из основной памяти (т.е. не заблокирована). Этот бит устанавливается равным 0 при первоначальной загрузке страницы и равным 1 — при обращении к ней для чтения или записи. Передний указатель проходит по страницам, содержащимся в списке пригодных для выгрузки страниц, и устанавливает бит обращений каждой из них равным 0. Несколько позже по тем же страницам проходит задний указатель и проверяет бит обращений. Если он равен 1, значит, к данной странице было обращение между проверками ее передним и задним указателями, и такая страница игнорируется. Страница же, бит обращений которой остался равным 0, переносится в список выгружаемых страниц.

Работа алгоритма определяется двумя параметрами.

1. **Частота сканирования.** Частота, с которой указатели сканируют список страниц (в страницах в секунду).
2. **Охват.** Промежуток между передним и задним указателями.

Эти параметры в процессе загрузки операционной системы получают значения по умолчанию, основанные на количестве физической памяти. Частота сканирования в процессе работы может изменяться, чтобы соответствовать изменяющимся условиям работы системы. Параметр линейно изменяется от минимального значения до максимального, определенных при настройке системы, с изменением количества свободной памяти от максимального до минимального. Иными словами, чем меньше свободной памяти в системе, чем чаще выполняется сканирование. Охват определяет интервал между указателями и вместе с частотой сканирования определяет промежуток времени, в течение которого к странице должно произойти обращение, чтобы она оставалась в основной памяти.

## Распределение памяти ядра

Ядро в процессе работы часто генерирует и уничтожает маленькие таблицы и буферы, память для каждого из которых выделяется динамически. В [261] перечислены следующие примеры.

- Преобразование имени пути может запросить буфер для копирования имени из пользовательского пространства.
- Подпрограмма `allocb()` выделяет буферы произвольного размера.
- Ряд реализаций UNIX выделяют “зомби”-структуры для хранения информации о состоянии выхода и использовании ресурсов завершенными процессами.
- В SVR4 и Solaris ядро динамически распределяет множество объектов (таких, как структуры процессов, блоки дескрипторов файлов и др.).

Размер большинства этих блоков гораздо меньше типичного размера страницы памяти, и, соответственно, использование страничного механизма в данном случае крайне неэффективно. В SVR4 используется модификация системы двойников (описанной в разделе 7.2).

Стоимость выделения свободного блока памяти в системе двойников меньше, чем в случае использования стратегий первого или наилучшего подходящего [136]. Однако при управлении памятью ядра выделение и освобождение памяти должно выполняться с максимально возможной скоростью. Недостатком же системы двойников являются затраты времени на разделение и слияние блоков.

Беркли (Barkley) и Ли (Lee) из AT&T предложили модификацию, известную как “ленивая” система двойников [18], которая принята в SVR4. Авторами было замечено, что UNIX часто демонстрирует устойчивое состояние памяти ядра, т.е. количество требующихся блоков определенного размера мало меняется со временем. Таким образом, вполне возможна ситуация, когда освобождающийся блок размером  $2^i$  сливаются со своим двойником в блок размером  $2^{i+1}$ , который тут же вновь разделяется на два блока размером  $2^i$  в соответствии с запросом системы. Чтобы избежать излишних слияний и разделений блоков, слияние освобожденных блоков откладывается до того момента, когда оно оказывается действительно необходимым (и тогда производится максимально возможное количество слияний блоков).

В модифицированной таким образом системе двойников используются следующие параметры.

- $N_i$  — текущее количество блоков размером  $2^i$ .
- $A_i$  — текущее количество занятых блоков размером  $2^i$ .
- $G_i$  — текущее количество глобально свободных блоков размером  $2^i$ . (Это блоки, пригодные для слияния со своими двойниками. Когда двойник такого блока становится глобально свободным, эти два блока сливаются в глобально свободный блок размером  $2^{i+1}$ . Все свободные блоки в системе двойников могут рассматриваться как глобально свободные.)
- $L_i$  — текущее количество локально свободных блоков размером  $2^i$ . (Это блоки, не пригодные для слияния. Даже если двойник такого блока становится свободным, эти два блока не сливаются, а остаются в ожидании последующих запросов на блоки данного размера.)

Выполняется следующее соотношение:

$$N_i = A_i + G_i + L_i$$

В целом такая “ленивая” система двойников пытается поддерживать пул локально свободных блоков и производит слияние, только когда количество локально свободных блоков превышает предопределенный порог (при наличии слишком большого количества локально свободных блоков возрастает вероятность недостатка блоков большего размера для удовлетворения требований системы). В основном при освобождении блока слияние не выполняется, что минимизирует накладные расходы. Никаких различий между локально и глобально свободными блоками при выделении блока в ответ на запрос системы не делается.

Для слияния используется критерий, согласно которому количество локально свободных блоков данного размера не должно превышать количество занятых блоков этого размера (т.е. должно выполняться условие  $L_i \leq A_i$ ). Это вполне разумный принцип для ограничения количества локально свободных блоков; эксперименты, описанные в [18], подтверждают, что такая схема приводит к значительному снижению стоимости распределения памяти.

Для реализации описанной схемы ее авторы определили переменную задержки

$$D_i = A_i - L_i = N_i - 2L_i - G_i$$

Алгоритм схемы приведен на рис. 8.22.

## 8.4. УПРАВЛЕНИЕ ПАМЯТЬЮ В LINUX

Многие характеристики схем управления памятью других реализаций UNIX применимы и к Linux, однако эта операционная система имеет и свои, присущие только ей, особенности. Вообще говоря, система управления памятью в Linux весьма сложна [71], и здесь мы дадим только краткое описание двух основных аспектов управления памятью в Linux: виртуальной памяти процесса и распределение памяти ядра. Основной единицей памяти является физическая страница, которая представлена в ядре Linux соответствующей структурой. Размер страницы зависит от архитектуры; обычно он равен 4 Кбайт.

Начальное значение  $D_i$  равно 0

После выполнения операций значение  $D$  изменяется следующим образом.

**(I)** Если следующая операция является запросом на выделение блока:

если имеется свободный блок, выбрать его

если выбранный блок локально свободен,

то  $D_i := D_i + 2$

иначе  $D_i := D_i + 1$

в противном случае

получить два блока путем разделения большего

блока на два меньших (рекурсивная операция).

выбрать один из них, пометив второй как локально свободный.

$D_i$  остается неизменным (при этом значение  $D$

других размеров блоков может измениться в связи с использованием рекурсивности).

**(II)** Если следующая операция – запрос на освобождение блока:

при  $D_i \geq 2$

пометить блок как локально свободный и освободить его локально

$D_i := D_i - 2$

при  $D_i = 1$

пометить блок как глобально свободный и освободить его глобально; если это возможно, выполнить слияние блоков.

$D_i := 0$

при  $D_i = 0$

Пометить блок как глобально свободный и освободить его глобально; если это возможно, выполнить слияние блоков.

Выбрать один локально свободный блок размером  $2^i$  и освободить его глобально; если это возможно, выполнить слияние блоков.

$D_i := 0$

**Рис. 8.22.** Алгоритм “ленивой” системы двойников

Linux также поддерживает возможность Hugepages, позволяющую задавать большие размеры страниц (например, 2 Мбайт). Существует несколько проектов, которые используют Hugepages для повышения производительности. Например, Data Plane Development Kit (<http://dpdk.org/>) использует Hugepages для буферов пакетов, и это применение уменьшает количество обращений к TLB в системе по сравнению с использованием страниц размером 4 Кбайт.

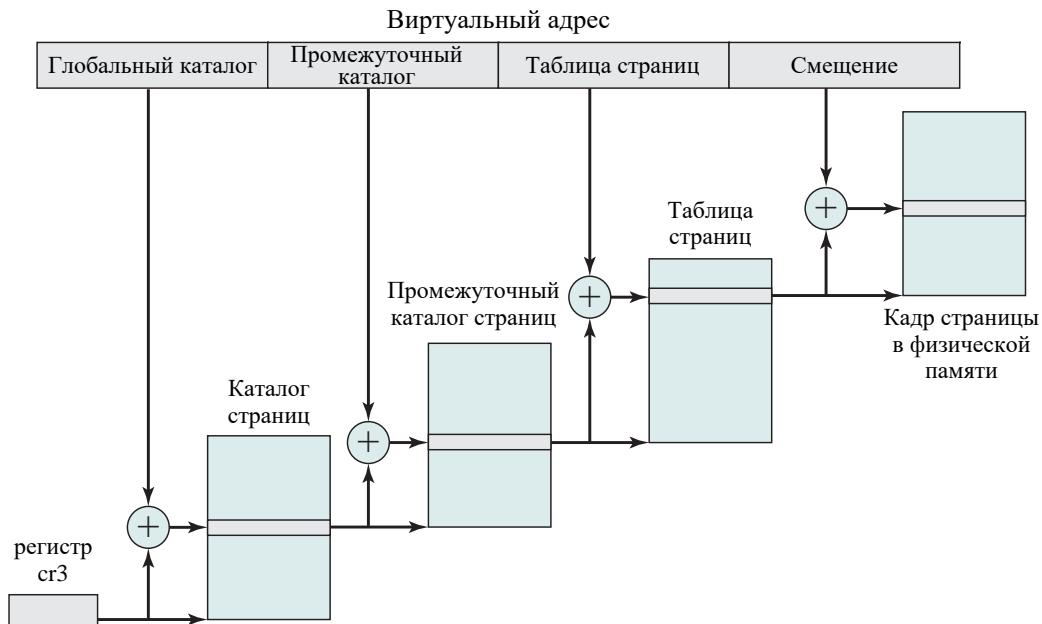
## Виртуальная память Linux

### Адресация виртуальной памяти

Linux использует трехуровневую структуру таблицы страниц, состоящую из следующих типов таблиц (каждая отдельная таблица имеет размер, равный одной странице).

- **Каталог страниц.** Активный процесс имеет единый каталог страниц, размер которого равен одной странице. Каждая запись в каталоге страниц указывает на одну страницу промежуточного каталога страниц. Каталог страниц активного процесса должен находиться в активной памяти.
- **Промежуточный каталог страниц.** Промежуточный каталог страниц может охватывать несколько страниц. Каждая запись промежуточного каталога указывает на одну страницу таблицы страниц.
- **Таблица страниц.** Таблица страниц также может охватывать несколько страниц. Каждая запись таблицы страниц указывает на одну виртуальную страницу процесса.

Для использования трехуровневой структуры таблицы страниц виртуальный адрес в Linux рассматривается как состоящий из четырех частей (рис. 8.23). Левое (наиболее значащее) поле используется в качестве индекса в каталоге страниц; следующее поле служит в качестве индекса в промежуточном каталоге страниц. Третье поле представляет собой индекс таблицы страниц, а четвертое — смещение в пределах выбранной страницы памяти.



**Рис. 8.23.** Трансляция адреса в схеме виртуальной памяти Linux

Структура таблицы страниц Linux платформонезависима и разработана для работы с 64-разрядным процессором Alpha, который обеспечивает аппаратную поддержку трехуровневой страничной организации. При использовании 64-разрядных адресов использование только двух уровней может привести к тому, что таблицы и каталоги страниц будут очень большими. 32-разрядная архитектура x86 обладает только двухуровневым механизмом страничной организации, и программное обеспечение Linux использует двухуровневую схему путем определения размера промежуточного каталога, равного одной странице. Обратите внимание, что все ссылки на дополнительный уровень косвенности устраняются оптимизатором во время компиляции, а не во время выполнения. Таким образом, при использовании обобщенного трехуровневого дизайна на платформах, которые аппаратно поддерживают только два уровня, снижение производительности не наблюдается.

## Распределение страниц

Чтобы повысить эффективность чтения страниц из основной памяти и записи страниц в нее, Linux определяет механизм для работы со смежными блоками страниц, отображаемых на смежные блоки кадров страниц. С этой целью Linux использует систему двойников. Ядро поддерживает список групп смежных кадров фиксированного размера; группа может состоять из 1, 2, 4, 8, 16 или 32 кадров страниц. При выделении и освобождении страниц в основной памяти доступные группы разделяются и объединяются с использованием алгоритма двойников.

## Алгоритм замещения страниц

Алгоритм замещения страниц в Linux до версии 2.6.28 был основан на часовом алгоритме, описанном в разделе 8.2 (см. рис. 8.15). В случае использования простого часового алгоритма с каждой страницей основной памяти связаны биты использования и модификации. В схеме, применяемой в Linux, бит использования заменен 8-битовой переменной возраста, значение которой увеличивается при каждом обращении к странице. В фоновом режиме Linux периодически сканирует страницы основной памяти и уменьшает значения их переменных возраста. Страницы с нулевым значением представляют собой “старые” страницы, к которым некоторое время не было обращений и которые являются наиболее подходящими кандидатами для замещения. Чем больше значение возраста страницы, тем чаще она использовалась в последнее время и тем менее она подходит для замещения. Таким образом, алгоритм, используемый в Linux, представляет собой разновидность стратегии замещения наименее часто используемых страниц.

Начиная с Linux версии 2.6.28 алгоритм замены страницы, описанный в предыдущем абзаце, был пересмотрен, и в ядро был внесен новый алгоритм, именуемый алгоритмом разделения последнего использованного (split LRU). Одной из проблем старого алгоритма является то, что периодическое сканирование пула страниц с ростом памяти потребляет все большее и большее количество времени процессора.

Новый алгоритм использует два флага, добавленных к каждой записи таблицы страниц: PG\_active и PG\_referenced. Вся физическая память делится на различные “зоны” Linux, основанные на их адресах. В каждой зоне имеются два связанных списка, которые используются диспетчером памяти, а именно — списки активных и неактивных страниц. Демон ядра kswapd периодически запускается в фоновом режиме для выполнения утилизации страниц в каждой зоне. Этот демон сканирует записи таблицы страниц, на которые отображены системные кадры страниц. Для всех записей таблицы страниц, помеченных как доступные, установлен бит PG\_referenced. Этот бит

устанавливается процессором при первом обращении к странице. На каждой итерации `kswapd` он проверяет, установлен ли бит доступа к странице в записи таблицы страниц. Каждый раз при чтении бита доступности страницы `kswapd` очищает этот бит. Мы можем собрать все шаги управления страницами памяти воедино на рис. 8.24.

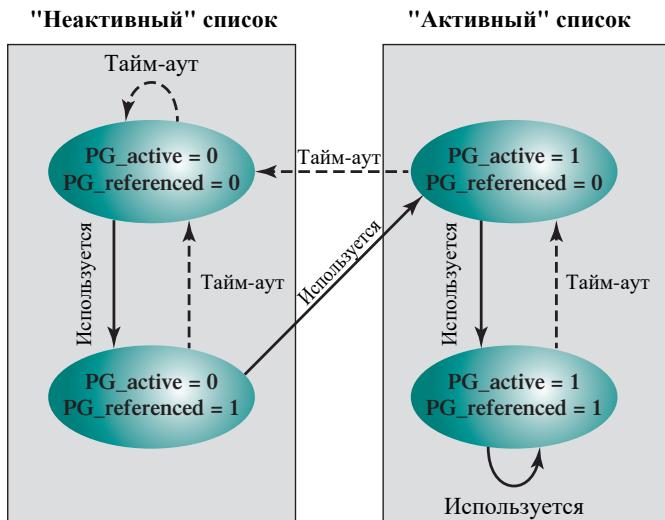


Рис. 8.24. Утилизация страниц в Linux

1. При первом обращении к странице в неактивном списке устанавливается флаг `PG_referenced`.
2. При следующем обращении к странице она перемещается в активный список. Таким образом, для объявления страницы активной требуется два обращения к ней. Точнее говоря, чтобы страница стала активной, к ней должны быть выполнены два обращения в различных сканированиях.
3. Если второе обращение не произошло достаточно быстро, бит `PG_referenced` сбрасывается.
4. Аналогично для перемещения активных страниц в список неактивных требуется два тайм-аута.

Страницы в списке неактивных доступны для замещения страниц с использованием одной из разновидностей алгоритма последнего использовавшегося.

## Распределение памяти ядра

Возможности работы с памятью ядра Linux управляют физическими кадрами страниц основной памяти. Возможные владельцы кадра включают пользовательские процессы (т.е. кадр является частью виртуальной памяти процесса, который в настоящий момент располагается в реальной памяти), динамически выделенную память для данных ядра, статический код ядра и кеш страниц<sup>7</sup>.

<sup>7</sup> Кеш страниц обладает свойствами, схожими со свойствами буфера диска, описанного в данной главе, а также дискового кеша, который будет рассмотрен в главе 11. “Управление вводом-выводом и планирование дисковых операций”. Мы отложим рассмотрение кеша страниц Linux до упомянутой главы.

Фундаментом распределения памяти ядра в Linux является механизм распределения страниц, используемый для управления пользовательской виртуальной памятью. Здесь, как и в схеме виртуальной памяти, используется алгоритм двойников, так что память для нужд ядра может выделяться и освобождаться на уровне страниц. Поскольку минимальное количество памяти, которое может быть выделено таким образом, составляет одну страницу, такое распределение неэффективно в связи с частыми запросами на выделение небольших участков памяти разного размера с малым временем жизни. Для повышения эффективности Linux использует схему, известную как **кусочное распределение** (slab allocation) [25] в пределах выделенной страницы. На машинах x86 размер страницы составляет 4 Кбайт, а участки памяти, выделяемые в пределах страницы, могут иметь размеры 32, 64, 128, 252, 508, 2040 и 4080 байт.

Такой SLAB-аллокатор является относительно сложным и детально здесь не рассматривается; его полное описание можно найти в [261]. По сути, Linux поддерживает множество связанных списков, по одному для участков каждого размера. Участки могут быть разделены на меньшие или объединены, подобно разделению и слиянию блоков в алгоритме двойников, и перемещаться из одного списка в другой соответственно изменению их размеров.

Наиболее часто в Linux используется именно SLAB, но есть и иные аллокаторы для распределения небольших фрагментов памяти.

1. SLAB. Предназначен для работы с кешем, минимизируя, насколько возможно, количество промахов кеша.
2. SLUB (SLAB без очереди). Максимально простой с минимальным количеством команд [53].
3. SLOB (simple list of blocks — простой список блоков). Максимально компактный; предназначен для систем с ограничениями памяти [160].

## 8.5. УПРАВЛЕНИЕ ПАМЯТЬЮ В WINDOWS

Система управления виртуальной памятью Windows контролирует распределение памяти и работу страничной организации. Диспетчер памяти сконструирован для работы на множестве платформ и использует страницы размером от 4 до 64 Кбайт. На платформах Intel и AMD64 размер страницы составляет 4 Кбайт, а в Intel Itanium — 8 Кбайт.

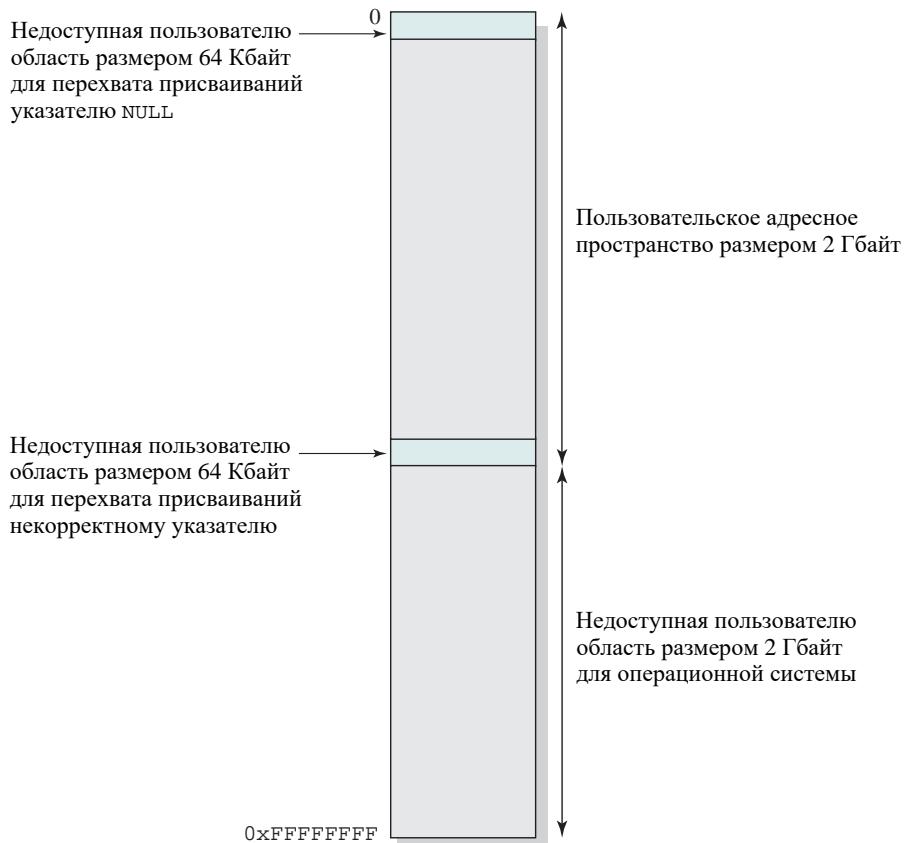
### Карта виртуальных адресов Windows

На 32-разрядных платформах каждый пользовательский процесс Windows получает отдельное 32-битовое адресное пространство, предоставляемое процессу до 4 Гбайт памяти. По умолчанию половина этой памяти зарезервирована для операционной системы, так что каждому пользователю на самом деле доступны 2 Гбайт виртуального адресного пространства, и все процессы совместно используют одни и те же 2 Гбайт системного пространства при работе в режиме ядра. Приложения с интенсивным использованием большого количества памяти (как клиенты, так и серверы) могут работать более эффективно с помощью 64-разрядной версии Windows. Помимо нетбуков, большинство современных персональных компьютеров используют архитектуру процессора AMD64, который способен работать либо как 32-разрядная, либо как 64-разрядная система.

На рис. 8.25 показано виртуальное адресное пространство, видимое обычному 32-разрядному пользовательскому процессу. Оно состоит из четырех областей.

- От 0x00000000 до 0x0000FFFF. Предназначено для помощи программисту в перехвате присвоений указателю NULL.
- От 0x00010000 до 0x7FFEFFFF. Адресное пространство, доступное пользователю. Это пространство разделяется на страницы, которые могут быть загружены в основную память.
- От 0xFFFF0000 до 0xFFFFFFFF. Защитная страница, недоступная пользователю. Эта страница облегчает операционной системе проверку выхода указателя за границы блока памяти.
- От 0x80000000 до 0xFFFFFFFF. Системное адресное пространство. Этот 2-гигабайтный участок памяти предназначен для исполняющей системы Windows, ядра, HAL и драйверов устройств.

На 64-разрядных платформах в Windows доступно пользовательское адресное пространство размером 8 Тбайт.



**Рис. 8.25.** 32-битное виртуальное адресное пространство Windows по умолчанию

## Страницчная организация Windows

При создании процесса ему полностью доступно пользовательское пространство размером почти 2 Гбайт (или 8 Тбайт в 64-разрядной Windows). Это пространство разделяется на страницы фиксированного размера, но операционная система управляет адресами в непрерывных областях, выравненных по границе 64 Кбайта. Такая область может находиться в одном из трех состояний.

1. **Доступна.** Адреса в настоящее время не используются процессом.
2. **Зарезервирована.** Адреса, которые диспетчер виртуальной памяти предназначает процессу, так что они не могут быть выделены для иного применения (например, сохранение непрерывного пространства для роста стека).
3. **Размещена.** Адреса, которые диспетчер виртуальной памяти инициализировал для использования процессом для доступа к страницам виртуальной памяти. Эти страницы могут располагаться либо на диске, либо в физической памяти. Находясь на диске, они могут либо быть в файлах (отображенные страницы), либо занимать место в файле подкачки (т.е. в дисковом файле, в который страницы записываются при их удалении из основной памяти).

Различие между зарезервированной и размещенной памятью 1) позволяет минимизировать дисковое пространство, предназначенное определенному процессу, тем самым сохранив это пространство для других процессов и делая файл подкачки меньшим по размеру, и 2) разрешает потоку или процессу резервировать адреса без того, чтобы делать их доступными программе за счет квот их ресурсов.

Схема управления резидентным множеством, используемая в Windows, — переменное распределение с локальной областью видимости (см. табл. 8.5). При первой активации процесса ему в качестве рабочего множества передается некоторое количество кадров основной памяти. Когда страницы, необходимые процессу, загружаются в физическую память, диспетчер памяти использует эти структуры данных для отслеживания страниц, назначенных процессу. Рабочие множества активных процессов настраиваются во время работы с использованием следующих общих соглашений.

- При большом размере основной памяти диспетчер виртуальной памяти позволяет резидентным множествам активных процессов расти. Для этого при генерации прерывания из-за отсутствия страницы новая страница загружается в память, но старая при этом не выгружается.
- При малом размере основной памяти диспетчер виртуальной памяти возвращает память системе, удаляя давно не использовавшиеся страницы из рабочих множеств активных процессов, снижая тем самым их размеры.
- Даже когда память в изобилии, Windows следит за крупными процессами, у которых быстро растет количество используемой памяти. Система начинает удалять из процесса страницы, которые давно не использовались. Эта стратегия снижает время отклика системы, потому что новая программа не приведет к внезапной нехватке памяти и ожиданию пользователями, пока система будет пытаться уменьшить резидентные множества уже выполняемых процессов.

## Свопинг в Windows

Одновременно с интерфейсом Metro появилась и новая система виртуальной памяти для обработки прерываний запросов от приложений Windows Store. К знакомому нам файлу Windows pagefile.sys присоединился его коллега swapfile.sys, предоставляющий доступ ко временному хранилищу памяти на жестком диске. Подкачка хранит элементы, к которым не было обращений в течение длительного времени, тогда как свопинг имеет дело с элементами, которые были недавно изъяты из памяти. К элементам в файле подкачки длительное время может не быть обращений, в то время как к элементам в файле свопинга обращение может произойти гораздо раньше. Файл swapfile.sys используют только приложения Microsoft Store, и из-за их относительно небольшого размера фиксированный размер составляет лишь 256 Мбайт. Файл же pagefile.sys имеет размер, который примерно в один-два раза больше размера физической памяти в системе. swapfile.sys работает путем сохранения всего процесса из системной памяти в файл свопинга. Таким образом, происходит немедленное освобождение памяти для использования другими приложениями. Файлы подкачки, напротив, работают путем перемещения “страниц” программы из системной памяти в файл. Размеры этих страниц — 4 Кбайт. При этом вся программа полностью в файл подкачки не сбрасывается.

## 8.6. УПРАВЛЕНИЕ ПАМЯТЬЮ В ANDROID

Android включает ряд расширений обычной системы управления памятью ядра Linux, включая следующие.

- **ASHMem.** Это расширение обеспечивает анонимную совместно используемую память, которая абстрагирует память в виде файловых дескрипторов. Файловый дескриптор может передаваться другому процессу для совместного использования памяти.
- **ION.** ION представляет собой диспетчер пула памяти, позволяющий также своим клиентам совместно использовать буферы. ION управляет одним или несколькими пулами памяти, некоторые из которых выделяются во время загрузки для борьбы с фрагментацией или для обслуживания особых потребностей оборудования. GPU, контроллеры дисплеев и камеры являются одними из аппаратных блоков, которые могут иметь особые требования к памяти. ION представляет свои пулы памяти в виде куч. Каждый тип устройства Android может работать со своим набором куч ION в соответствии с требованиями устройства к памяти.
- **Low Memory Killer.** Большинство мобильных устройств не имеют возможностей свопинга (по соображениям времени жизни флеш-памяти). Когда основная память исчерпана, приложение или приложения, использующие большую часть памяти, должны либо вернуть используемую память, либо быть остановлены. Эта функциональная возможность позволяет системе уведомлять приложение или приложения, которые должны освободить память. Если приложение не освобождает память, его выполнение прекращается.

## 8.7. РЕЗЮМЕ

Для эффективного использования процессора и систем ввода-вывода желательно поддерживать в основной памяти как можно больше процессов одновременно. Кроме того, желательно освободить программиста от ограничений, накладываемых на размер создаваемых им программ.

Решение обеих задач состоит в использовании виртуальной памяти. При использовании виртуальной памяти все адреса являются логическими, транслируемыми в процессе работы в физические. Это позволяет процессу располагаться в произвольном месте основной памяти, а также изменять свое местоположение в памяти в процессе работы. Кроме того, виртуальная память позволяет разделить процесс на несколько частей, которые не обязательно должны располагаться в смежных блоках основной памяти, более того — не все они должны находиться в основной памяти во время работы процесса.

Виртуальная память опирается на два основных подхода — страничную организацию и сегментацию. При страничной организации каждый процесс разделяется на относительно малые страницы фиксированного размера. Сегментация же позволяет использовать части различного размера. В одной схеме управления памятью могут совместно использоваться и сегментация, и страничная организация.

Схема управления виртуальной памятью требует как программной, так и аппаратной поддержки. Аппаратная поддержка, обеспечиваемая процессором, включает динамическое преобразование виртуальных адресов в физические и генерацию прерывания при отсутствии адресуемой страницы или сегмента в основной памяти. Это прерывание обрабатывается программным обеспечением управления памятью.

При разработке системы управления памятью следует решить множество связанных с ней вопросов, включающих следующие.

- **Стратегия выборки.** Страницы могут загружаться в основную память как по требованию процесса, так и с использованием стратегии предварительной выборки, при которой происходит загрузка страниц кластерами.
- **Стратегия размещения.** В случае выбора системы с использованием только сегментации все вновь загружаемые сегменты должны быть размещены в доступном пространстве в памяти.
- **Стратегия замещения.** При заполнении памяти следует принять решение о том, какая страница (или страницы) будет замещена загружаемыми в память.
- **Управление резидентным множеством.** Операционная система должна решать, какой именно объем памяти должен быть отведен тому или иному процессу при его загрузке в память. Этот объем может быть выделен статически, в момент создания процесса, либо изменяться динамически в процессе работы.
- **Стратегия очистки.** Измененные страницы процесса должны быть записаны при их замещении; однако возможно использование стратегии предварительной очистки, при которой за одну операцию производится запись кластера страниц.
- **Управление загрузкой.** Управление загрузкой заключается в определении количества процессов, которые должны быть резидентны в основной памяти в данный момент.

## 8.8. КЛЮЧЕВЫЕ ТЕРМИНЫ, КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАЧИ

### Ключевые термины

Ассоциативное отображение	Прерывание из-за отсутствия страницы	Страницчная организация
Буфер быстрой переадресации (TLB)	Рабочее множество	Стратегия выборки
Виртуальная память	Реальная память	Стратегия замещения страниц
Внешняя фрагментация	Резидентное множество	Стратегия размещения страниц
Внутренняя фрагментация	Сегмент	Таблица сегментов
Выборка по требованию	Сегментация	Таблица страниц
Кадр	Снижение пропускной способности	Управление резидентным множеством
Кусочное распределение	Страница	Хеширование
Локальность		Хеш-таблица
Предварительная выборка		

### Контрольные вопросы

- 8.1. В чем состоит различие между простой страницочной организацией и страницочной организацией виртуальной памяти?
- 8.2. Поясните эффект снижения пропускной способности системы.
- 8.3. Почему принцип локальности так важен для использования виртуальной памяти?
- 8.4. Какие элементы обычно содержатся в записи таблицы страниц? Вкратце опишите каждый элемент.
- 8.5. В чем заключается цель буфера быстрой переадресации (TLB)?
- 8.6. Вкратце опишите различные стратегии выборки страниц.
- 8.7. В чем заключается различие между управлением резидентным множеством и стратегией замещения страниц?
- 8.8. Как соотносятся между собой алгоритм замещения “первым вошел — первым вышел” и часовой?
- 8.9. В чем заключается буферизация страниц?
- 8.10. Почему невозможно объединить стратегию глобального замещения со стратегией фиксированного размещения?
- 8.11. В чем состоит разница между резидентным и рабочим множествами?
- 8.12. В чем состоит разница между очисткой по требованию и предварительной очисткой?

## Задачи

- 8.1.** Предположим, что таблица страниц текущего процесса выглядит так, как показано ниже. Все числа в таблице — десятичные, вся нумерация начинается с нуля, а все адреса представляют собой адреса отдельных байтов памяти. Размер страницы равен 1024 байтам.

Номер виртуальной страницы	Бит присутствия в памяти	Бит обращений	Бит модификации	Номер кадра
0	1	1	0	4
1	1	1	1	7
2	0	0	0	—
3	1	0	0	2
4	0	0	0	—
5	1	0	1	0

- Опишите, как именно виртуальный адрес транслируется в физический адрес основной памяти.
- Какой физический адрес (если таковой имеется) соответствует каждому из приведенных виртуальных адресов? (Вы не должны пытаться обработать прерывание из-за отсутствия страницы).
  - 1052
  - 2221
  - 5499

- 8.2.** Рассмотрим следующую программу.

```
#define Size 64
int A[Size][Size], B[Size][Size], C[Size][Size];
int register i, j;
for (j = 0; j < Size; j++)
    for (i = 0; i < Size; i++)
        C[i][j] = A[i][j] + B[i][j];
```

Предположим, что данная программа выполняется в системе с выборкой по требованию и размером страницы 1 Кбайт. Размер целого числа — 4 байта. Ясно, что каждому массиву требуется пространство из 16 страниц. В качестве примера  $A[0][0] - A[0][63]$ ,  $A[1][0] - A[1][63]$ ,  $A[2][0] - A[2][63]$  и  $A[3][0] - A[3][63]$  будут храниться в первой странице. Аналогичная схема хранения может быть записана для оставшейся части массива  $A$  и для массивов  $B$  и  $C$ . Предположим, что система выделила 4-страничное рабочее множество этому процессу. Одна из страниц используется программой, а три могут использоваться для хранения данных.

Кроме того, два индексных регистра хранят  $i$  и  $j$  (так что для этих переменных не требуются обращения к памяти).

- Рассмотрите, насколько часто будут происходить прерывания отсутствия страниц (в терминах количества выполнений присваиваний  $C[i][j] = A[i][j] + B[i][j]$ ).
- Можете ли вы изменить программу для минимизации частоты прерываний отсутствия страниц?
- Какой будет частота прерываний отсутствия страниц после ваших изменений?

- 8.3.**
- Сколько памяти требуется для пользовательской таблицы страниц на рис. 8.3?
  - Предположим, что вы хотите реализовать хешированную инвертированную таблицу страниц для той же схемы адресации, которая показана на рис. 8.3, используя хеш-функцию, отображающую 20-битный номер страницы в 6-битное хеш-значение. Записи в таблице содержат номер страницы, номер кадра и указатель цепочки. Если в таблице страницы выделена память для менее чем трех дополнительных записей на одно хеш-значение, то сколько памяти требуется для такой инвертированной таблицы страниц?

**8.4. Рассмотрим последовательность обращений к страницам**

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2.

Изобразите диаграмму, подобную показанной на рис. 8.14 и демонстрирующую распределение кадров для стратегий.

- FIFO (“первым вошел — первым вышел”).
- LRU (последний использовавшийся).
- Часовой.
- Оптимальный (в предположении, что последовательность обращений продолжается как 1, 2, 0, 1, 7, 0, 1).
- Перечислите общее количество ошибок страниц и частоту промахов для каждой стратегии. Подсчитайте количество ошибок страницы, произошедших после того, как все кадры были инициализированы.

**8.5. Процесс обращается к страницам A, B, C, D и E в следующем порядке:**

A B C D A B E A B C D E

Примените алгоритм замещения “первым вошел — первым вышел” и определите количество пересылок страниц в процессе выполнения указанных обращений, если работа процесса выполняется с тремя изначально пустыми кадрами основной памяти. Решите ту же задачу для четырех кадров.

- 8.6. Процесс содержит восемь виртуальных страниц на диске, и ему выделено четыре фиксированных кадра в основной памяти. Далее выполняются обращения к следующим страницам:**

1, 0, 2, 2, 1, 7, 6, 7, 0, 1, 2, 0, 3, 0, 4, 5, 1, 5, 2, 4, 5, 6, 7, 6, 7, 2, 4, 2, 7, 3, 3, 2, 3.

а. Укажите последовательность размещения страниц в кадрах при использовании алгоритма замещения наиболее долго не использовавшейся страницы. Вычислите результативность обращения к основной памяти (считаем, что изначально все кадры пусты).

б. Выполните то же задание для алгоритма “первым вошел — первым вышел”.

в. Сравните результативности обращения к основной памяти, вычисленные в первых двух заданиях, и прокомментируйте эффективность использования указанных алгоритмов применительно к данной последовательности обращений.

**8.7.** Пользовательские таблицы страниц в VAX располагаются в виртуальных адресах системного пространства. Каковы преимущества такого размещения по сравнению с размещением таблиц в основной памяти? Каковы недостатки этого размещения?

**8.8.** Предположим, что фрагмент кода

```
for (i = 1; i <= n; i++) a[i] = b[i] + c[i]
```

выполняется в памяти с размером страницы, равным 1000 слов. Пусть  $n = 1000$ . Напишите гипотетическую машинную программу, реализующую приведенный фрагмент (считая, что машина имеет полный набор команд для пересылки информации между регистрами и может использовать индексные регистры). Затем покажите последовательность обращения к страницам в процессе выполнения кода.

**8.9.** Архитектура IBM System/370 использует двухуровневую структуру памяти, в которой эти уровни названы сегментами и страницами (несмотря на то, что такая сегментация не обладает многими возможностями, рассмотренными в этой главе). Размер страницы в данной архитектуре может быть равен 2 или 4 Кбайт, а размер сегмента, являющийся в данной архитектуре фиксированным, — либо 64 Кбайт, либо 1 Мбайт. В архитектурах 370/XA и 370/ESA размер страницы равен 4 Кбайт, а размер сегмента — 1 Мбайт. Какие преимущества сегментации утрачены данной архитектурой? Какие выгоды приносит сегментация System/370?

**8.10.** Предположим, что размер страницы — 4 Кбайт и что запись таблицы страниц занимает 4 байт. Сколько уровней таблиц страниц потребуется для отображения 64-битового адресного пространства, если таблица верхнего уровня занимает одну страницу?

**8.11.** Рассмотрим систему с отображением памяти на уровне страниц и использованием одноуровневой таблицы страниц. Предположим, что нужная нам таблица страниц уже находится в основной памяти.

а. Если обращение к памяти занимает 200 нс, чему будет равно время доступа к страничной памяти?

б. Добавим блок управления памятью, который создает накладные расходы в 20 нс как при успешном, так и при неудачном поиске. Предполагая, что результативность поиска в TLB составляет 85%, вычислите эффективное время доступа к памяти.

в. Поясните, как результативность поиска в TLB влияет на эффективное время доступа к памяти.

- 8.12.** Рассмотрим последовательность обращения к страницам процесса с изначально пустым рабочим множеством из  $M$  кадров. Последовательность обращений к страницам длиной  $P$  содержит  $N$  различных номеров страниц. Для произвольного алгоритма замещения страниц определите
- нижнюю границу количества прерываний из-за отсутствия страницы;
  - верхнюю границу количества прерываний из-за отсутствия страницы.
- 8.13.** При обсуждении алгоритма замещения страниц один из авторов провел аналогию сдвигающейся по кругу снегоуборочной машиной. Снег равномерно засыпает кольцевую дорогу, по которой с постоянной скоростью движется снегоочиститель. Снег, отброшенный снегоочистителем, исчезает из рассматриваемой системы.
- Какому из рассмотренных в разделе 8.2 алгоритмов соответствует эта аналогия?
  - Какое предположение о поведении рассматриваемого алгоритма замещения используется в данной аналогии?
- 8.14.** В архитектуре S/370 ключ управления памятью представляет собой управляющее поле, связанное с каждым кадром в основной памяти, с размером кадра, равным размеру страницы. Два бита этого ключа относятся к замещению страниц и представляют собой биты обращения и изменения. Бит обращения устанавливается равным 1, когда происходит чтение или запись ячейки памяти по адресу, находящемуся внутри кадра, и равным 0 — при первой загрузке новой страницы в кадр. Бит изменения становится равным 1 при выполнении операции записи в любую ячейку памяти в пределах кадра. Предложите способ определения страницы, которая не использовалась больше других, с помощью только бита обращения.
- 8.15.** Рассмотрим последовательность обращения к страницам (каждый элемент последовательности представляет номер страницы):

1 2 3 4 5 2 1 3 3 2 3 4 5 4 5 1 1 3 2 5

Определим средний размер рабочего множества после  $k$ -го обращения как

$$s_k(\Delta) = \frac{1}{k} \sum_{i=1}^k |W(t, \Delta)|$$

а вероятность отсутствия страницы как

$$m_k(\Delta) = \frac{1}{k} \sum_{i=1}^k |F(t, \Delta)|$$

Здесь  $F(t, \Delta) = 1$ , если в момент виртуального времени  $t$  наблюдается отсутствие страницы, и 0 — в противном случае.

- Изобразите диаграмму, подобную представленной на рис. 8.17, для указанной последовательности обращений для значений  $\Delta = 1, 2, 3, 4, 5, 6$ .
- Изобразите график  $s_{20}(\Delta)$  как функцию от  $\Delta$ .
- Изобразите график  $m_{20}(\Delta)$  как функцию от  $\Delta$ .

- 8.16.** Ключевым параметром производительности стратегии управления резидентным множеством VSWI является значение  $Q$ . Опыт показывает, что при фиксированном значении  $Q$  наблюдаются значительные различия в частоте генерации прерываний на разных стадиях выполнения процесса. Кроме того, если для разных процессов используется одно и то же значение  $Q$ , их частоты генерации прерываний существенно различаются. Эти наблюдения недвусмысленно указывают на то, что динамическое изменение значения  $Q$  в процессе жизни процесса может улучшить работу алгоритма. Предложите простейший механизм для реализации этой идеи.
- 8.17.** Предположим, что задание разделено на четыре сегмента одинакового размера и что для каждого сегмента система строит таблицу дескрипторов страниц с восемью записями. Таким образом, описанная система представляет собой комбинацию сегментации и страницной организации. Предположим также, что размер страницы равен 2 Кбайт.
- Чему равен максимальный объем каждого сегмента?
  - Каково максимальное логическое адресное пространство одного задания?
  - Предположим, что рассматриваемое задание обратилось к ячейке памяти с физическим адресом 00021ABC. Каков формат генерируемого для этого логического адреса? Каково максимально возможное физическое адресное пространство в этой системе?
- 8.18.** Рассмотрим страницочное логическое адресное пространство, состоящее из 32 страниц по 2 Кбайт, отображенное на 1-мегабайтовое физическое пространство.
- Каков формат логического адреса процессора?
  - Чему равны длина и ширина таблицы страниц (без учета битов прав доступа)?
  - Как повлияет на размер таблицы страниц уменьшение физической памяти в два раза?
- 8.19.** Ядро UNIX при необходимости динамически увеличивает стек процесса в виртуальной памяти, но никогда не уменьшает его. Рассмотрим вызов подпрограммы на языке программирования C, в которой имеется локальный массив размером 10 Кбайт, размещаемый в стеке. Ядро увеличит сегмент стека, для того чтобы этот массив мог быть успешно размещен в стеке. При возврате из подпрограммы указатель стека перемещается, и выделенное пространство может быть освобождено ядром, но оно этого не делает. Поясните, почему в этот момент можно уменьшить размер стека и почему ядро UNIX не делает этого.

ЧАСТЬ IV

---

# ПЛАНИРОВАНИЕ



# 9

## ГЛАВА

# ОДНОПРОЦЕССОРНОЕ ПЛАНИРОВАНИЕ

В ЭТОЙ ГЛАВЕ...

### 9.1. Типы планирования процессора

- Долгосрочное планирование
- Среднесрочное планирование
- Краткосрочное планирование

### 9.2. Алгоритмы планирования

- Критерии краткосрочного планирования
- Использование приоритетов
- Альтернативные стратегии планирования
  - Первым поступил — первым обслужен
  - Круговое планирование
  - Выбор самого короткого процесса
  - Наименьшее остающееся время
  - Наивысшее отношение отклика
  - Снижение приоритета
- Сравнение производительности
- Анализ очередей
- Имитационное моделирование
- Справедливое планирование

### 9.3. Традиционное планирование UNIX

### 9.4. Резюме

### 9.5. Ключевые термины, контрольные вопросы и задачи

- Ключевые термины
- Контрольные вопросы
- Задачи

## УЧЕБНЫЕ ЦЕЛИ

- Пояснить разницу между долгосрочным, среднесрочным и краткосрочным планированием.
- Описать производительность различных стратегий планирования.
- Понимать методы планирования, используемые в традиционном UNIX.

В многозадачных системах в основной памяти одновременно содержится код нескольких процессов. В работе каждого процесса периоды использования процессора чередуются с ожиданием некоторых событий, таких как завершение выполнения операций ввода-вывода. Процессор (или процессоры) занят выполнением одного процесса, в то время как остальные находятся в состоянии ожидания.

Ключом к многозадачности является планирование. Обычно используются четыре типа планирования (табл. 9.1). Одно из них — планирование ввода-вывода — рассматривается в главе 11, “Управление вводом-выводом и планирование дисковых операций”, посвященной вопросам ввода-вывода. Планирование остальных трех типов, являющееся планированием процессора, будет рассматриваться в этой и следующей главах.

**Таблица 9.1. Типы планирования**

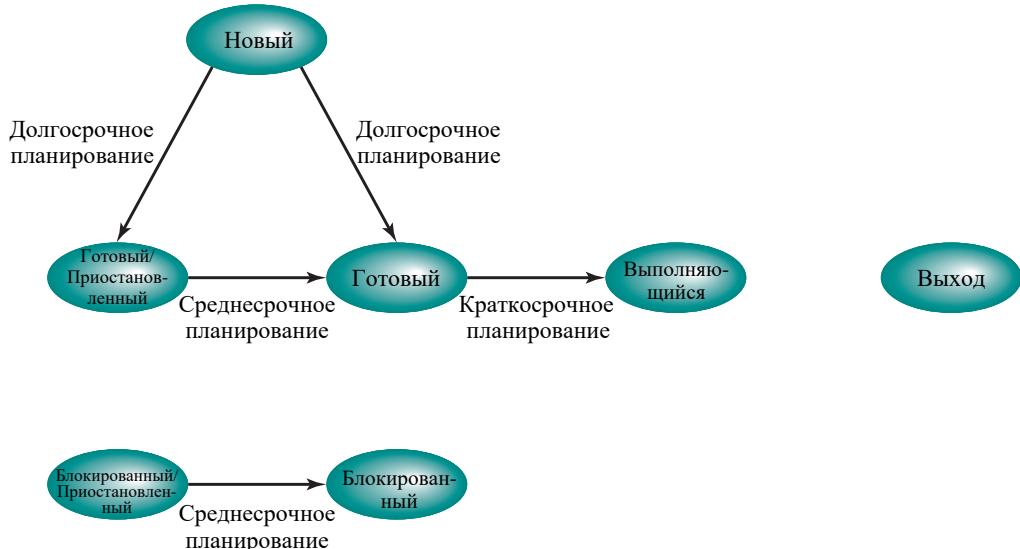
<b>Долгосрочное планирование</b>	Решение о добавлении процесса в пул выполняемых процессов
<b>Среднесрочное планирование</b>	Решение о добавлении процесса к числу процессов, полностью или частично размещенных в основной памяти
<b>Краткосрочное планирование</b>	Решение о том, какой из доступных процессов будет выполняться процессором
<b>Планирование ввода-вывода</b>	Решение о том, какой из запросов процессов на операции ввода-вывода будет обработан доступным устройством ввода-вывода

Данная глава начинается с рассмотрения трех типов планирования процессора и выявления их взаимосвязей. Вы увидите, что долгосрочное и среднесрочное планирование определяется, в первую очередь, вопросами производительности, связанными со степенью многозадачности (материал, с которым вы вкратце познакомились в главе 3, “Описание процессов и управление ими”, и который более детально был рассмотрен в главах 7, “Управление памятью”, и 8, “Виртуальная память”). В остальной части данной главы рассматриваются вопросы краткосрочного планирования в однопроцессорной системе. Поскольку изучение планирования при использовании нескольких процессоров сопряжено с дополнительными сложностями, методологически правильнее сначала рассмотреть работу одного процессора, чтобы отчетливее увидеть различия алгоритмов планирования.

Раздел 9.2 посвящен различным алгоритмам, которые могут использоваться при краткосрочном планировании.

## 9.1. ТИПЫ ПЛАНИРОВАНИЯ ПРОЦЕССОРА

Цель планирования процессора состоит в распределении во времени процессов, выполняемых процессором (или процессорами) таким образом, чтобы удовлетворять требованиям системы, таким как время отклика, пропускная способность и эффективность работы процессора. Во многих системах планирование разбивается на три отдельные функции — долгосрочного, среднесрочного и краткосрочного планирования. Их названия соответствуют временным масштабам выполнения этих функций.

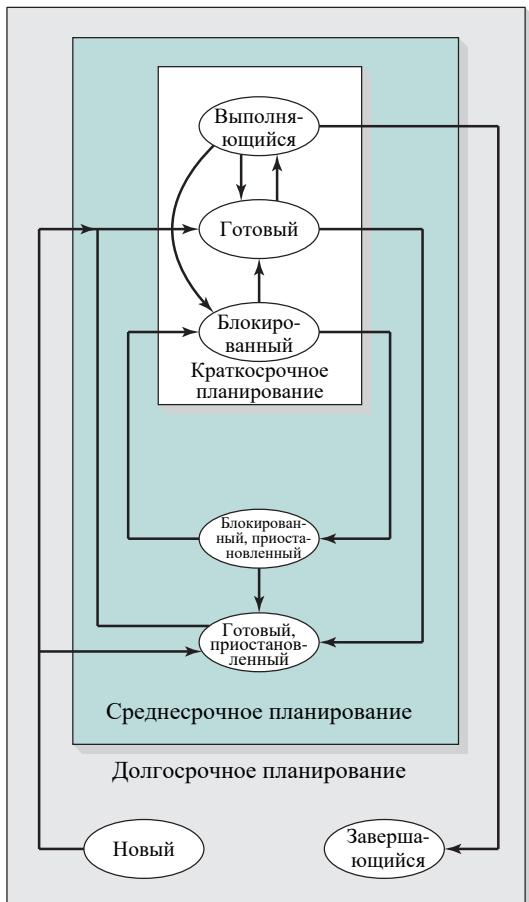


**Рис. 9.1.** Место планирования в диаграмме переходов состояний процесса

На рис. 9.1 функции планирования привязаны к диаграмме переходов состояния процесса (впервые показанной на рис. 3.9, б). Долгосрочное планирование осуществляется при создании нового процесса и представляет собой решение о добавлении нового процесса к множеству активных в настоящий момент процессов. Среднесрочное планирование является частью свопинга и представляет собой решение о добавлении процесса к множеству по крайней мере частично расположенных в основной памяти (и следовательно, доступных для выполнения) процессов. Краткосрочное планирование является решением о том, какой из готовых к выполнению процессов будет выполняться следующим. На рис. 9.2 диаграмма переходов реорганизована таким образом, чтобы показать вложенность функций планирования.

Планирование оказывает большое влияние на производительность системы, поскольку именно оно определяет, какой процесс будет выполняться, а какой — ожидать выполнения. Эта точка зрения представлена на рис. 9.3, где показаны очереди, включенные в диаграмму переходов состояний процесса.<sup>1</sup> По сути, планирование представляет собой

<sup>1</sup> Для простоты восприятия на рис. 9.3 новые процессы показаны как непосредственно становящиеся в очередь готовых к выполнению, в то время как на рис. 9.1 и 9.2 видно, что новый процесс может быть как в состоянии готовности, так и приостановленным.



**Рис. 9.2.** Уровни планирования

дует превратить в процесс (процессы). Рассмотрим вкратце эти решения.

Решение о том, когда следует создавать новый процесс, в общем определяется желаемым уровнем многозадачности. Чем больше процессов будет создано, тем меньший процент времени будет тратиться на выполнение каждого из них (поскольку в борьбе за одно и то же процессорное время конкурирует большее количество процессов). Таким образом, долгосрочный планировщик может ограничить степень многозадачности, с тем чтобы обеспечить удовлетворительный уровень обслуживания текущего множества процессов. Каждый раз при завершении задания планировщик решает, следует ли добавить в систему один или несколько новых процессов. Кроме того, долгосрочный планировщик может быть вызван в случае, когда относительное время простоя процессора превышает некоторый предопределенный порог.

Решение о том, какое из заданий должно быть добавлено в систему, может основываться на простейшем принципе “первым поступил — первым обслужен”; кроме того, для управления производительностью системы может использоваться специальный инструментарий. Используемые в последнем случае критерии могут включать приоритет заданий, ожидаемое время выполнения и требования к работе устройств ввода-выво-

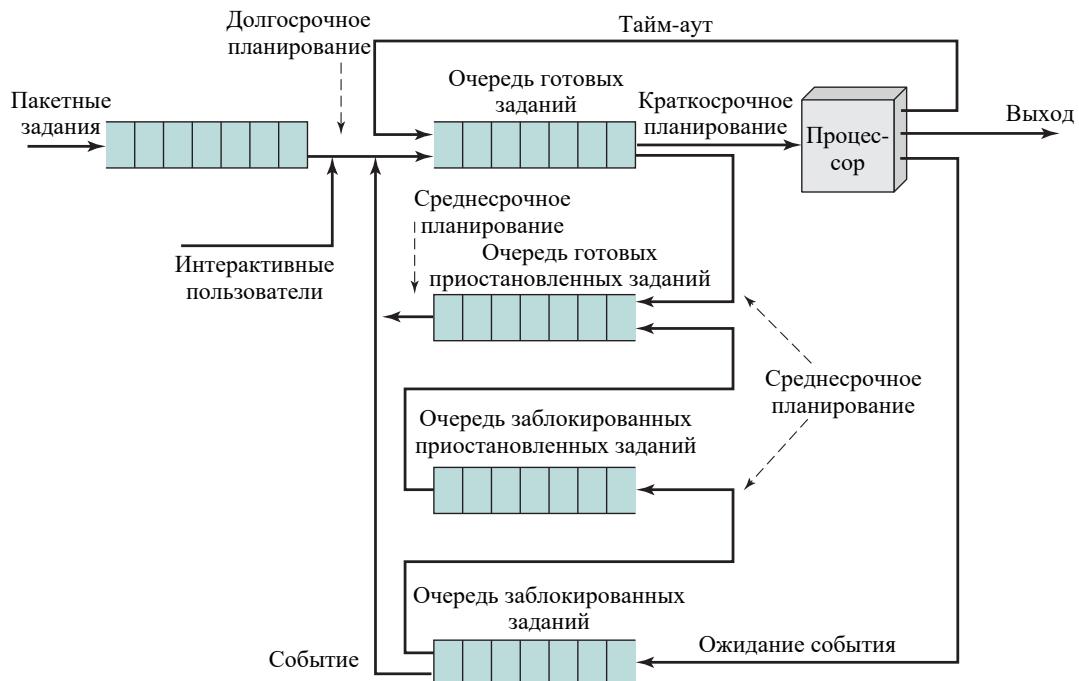
дования с целью минимизации задержек и оптимизации производительности системы.

## Долгосрочное планирование

Долгосрочное планирование определяет, какие программы допускаются к выполнению системой, и тем самым определяет степень многозадачности. Будучи допущенным к выполнению, задание (или пользовательская программа) становится процессом, который добавляется в очередь для краткосрочного планирования. В некоторых системах вновь созданный процесс добавляется к очереди среднесрочного планировщика, будучи целиком сброшенным на диск.

В пакетных системах (или в пакетной части операционной системы общего назначения) новое задание направляется на диск и хранится в очереди пакетных заданий, а долгосрочный планировщик по возможности создает процессы для заданий из очереди. В такой ситуации планировщик должен принять решение, во-первых, о том, способна ли операционная система работать с дополнительными процессами, а во-вторых, о том, какое именно задание (или задания) сле-

да. Например, если заранее доступна детальная информация о процессах, планировщик может пытаться поддерживать в системе комбинацию из процессов, ориентированных на вычисления и на ввод-вывод<sup>2</sup>. Принимаемое решение, в попытках сбалансировать использование устройств ввода-вывода, может также зависеть от того, какие именно ресурсы ввода-вывода будут запрашиваться процессом.



9.3 Диаграмма планирования с участием очередей

В случае использования интерактивных программ в системах с разделением времени запрос на запуск процесса может генерироваться действиями пользователя по подключению к системе. Пользователи не просто вносятся в очередь в ожидании, когда система обработает их запрос на подключение. Вместо этого операционная система принимает всех авторизованных пользователей до насыщения системы (пороговое значение которого определяется заранее). После достижения состояния насыщения на все запросы на вход в систему будет получено сообщение о заполненности системы и временном прекращении доступа к ней с предложением повторить операцию входа попозже.

## Среднесрочное планирование

Среднесрочное планирование является частью системы свопинга, вопросы которого рассматривались в главах 3, “Описание процессов и управление ими”, 7, “Управление памятью”, и 8, “Виртуальная память”.

<sup>2</sup> Процесс рассматривается как *связанный с процессором* (processor bound), если он в основном выполняет вычисления и только изредка работает с устройствами ввода-вывода. Процесс рассматривается как *связанный с вводом-выводом* (I/O bound), если время работы процесса затрачивается в основном на ожидание выполнения операций ввода-вывода.

Обычно решение о загрузке процесса в память принимается в зависимости от степени многозадачности; кроме того, в системе с отсутствием виртуальной памяти среднесрочное планирование также тесно связано с вопросами управления памятью. Таким образом, решение о загрузке процесса в память должно учитывать требования к памяти выгружаемого процесса.

## Краткосрочное планирование

Рассматривая частоту работы планировщика, можно сказать, что долгосрочное планирование выполняется сравнительно редко, принимая крупные решения о том, следует ли принять новый процесс, и какой именно. Среднесрочное планирование работает несколько чаще и принимает решения о свопинге. Краткосрочный же планировщик, известный также как диспетчер (*dispatcher*), работает чаще всего и на самом “мелкозернистом” уровне, определяя, какой именно процесс будет выполняться следующим.

Краткосрочный планировщик вызывается при наступлении события, которое может приостановить текущий процесс или предоставить возможность прекратить выполнение данного процесса в пользу другого. Вот некоторые примеры таких событий.

- Прерывание таймера
- Прерывания ввода-вывода
- Вызовы операционной системы
- Сигналы (например, семафоры)

## 9.2. Алгоритмы планирования

### Критерии краткосрочного планирования

Основная цель краткосрочного планирования состоит в распределении процессорного времени таким образом, чтобы оптимизировать один или несколько аспектов поведения системы. Вообще говоря, имеется множество критериев оценки различных стратегий планирования.

Наиболее распространенные критерии могут быть классифицированы двумя способами. Во-первых, мы можем разделить их на пользовательские и системные. Пользовательские критерии связаны с поведением системы по отношению кциальному пользователю или процессу. В качестве примера можно привести время отклика в интерактивной системе. Время отклика представляет собой интервал между передачей запроса и началом ответа на него. С ним пользователь сталкивается непосредственно и, самой, продолжительность этого интервала очень его интересует. Мы намерены создать стратегию планирования, обеспечивающую качественный сервис для пользователей? В таком случае для времени отклика следует установить порог, например, в 2 секунды. Тогда цель механизма планирования должна заключаться в максимизации количества пользователей, среднее время отклика для которых не превышает 2 секунд.

Системные критерии ориентированы на эффективность и полноту использования процессора. В качестве примера можно привести пропускную способность, которая представляет собой скорость завершения процессов. Это, безусловно, эффективная мера производительности системы, которая должна быть максимизирована. Однако она в

большой степени ориентирована на производительность системы, а не на обслуживание пользователя, так что и удовлетворять она будет системного администратора, а не пользователей системы.

В то время как пользовательские критерии важны почти для всех систем, системные критерии для однопользовательских систем в общем случае не так значимы. В этом случае, пожалуй, достижение высокой эффективности использования процессора или высокая производительность не так существенна, как скорость ответа системы приложению пользователя.

Еще один способ классификации критериев — на те, которые связаны с производительностью, и те, которые с производительностью непосредственно не связаны. Ориентированные на производительность критерии выражаются числовыми значениями и обычно достаточно легко измеримы — их примерами могут служить время отклика и пропускная способность. Критерии, не связанные с производительностью непосредственно, либо качественны по своей природе, либо трудно поддаются измерениям и анализу. Примером такого критерия служит предсказуемость. Желательно, чтобы предоставляемые пользователю сервисы в разное время имели одни и те же характеристики, не зависящие от других задач, выполняемых в настоящее время системой. До некоторой степени этот критерий является измеримым — путем вычисления отклонений как функции от загрузки системы. Однако провести такие измерения оказывается вовсе не просто.

В табл. 9.2 приведены ключевые критерии планирования. Все они взаимозависимы, и достичь оптимального результата по каждому из них одновременно невозможно. Например, обеспечение удовлетворительного отклика может потребовать применения алгоритма с высокой частотой переключения процессов, что повысит накладные расходы и, соответственно, снизит пропускную способность системы. Таким образом, разработка стратегии планирования представляет собой поиск компромисса среди противоречивых требований; относительный вес каждого из критериев определяется природой и назначением разрабатываемой системы.

**Таблица 9.2. Критерии планирования**

---

#### Пользовательские, связанные с производительностью

---

**Время оборота** Интервал времени между передачей процесса для выполнения и его завершением. Включает время выполнения, а также время, затраченное на ожидание ресурсов, в том числе процессора. Критерий вполне применим для пакетных заданий

**Время отклика** В интерактивных процессах это время, истекшее между подачей запроса и началом получения ответа на него. Зачастую процесс может начать вывод информации пользователю, еще не окончив полной обработки запроса, так что описанный критерий — наиболее подходящий с точки зрения пользователя. Стратегия планирования должна пытаться сократить время получения ответа при максимизации количества интерактивных пользователей, время отклика для которых не выходит за заданные пределы

**Предельный срок** При указании предельного срока завершения процесса планирование должно подчинить ему все прочие цели максимизации количества процессов, завершающихся в срок

---

---

### Пользовательские, иные

---

**Предсказуемость** Данное задание должно выполняться примерно за одно и то же количество времени и с одной и той же стоимостью, независимо от загрузки системы. Большие вариации времени выполнения или времени отклика дезориентируют пользователей. Это явление может сигнализировать о больших колебаниях загрузки или о необходимости дополнительной настройки системы для устранения нестабильности ее работы

---

### Системные, связанные с производительностью

---

**Пропускная способность** Стратегия планирования должна пытаться максимизировать количество процессов, завершающихся за единицу времени, что является мерой количества выполненной системой работы. Очевидно, что эта величина зависит от средней продолжительности процесса; однако на нее влияет и используемая стратегия планирования

**Использование процессора** Этот показатель представляет собой процент времени, в течение которого процессор оказывается занятым. Для дорогих совместно используемых систем этот критерий достаточно важен; в однопользовательских же и некоторых других системах (типа систем реального времени) — менее важен по сравнению с рядом других

---

### Системные, иные

---

**Беспристрастность** При отсутствии дополнительных указаний от пользователя или системы все процессы должны рассматриваться как равнозначные и ни один процесс не должен подвергнуться голоданию

**Использование приоритетов** Если процессам назначены приоритеты, стратегия планирования должна отдавать предпочтение процессам с более высоким приоритетом

**Баланс ресурсов** Стратегия планирования должна поддерживать занятость системных ресурсов. Предпочтение должно быть отдано процессу, который недостаточно использует важные ресурсы. Этот критерий включает использование долгосрочного и среднесрочного планирования

---

В большинстве интерактивных операционных систем, как однопользовательских, так и с разделяемым временем, критичным требованием является время отклика. В связи с важностью этого критерия и тем, что определение его величины меняется от одной программы к другой, этот вопрос вынесен в приложение Ж, “Время отклика”.

## Использование приоритетов

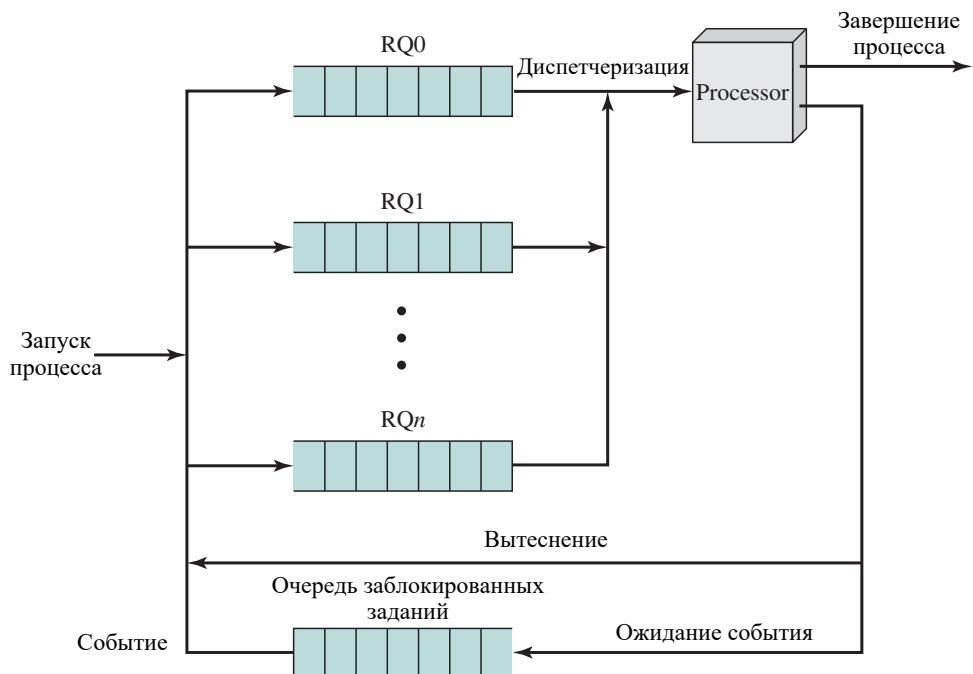
Во многих системах каждому процессу присвоен некоторый приоритет, и планировщик всегда должен среди процессов выбирать тот, приоритет которого наибольший. На рис. 9.4 показано использование приоритетов. Для большей ясности диаграмма упрощена и игнорирует существование нескольких очередей заблокированных или приостановленных процессов (ср. с рис. 3.8, а). Вместо одной очереди готовых к выполнению процессов у нас имеется их множество, упорядоченное по убыванию приоритета:  $RQ_0, RQ_1, \dots, RQ_n$ , т.е.

$$\text{Приоритет}[RQ_i] > \text{Приоритет}[RQ_j] \text{ при } i < j.^3$$

---

<sup>3</sup> В UNIX и многих других системах большие значения приоритетов соответствуют процессам с низким приоритетом; если не указано иное, мы придерживаемся именно этого соглашения. Некоторые системы, такие как Windows, используют обратное соглашение: большее значение указывает на более высокий приоритет.

При выборе процесса планировщик начинает с очереди процессов с наивысшим приоритетом (RQ0). Если в очереди имеется один или несколько процессов, процесс для работы выбирается с использованием некоторой стратегии планирования. Если очередь RQ0 пуста, рассматривается очередь RQ1 и т.д.



**Рис. 9.4.** Планирование с учетом приоритета процессов

Одна из основных проблем в такой чисто приоритетной схеме планирования состоит в том, что процессы с низким приоритетом могут оказаться в состоянии голодания. Это будет происходить при постоянном поступлении новых готовых к выполнению процессов с высоким приоритетом. Если такое поведение нежелательно, приоритет процесса может изменяться с его “возрастом” или историей выполнения (пример такой стратегии планирования будет приведен ниже).

## Альтернативные стратегии планирования

В табл. 9.3 представлена некоторая информация о различных стратегиях планирования, рассматриваемых в данном подразделе. Функция выбора определяет, какой из готовых к выполнению процессов будет выбран следующим для выполнения. Функция может быть основана на приоритете, требованиях к ресурсам или характеристиках выполнения процессов. В последнем случае имеют значение три величины:

- время, затраченное к этому моменту системой (ожидание и выполнение);
- время, затраченное к этому моменту на выполнение;
- общее время обслуживания, требующееся процессу, включая  $e$  (обычно эта величина оценивается или задается пользователем).

Таблица 9.3. ХАРАКТЕРИСТИКИ РАЗЛИЧНЫХ СТРАТЕГИЙ ПЛАНИРОВАНИЯ

Функция выбора	FSCF	Круговая	SPN	SRT	HRRN	Со снижением приоритета
Режим решения	max[w]	const	min[s]	min[s - e]	$\max\left(\frac{w+s}{s}\right)$	См. текст
Пропускная способность	Невытесняющий	Вытесняющий (по времени)	Невытесняющий	Вытесняющий (по решению)	Невытесняющий	Вытесняющий (по времени)
Время отклика	Не важна	Может быть низкой при малом кванте времени	Высокая	Высокая	Высокая	Не важна
Накладные расходы	Минимальны	Может быть большим, в особенности при больших отклонениях во времени выполнения процесса	Минимальны	Обеспечивает хорошее время отклика для коротких процессов	Обеспечивает хорошее время отклика	Обеспечивает хорошее время отклика
Влияние на процессы	Плохо оказывается на коротких процессах и процессах с интенсивным вводом-выводом	Беспристрастна	Плохо оказывается на длинных процессах	Плохо оказывается на длинных процессах	Хороший баланс	Может привести к предпочтению процессов с интенсивным вводом-выводом
Голодание	Отсутствует	Возможно	Возможно	Отсутствует	Возможно	Возможно

Например, выбор функции  $\max[w]$  определяет стратегию “первым поступил — первым обслужен” (first-come-first-served — FCFS).

**Режим принятия решения** определяет, в какие моменты времени выполняется функция выбора. Режимы принятия решения подразделяются на две основные категории.

- **Невытесняющие.** В этом случае находящийся в состоянии выполнения процесс продолжает выполнение до тех пор, а) пока он не завершится или б) пока не окажется в заблокированном состоянии ожидания завершения операции ввода-вывода или запроса некоторого системного сервиса.
- **Вытесняющие.** Выполняющийся в настоящий момент процесс может быть прерван и переведен операционной системой в состояние готовности к выполнению. Решение о вытеснении может приниматься при запуске нового процесса по прерыванию, которое переводит заблокированный процесс в состояние готовности к выполнению, или периодически — на основе прерываний таймера.

Вытесняющие стратегии приводят к повышенным накладным расходам по сравнению с невытесняющими, но при этом обеспечивают лучший уровень обслуживания всего множества процессов, поскольку предотвращают монопольное использование процессора в течение продолжительного времени одним из процессов. Кроме того, использование эффективных механизмов переключения процессов (по возможности реализованное аппаратно) и большой объем основной памяти (для хранения в ней как можно большего количества процессов) позволяют поддерживать относительно небольшую стоимость вытеснения.

При описании различных стратегий планирования в качестве примера мы будем использовать набор процессов из табл. 9.4. Будем рассматривать их как пакетные задания со временем обслуживания, равным общему времени выполнения. Эти же процессы можно рассматривать как непрерывные процессы, требующие циклического чередования работы процессора и устройств ввода-вывода. В этом последнем случае время обслуживания будет представлять собой процессорное время, требующееся в одном цикле. В любом случае в терминах модели с очередями эта величина соответствует времени обслуживания.<sup>4</sup>

**Таблица 9.4. Пример планирования процессов**

Процесс	Время запуска	Время обслуживания
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

На рис. 9.5 приведен пример выполнения одного цикла описанного в табл. 9.4 примера, а в табл. 9.5 представлены некоторые ключевые результаты. Во-первых, определено время завершения каждого процесса, так что мы можем найти **время оборота**  $T_r$  (тур-

<sup>4</sup> См. терминологию в приложении 3, “Концепции теории массового обслуживания”, и более подробное обсуждение анализа очередей в главе 20, “Обзор вероятности и стохастических процессов”.

turnaround time — TAT), которое представляет собой полное время, затраченное процессом в системе (время ожидания и время обслуживания). Более полезной величиной является нормализованное время оборота, которое определяется как отношение времени оборота ко времени обслуживания и указывает относительную задержку, испытываемую процессом. Обычно, чем больше время выполнения процесса, тем больше абсолютная величина задержки, которая может быть приемлемой. Минимальное значение этого отношения — 1,0. Возрастание значения отношения соответствует снижению уровня обслуживания.

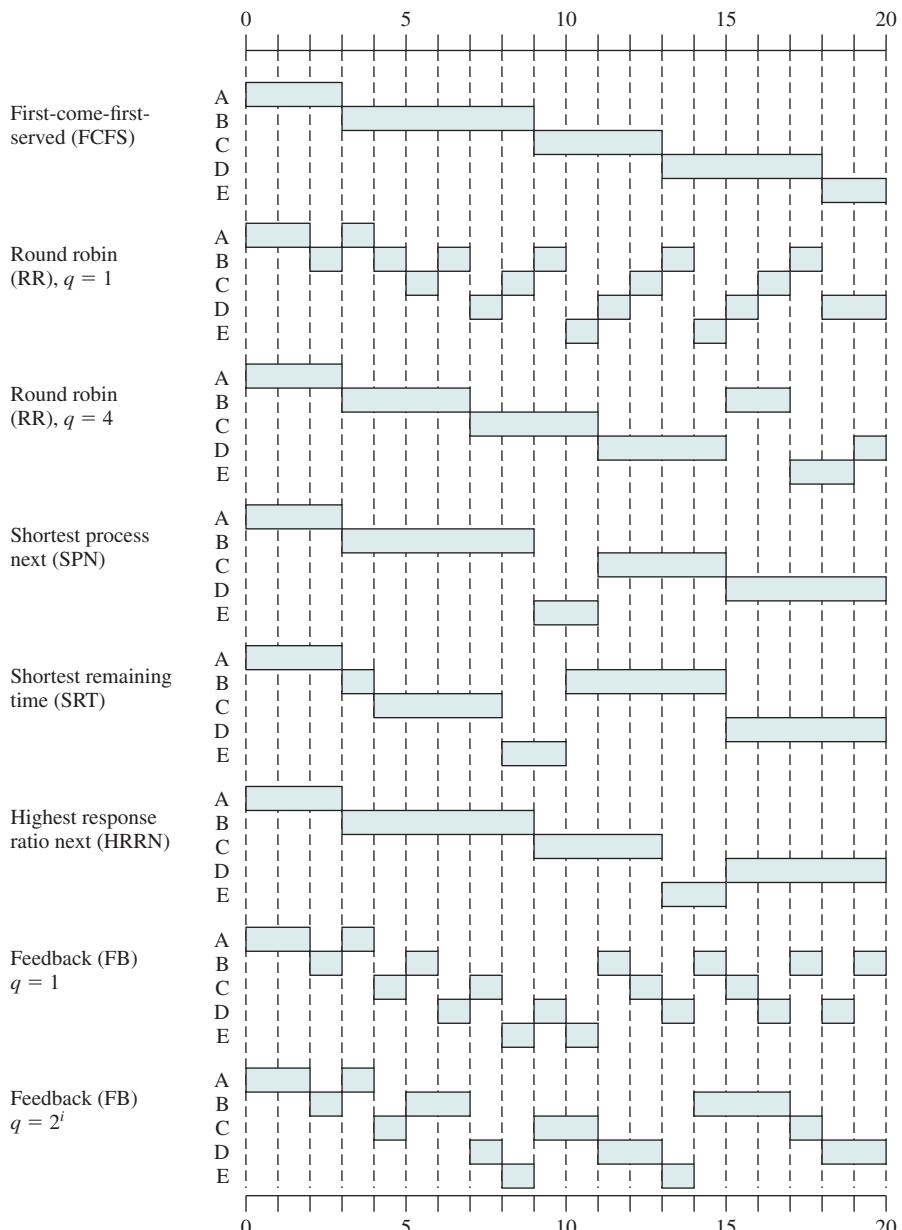


Рис. 9.5. Сравнение стратегий планирования

Таблица 9.5. СРАВНЕНИЕ СТРАТЕГИЙ ПЛАНИРОВАНИЯ

Процесс	A	B	C	D	E	Среднее
Время входа	0	2	4	6	8	
Время обслуживания $T_s$	3	6	4	5	2	
<b>FCFS</b>						
Время завершения	3	9	13	18	20	
Время оборота $T_r$	3	7	9	12	12	8,60
$T_r/T_s$	1,00	1,17	2,25	2,40	6,00	2,56
<b>RR, q = 1</b>						
Время завершения	4	18	17	20	15	
Время оборота $T_r$	4	16	13	14	7	10,80
$T_r/T_s$	1,33	2,67	3,25	2,80	3,50	2,71
<b>RR, q = 4</b>						
Время завершения	3	17	11	20	19	
Время оборота $T_r$	3	15	7	14	11	10,00
$T_r/T_s$	1,00	2,5	1,75	2,80	5,50	2,71
<b>SPN</b>						
Время завершения	3	9	15	20	11	
Время оборота $T_r$	3	7	11	14	3	7,60
$T_r/T_s$	1,00	1,17	2,75	2,80	1,50	1,84
<b>SRT</b>						
Время завершения	3	15	8	20	10	
Время оборота $T_r$	3	13	4	14	2	7,20
$T_r/T_s$	1,00	2,17	1,00	2,80	1,00	1,59
<b>HRRN</b>						
Время завершения	3	9	13	20	15	
Время оборота $T_r$	3	7	9	14	7	8,00
$T_r/T_s$	1,00	1,17	2,25	2,80	3,50	2,14
<b>FB q = 1</b>						
Время завершения	4	20	16	19	11	
Время оборота $T_r$	4	18	12	13	3	10,00
$T_r/T_s$	1,33	3,00	3,00	2,60	1,50	2,29
<b>FB q = <math>2^i</math></b>						
Время завершения	4	17	18	20	14	
Время оборота $T_r$	4	15	14	14	6	10,60
$T_r/T_s$	1,33	2,50	3,50	2,80	3,00	2,63

## Первым поступил — первым обслужен

Простейшая стратегия планирования “первым поступил — первым обслужен” (first-come-first-served — FCFS) известна также как схема “первым пришел — первым вышел” (first-in, first-out — FIFO) или схема строгой очередности. Как только процесс становится готовым к выполнению, он присоединяется к очереди готовых процессов. При прекращении выполнения текущего процесса для выполнения выбирается процесс, который находился в очереди дольше других.

Как видите, стратегия FCFS гораздо лучше работает для длинных процессов, чем для коротких. Вот данные, приведенные в работе [81].

Процесс	Время входа	Время обслуживания $T_s$	Время начала	Время завершения	Время оборота $T_r$	$T_r/T_s$
W	0	1	0	1	1	1
X	1	100	1	101	100	1
Y	2	1	101	102	100	100
Z	3	100	102	202	199	1,99
<b>Среднее</b>					100	26

Нормализованное время оборота для процесса Y оказывается существенно большим, чем для других процессов. Общее время нахождения процесса в системе в 100 раз превышает время, необходимое для обработки процесса. Такая ситуация возникает, когда короткий процесс поступает в систему сразу после длинного. С другой стороны, даже в таком экстремальном случае длинные процессы хорошо обрабатываются. Так, время оборота процесса Z почти в два раза превышает время оборота Y, но нормализованное время ожидания процесса Z меньше 2.

Другая трудность при использовании стратегии FCFS связана с тенденцией процессов, ориентированных на работу с процессором, к получению преимущества над процессами, ориентированными на ввод-вывод. Рассмотрим множество процессов, один из которых ориентирован на использование процессора (связан с процессором), а остальные — на работу с устройствами ввода-вывода (связаны с вводом-выводом). При работе процесса, ориентированного на процессор, все остальные процессы вынуждены находиться в состоянии ожидания. Некоторые из них могут находиться в очереди ввода-вывода, в заблокированном состоянии, но могут и вернуться в очередь готовых к выполнению процессов за то время, пока выполняется процесс, ориентированный на использование процессора. В этот момент большинство или все устройства ввода-вывода могут простаивать, несмотря на то, что для них имеется потенциальная работа. Когда текущий выполняемый процесс покидает состояние выполняющегося, готовые ориентированные на ввод-вывод процессы быстро проходят через состояние выполнения и тут же оказываются заблокированными очередной операцией ввода-вывода. Если же в этот момент окажется заблокированным и процесс, ориентированный на использование процессора, то последний будет находиться в состоянии простоя, и, таким образом, стратегия FCFS может привести к неэффективному использованию как устройств ввода-вывода, так и процессора.

Для однопроцессорных систем FCFS — не самая подходящая стратегия, но для получения эффективного планирования она часто комбинируется с использованием приоритетов. В этом случае планировщик поддерживает ряд очередей, по одной для каждого уровня приоритета, и работает с процессами в каждой очереди в соответствии со стратегией FCFS. С примером такой системы мы познакомимся позже, при рассмотрении планирования со снижением приоритета.

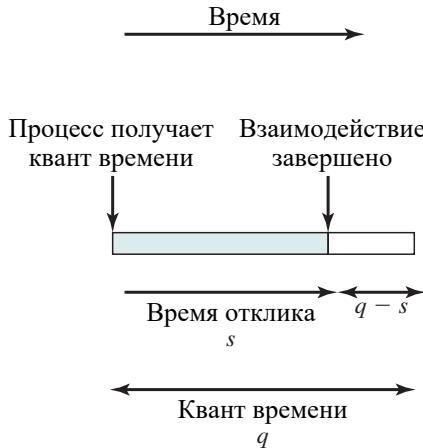
### Круговое планирование

Очевидный путь повышения эффективности работы с короткими процессами в схеме FCFS — использование вытеснения на основе таймера. Простейшая стратегия, основанная на этой идеи, — стратегия кругового (карусельного) планирования (round robin — RR). Таймер генерирует прерывания через определенные интервалы времени. При каждом прерывании выполняющийся в настоящий момент процесс помещается в очередь готовых к выполнению процессов, и начинает выполняться очередной процесс, выбранный в соответствии со стратегией FCFS. Эта методика известна также как **квантование времени** (time slicing), поскольку перед тем как оказаться вытесненным, каждый процесс получает квант времени для выполнения.

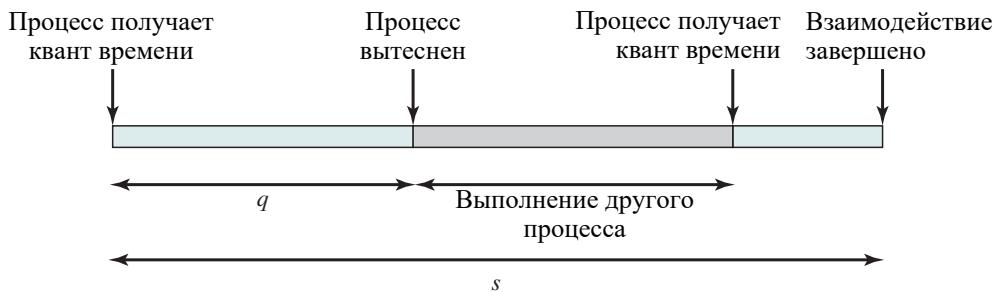
При круговом планировании принципиальным становится вопрос о продолжительности кванта времени. При малом кванте времени короткие процессы будут относительно быстро проходить через систему, но при этом возрастают накладные расходы, связанные с обработкой прерывания и выполнением функций планирования. Следовательно, очень коротких квантов времени следует избегать. Одно из полезных правил в этом случае звучит так: квант времени должен быть немного больше, чем время, требующееся для типичного полного обслуживания. Если квант оказывается меньшего размера, большинство процессов потребует как минимум двух квантов времени. На рис. 9.6 проиллюстрировано влияние продолжительности кванта времени на время отклика. Обратите внимание, что в предельном случае, когда квант времени превышает продолжительность самого длинного процесса, круговое планирование вырождается в планирование FCFS.

На рис. 9.5 и в табл. 9.5 показаны результаты работы круговой стратегии при использовании кванта времени  $q$  продолжительностью 1 и 4. Обратите внимание, что наиболее короткий процесс Е значительно быстрее проходит через систему при малом кванте времени.

Круговая стратегия особенно эффективна в системах общего назначения с разделением времени и в системах обработки транзакций. Чтобы обнаружить один из основных недостатков круговой схемы, рассмотрим работу с набором процессов, ориентированных как на процессор, так и на ввод-вывод. Обычно у процессов с интенсивным вводом-выводом промежуток времени между двумя операциями ввода-вывода, когда процесс использует процессор, меньше, чем у процесса, ориентированного на использование процессора. В результате возможна следующая ситуация: процесс с интенсивным вводом-выводом использует процессор в течение короткого промежутка времени и оказывается в заблокированном состоянии в ожидании завершения операции ввода-вывода. По завершении этой операции он вновь присоединяется к очереди готовых к выполнению процессов. С другой стороны, процесс с интенсивным использованием процессора обычно использует отпущененный ему квант времени полностью и немедленно возвращается в очередь готовых к выполнению процессов. Следовательно, процесс, ориентированный на работу с процессором, получает значительно большее процессорное время, что приводит к снижению производительности процессов с интенсивным вводом-выводом, неэффективному использованию устройств ввода-вывода и увеличению времени отклика.



а) Квант времени больше типичного взаимодействия

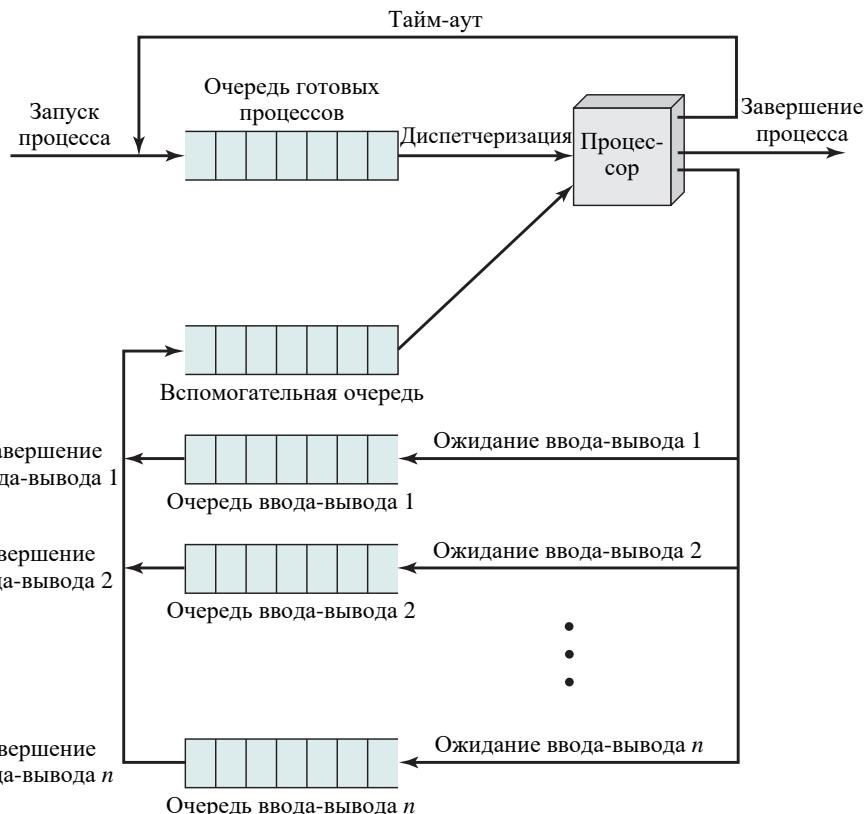


б) Квант времени меньше типичного взаимодействия

**Рис. 9.6.** Влияние продолжительности кванта времени на время отклика

В [100] предложено улучшение кругового планирования, которое названо в работе виртуальным круговым планированием (virtual round robin — VRR) и позволяет избежать пристрастности в работе. Данная схема показана на рис. 9.7. Новый процесс присоединяется к очереди готовых к выполнению процессов, управление которой осуществляется на основе стратегии FCFS. Когда исчерпывается время работающего процесса, он возвращается в очередь готовых к выполнению процессов; при блокировании процесса для ожидания завершения операции ввода-вывода он поступает в очередь процессов, ожидающих завершения операции ввода-вывода. Пока что все, как обычно. Новым оказывается наличие вспомогательной очереди, в которую переносятся процессы после их разблокирования по завершении операций ввода-вывода. При выборе процесса для выполнения преимущество отдается процессам из вспомогательной очереди. Когда диспетчеризуется процесс из вспомогательной очереди, он работает не дольше, чем время, равное продолжительности основного кванта времени минус общее время, затраченное на работу со времени последнего выбора из основной очереди готовых процессов. Из-у-

чение производительности такой схемы показывает, что с точки зрения беспристрастности данный подход лучше простого кругового планирования.



**Рис. 9.7.** Диаграмма работы виртуального кругового планирования

### Выбор самого короткого процесса

Еще один путь к снижению перекоса в пользу длинных процессов — использование стратегии выбора самого короткого процесса (shortest process next — SPN). Это невытесняющая стратегия, при которой для выполнения выбирается процесс с наименьшим ожидаемым временем выполнения.

На рис. 9.5 и в табл. 9.5 приведены результаты применения данной стратегии к нашему примеру. Обратите внимание, что процесс Е обслуживается гораздо раньше, чем в случае применения FCFS-стратегии. В отношении времени отклика общая производительность системы также возрастает, но при этом увеличивается разброс его величины, в особенности для длинных процессов (и, соответственно, снижается предсказуемость).

Основная трудность в применении стратегии SPN состоит в том, что для ее осуществления необходима по меньшей мере оценка времени выполнения, требующегося каждому процессу. При выполнении пакетных заданий может понадобиться оценка этого значения программистом и его предоставление операционной системе. Если оценка программиста существенно ниже реального времени выполнения, система может пре-

кратить выполнение задания. В промышленных системах часто выполняются одни и те же задания, так что можно собрать достаточно точную статистику. В случае выполнения интерактивных процессов операционная система может поддерживать во время выполнения средний “разрыв” для каждого процесса. Простейшее вычисление выглядит следующим образом:

$$S_{n+1} = \frac{1}{n} \sum_{i=1}^n T_i, \quad (9.1)$$

где

$T_i$  — время работы процессора для  $i$ -го экземпляра данного процесса (общее время работы для пакетного задания, время разрыва при интерактивной работе);

$S_i$  — предсказанное значение для  $i$ -го экземпляра;

$S_1$  — предсказанное значение для первого экземпляра (не вычисляется).

Для того чтобы избежать повторного вычисления всей суммы, уравнение (9.1) можно записать следующим образом:

$$S_{n+1} = \frac{1}{n} T_n + \frac{n-1}{n} S_n, \quad (9.2)$$

Заметим, что в данной формуле все экземпляры при усреднении имеют одинаковый вес, т.е. каждый член умножается на одинаковый коэффициент  $1/n$ ; обычно следует давать больший вес экземплярам, выполнившимся последними, так как они в большей степени отражают будущее поведение процесса. Обычная технология предсказания будущего значения на основе значений прошедших серий представляет собой взвешенное усреднение

$$S_{n+1} = \alpha T_n + (1 - \alpha) S_n, \quad (9.3)$$

где  $\alpha$  — постоянный весовой множитель ( $0 < \alpha < 1$ ), определяющий относительный вес последнего и предыдущих наблюдений (ср. с (9.2)). При использовании постоянного значения  $\alpha$ , не зависящего от количества наблюдений, мы получаем ситуацию, когда рассматриваются все предыдущие значения, причем чем значение более давнее, тем меньше его вес. Чтобы было понятнее, распишем (9.3) как

$$S_{n+1} = \alpha T_n + (1 - \alpha) \alpha T_{n-1} + \dots + (1 - \alpha)^i \alpha T_{n-i} + \dots + (1 - \alpha)^n S_1 \quad (9.4)$$

Поскольку и  $\alpha$ , и  $1 - \alpha$  меньше единицы, каждый последующий множитель в (9.4) меньше предыдущего. Например, при  $\alpha = 0,8$  уравнение (9.4) записывается как

$$S_{n+1} = 0.8 T_n + 0.16 T_{n-1} + 0.032 T_{n-2} + 0.0064 T_{n-3} + \dots + (0.2)^n S_1$$

т.е. чем старее наблюдение, тем меньший вклад оно вносит в вычисляемое среднее значение.

Значение коэффициента как функции от положения члена в сумме показано на рис. 9.8. Чем больше значение  $\alpha$ , тем больший вес имеют последние наблюдения. При  $\alpha = 0,8$  в вычислении среднего значения, по сути, участвуют только три-четыре последние наблюдения, в то время как при  $\alpha = 0,2$  заметный вклад вносят восьмое и бо-

лее поздние наблюдения. Значения  $\alpha$ , близкие к 1, позволяют нашему методу быстро реагировать на любые изменения, но при этом увеличивается и реакция на случайные отклонения от среднего значения при наблюдениях, что приводит к излишне резким изменениям вычисляемого значения.

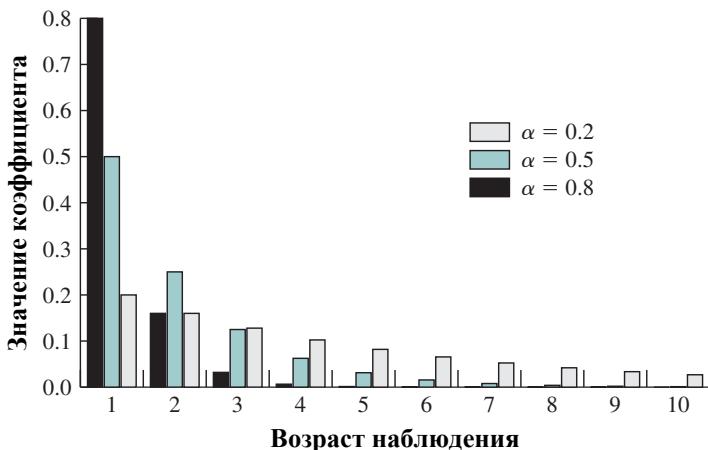


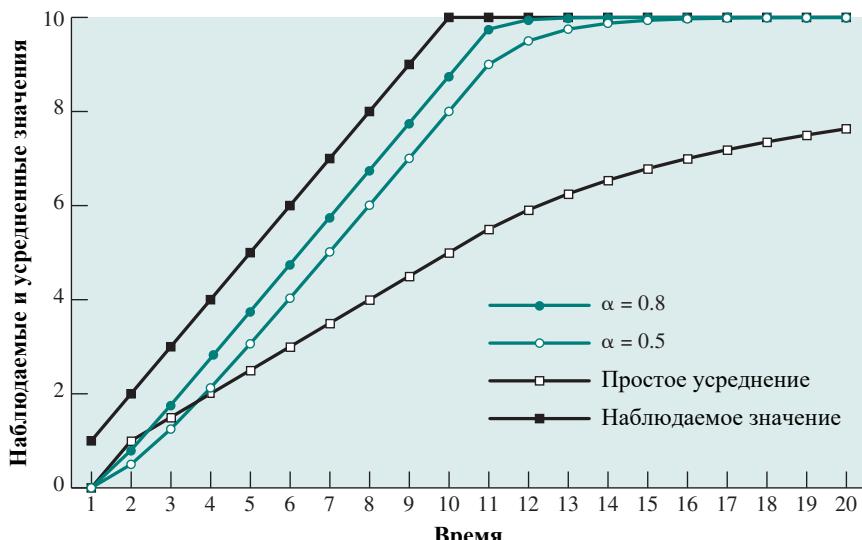
Рис. 9.8. Весовые коэффициенты при разных значениях  $\alpha$

На рис. 9.9 приведено сравнение простого и взвешенного усреднений для двух разных значений  $\alpha$ . На рис. 9.9, *a* значения  $\alpha$  начинаются с 1 и постепенно вырастают до 10, после чего продолжают удерживаться на этом уровне; на рис. 9.9, *b* наблюдаемые значения уменьшаются от 20 до 10. В обоих случаях мы начинаем с оценки  $S_1 = 0$ . Это обеспечивает больший приоритет новым процессам. Обратите внимание на то, насколько быстрее реагирует на изменение наблюдаемых значений взвешенное среднее по сравнению с обычным средним, и чем больше значение  $\alpha$ , тем выше скорость реакции.

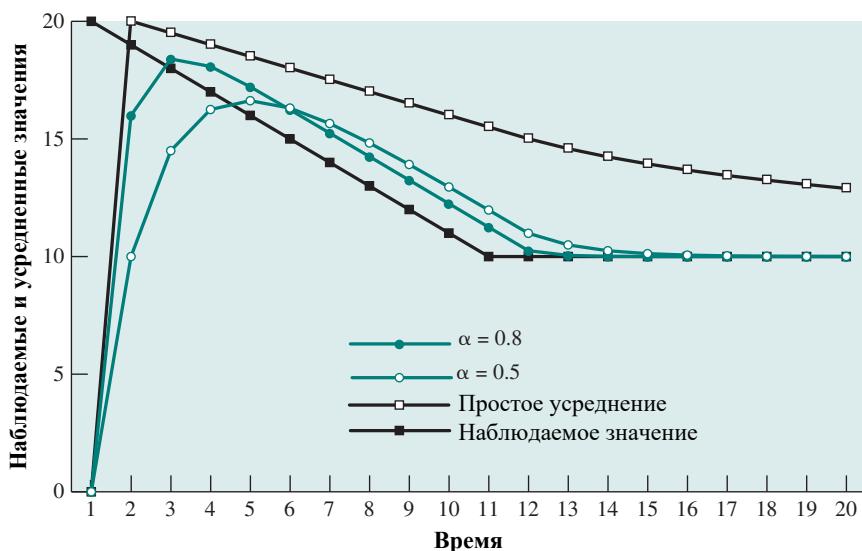
Основной риск при использовании стратегии SPN заключается в возможном голодании длинных процессов при стабильной работе коротких процессов. Кроме того, хотя SPN снижает перекос в пользу длинных процессов, его применение нежелательно в системах с разделением времени или в системах обработки транзакций из-за отсутствия вытеснения. Возвращаясь к анализу наихудшего случая для метода FCFS, мы увидим, что процессы W, X, Y и Z будут выполняться в том же порядке, причем обслуживание процесса Y имеет значительно худшие параметры, чем обслуживание других процессов.

### Наименьшее остающееся время

Стратегия наименьшего остающегося времени (shortest remaining time — SRT) представляет собой вытесняющую версию стратегии SPN. В этом случае планировщик выбирает процесс с наименьшим ожидаемым временем до окончания процесса. При присоединении нового процесса к очереди готовых к выполнению процессов может оказаться, что его оставшееся время в действительности меньше, чем оставшееся время выполняемого в настоящий момент процесса. Планировщик, соответственно, может применить вытеснение при готовности нового процесса.



а) Возрастающая функция



б) Убывающая функция

Рис. 9.9. Использование взвешенного усреднения

Как и при использовании стратегии SPN, планировщик для корректной работы функции выбора должен оценивать время выполнения процесса; в этом случае также имеется риск голодаания длинных процессов.

В случае использования стратегии SRT таких больших перекосов в пользу длинных процессов, как при использовании стратегии FCFS, нет; в отличие от стратегии кругового планирования здесь не генерируются дополнительные прерывания, что снижает накладные расходы. С другой стороны, в этом случае необходимость фиксировать и записывать время выполнения процессов приводит к увеличению накладных расходов.

В связи с тем что короткие задания немедленно получают преимущество перед выполняющимися длинными заданиями, стратегия SRT существенно выигрывает у стратегии SPN во времени оборота.

Обратите внимание, что в нашем примере (табл. 9.5) три наиболее кратких процессы обслуживаются немедленно, что приводит к нормализованному времени оборота для каждого из них, равному 1,0.

### **Наивысшее отношение отклика**

В табл. 9.5 мы использовали как показатель нормализованное время оборота, представляющее собой отношение времени оборота к фактическому времени обслуживания. Для каждого отдельного процесса этот показатель желательно минимизировать, так же, как и среднее значение по всем процессам. В общем случае мы не можем знать время обслуживания заранее, но можем оценить его либо на основе предыдущих выполнений, либо на основе информации, вводимой пользователем или задаваемой при настройке. Рассмотрим соотношение

$$R = \frac{w + s}{s},$$

где

$R$  — отношение отклика;

$w$  — время, затраченное процессом на ожидание;

$s$  — ожидаемое время обслуживания.

Если процесс будет немедленно диспетчеризован, его значение  $R$  будет равно нормализованному времени оборота. Заметим, что минимальное значение (равное 1,0)  $R$  принимает при входе процесса в систему.

Таким образом, правило стратегии планирования наивысшего отношения отклика (highest response ratio next — HRRN) можно сформулировать так: при завершении или блокировании текущего процесса для выполнения из очереди готовых процессов выбирается тот, который имеет наибольшее значение  $R$ . Такой подход довольно привлекателен, поскольку учитывает возраст процесса. Короткие процессы получают преимущество перед продолжительным (в силу меньшего знаменателя, увеличивающего отношение), однако и увеличение возраста процесса приводит к тому же результату, так что в конечном счете длинные процессы смогут конкурировать с короткими.

Как и в случае использования стратегий SRT и SPN, в описанной стратегии требуется оценка времени обслуживания для определения максимального значения  $R$ .

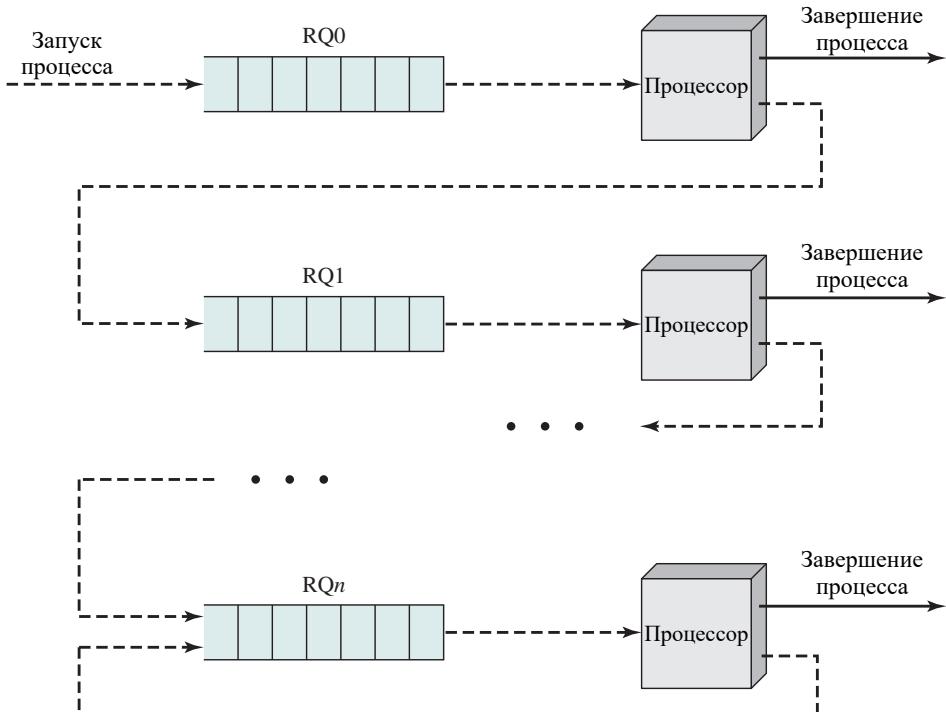
### **Снижение приоритета**

Если у нас нет никаких указаний об относительной продолжительности процессов, то мы не можем использовать ни одну из рассмотренных стратегий, SPN, SRT или HRRN. Еще один путь предоставления преимущества коротким процессам состоит в применении штрафных санкций к долго выполняющимся процессам. Другими словами, раз уж мы не можем работать с оставшимся временем выполнения, мы будем работать с затраченным до настоящего момента временем.

Вот как этого можно достичь. Выполняется вытесняющее (по квантам времени) планирование с использованием динамического механизма. Процесс при входе в систему помещается в очередь RQ0 (см. рис. 9.4). После первого вытеснения и возвращения в состояние готовности процесс помещается в очередь RQ1. В дальнейшем при каждом

вытеснении этот процесс вносится в очередь со все меньшим и меньшим приоритетом. Соответственно, быстро выполняющиеся короткие процессы не могут далеко зайти в иерархии приоритетов, в то время как длинные процессы постепенно теряют свой приоритет. Таким образом, новые короткие процессы получают преимущество в выполнении перед старыми длинными процессами. В рамках каждой очереди для выбора процесса используется стратегия FCFS. По достижении очереди с наименее высоким приоритетом процесс уже не покидает ее, всякий раз после вытеснения попадая в нее вновь (таким образом, эта очередь, по сути, обрабатывается с использованием кругового планирования).

На рис. 9.10 проиллюстрирован этот механизм планирования; пунктирной линией показан путь длинного процесса по различным очередям. Такой подход известен как **многоуровневый возврат** (multilevel feedback), поскольку при блокировании или вытеснении процесса осуществляется его возврат на очередной уровень приоритетности.



**Рис. 9.10.** Планирование со снижением приоритета

Имеется несколько разновидностей данной схемы. В простейшем случае вытеснение выполняется так же, как и в случае применения стратегии кругового планирования, — через периодические интервалы времени. В нашем примере (рис. 9.5 и табл. 9.5) использован именно этот метод, с квантом времени, равным 1.

При такой простой схеме имеется один недостаток, заключающийся в том, что время оборота длинных процессов резко растягивается. В системе с частым запуском новых процессов в связи с этим вполне вероятно появление голодания. Для уменьшения отрицательного эффекта мы можем использовать разное время вытеснения для процессов из разных очередей: процесс из очереди RQ0 выполняется в течение одной единицы времени и вытесняется; процесс из очереди RQ1 выполняется в течение двух единиц времени

и т.д. — процесс из очереди  $RQ_i$  выполняется до вытеснения в течение  $2^i$  единиц времени. Использование этой схемы также проиллюстрировано на рис. 9.5 и в табл. 9.5.

Даже при выделении процессу с более низким приоритетом большего количества времени для выполнения не удается полностью избежать голодания. Еще одним средством против голодания может служить перемещение процесса в очередь с более высоким приоритетом, если процесс не был обслужен в течение некоторого порогового времени для данной очереди.

## Сравнение производительности

Очевидно, что при выборе стратегии планирования критическим становится вопрос производительности. Однако точное сравнение стратегий невозможно в силу того, что относительная производительность зависит от ряда факторов, включая распределение времени обслуживания различных процессов, эффективность планирования и механизм переключения контекстов, а также природу запросов к устройствам ввода-вывода и их производительность. Тем не менее мы попытаемся сделать хотя бы самые общие выводы.

### Анализ очередей

В этом разделе мы используем основные формулы, описывающие работу очередей, в предположении запуска процессов в соответствии с распределением Пуассона и экспоненциального времени обслуживания.<sup>5</sup>

Сначала мы заметим, что любая стратегия планирования, которая выбирает очередной обслуживаемый процесс независимо от времени обслуживания, подчиняется следующему соотношению:

$$\frac{T_r}{T_s} = \frac{1}{1 - \rho},$$

где

$T_r$  — время оборота (общее время пребывания в системе, ожидания плюс выполнения);

$T_s$  — среднее время обслуживания (среднее время нахождения в состоянии выполняющегося процесса);

$\rho$  — степень использования процесса.

В частности, планирование с использованием приоритетов, при котором приоритет каждому процессу назначается независимо от ожидаемого времени обслуживания, обеспечивает то же нормализованное среднее время оборота, что и простейшая стратегия FCFS. Более того, эти средние значения не зависят от наличия или отсутствия вытеснения.

За исключением стратегий FCFS и RR, различные рассматривавшиеся до сих пор стратегии планирования осуществляют выбор процесса с учетом ожидаемого времени обслуживания. К сожалению, очень сложно разработать точные аналитические модели этих стратегий. Однако мы можем попытаться рассмотреть относительную производительность стратегий — в сравнении со стратегией FCFS, рассматривая приоритетное планирование, в котором приоритет основан на времени обслуживания.

---

<sup>5</sup> Подробнее с терминологией, связанной с очередями, можно познакомиться в приложении 3, "Концепции теории массового обслуживания".

Если планирование выполняется с учетом приоритетов и если классы приоритетов процессам назначаются на основе времени обслуживания, выявляются определенные отличия разных стратегий. В табл. 9.6 приведены формулы, получающиеся при использовании двух классов приоритетов с различным временем обслуживания для каждого класса.  $\lambda$  в таблице обозначает частоту поступления новых процессов в систему. Приведенные результаты могут быть обобщены для произвольного количества классов приоритетов. Обратите внимание, что для невытесняющего и вытесняющего планирования получаются разные результаты. При вытесняющем планировании предполагается, что низкоприоритетный процесс немедленно прерывается, как только становится готовым к выполнению процесс с более высоким приоритетом.

В качестве примера рассмотрим случай с двумя классами приоритетов, равным количеством поступающих в систему процессов каждого класса и средним временем обслуживания низкоприоритетного класса, в 5 раз превышающим время обслуживания высокоприоритетного класса. Тем самым мы хотим отдать предпочтение коротким процессам.

**Таблица 9.6. Формулы для двух классов приоритетов**

#### Предположения

1. Поступление новых процессов подчиняется распределению Пуассона
2. Процессы с приоритетом 1 обслуживаются перед процессами с приоритетом 2
3. Для процессов с равным приоритетом применяется стратегия “первым поступил — первым обслужен”
4. Во время обслуживания процессы не прерываются
5. Процессы не покидают очередь

#### a) Общие формулы

$$\lambda = \lambda_1 + \lambda_2$$

$$\rho_1 = \lambda_1 T_{s1}; \quad \rho_2 = \lambda_2 T_{s2}$$

$$\rho = \rho_1 + \rho_2$$

$$T_s = \frac{\lambda_1}{\lambda} T_{s1} + \frac{\lambda_2}{\lambda} T_{s2}$$

$$T_r = \frac{\lambda_1}{\lambda} T_{r1} + \frac{\lambda_2}{\lambda} T_{r2}$$

#### б) Экспоненциальное время обслуживания; прерываний нет

$$T_{r1} = T_{s1} + \frac{\rho_1 T_{s1} + \rho_2 T_{s2}}{1 + \rho_1}$$

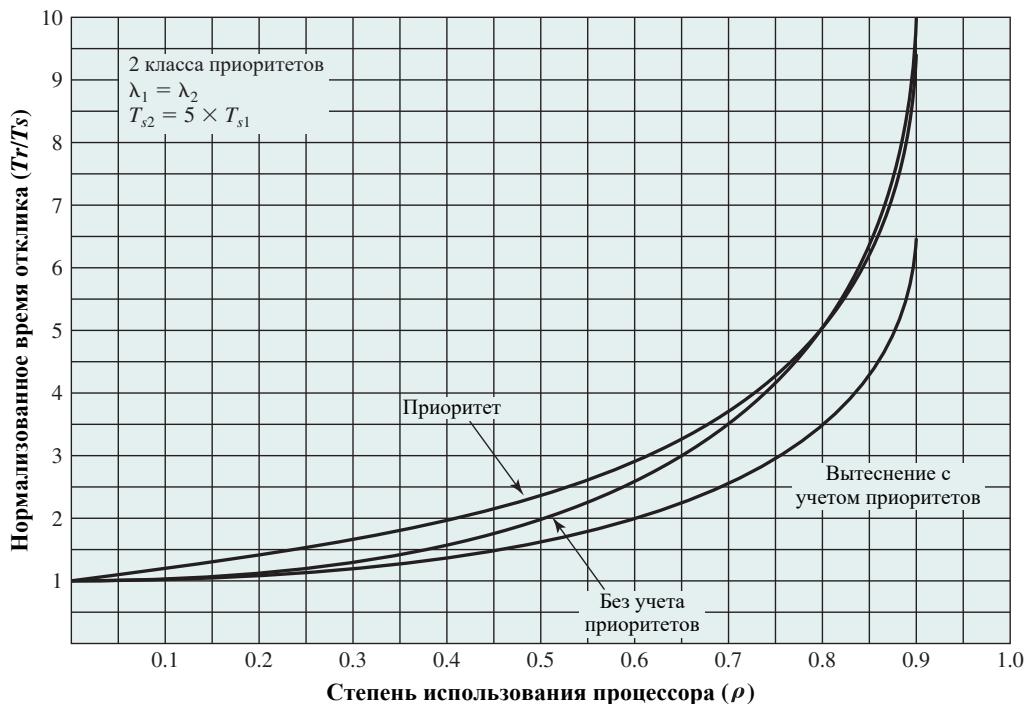
$$T_{r2} = T_{s2} + \frac{T_{r1} - T_{s1}}{1 - \rho}$$

#### в) Экспоненциальное время обслуживания с наличием вытеснения

$$T_{r2} = T_{s2} + \frac{T_{r1} - T_{s1}}{1 - \rho}$$

$$T_{r2} = T_{s2} + \frac{1}{1 - \rho_1} \left( \rho_1 T_{s2} + \frac{\rho_2 T_2}{1 - \rho} \right)$$

На рис. 9.11 показан общий результат. При предпочтении коротких процессов среднее нормализованное время оборота уменьшается. Как и следовало ожидать, результат еще лучше при использовании вытеснения. Обратите, однако, внимание на то, что на общую производительность это влияет не так сильно.



**Рис. 9.11.** Общее нормализованное время отклика

Однако при отдельном рассмотрении классов приоритетов наблюдаются значительные отличия. На рис. 9.12 показаны результаты для коротких процессов с высоким приоритетом. Для сравнения верхняя кривая на графике построена в предположении, что приоритеты не используются, и мы просто рассматриваем относительную производительность той половины процессов, продолжительность выполнения которых меньше. Две другие линии построены с учетом того, что эти процессы имеют более высокий приоритет.

На рис. 9.13 показаны результаты того же анализа для низкоприоритетных длительных процессов. Как и следовало ожидать, у этих процессов при приоритетном планировании наблюдается снижение производительности.

### Имитационное моделирование

Некоторые трудности аналитического моделирования преодолеваются посредством использования имитации дискретных событий, что позволяет моделировать широкий диапазон различных стратегий. Недостаток имитационного моделирования состоит в том, что полученный результат применим только для конкретного множества процессов при заданных предположениях. Несмотря на это имитационное моделирование позволяет получить интересные и полезные результаты.

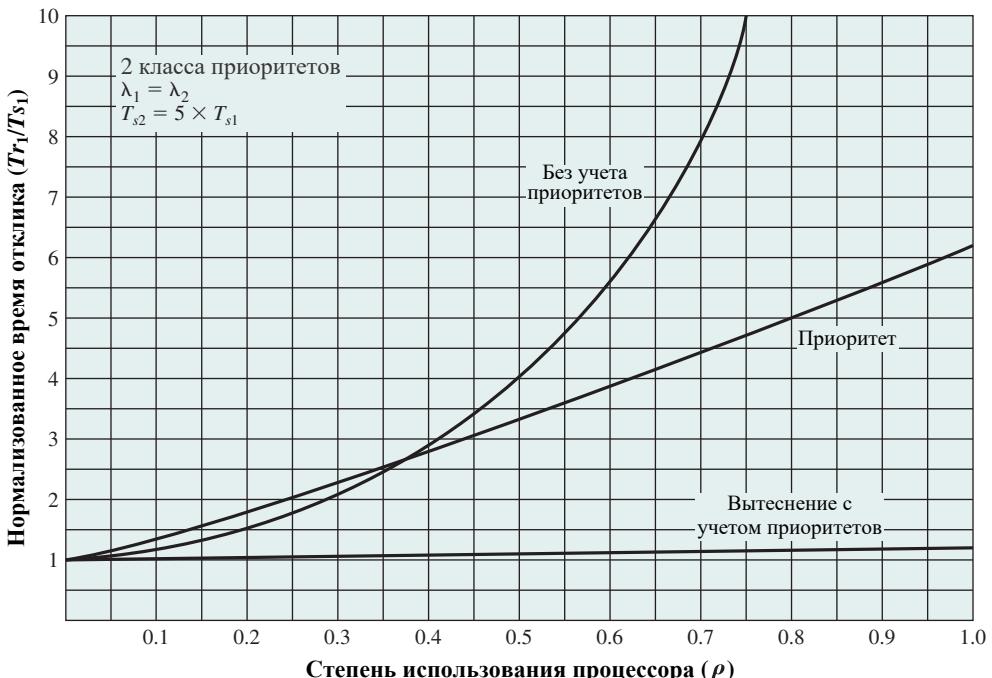


Рис. 9.12. Нормализованное время отклика для коротких процессов

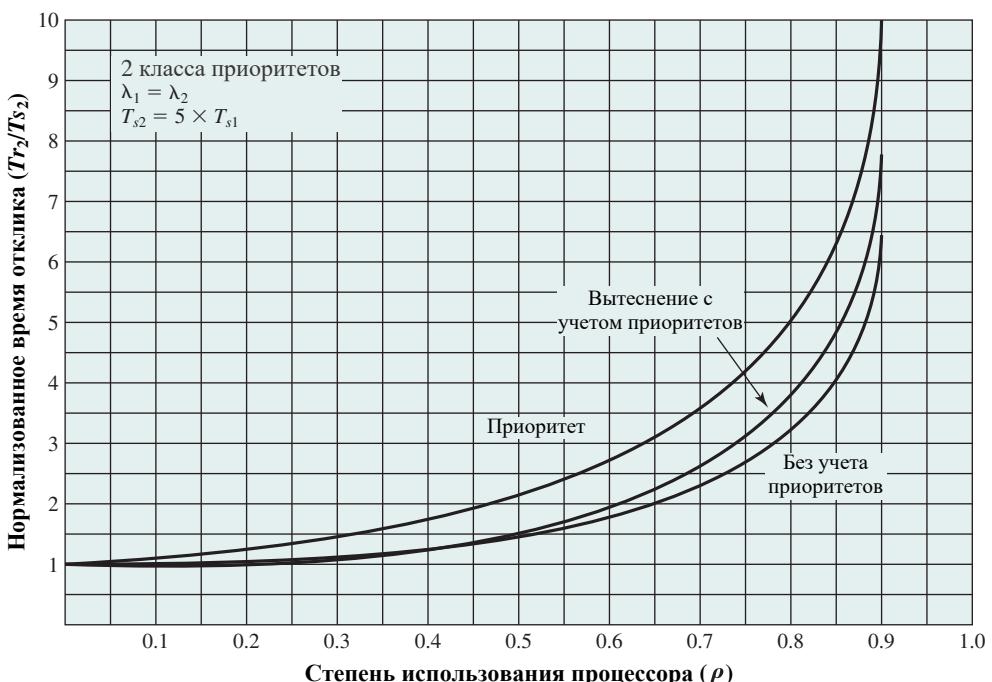
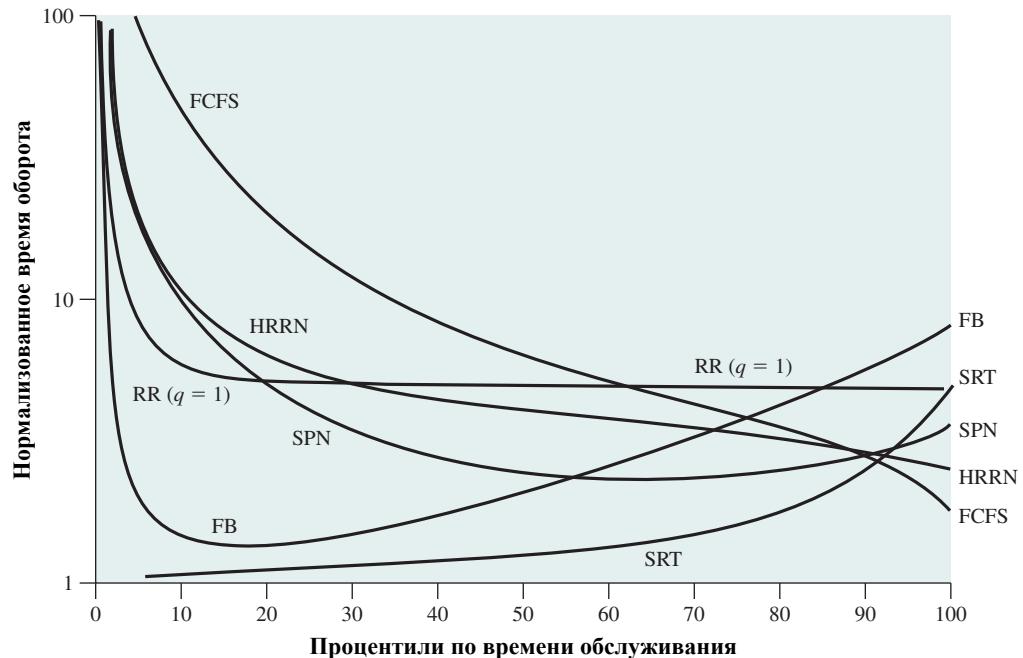
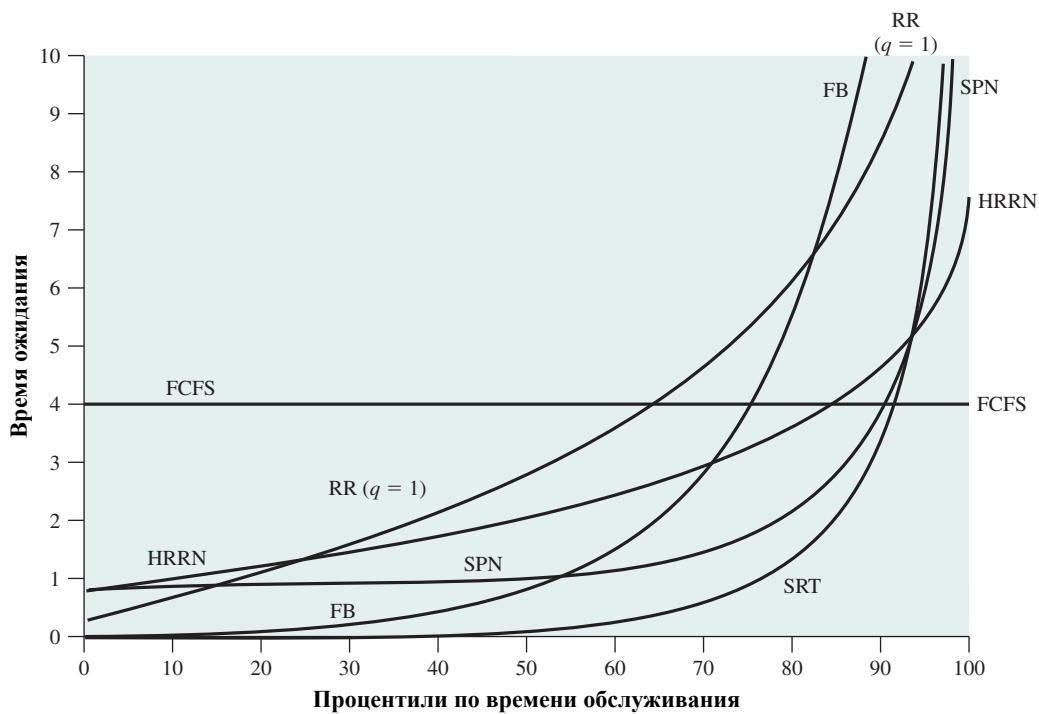


Рис. 9.13. Нормализованное время отклика для длинных процессов



**Рис. 9.14.** Результаты имитационного моделирования для нормализованного времени оборота



**Рис. 9.15.** Результаты имитационного моделирования для времени ожидания

Отчет об одном из таких исследований имеется в [81]. Имитировалось поведение 50 000 процессов со скоростью входа в систему  $\lambda = 0,8$  и средним временем обслуживания  $T_s = 1$ . Таким образом, предполагается, что степень использования процессора равна  $\rho = \lambda T_s = 0,8$ . Обратите внимание: здесь исследуется только одна степень использования процессора.

Для представления результатов моделирования процессы группируются в процентили по 500 процессов в соответствии со временем их обслуживания. Таким образом, 500 процессов с минимальным временем обслуживания содержатся в первом процентиле; если исключить эти процессы, 500 процессов с минимальным временем обслуживания среди оставшихся попадают во второй процентиль и т.д. Это позволяет рассматривать влияние различных стратегий планирования на процесс как функцию от продолжительности процесса.

На рис. 9.14 показано нормализованное время оборота, а на рис. 9.15 — среднее время ожидания. Взглянув на время оборота, мы можем увидеть, что производительность FCFS получается весьма непривлекательной — примерно у трети процессов время оборота более чем в 10 раз превышает время обслуживания; к тому же это самые короткие процессы. С другой стороны, время ожидания при этом одинаково для всех процессов, что обусловлено независимостью стратегии от времени обслуживания.

При применении стратегии кругового планирования используется квант, равный одной единице времени. За исключением очень коротких процессов, продолжительность которых — менее одного кванта, нормализованное время оборота при этой стратегии равняется примерно 5 квантам для всех процессов, тем самым обеспечивается беспристрастность. Производительность при использовании стратегии SPN выше, если не принимать во внимание короткие процессы. Стратегия SRT, представляющая собой версию SPN с вытеснением, превосходит SPN по производительности, за исключением 7% самых длинных процессов. Как видно по результатам исследований, среди невытесняющих стратегий планирования FCFS отдает предпочтение длинным процессам, а SPN — коротким. Стратегия HRRN разрабатывалась как компромиссное решение, и именно таковой, по результатам исследования, она и является. И наконец, как и ожидалось, стратегия со снижением приоритета с одинаковыми квантами времени для всех очередей неплохо работает с короткими процессами.

## Справедливое планирование

Все рассмотренные к этому моменту алгоритмы планирования рассматривают множество готовых к выполнению процессов как единый пул, из которого выбирается очередной процесс для выполнения. Этот пул может быть разделен по степени приоритета процессов, но в противном случае он остается гомогенным.

Однако в многопользовательских системах при организации приложений или задачий отдельных пользователей как множества процессов (или потоков) у них имеется структура, не распознаваемая традиционными планировщиками. С точки зрения пользователя, важно не то, как будет выполняться отдельный процесс, а то, как будет выполняться множество процессов, составляющих единое приложение. Таким образом, было бы неплохо, если бы планирование осуществлялось с учетом наличия таких множеств процессов. Данный подход в целом известен как справедливое (fair-share) планирование. Эта же концепция может быть распространена на группы пользователей, даже если каждый из пользователей представлен единственным процессом. Например, в системе с разделением времени мы можем рассматривать всех пользователей данного отдела как членов одной группы. Планировщик принимает решения с учетом необходимости предоставить каждой группе пользователей по возможности одинаковый сервис. Таким

образом, если в системе находится много пользователей из одного отдела, то изменение времени отклика должно, в первую очередь, коснуться именно пользователей этого отдела, не затрагивая прочих пользователей.

Термин *справедливое планирование* указывает на философию, лежащую в основе такого планирования. Каждому пользователю назначен определенный вес, который определяет долю использования системных ресурсов данным пользователем. В частности, каждый пользователь использует процессор. Данная схема работает более или менее линейно, так что если вес пользователя А в два раза превышает вес пользователя В, то в течение достаточно длительного промежутка времени пользователь А должен выполнить в два раза большую работу, чем пользователь В. Цель справедливого планирования состоит в отслеживании использования ресурсов и предоставлении меньшего количества ресурсов тому пользователю, который уже получил лишнее, и большего количества — тому, доля которого оказалась меньше справедливой.

Был предложен ряд алгоритмов справедливого планирования [103, 126, 270]. В этом разделе мы рассмотрим описанную в [103] схему планирования, реализованную в ряде систем UNIX. Эта схема известна как *справедливый планировщик* (fair-share scheduler — FSS). FSS при принятии решения рассматривает историю выполнения связанной группы процессов вместе с индивидуальными историями выполнения каждого процесса. Система разделяет пользовательское сообщество на множество групп со справедливым планированием и распределяет процессорное время между ними. Так, если у нас имеется четыре группы, каждая из них получит по 25% процессорного времени. В результате каждая группа обеспечивается виртуальной системой, работающей, соответственно, медленнее, чем система в целом.

Планирование осуществляется исходя из приоритетов с учетом приоритета процесса, недавнего использования им процессора и недавнего использования процессора группой, к которой он принадлежит. Чем больше числовое значение приоритета, тем ниже сам приоритет. Для процесса  $j$  из группы  $k$  применимы следующие формулы:

$$\begin{aligned} CPU_j(i) &= \frac{CPU_j(i-1)}{2} \\ GCPU_k(i) &= \frac{GCPU_k(i-1)}{2} \\ P_j(i) &= Base_j + \frac{CPU_j(i)}{2} + \frac{GCPU_k(i)}{4 \times W_k} \end{aligned}$$

где

- $CPU_j(i)$  — мера загруженности процессора процессом  $j$  на интервале  $i$ ;
- $GCPU_k(i)$  — мера загруженности процессора группой  $k$  на интервале  $i$ ;
- $P_j(i)$  — приоритет процесса  $j$  в начале интервала  $i$  (меньшее значение соответствует большему приоритету);
- $Base_j$  — базовый приоритет процесса  $j$ ;
- $W_k$  — вес, назначенный группе  $k$  ( $0 < W_k \leq 1$  и  $\sum_k W_k = 1$ ).

Каждому процессу назначается базовый приоритет. Приоритет процесса снижается по мере использования им процессора, так же как и по мере использования процессора

группой в целом. В случае использования процессора группой среднее значение нормализуется делением на вес группы. Чем больший вес назначен группе, тем меньше использование ею процессора влияет на приоритет.

На рис. 9.16 приведен пример, в котором процесс A находится в одной группе, а процессы B и C — в другой; вес каждой группы равен 0,5. Предположим, что все процессы ориентированы на вычисления и всегда готовы к выполнению. Базовый приоритет всех процессов — 60. Степень использования процессора определяется следующим образом: процессор прерывается 60 раз в секунду; при каждом прерывании увеличивается значение счетчика использования процессора текущего процесса, так же как и соответствующего счетчика группы. Один раз в секунду происходит перерасчет приоритетов.

В приведенной схеме первым запускается процесс A. Позже его вытесняет другой. Процессы B и C теперь имеют более высокий по сравнению с A приоритет, и на выполнение передается процесс B. По окончании второй единицы времени наивысший приоритет снова имеет процесс A, который и передается на выполнение. После этого ситуация повторяется с тем отличием, что теперь наивысший приоритет имеет процесс C. Очередность выполнения процессов такова: A, B, A, C, A, B, A, C, ... В результате 50% времени процессор занят выполнением процесса A и 50% — выполнением процессов B и C.

## 9.3. ТРАДИЦИОННОЕ ПЛАНИРОВАНИЕ UNIX

В этом разделе мы изучим традиционное планирование UNIX, используемое как в SVR3, так и в 4.3 BSD UNIX. Эти системы предназначены, в первую очередь, для работы в интерактивной среде с разделением времени. Алгоритм планирования разработан таким образом, чтобы обеспечить приемлемое время отклика для интерактивных пользователей, одновременно гарантируя отсутствие голодания низкоприоритетных задач. Хотя описываемый алгоритм и был заменен в более современных версиях UNIX, его изучение как представителя практически используемых алгоритмов с разделением времени не лишено основания. Схема планирования SRV4 соответствует требованиям реального времени, но этот вопрос мы обсудим в главе 10, “Многопроцессорное планирование и планирование реального времени”.

Традиционный планировщик UNIX использует многоуровневый возврат с применением кругового планирования в пределах очередей каждого приоритета, а также односекундное вытеснение. Таким образом, если текущий процесс не блокируется или не завершается в пределах одной секунды, он вытесняется. Приоритет основан на типе процесса и истории выполнения. Применяются следующие формулы:

$$CPU_j(i) = \frac{CPU_j(i-1)}{2}$$

$$P_j(i) = Base_j + \frac{CPU_j(i)}{2} + nice_j$$

где

$CPU_j(i)$  — мера использования процессора процессом  $j$  на протяжении интервала  $i$ ;

$P_j(i)$  — приоритет процесса  $j$  в начале интервала  $i$  (меньшее значение соответствует большему приоритету);

$Base_j$  — базовый приоритет процесса  $j$ ;

$nice_j$  — указываемый пользователем коэффициент настройки.

	Процесс А			Процесс В			Процесс С		
Время	Приоритет	Процесс	Группа	Приоритет	Процесс	Группа	Приоритет	Процесс	Группа
0	60	0	0	60	0	0	60	0	0
	1	1	1						
	2	2	2						
	•	•							
	•	•							
	60	60							
1	90	30	30	60	0	0	60	0	0
				1	1				1
				2	2				2
				•	•				•
				•	•				•
				60	60				60
2	74	15	15	90	30	30	75	0	30
		16	16						
		17	17						
		•	•						
		•	•						
		75	75						
3	96	37	37	74	15	15	67	0	15
				16				1	16
				17				2	17
				•				•	
				•				•	
				75				60	75
4	78	18	18	81	7	37	93	30	37
		19	19						
		20	20						
		•	•						
		•	•						
		78	78						
5	98	39	39	70	3	18	76	15	18

Группа 1

Группа 2

*Выполняющийся процесс указан на цветном фоне.***Рис. 9.16.** Пример справедливого планирования — три процесса, две группы

Приоритет каждого процесса пересчитывается один раз в секунду, в момент принятия решения о том, какой процесс будет выполняться следующим. Назначение базового приоритета состоит в разделении процессов на фиксированные группы уровней приоритетов. Значения компонентов *CPU* и *nice* ограничены требованием того, чтобы процесс не мог выйти из назначенной ему на основании базового приоритета группы. Эти группы используются для оптимизации доступа к блочным устройствам (например, к диску) и обеспечения быстрого отклика операционной системы на системные вызовы. Имеются следующие группы приоритетов (приведены в порядке снижения приоритетов).

- Свопинг
- Управление блочными устройствами ввода-вывода
- Управление файлами
- Управление символьными устройствами ввода-вывода
- Пользовательские процессы

Такая иерархия должна обеспечить наиболее эффективное использование устройств ввода-вывода. В группе пользовательских процессов использование истории выполнения приводит к применению штрафных санкций к процессам, ориентированным на вычисления, что также должно способствовать повышению эффективности системы. В сочетании с круговой схемой с вытеснением данная стратегия неплохо удовлетворяет требованиям к системе общего назначения с разделением времени.

Пример работы планировщика приведен на рис. 9.17. Процессы А, В и С создаются одновременно с одним и тем же базовым приоритетом 60 (мы игнорируем наличие значения параметра *nice*). Таймер прерывает выполнение процесса 60 раз в секунду и увеличивает значение счетчика текущего процесса. В примере предполагается, что ни один процесс не блокируется сам по себе и нет никаких других процессов, готовых к выполнению. Сравните этот рисунок с рис. 9.16.

## 9.4. РЕЗЮМЕ

Операционная система должна принимать во время выполнения процессов три типа решений, связанных с планированием. Долгосрочное планирование определяет, когда новый процесс должен поступить в систему. Среднесрочное планирование является частью свопинга и определяет, когда программа должна быть полностью или частично загружена в основную память, с тем чтобы она могла выполняться. Краткосрочное планирование определяет, какой из готовых к выполнению процессов будет выполняться процессором следующим. Эта глава была посвящена вопросам, связанным, в первую очередь, с краткосрочным планированием.

При разработке краткосрочного планировщика может использоваться ряд различных критериев. В соответствии с некоторыми из них поведение системы рассматривается с точки зрения пользователя (пользовательско-ориентированные), другие же ориентированы на общую эффективность системы, что отвечает нуждам всех пользователей (системно-ориентированные). Одни из критериев можно легко выразить количественно, другие же по своей природе в большей степени качественные. С точки зрения пользователя, наиболее важным критерием является время отклика, в то время как с позиции системы более важна степень использования процессора.

Время	Процесс А		Процесс В		Процесс С	
	Приоритет	Счетчик	Приоритет	Счетчик	Приоритет	Счетчик
0	60 1 2 • • 60	0	60	0	60	0
1	75	30	60 1 2 • • 60	0	60	0
2	67 15		75	30	60 1 2 • • 60	0
3	63 8 9 • • 67	7	67	15	75	30
4	76	33	63 8 9 • • 67	7	67	15
5	68	16	76	33	63	7

Выполняющийся процесс указан на цветном фоне.

**Рис. 9.17.** Пример традиционного планирования процессов в UNIX

Имеется ряд алгоритмов краткосрочного планирования, осуществляющих выбор среди готовых к выполнению процессов.

- **Первым поступил — первым обслужен.** Выбирается процесс, ожидающий обслуживания дольше других.
- **Круговое планирование.** Использует кванты времени для ограничения времени непрерывного выполнения процесса, циклически обслуживая имеющиеся процессы.
- **Выбор самого короткого процесса.** Выбирается процесс с наименьшим ожидаемым временем работы; вытеснение процессов не применяется.
- **Наименьшее оставшееся время.** Выбирается процесс с наименьшим ожидаемым временем оставшейся работы. Процесс может быть вытеснен другим готовым к выполнению процессом.
- **Наивысшее отношение отклика.** Принимаемое решение опирается на оценку нормализованного времени оборота.
- **Снижение приоритета.** Определяет множество очередей и распределяет в них процессы, основываясь на истории выполнения и других критериях.

Выбор алгоритма планирования зависит от ожидаемой производительности и сложности реализации.

## 9.5. Ключевые термины, контрольные вопросы и задачи

### Ключевые термины

Беспристрастность	Долгосрочное планирование	Предсказуемость
Время оборота (пребывания в системе)	Квант времени	Пропускная способность
Время обслуживания	Краткосрочное планирование	Скорость поступления процессов
Время ожидания	Круговое планирование	Справедливое планирование
Время отклика	Первым поступил — первым обслужен	Среднесрочное планирование
Диспетчер	Планирование с учетом приоритетов	Степень использования процессора

### Контрольные вопросы

- 9.1. Кратко опишите три типа планирования процессов.
- 9.2. Какое требование к производительности системы является критическим в случае использования интерактивной операционной системы?
- 9.3. В чем заключается отличие времени оборота от времени отклика?

- 9.4. Какой приоритет (высокий или низкий) представляет малое значение в случае планирования процессов?
- 9.5. В чем заключается отличие планирования с вытеснением от невытесняющего планирования?
- 9.6. Кратко опишите FCFS-планирование.
- 9.7. Кратко опишите круговое планирование.
- 9.8. Кратко опишите стратегию выбора самого короткого процесса.
- 9.9. Кратко опишите стратегию наименьшего оставшегося времени.
- 9.10. Кратко опишите стратегию наивысшего отношения отклика.
- 9.11. Кратко опишите стратегию со снижением приоритета.

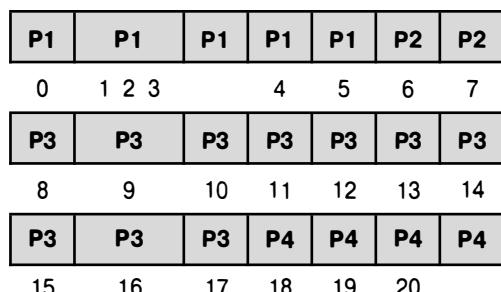
## Задачи

9.1. Рассмотрим следующий набор процессов.

Процесс	Время разрыва, мс	Приоритет	Время поступления, мс
P1	50	4	0
P2	20	1	20
P3	100	3	40
P4	40	2	60

а. Покажите планы, получающиеся с применением наименьшего оставшегося времени, невытесняющего приоритета (меньшее значение означает более высокий приоритет) и кругового планирования с квантом времени 30 мс. Используйте диаграмму наподобие приведенной ниже для плана FCFS планирования.

Пример для FCFS (1 единица = 10 мс):



б. Чему равно среднее время ожидания для всех указанных выше стратегий?

## 9.2. Рассмотрим следующее множество процессов.

Процесс	Время поступления	Время обработки
A	0	3
B	1	5
C	3	2
D	9	5
E	12	5

Выполните для данного множества анализ, аналогичный анализу, представленному на рис. 9.5 и в табл. 9.5.

- 9.3. Докажите, что среди невытесняющих алгоритмов планирования стратегия SPN обеспечивает минимальное среднее время ожидания для пакета одновременно поступивших заданий. Считаем, что планировщик всегда должен выполнять задание, если таковое доступно.
- 9.4. Предположим, что у нас имеется набор значений времени разрыва 6, 4, 6, 4, 13, 13, 13, а начальное приближение равно 10. Постройте график, аналогичный графику, приведенному на рис. 9.9.

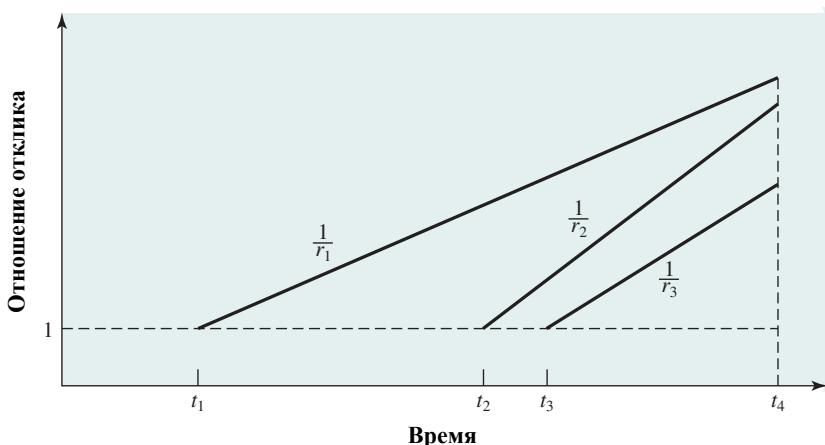
- 9.5. Рассмотрим следующую альтернативу формуле (9.3):

$$S_{n+1} = \alpha T_n + (1 - \alpha)S_n$$

$$X_{n+1} = \min [Ubound, \max[Lbound, (\beta S_{n+1})]]$$

Здесь *Ubound* и *Lbound* представляют собой заранее определенные верхнюю и нижнюю границы оценки значения *T*. Значение *X<sub>n+1</sub>* используется в алгоритме SPN вместо значения *S<sub>n+1</sub>*. Какие функции выполняют параметры  $\alpha$  и  $\beta$  и какое влияние оказывают высокие и низкие значения данных параметров?

- 9.6. В нижнем примере на рис. 9.5 процесс А работает две единицы времени, прежде чем управление передается процессу В. Другим вероятным сценарием является работа А на протяжении трех единиц времени до того, как управление будет передано процессу В. В чем состоит отличие стратегий в алгоритме со снижением приоритета для этих двух сценариев?
- 9.7. В невытесняющей однопроцессорной системе очередь готовых процессов содержит три задания в некоторый момент *t* (непосредственно после завершения предыдущего задания), поступивших в систему в моменты времени *t<sub>1</sub>*, *t<sub>2</sub>* и *t<sub>3</sub>*, ожидаемое время выполнения которых — *r<sub>1</sub>*, *r<sub>2</sub>* и *r<sub>3</sub>* соответственно. На рис. 9.18 показано линейное увеличение отношения отклика этих процессов со временем. Используйте данный пример для поиска стратегии планирования, известной как планирование минимакса отношения отклика, которая минимизирует максимальное отношение отклика для данного пакета заданий, игнорируя другие возможные поступления заданий. (Указание: сначала примите решение о том, какое задание будет выполняться последним.)



**Рис. 9.18.** Отношение отклика как функция времени

- 9.8. Докажите, что минимаксный алгоритм из предыдущей задачи минимизирует максимальное время отношения для данного пакета заданий. (Указание: рассмотрите задание с максимальным отношением отклика, до которого выполняются все остальные задания. Рассмотрите то же подмножество заданий, спланированных в другом порядке, и определите отношение отклика задания, выполняющегося последним. Обратите внимание: теперь наряду с заданиями из пакета в системе могут выполняться и другие задания.)
- 9.9. Определим время пребывания  $T_r$  как среднее общее время, затрачиваемое процессом на ожидание и обслуживание. Покажите, что в случае обслуживания “первым вошел — первым вышел” со средним временем обслуживания  $T_s$  и степенью загруженности процессора  $\rho$  справедливо соотношение  $T_r = T_s / (1 - \rho)$ .
- 9.10. Процессор переключается между готовыми к выполнению процессами с бесконечной скоростью без накладных расходов (это идеализированная модель кругового планирования с использованием квантов времени, которые гораздо меньше времени обслуживания). Покажите, что при распределенных в соответствии с законом Пуассона процессах с экспоненциальным временем обслуживания, входящих из бесконечного источника, среднее время отклика  $R_x$  процесса со временем обслуживания  $x$  задается соотношением  $R_x = x / (1 - \rho)$ . (Указание: обратитесь к основным уравнениям теории очередей, которые можно найти в приложении 3, “Концепции теории массового обслуживания”, и главе 20, “Обзор вероятности и стохастических процессов”. Затем примите количество ожидающих процессов в системе перед поступлением данного равным  $w$ .)
- 9.11. Рассмотрим вариант алгоритма кругового планирования, в котором элементы в очереди готовых процессов представляют собой указатели на РСВ.
- Что произойдет при помещении двух указателей на один и тот же процесс в очередь готовых процессов?
  - В чем состоит главное преимущество данной схемы?
  - Как можно модифицировать базовый алгоритм кругового планирования для достижения того же эффекта без дублирования указателей?

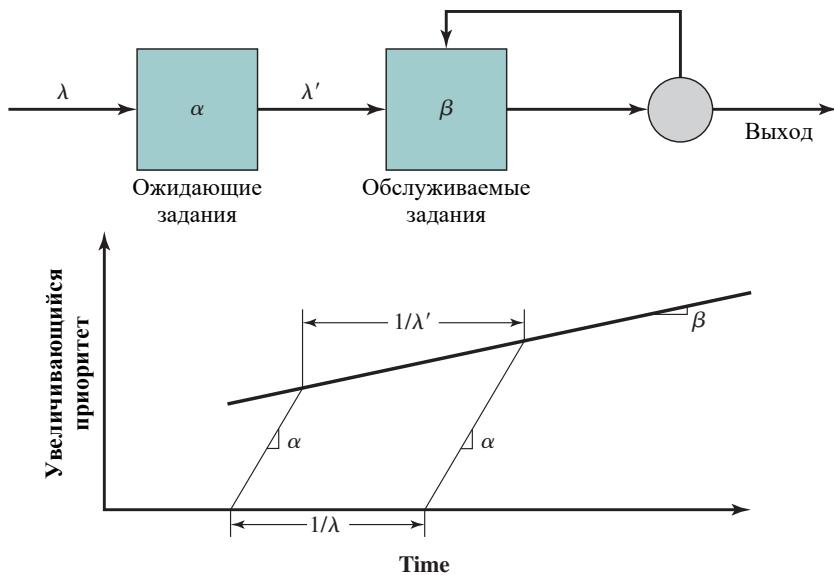
- 9.12. В системе с очередями новое задание должно ожидать своей очереди на обслуживание. В то время, когда задание находится в состоянии ожидания, его приоритет линейно возрастает со временем со скоростью  $\alpha$  от нулевого значения. Задание находится в состоянии ожидания до тех пор, пока его приоритет не сравняется с приоритетом обслуживаемых заданий, после чего оно будет обслуживаться наравне с другими процессами с использованием кругового планирования (при этом его приоритет возрастает со скоростью  $\beta$ , меньшей, чем  $\alpha$ ). Этот алгоритм известен как круговой эгоистичный, поскольку задания пытаются (понапрасну) монополизировать процессор путем постоянного повышения приоритета. Воспользуйтесь рис. 9.19, чтобы показать, что среднее время отклика  $R_x$  задания со временем обслуживания  $x$  задается формулой

$$R_x = \frac{s}{1 - \rho} + \frac{x - s}{1 - \rho'},$$

где

$$\rho = \lambda s \quad \rho' = \rho \left(1 - \frac{\beta}{\alpha}\right) \quad 0 \leq \beta < \alpha$$

в предположении, что время поступления и обслуживания распределено экспоненциально, со средними значениями  $1/\lambda$  и  $s$  соответственно. (Указание: рассмотрите систему в целом и две подсистемы в отдельности.)



**Рис. 9.19.** Эгоистичное круговое планирование

- 9.13. Интерактивная система с использованием кругового планирования и свопинга пытается обеспечить гарантированный отклик на тривиальные запросы следующим образом. После завершения кругового цикла по всем процессам в состоянии готовности система определяет квант времени для каждого готового процесса на следующий цикл посредством деления максимального времени отклика на количество процессов, требующих обслуживания. Насколько обоснована такая стратегия?

- 9.14. Какой тип процессов в целом получает преимущества при многоуровневом возврате (со снижением приоритета) — ориентированные на вычисления или ориентированные на операции ввода-вывода? Вкратце поясните свой ответ.
- 9.15. При использовании стратегии планирования на основе приоритетов процессов планировщик передает управление определенному процессу, если только в состоянии ожидания не находится процесс с более высоким приоритетом. Предположим, что при принятии решения о планировании не используется никакая иная информация. Примем также, что приоритеты процессов определяются при их создании и в дальнейшем не изменяются. Почему в такой системе опасно использование алгоритма Деккера для обеспечения взаимоисключений (см. раздел 5.1)? Поясните это, описав, какая нежелательная ситуация может сложиться и как это может произойти.
- 9.16. Пять пакетных заданий, от А до Е, поступают в вычислительный центр одновременно. Их ожидаемое время работы — 15, 9, 3, 6 и 12 минут соответственно. Их приоритеты, определенные при передаче заданий, равны соответственно 6, 3, 7, 9 и 4, причем меньшее значение означает более высокий приоритет. Для каждого из перечисленных ниже алгоритмов определите время оборота каждого процесса и среднее время оборота всех процессов. Накладные расходы, связанные с переключением процессов, не учитываются. Поясните, как вы пришли к данному ответу. В трех последних случаях предполагается, что в определенный момент времени работает только один процесс, вытеснения не происходит и все задания ориентированы на вычисления.
- Круговое планирование с размером кванта, равным 1 минуте.
  - Планирование с учетом приоритетов.
  - FCFS при запуске процессов в следующем порядке: 15, 9, 3, 6 и 12.
  - Первым выполняется самое короткое задание.



# 10

## ГЛАВА

# МНОГОПРОЦЕССОРНОЕ ПЛАНИРОВАНИЕ И ПЛАНИРОВАНИЕ РЕАЛЬНОГО ВРЕМЕНИ

В ЭТОЙ ГЛАВЕ...

## 10.1. Многопроцессорное и многоядерное планирование

Зернистость

- Независимые процессы
- Большая и очень большая зернистость
- Средняя зернистость
- Малая зернистость

Вопросы проектирования

- Назначение процессов процессорам
- Использование многозадачности отдельных процессоров
- Диспетчеризация процессов

Планирование процессов

Планирование потоков

- Разделение загрузки
- Бригадное планирование
- Назначение процессоров
- Динамическое планирование

Планирование потоков в многоядерных системах

## 10.2. Планирование реального времени

Введение

Характеристики операционных систем реального времени

Планирование реального времени

Планирование с предельными сроками

Частотно-монотонное планирование

Инверсия приоритета

### 10.3. Планирование в Linux

- Планирование реального времени
- Обычное планирование

### 10.4. Планирование в UNIX SVR4

### 10.5. Планирование в UNIX FreeBSD

- Классы приоритетов
- Поддержка SMP и многоядерности
  - Структура очереди
  - Учет интерактивности
  - Миграция потоков

### 10.6. Планирование в Windows

- Приоритеты процессов и потоков
- Многопроцессорное планирование

### 10.7. Резюме

### 10.8. Ключевые термины, контрольные вопросы и задачи

- Ключевые термины
- Контрольные вопросы
- Задачи

## УЧЕБНЫЕ ЦЕЛИ

- Пояснить концепцию зернистости потоков.
- Обсудить ключевые вопросы дизайна планирования потоков в многопроцессорной среде и некоторые ключевые подходы к планированию.
- Понимать требования, налагаемые планированием реального времени.
- Пояснить методы планирования, используемые в Linux, UNIX SVR4 и Windows 10.

В этой главе мы продолжим обзор планирования процессов. Начнем с изучения вопросов, возникающих при наличии в системе нескольких процессоров, и изучим ряд вопросов проектирования таких систем. Затем мы перейдем к планированию процессов в многопроцессорной системе, рассматривая проектные решения для многопроцессорного планирования потоков. Во втором разделе этой главы рассматривается планирование в реальном времени. Раздел начинается с обсуждения характеристик процессов реального времени, а затем мы переходим к изучению природы планируемых процессов. В главе рассматриваются два подхода к планированию реального времени — планирование с предельными сроками и частотно-монотонное планирование.

## 10.1. МНОГОПРОЦЕССОРНОЕ И МНОГОЯДЕРНОЕ ПЛАНИРОВАНИЕ

Если вычислительная система содержит более одного процессора, при разработке планирования встает ряд новых вопросов. Начнем с краткого обзора многопроцессорности и разницы в планировании на уровне процессов и на уровне потоков.

Многопроцессорные системы можно классифицировать следующим образом.

- **Слабосвязанные, или распределенные, системы, или кластеры.** Состоят из набора относительно автономных систем; каждый процессор имеет собственную основную память и каналы ввода-вывода. Этот тип систем будет рассмотрен в главе 16, “Облачные операционные системы и операционные системы Интернета вещей”.
- **Функционально специализированные процессоры.** В качестве примера можно привести процессор ввода-вывода. В такой ситуации у нас имеется ведущий процессор общего назначения; специализированные процессоры, предоставляющие ему свои услуги, являются ведомыми, управляемыми ведущим процессором. Вопросы, связанные с процессорами ввода-вывода, будут затронуты в главе 11, “Управление вводом-выводом и планирование дисковых операций”.
- **Сильносвязанные системы.** Состоят из множества процессоров, совместно использующих общую основную память и находящихся под общим управлением операционной системы.

Материал, изложенный в данной главе, касается последней категории многопроцессорных систем, а еще точнее — вопросов планирования в таких системах.

## Зернистость

Неплохой характеристикой многопроцессорной системы может служить зернистость синхронизации (synchronization granularity), которая представляет собой частоту синхронизации между процессорами в системе. Мы будем различать пять типов параллельных вычислений по степени их зернистости, приведенные в табл. 10.1.

**Таблица 10.1. Зернистость синхронизации**

Размер зерна	Описание	Интервал синхронизации, команды
Малый	Параллельные вычисления на уровне отдельных команд	Менее 20
Средний	Параллельные вычисления или многозадачность в пределах одного приложения	20–200
Большой	Многозадачность на уровне процессов в многозадачной среде	200–2000
Очень большой	Распределенная работа в сети для формирования единой вычислительной среды	2000–1 млн.
Независимые процессы	Не связанные между собой процессы	—

### Независимые процессы

В этом случае явной синхронизации между процессами нет. Каждый из них представляет собой отдельное, независимое приложение или задание. Использование такого типа параллельных вычислений характерно для систем с разделением времени. Каждый пользователь работает со своим приложением, таким как текстовый редактор или электронные таблицы, а многопроцессорная система обеспечивает для них тот же сервис, что и многозадачный режим в системе с одним процессором. Поскольку при этом доступно несколько процессоров, среднее время отклика будет меньше, чем в системе с одним процессором.

Тот же эффект можно получить и при предоставлении каждому конечному пользователю персонального компьютера или рабочей станции. Если пользователи совместно работают с какими-то файлами или другой информацией, индивидуальные системы должны быть объединены в распределенную систему, поддерживаемую с помощью сети. Такой подход рассматривается в главе 16, “Облачные операционные системы и операционные системы Интернета вещей”. С другой стороны, по многим параметрам с точки зрения стоимости единая многопроцессорная система по сравнению с распределенной получается более эффективной, позволяя, например, сэкономить на дисках и других периферийных устройствах.

### Большая и очень большая зернистость

В этом случае между процессами наблюдается синхронизация, хотя и весьма редкая. Такую ситуацию можно трактовать как множество параллельно выполняющихся процес-

сов, которые работают в многозадачном режиме на одном процессоре и с минимальным изменением программного обеспечения (или вовсе без изменений) могут поддерживаться многопроцессорной системой.

Простой пример приложения, способного использовать наличие нескольких процессоров, приведен в [269]. Авторы разработали программу, которая получает список файлов, требующих перекомпиляции, и определяет, какие из них могут компилироваться одновременно (обычно это все файлы). Затем программа запускает по одному процессу для компиляции каждого файла параллельно с другими. Авторы сообщают, что ускорение в многопроцессорной системе превысило ожидаемое (рассчитанное на основе простого добавления процессоров), что связано с совместным использованием дисковых кешей (этот вопрос подробнее изучается в главе 11, “Управление вводом-выводом и планирование дисковых операций”) и кода компилятора, который загружается в основную память только один раз.

В общем случае от использования многопроцессорной архитектуры выигрывает любое множество параллельно выполняющихся процессов, для которых требуется синхронизация или обмен информацией. При очень редком взаимодействии процессов целесообразнее использовать распределенную систему, но при более частом взаимодействии накладные расходы по передаче информации по сети могут привести к снижению производительности, и в этом случае более эффективным решением будет использование многопроцессорной системы.

### **Средняя зернистость**

В главе 4, “Потоки”, рассказывалось, что единое приложение может быть эффективно реализовано в виде множества потоков в пределах одного процесса. В этом случае потенциальная параллельность приложения должна быть явно указана программистом. Обычно между потоками одного приложения требуется более высокая степень координации и взаимодействия, чем между различными процессами, что и приводит к среднему уровню синхронизации.

В то время как параллельность независимых процессов, а также процессов с большим и очень большим зерном может поддерживаться как многозадачной системой с одним процессором, так и многопроцессорной системой с минимальным влиянием стратегии планирования, при работе многопоточных приложений планирование становится важной задачей, требующей пересмотра с учетом новых требований. В силу частого взаимодействия потоков одного приложения правильное решение о планировании одного потока может существенно повлиять на производительность приложения в целом. Позже мы вернемся к этому вопросу.

### **Малая зернистость**

Малая зернистость представляет более сложное использование возможностей параллельных вычислений, чем использование потоков. Хотя основная работа выполняется высокопараллельными приложениями, к настоящему моменту это специализированная и фрагментированная область, в которой используется множество различных подходов.

В главе 4, “Потоки”, имеется пример использования зернистости для игровой программы Valve.

## Вопросы проектирования

Планирование в многопроцессорной системе включает три взаимосвязанных вопроса:

1. назначение процессов процессорам;
2. использование многозадачности на отдельных процессорах;
3. диспетчеризация процесса.

При рассмотрении этих вопросов не следует забывать о том, что конкретный рассматриваемый подход зависит, вообще говоря, от степени зернистости приложений и количества доступных процессоров.

### Назначение процессов процессорам

В предположении единства архитектуры многопроцессорной системы (в том смысле, что ни один процессор физически не отличается от других в плане доступа к основной памяти или устройствам ввода-вывода) простейший подход к планированию состоит в рассмотрении процессоров как единого ресурса и назначении процессов процессорам по требованию. При таком подходе возникает один вопрос: каким должно быть такое назначение: статическим или динамическим?

Если процесс назначается одному процессору постоянно — от момента активации и до его завершения, — то для каждого процессора следует поддерживать отдельную краткосрочную очередь. Такой подход способствует уменьшению накладных расходов планирования процессов (поскольку назначение процесса процессору выполняется только один раз). Кроме того, использование выделенных процессоров обеспечивает возможность применения стратегии, известной как групповое (group), или бригадное (gang), планирование (о нем мы поговорим ниже).

Недостатком статического распределения является то, что когда один процесс загружен работой, другой может простоять. Для предотвращения такой ситуации можно использовать очередь, общую для всех процессоров. Все процессы попадают в одну глобальную очередь и передаются для выполнения любому свободному процессору. Таким образом, при этом подходе в течение жизни процесс может в разное время выполнятьсь на разных процессорах. В сильносвязанных системах с общей памятью информация контекста всех процессов доступна всем процессорам и, следовательно, стоимость планирования процесса не зависит от того, какой из процессов оказывается выбранным. Еще одним вариантом является динамическая балансировка нагрузки, при которой потоки перемещаются из очереди одного процессора в очередь другого процессора; этот подход использует Linux.

Независимо от выбранного типа распределения, используются два основных подхода к назначению процессов процессорам: ведущий/ведомый процессоры и равноправные процессоры. В случае использования архитектуры “ведущий/ведомый” ключевые функции операционной системы всегда выполняются на одном специально выделенном процессоре; все остальные процессоры могут выполнять только пользовательские приложения. Ведущий процессор отвечает за планирование заданий. Когда активный процесс на ведомом процессоре требует определенного обслуживания (например, осуществляет вызов ввода-вывода), он должен послать запрос ведущему процессору и ожидать завершения сервиса. Такой подход достаточно прост и требует внесения небольших дополнений в однопроцессорную многозадачную операционную систему. Упрощается при таком

подходе и разрешение конфликтов, поскольку один процессор управляет всей памятью и ресурсами ввода-вывода. Однако у такого подхода имеются два недостатка: 1) сбой ведущего процессора приводит к неработоспособности всей системы в целом и 2) ведущий процессор превращается в узкое место системы, определяющее ее производительность в целом.

При использовании архитектуры равноправных процессоров ядро операционной системы может выполняться на любом из процессоров, и каждый процессор самостоятельно планирует свою работу, беря процессы для выполнения из общего пула. Такой подход усложняет операционную систему, которая должна гарантировать, что никакие два процессора не выберут одновременно один и тот же процесс для выполнения и что не будет никаких потерь из очереди. Здесь должна быть применена технология, использующаяся для разрешения конфликтов и синхронизации запросов к ресурсам.

Конечно, кроме этих крайностей, имеется ряд других подходов. Например, для работы операционной системы может быть выделен не один процессор, а несколько; возможно использование системы приоритетов и истории выполнения для отделения процессов ядра от прочих процессов.

### **Использование многозадачности отдельных процессоров**

Когда каждый процесс назначается процессору статически на все время жизни, возникает новый вопрос: должен ли этот процессор быть многозадачным? На первый взгляд, такой вопрос кажется просто странным: ведь мы уже выяснили, что использование процессора в однозадачном режиме, при том что процесс может обращаться к устройствам ввода-вывода, является попросту расточительным.

При рассмотрении традиционных многопроцессорных систем, работающих с независимыми процессами или с большим зерном синхронизации (см. табл. 10.1), очевидно, что каждый отдельный процессор должен иметь возможность переключаться между процессами для повышения загруженности, а следовательно, и производительности. Однако в случае использования приложений со средней зернистостью синхронизации ситуация не столь ясна. При доступности множества процессоров правило максимально возможной загруженности процессора перестает быть первостепенным; вместо этого первостепенную важность приобретает обеспечение максимальной средней производительности приложений. Приложение, состоящее из множества потоков, может плохо работать до тех пор, пока его потоки не получат возможность одновременного выполнения.

### **Диспетчеризация процессов**

Последний вопрос проектирования, связанный с многопроцессорным планированием, является вопросом фактического выбора процесса для выполнения. Мы видели, что в многозадачной системе с одним процессором использование приоритетов или сложных алгоритмов, учитывающих время работы процессов, может существенно повысить производительность системы по сравнению с использованием стратегии “первым пришел — первым обслужен”. При рассмотрении многопроцессорных систем более простой подход может оказаться более эффективным в силу уменьшения накладных расходов. В случае планирования потоков в действие вступают новые характеристики, которые могут оказаться более важными, чем приоритеты или история выполнения. Позже мы детально рассмотрим эти вопросы.

## Планирование процессов

В большинстве традиционных многопроцессорных систем назначение процессов процессорам отсутствует. Вместо этого используется общая очередь для всех процессов либо — в случае применения схемы с учетом приоритетов — множество очередей на основе приоритетов с общим пулом процессоров. В любом случае мы можем рассматривать систему как очередь с несколькими серверами.

Рассмотрим систему с двумя процессорами, в которой каждый процессор обладает производительностью, в два раза меньшей, чем производительность процессора в однопроцессорной системе. В [215] проведено сравнение стратегии планирования FCFS с круговым планированием и планированием на основе минимального оставшегося времени выполнения. При круговом планировании квант времени считался большим по сравнению с затратами на переключение контекстов заданий и малым — по сравнению со временем обслуживания, которое и было основным объектом исследования. Полученные результаты в значительной степени зависят от того, насколько велика разница во времени обслуживания разных процессов. Количественно эту разницу можно выразить как коэффициент вариации  $C_s$ .<sup>1</sup> Значение  $C_s = 0$  соответствует отсутствию вариации времени обслуживания (время обслуживания всех процессов одинаково). Увеличение значения  $C_s$  соответствует росту изменчивости времени обслуживания, т.е. чем выше это значение, тем больше отклонения отдельных значений времени обслуживания от среднего. Значения коэффициента вариации, превышающие 5, не являются чем-то необычным для распределения времени обслуживания процессов.

На рис. 10.1, а приведено сравнение пропускных способностей кругового планирования и планирования FCFS как функций коэффициента вариации  $C_s$ . Обратите внимание, что разница между ними существенно меньше в системах с двумя процессорами. При наличии двух процессоров длинный процесс не так сильно влияет на систему в целом, поскольку другие процессы могут использовать второй процессор. Аналогичные результаты можно увидеть и на рис. 10.1, б.

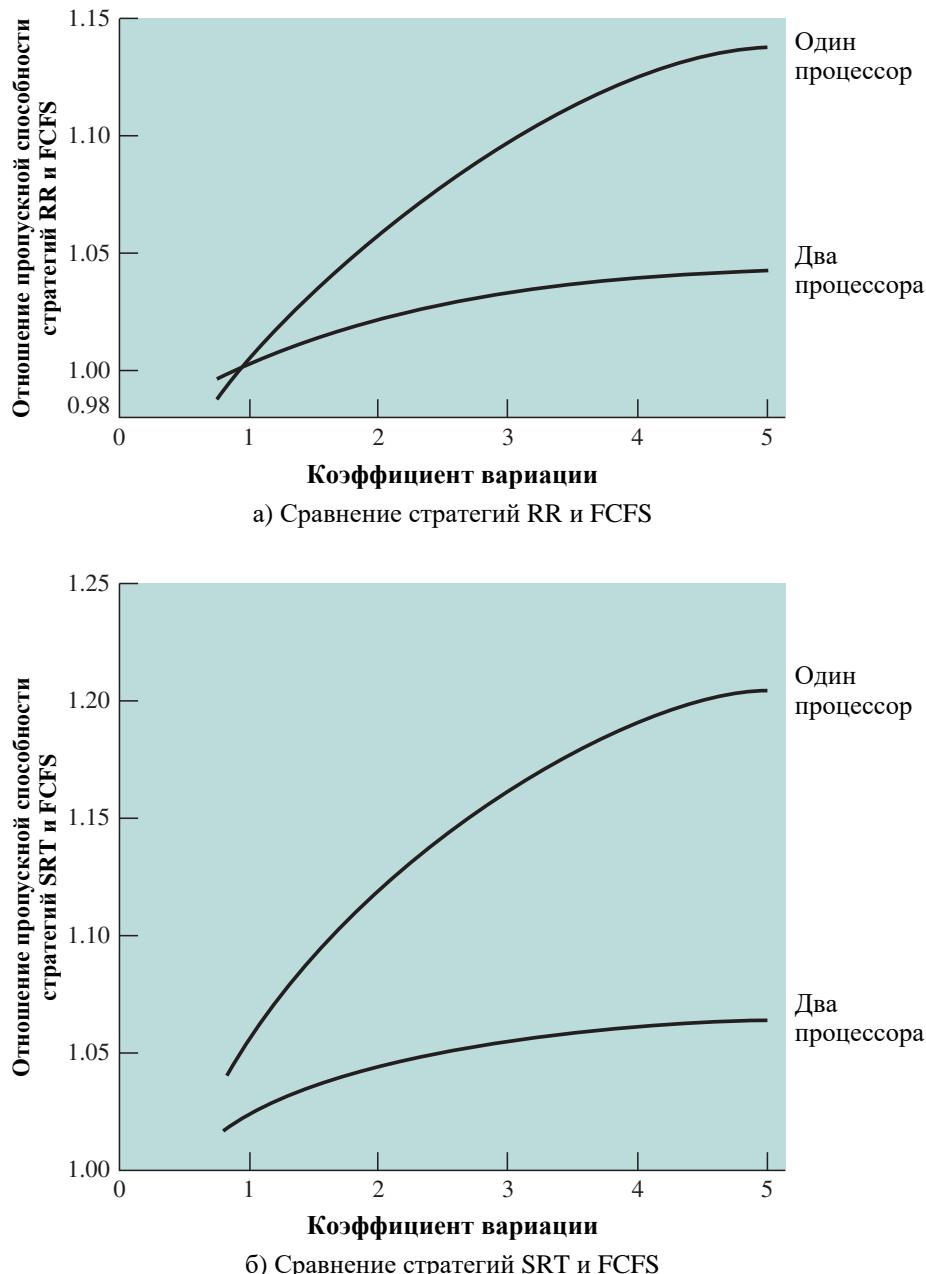
В работе [215] такой же анализ был проведен для самых разных предположений о степени многозадачности, соотношении количества процессов, ориентированных на вычисления и на ввод-вывод, и использовании приоритетов. Общий вывод, сделанный в этой работе, следующий: влияние выбора стратегии планирования на производительность при наличии двух процессоров существенно меньше, чем при наличии лишь одного. Очевидно, что с ростом количества процессоров рассматриваемое влияние уменьшается. Следовательно, простейшую стратегию FCFS (в том числе и в схеме со статическими приоритетами) можно вполне успешно применять в многопроцессорной системе.

## Планирование потоков

Как мы видели, в случае использования потоков концепция выполнения отделена от остальной части определения процесса. Приложение может быть реализовано как множество взаимодействующих потоков, выполняющихся параллельно в одном адресном пространстве.

---

<sup>1</sup> Значение  $C_s$  вычисляется как  $\sigma_s / T_s$ , где  $\sigma_s$  — стандартное отклонение времени обслуживания, а  $T_s$  — среднее время обслуживания.



**Рис. 10.1.** Сравнение стратегий планирования при наличии одного и двух процессоров

В однопроцессорной системе потоки могут использоваться для структуризации программы и с целью перекрытия операций ввода-вывода вычислительной работой. Поскольку по сравнению с переключением процессов переключение потоков осуществляется с гораздо меньшими затратами, оно имеет преимущество в стоимости. Однако полная мощь потоков проявляется только в многопроцессорных системах, в которых потоки могут использоваться для достижения истинно параллельных вычислений в рамках одного приложения. Если различные потоки одного приложения выполняются одновременно на разных процессорах, можно достичь резкого повышения производительности. Однако в приложении с интенсивным взаимодействием потоков (среднезернистая синхронизация) небольшие различия в стратегии управления потоками и их планирования могут привести к существенному изменению производительности [7].

Среди множества вариантов планирования потоков в многопроцессорных системах и назначения процессов процессорам можно выделить четыре основных подхода.

- Разделение загрузки.** Процессы не назначаются конкретным процессорам. Поддерживается глобальная очередь готовых к выполнению потоков, и каждый процессор в состоянии простоя выбирает поток из этой очереди. Термин *разделение загрузки* (*load sharing*) используется, чтобы отличать эту стратегию от схем со сбалансированной загрузкой, в которых работа распределяется на более постоянной основе (см., например, [77]).<sup>2</sup>
- Бригадное планирование** (*gang scheduling*). Множество связанных потоков распределяются для одновременной работы среди множества процессоров, по одному потоку на процессор.
- Назначение выделенных процессоров** (*dedicated processor assignment*). В противоположность подходу с разделением нагрузки этот подход обеспечивает неявное планирование путем назначения потоков процессорам. Каждой программе на время ее выполнения выделяется количество процессоров, равное количеству потоков программы. По окончании работы программы процессоры возвращаются в общий пул для распределения другим программам.
- Динамическое планирование** (*dynamic scheduling*). Количество потоков процесса может изменяться во время работы.

### Разделение загрузки

Разделение загрузки представляет собой, пожалуй, наиболее простой подход, непосредственно переносимый из однопроцессорной среды в многопроцессорную. Он имеет ряд преимуществ.

- Загрузка равномерно распределяется между процессорами, обеспечивая отсутствие простоя процессоров при наличии работы.
- Не требуется централизованный планировщик; когда процессор становится доступным, он сам выполняет подпрограмму планирования операционной системы для выбора очередного потока.

---

<sup>2</sup> В литературе о таком подходе говорят как о *самопланировании* (*self-scheduling*), поскольку каждый процессор планирует свою работу самостоятельно, независимо от других процессоров. Однако этот термин используется в литературе и для обозначения программ, написанных на языке, обеспечивающем программисту возможность определить стратегию планирования (см., например, [82]).

- Глобальная очередь может быть организована с использованием любой из приведенных в главе 9, “Однопроцессорное планирование”, схем, включая схемы с использованием приоритетов и с учетом истории выполнения и требований обработки.

В [151] проанализированы три различные версии разделения загрузки.

1. **Первым пришел — первым обслужен (FCFS).** При поступлении в систему нового задания каждый из его потоков помещается в конец разделяемой очереди. Когда процессор становится свободным, он выбирает очередной готовый к выполнению поток и работает с ним до его завершения или блокирования.
2. **Выбор процесса с наименьшим числом потоков.** Совместно используемая очередь готовых к выполнению потоков организована как приоритетная, причем наивысший приоритет отдается потокам тех заданий, у которых осталось наименьшее число нераспределенных потоков. Задания с одинаковым приоритетом упорядочиваются в соответствии со временем поступления в систему. Как и в случае использования стратегии FCFS, поток выполняется до его завершения или блокирования.
3. **Выбор процесса с наименьшим числом потоков с применением вытеснения.** Наивысший приоритет получают задания с наименьшим количеством нераспределенных потоков. Поступившие задания с меньшим, чем у выполняющегося, количеством потоков вытесняют потоки, выполняющиеся в настоящее время.

Исследования, проведенные с помощью имитационного моделирования, показали, что по большому количеству характеристик стратегия FCFS превосходит две остальные стратегии из приведенного списка. Кроме того, авторы обнаружили, что некоторые виды бригадного планирования, обсуждающиеся в следующем подразделе, в целом превосходят разделение загрузки.

Разделение загрузки обладает рядом недостатков.

- Центральная очередь занимает область памяти, обращение к которой должно производиться с обеспечением взаимоисключений. Таким образом, это может оказаться узким местом, если несколько процессоров одновременно обращаются к очереди за порцией работы. При небольшом количестве процессоров эта проблема не является критической, но при наличии в системе десятков, а то и сотен процессоров затор в этом месте представляется совершенно реальным.
- Низкая вероятность того, что вытесняемые потоки продолжат выполнение на тех же процессорах, снижает эффективность использования локальных кешей процессоров.
- Если все потоки рассматриваются как один общий пул, маловероятно, чтобы все потоки одной программы получили одновременный доступ к процессорам. При необходимости высокой степени координации между потоками программы это приводит к серьезному снижению ее общей производительности.

Несмотря на потенциальные недостатки, это одна из наиболее часто используемых в современных многопроцессорных системах схем.

Усовершенствованная схема разделения загрузки используется в операционной системе Mach [23, 267]. Операционная система поддерживает локальную очередь для каждого процессора и совместно используемую глобальную очередь. Локальная очередь используется потоками, которые временно связаны с определенным процессором. Процессор

сначала исследует локальную очередь, тем самым отдавая приоритет связанным с ним потокам перед не связанными. Примером связанных потоков может служить использование одного или нескольких процессоров для выполнения процессов, являющихся частью операционной системы. Еще одним примером являются потоки одного приложения, которые могут быть распределены между рядом процессоров; при наличии соответствующего дополнительного программного обеспечения поддерживается обсуждающееся ниже бригадное планирование.

### Бригадное планирование

Концепция одновременного выполнения множества процессов на множестве процессоров предшествует использованию потоков. В [120] эта концепция названа *групповым планированием* (group scheduling), и там же перечислены ее основные преимущества.

- Если процессы в группе тесно связаны или координируются тем или иным образом, блокирование, вызванное синхронизацией, может быть уменьшено; кроме того, требуется меньше переключений процессов, так что общая производительность процесса растет.
- Поскольку одно решение влияет одновременно на целый ряд процессов и процессоров, это приводит к уменьшению накладных расходов, связанных с планированием.

В многопроцессорной системе Сm\* используется термин *сопланирование* (coscheduling) [87]. Сопланирование основано на концепции планирования связанных множества заданий, отдельные элементы которого достаточно малы и, следовательно, близки к концепции потоков.

Термин *бригадное планирование* (gang scheduling) применялся к одновременному планированию потоков, составляющих единый процесс [78]. Бригадное планирование полезно для приложений с синхронизацией от среднезернистой до мелкозернистой, производительность которых резко падает, если какая-то часть приложения не работает, в то время как другие готовы к выполнению. Этот метод вполне применим и к другим приложениям с параллельными вычислениями, которые не настолько чувствительны в плане зависимости общей производительности от синхронизации работы потоков. Бригадное планирование общепризнанно, и его реализации имеются во множестве многопроцессорных операционных систем.

Один очевидный путь улучшения производительности отдельного приложения при бригадном планировании заключен в минимизации переключения процессов. Предположим, что один поток процесса выполняется и достигает точки синхронизации с другим потоком того же процесса. Если этот второй поток не выполняется, но готов к выполнению, то первый поток приостанавливается в ожидании, когда некоторым другим процессором не будет выполнено переключение процессов для запуска требующегося потока. В приложении с тесно связанными потоками такие переключения существенно снижают производительность. Одновременное планирование сотрудничающих потоков может также сохранить время на выделение ресурсов. Например, ряд потоков при бригадном планировании может обращаться к файлу без дополнительных расходов на блокирование в процессе операций чтения/записи или позиционирования в файле.

Использование бригадного планирования выдвигает требования по распределению процессоров. Предположим, что у нас имеется  $N$  процессоров и  $M$  приложений, каждое из которых состоит из  $N$  или меньшего числа потоков. Тогда каждое приложение при использовании квантования времени может получить  $1/M$  доступного времени  $N$  процес-

солов. В [77] замечено, что такая стратегия может оказаться неэффективной. Рассмотрим пример, в котором имеются два приложения, одно — с четырьмя потоками, а второе — с одним. Использование равномерного распределения времени приводит к потере 37,5% вычислительного ресурса, поскольку при выполнении однопоточного приложения три процессора простаивают (см. рис. 10.2). При наличии ряда однопоточных приложений они могут распределяться совместно для увеличения загрузки процессоров. Если это невозможно, вторым решением может стать применение взвешенного планирования с учетом количества потоков каждого процесса. В этом случае приложение с четырьмя потоками из нашего примера может занять 80% (4/5) всего времени, а однопоточное — только 20% (1/5), что приведет к снижению потерь вычислительного ресурса до 15%.

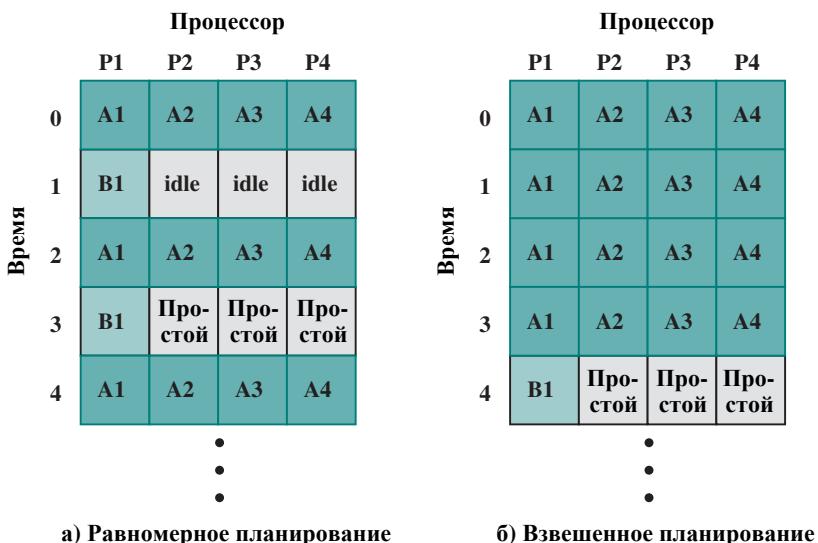


Рис. 10.2. Бригадное планирование

### Назначение процессоров

Экстремальной формой бригадного планирования является предложенное в [260] выделение приложению групп процессоров на все время работы данного приложения. Таким образом, когда приложение спланировано, каждый из его потоков назначается определенному процессору, на котором он и выполняется до завершения работы данного приложения.

Такой подход представляется очень неэффективным в плане процессорного времени. Если поток приложения блокируется операцией ввода-вывода или необходимостью синхронизации с другим потоком, то процессор этого потока простаивает: многозадачность процессора в этом методе отсутствует. В защиту данной стратегии можно привести два соображения.

1. В высокопараллельной системе с десятками или сотнями процессоров, каждый из которых представляет собой лишь малую часть стоимости системы, загруженность процессора не так важна, как общая эффективность или производительность.
2. Полное устранение переключений процессов во время работы программы должно существенно повысить ее скорость работы.

В работах [260] и [271] подтверждено второе высказывание. В табл. 10.2 показаны результаты одного эксперимента из [260]. Авторы запускали два приложения (умножение матриц и быстрое преобразование Фурье) в системе с 16 процессорами. Каждое приложение разбивало свою задачу на ряд заданий, которые отображались на потоки, выполняющие данное приложение. Программы были написаны таким образом, что позволяли варьировать количество используемых потоков. По существу, количество заданий определялось самим приложением. Если потоков оказывалось меньше, чем заданий, последние ставились в очередь и изымались из нее потоками, завершившими свои задания. Понятно, что далеко не все приложения можно структурировать подобным образом, но многие вычислительные задачи и ряд других приложений вполне могут быть преобразованы таким образом.

**Таблица 10.2. Ускорение приложения как функция от количества потоков**

Количество потоков на приложение	Умножение матриц	Быстрое преобразование Фурье
1	1	1
2	1,8	1,8
4	3,8	3,8
8	6,5	6,1
12	5,2	5,1
16	3,9	3,8
20	3,3	3
24	2,8	2,4

В табл. 10.2 показано ускорение работы приложений как функция от количества используемых потоков (которое в каждом приложении варьировалось от 1 до 24). Например, если оба приложения с 24 потоками каждое запускались одновременно, то ускорение их работы по сравнению с однопоточными приложениями составляло 2,8 для быстрого преобразования Фурье и 2,4 — для умножения матриц. Из таблицы видно, что производительность обоих приложений снижается при количестве потоков, превышающем 8, когда общее количество потоков превышает количество процессоров. Чем больше потоков, тем хуже производительность из-за большей частоты вытеснения потоков и дополнительных операций планирования. Снижение эффективности при этом определяется целым рядом факторов — от времени, затрачиваемого на ожидание, когда приостановленный поток покинет критический раздел, до снижения эффективности кешей и времени, затраченного на переключение процессоров.

Авторы приходят к выводу, что эффективная стратегия состоит в ограничении общего количества активных потоков в системе количеством процессоров. Если большинство приложений либо однопоточные, либо могут использовать очереди заданий, то такая стратегия обеспечивает эффективное использование процессоров.

Как стратегия назначения процессоров, так и бригадное планирование решают задачу планирования путем распределения процессоров. Однако можно заметить, что

распределение процессоров в многопроцессорной системе гораздо больше похоже на задачу распределения памяти в однопроцессорной системе, чем на задачу планирования в однопроцессорной системе. Вопрос о том, сколько процессоров могут быть привлечены к выполнению приложения в конкретный момент времени, аналогичен вопросу о том, сколько кадров страниц может быть предоставлено данному процессу в некоторый момент времени. В [87] предложен термин *рабочее множество активности* (activity working set), аналогичный рабочему множеству виртуальной памяти, как минимальное количество единиц активности (потоков), которое должно быть одновременно распределено между процессорами для приемлемого выполнения приложения. Как и в схемах управления памятью, ошибки в планировании элементов рабочего множества активности могут привести к снижению пропускной способности процессора. Это происходит в тех случаях, когда запускаются потоки, обслуживание которых вызывает приостановку других потоков, услуги которых понадобятся в ближайшее время. Аналогично фрагментация процессоров означает ситуацию, когда ряд процессоров оказываются распределенными, а оставшихся процессоров либо недостаточно, либо они не соответствуют требованиям ожидающего приложения. Бригадное планирование и распределение процессоров предназначено для устранения таких проблем.

## Динамическое планирование

В ряде приложений количество потоков динамически изменяется в процессе их работы, что позволяет операционной системе изменять загрузку для повышения степени использования процессоров.

В [271] предложен подход, в котором в планирование вовлекаются как операционная система, так и само приложение. Операционная система отвечает за распределение процессоров между заданиями. Каждое задание использует выделенные ей в настоящий момент процессоры для выполнения некоторого подмножества доступных в настоящий момент для выполнения заданий, распределяя их по потокам. Принятие решения о выделении такого подмножества для работы, как и о том, какой поток должен быть приостановлен при вытеснении процесса, выполняют отдельные приложения (возможно, посредством множества подпрограмм библиотеки времени выполнения). Такой подход применим не для всех приложений, однако возможна ситуация, когда одни приложения будут работать как однопоточные, в то время как другие приложения изначально будут создаваться с использованием этой возможности операционной системы.

При таком подходе ответственность операционной системы за планирование ограничена распределением процессоров и выполняется в соответствии с изложенными далее принципами. Когда задание требует одного или нескольких процессоров (не важно, в момент ли поступления в систему или в связи с изменившимися требованиями), происходит следующее.

1. Если в системе имеются простаивающие процессоры, они используются для удовлетворения этого запроса.
2. В противном случае, если запрос подается вновь поступившим в систему заданием, ему выделяется единственный процессор, забираемый у задания, которое в настоящий момент использует более одного процессора.
3. Если некоторая часть запроса не может быть удовлетворена, он остается невыполненным до тех пор, пока либо не появится свободный процессор, либо задание не снимет свой запрос (например, ему больше не требуется лишний процессор).

При освобождении одного или нескольких процессоров происходит следующее.

4. Сканируется текущая очередь неудовлетворенных запросов на процессоры. Каждому заданию в списке, до сих пор не имеющему ни одного процессора (т.е. ожидающему поступления в систему), назначается по одному процессору. После этого, при наличии свободных процессоров, список сканируется вновь, распределяя оставшиеся процессы на основе стратегии FCFS.

Анализ работ [271] и [162] дает основания полагать, что для приложений, которые могут использовать преимущества динамического планирования, этот подход предпочтительнее бригадного планирования или назначения выделенных процессоров. Однако накладные расходы при описываемом подходе могут свести на нет получаемые преимущества, и для доказательства превосходства динамического планирования требуется опыт работы реальных операционных систем.

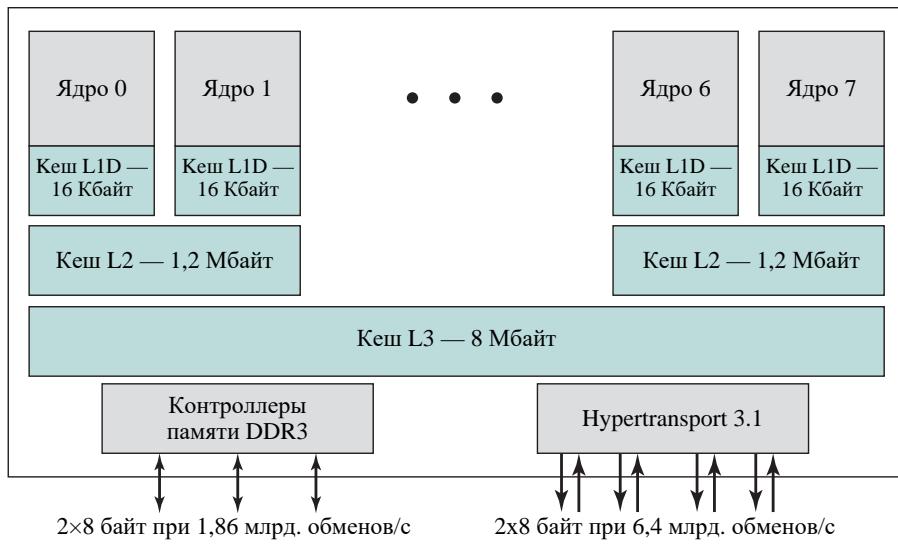
## Планирование потоков в многоядерных системах

Наиболее широко используемые современные операционные системы, такие как Windows и Linux, по существу рассматривают планирование в многоядерных системах таким же образом, как и в многопроцессорных системах. Такие планировщики, как правило, сосредоточиваются на поддержании занятости процессоров путем балансировки нагрузки, так что готовые к работе потоки равномерно распределяются между процессорами. Однако эта стратегия вряд ли даст желаемую выгоду с точки зрения производительности в многоядерной архитектуре.

По мере увеличения количества ядер в процессорах необходимость минимизировать доступ к памяти вне процессора превалирует над стремлением максимизировать использование процессора. Традиционное (и до сих пор основное) средство сведения к минимуму обращений к памяти вне процессора представляет собой использование преимуществ локальности с помощью кешей. Этот подход осложняется некоторыми архитектурами кешей, используемыми с многоядерными чипами, в частности когда кеш совместно используется некоторыми, но не всеми, ядрами. Хорошим примером является чип AMD Bulldozer, используемый в системе Operton FX-8000 (рис. 10.3). В этой архитектуре каждое ядро имеет выделенный кеш L1; каждая пара ядер совместно использует кеш L2; а все ядра совместно используют кеш L3. Сравните это с Intel Core i7-5960 X (см. рис. 1.20), в котором кеши L1 и L2 выделены для каждого ядра.

Когда некоторые, но не все, ядра совместно используют кеш, способ назначения потоков ядрам во время планирования значительно влияет на производительность. Давайте назовем два ядра, которые совместно используют один и тот же кеш L2, смежными или соседними. Таким образом, на рис. 10.3 ядра 0 и 1 являются смежными, а ядра 1 и 2 — не смежными. В идеале, если два потока совместно используют ресурсы памяти, то они должны быть назначены соседним ядрам для повышения локальности; если же они не используют ресурсы памяти, они могут быть назначены несмежным ядрам для достижения баланса нагрузки.

Фактически есть два различных аспекта совместного использования кешей, которые следует принимать во внимание: кооперативное совместное использование ресурсов и конкуренция за ресурсы. При кооперативном совместном использовании ресурсов несколько потоков обращаются к одним и тем же местоположениям основной памяти. Примерами являются многопоточные приложения и взаимодействие потоков в стиле “производитель–потребитель”.



**Рис. 10.3. Архитектура AMD Bulldozer**

В обоих случаях к данным, вносимым в кеш в одном потоке данных, должны обращаться сотрудничающие потоки, и желательно планирование выполнения сотрудничающих потоков на смежных ядрах.

Другой случай происходит, когда потоки, работающие на смежных ядрах, конкурируют за кеш-память. Какой бы метод не использовался для замены кеша, например такой, как последние использовавшиеся записи (least-recently-used — LRU), чем больше кеш-памяти динамически выделяется одному потоку, тем меньше кеш-памяти получает конкурирующий поток и, таким образом, страдает производительность. Цель планирования в случае конкурирующих потоков заключается в выделении потоков ядрам таким образом, чтобы максимально повысить эффективность совместно используемой кеш-памяти и, таким образом, свести к минимуму необходимость доступа к памяти за пределами процессора. Разработка алгоритмов для этой цели является активно исследуемой тематикой и выходит за рамки данной книги. Если вас интересует эта тема, обратитесь к обзору [273].

## 10.2. ПЛАНИРОВАНИЕ РЕАЛЬНОГО ВРЕМЕНИ

### Введение

Вычисления в реальном времени становятся все более и более важной отраслью знаний. Операционная система, в частности — планировщик, представляет собой, вероятно, наиболее важный компонент системы реального времени. Примерами приложений систем реального времени могут служить управление лабораторными экспериментами, управление работой цехов, роботы, системы управления воздушным движением, телекоммуникации и многое другое. Следующее поколение систем будет включать автономные системы передвижения, роботов с эластичными суставами, интеллектуальные системы управления, управление космическими и подводными станциями.

Вычисления реального времени — тип вычислений, в которых корректность системы зависит не только от логического результата вычислений, но и от времени получения этого результата. Мы можем определить систему реального времени путем определения того, что же собой представляет процесс, или задание реального времени.<sup>3</sup> Вообще говоря, в системах реального времени часть заданий являются заданиями реального времени, у каждого из которых своя степень срочности. Такие задания пытаются управлять событиями или реагировать на события, происходящие во внешнем мире. Поскольку эти события происходят в “реальном времени”, то и задания, связанные с ними, не должны отставать от реальных событий. Следовательно, с каждым заданием можно связать крайний срок, определяющий время начала (или завершения) задания. Такие задания можно охарактеризовать как жесткие и мягкие. **Жесткие задания реального времени** (hard real-time task) представляют собой задания, которые должны соответствовать этим предельным срокам; в противном случае неизбежны нежелательные повреждения или фатальные сбои системы. **Мягкие задания реального времени** (soft real-time task) также имеют предельные сроки, но их выполнение — скорее пожелание, чем обязанность; если даже такое задание не уложилось в отведенное ему время, его все равно имеет смысл продолжать планировать и довести до завершения.

Еще одной характеристикой заданий реального времени является их периодичность. **Непериодические задания** имеют предельные сроки начала или конца работы (или и те, и другие); в **периодических заданиях** требования могут быть указаны в виде “один раз за период  $T$ ” или “точно через  $T$  единиц времени”.

## Характеристики операционных систем реального времени

Операционные системы реального времени должны удовлетворять следующим пятью общим требованиям [174].

1. Детерминизм
2. Чувствительность
3. Управление со стороны пользователя
4. Надежность
5. Восстановление после сбоев

Операционная система **детерминирована** (deterministic), если она выполняет операции в фиксированное, предопределенное время или в пределах предопределенных интервалов времени. При конкуренции процессов за обладание ресурсами и временем процессора система не является полностью детерминированной. В операционных системах реального времени запросы процесса на обслуживание диктуются внешними событиями

<sup>3</sup> Как обычно, с терминологией возникают определенные проблемы, поскольку в литературе разные термины используются в разных значениях. Обычной практикой является циклическая природа процесса, работающего с временными ограничениями, т.е. процесс работает длительное время, выполняя при этом некоторые повторяющиеся функции в ответ на события реального времени. В этом разделе об индивидуальной функции мы будем говорить как о задании. Следовательно, процесс может рассматриваться как продвижение по последовательности заданий. В любой конкретный момент процесс занимается одним заданием и планироваться должны как процессы, так и задания.

и синхронизацией. Насколько детерминирована система способна удовлетворять запросы, зависит, в первую очередь, от скорости, с которой она способна реагировать на прерывания, а также от того, обладает ли система достаточной пропускной способностью для обработки всех запросов за требуемое время.

Мерой способности операционной системы к детерминированному функционированию служит длительность максимальной задержки между поступлением в систему прерывания от высокоприоритетного устройства и началом его обработки. В операционных системах, не являющихся системами реального времени, эта задержка может составлять от десятков до сотен миллисекунд, в то время как в операционной системе реального времени такая задержка должна иметь верхнюю границу от нескольких микросекунд до миллисекунды.

Сходной с детерминизмом характеристикой является **чувствительность** (responsiveness) системы. Детерминизм сосредоточен на времени задержки перед распознаванием прерывания; чувствительность же рассматривает вопрос о том, сколько времени требуется операционной системе для обслуживания прерывания после распознавания. Чувствительность включает следующее.

1. Количество времени, требующегося для начальной обработки прерывания и начала выполнения подпрограммы обработки прерывания (interrupt service routine — ISR). Если выполнение ISR требует переключения процессов, то задержка оказывается большей, чем при ISR, которая может выполняться в пределах контекста текущего процесса.
2. Количество времени, требующегося для выполнения ISR. В общем случае это время зависит от используемой аппаратной платформы.
3. Влияние вложенных прерываний. Если ISR может быть прервана поступлением нового прерывания, то обслуживание текущего прерывания будет отложено.

Детерминизм и чувствительность в совокупности образуют время отклика на внешнее событие. Требования ко времени отклика являются критическими для систем реального времени, которые должны отвечать временным требованиям со стороны устройств, внешних потоков данных и т.п.

**Управление со стороны пользователя** в системах реального времени обычно значительно шире, чем в обычной операционной системе. В типичной операционной системе, не являющейся системой реального времени, пользователь либо не в состоянии управлять функцией планирования операционной системы, либо может осуществлять только самое общее руководство типа группирования пользователей по нескольким классам приоритетов; однако в системах реального времени важной составляющей является обеспечение возможности тонкой настройки приоритетов заданий. Пользователь должен иметь возможность разделять задания на жесткие и мягкие и определять относительные приоритеты в пределах каждого класса. Кроме того, системы реального времени позволяют пользователю определять и такие характеристики, как использование страничной организации памяти или свопинг процессов, а также определять, какие процессы должны постоянно находиться в основной памяти, какой алгоритм дисковых операций должен использоваться, каковы права процессов из разных групп и многое другое.

**Надежность** в системах реального времени — очень важный вопрос. Случайная ошибка в обычной системе может быть в худшем случае обработана просто посредством перезагрузки системы; сбой одного из процессоров в обычной многопроцессорной системе

приведет к снижению уровня обслуживания до замены или починки этого процессора. Но система реального времени работает, как и следует из ее названия, с событиями в реальном времени, и потеря производительности может привести к катастрофическим последствиям — от финансовых потерь до потерь оборудования и даже человеческих жизней.

Система реального времени должна быть разработана таким образом, чтобы уметь реагировать на ошибки разного типа. **Восстановление после сбоев** — это характеристика системы, которая описывает способность системы сохранить максимальную функциональность и не потерять данные при сбое. Например, типичные традиционные системы UNIX при обнаружении повреждения данных ядра выводят соответствующее сообщение на системную консоль, сбрасывают дамп памяти на диск для последующего анализа и прекращают работу системы. Система же реального времени будет пытаться либо исправить ситуацию полностью, либо минимизировать ее влияние на продолжающуюся работу системы. Обычно система информирует пользователя или пользовательский процесс о необходимости корректирующих действий и продолжает работу, возможно, со сниженным уровнем обслуживания. Если необходимо выключение системы, она пытается сохранить согласованность файлов и данных.

Важным аспектом восстановления после сбоев является стабильность системы. Система стабильна, если в случаях, когда невозможно выдержать предельные сроки всех задач, она выдерживает предельные сроки для наиболее критических, высокоприоритетных задач (даже если не всегда удается выдержать условия работы над некоторыми менее критическими задачами).

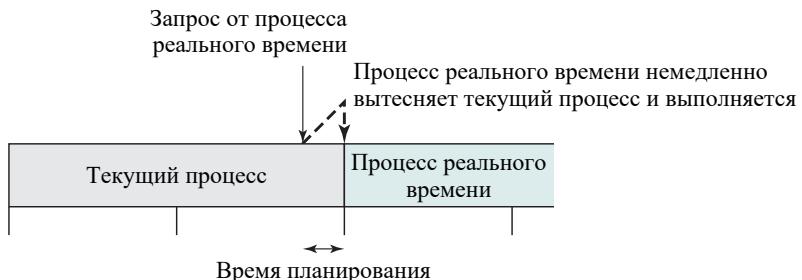
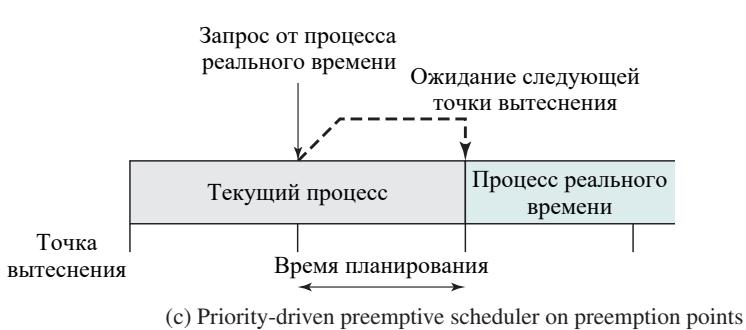
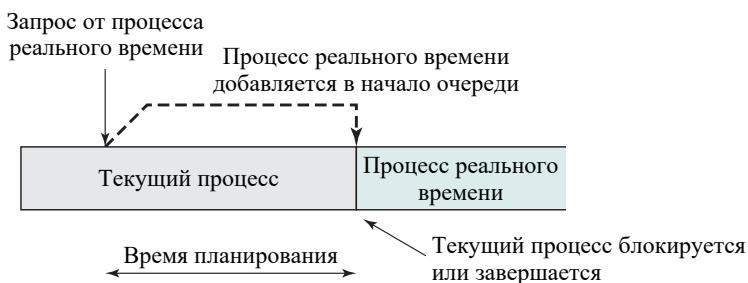
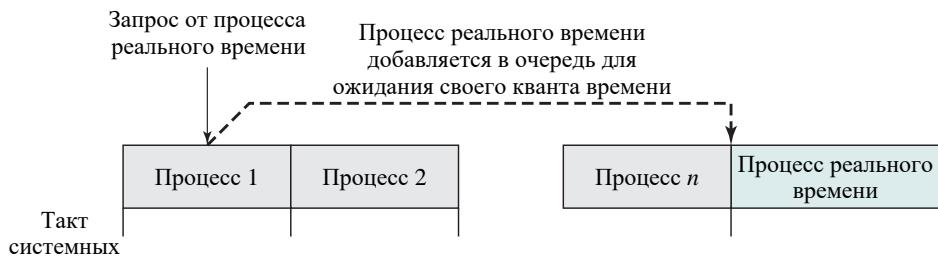
Для того чтобы соответствовать приведенным требованиям, современная система реального времени обычно включает следующее.

- Более строгое использование приоритетов, чем в обычной операционной системе, с вытесняющим планированием, разработано таким образом, чтобы отвечать требованиям работы в реальном времени.
- Задержка прерываний (время между моментом генерации прерывания устройством и моментом завершения обслуживания устройства) ограничена и относительно невелика.
- Более точные и предсказуемые временные характеристики, чем у операционной системы общего назначения.

Сердцем системы реального времени является краткосрочный планировщик заданий. При его разработке беспристрастность и минимизация среднего времени отклика первостепенными задачами не являются; зато на первый план выходит завершение (или начало) работы всех жестких заданий реального времени в рамках их предельного времени, и по возможности максимального количества мягких заданий.

Большинство современных операционных систем реального времени не работают с предельным временем непосредственно. Вместо этого они разработаны таким образом, чтобы обеспечить их максимальную чувствительность, с тем чтобы при подходе с предельным временем задание немедленно поступало на выполнение. С этой точки зрения приложения реального времени обычно требуют, чтобы детерминированное время отклика находилось в интервале от долей миллисекунд до нескольких миллисекунд при широком диапазоне условий. Некоторые специализированные приложения, например в моделировании военных самолетов, часто налагают ограничение от 10 до 100 мкс.

На рис. 10.4 продемонстрирован ряд возможных вариантов планирования реального времени.



г) Планирование с немедленным вытеснением

Рис. 10.4. Планирование процессов реального времени

В вытесняющем планировщике с использованием круговой стратегии задание реального времени добавляется к очереди готовых к выполнению заданий, ожидающих наступления следующего кванта времени (рис. 10.4, а). В этом случае время планирования в целом неприемлемо для приложений реального времени. В невытесняющем планировщике мы можем использовать механизм приоритетного планирования, дающий заданиям реального времени наивысший приоритет. В этом случае готовые задания реального времени будут выполняться сразу же после блокирования текущего процесса либо его выполнения до конца (рис. 10.4, б). Такая методика может привести к большим, до нескольких секунд, задержкам при выполнении в критический момент медленного низкоприоритетного задания, так что этот подход также неприменим. Более приемлемый подход состоит в комбинировании приоритетов с прерываниями таймера. При этом точки вытеснения равноудалены одна от другой. При достижении точки вытеснения выполняющееся в настоящий момент задание вытесняется, если в наличии имеется более высокоприоритетное задание в состоянии ожидания; таким образом, вытеснение заданий в этом случае оказывается частью ядра операционной системы. Здесь задержки могут быть порядка нескольких миллисекунд (рис. 10.4, в). Для ряда приложений реального времени задержки такого уровня вполне адекватны, но могут оказаться неприемлемыми для более требовательных приложений. В таком случае можно применить подход, иногда именуемый немедленным вытеснением. Он заключается в том, что операционная система отвечает на прерывание практически немедленно, если только она не выполняет код критического раздела, который не может быть прерван. Задержки планирования при этом снижаются до 100 мкс и менее.

## Планирование реального времени

Планирование реального времени является одной из областей, в которых ведутся активные исследовательские работы. В этом подразделе мы приведем краткий обзор различных подходов к проблеме планирования реального времени и рассмотрим два распространенных класса алгоритмов планирования.

В обзоре алгоритмов планирования реального времени [196] замечено, что различные алгоритмы планирования зависят от того, 1) выполняет ли система анализ планируемости, и если выполняет, то 2) как именно, статически или динамически, и 3) к чему он приводит — к непосредственному выполнению функции планирования или же к построению расписания, согласно которому в процессе работы осуществляется диспетчеризация заданий. На основе этого автор определяет следующие классы алгоритмов.

- **Статическое планирование с использованием таблиц.** При этом выполняется статический анализ осуществимости планирования; результатом анализа является план, который в процессе работы системы определяет, когда должно начаться выполнение заданий.
- **Статическое вытесняющее планирование на основе приоритетов.** В этом случае также выполняется статический анализ, но расписание не создается. Вместо этого на основе проведенного анализа заданиям назначаются приоритеты, с тем чтобы далее можно было использовать традиционный вытесняющий планировщик, работающий с учетом приоритетов заданий.
- **Динамическое планирование на основе расписания.** Осуществимость планирования определяется не статически, а динамически, в процессе выполнения.

Поступающее в систему задание принимается только в том случае, если определена возможность его выполнения с учетом всех временных требований. Одним из результатов анализа является расписание, используемое для принятия решения о диспетчеризации задания.

- **Динамическое планирование наилучшего результата.** При этом анализ осуществимости планирования не выполняется; система пытается удовлетворить все предельные сроки и снимает те выполняющиеся процессы, предельные сроки которых нарушены.

**Статическое планирование с использованием таблиц** применимо для периодических заданий. Входной информацией для анализа являются время поступления заданий в систему, время выполнения, предельные сроки выполнения и относительный приоритет каждого задания. Планировщик пытается разработать такой план работы, который удовлетворял бы всем временным требованиям заданий. Такой подход является предсказуемым, но абсолютно не гибким, поскольку любое изменение требований любого задания приводит к необходимости пересмотра всего расписания. Типичными представителями этой категории алгоритмов планирования являются планирование наиболее раннего предельного срока и другие, рассматриваемые далее, алгоритмы планирования периодических заданий.

**Статическое вытесняющее планирование на основе приоритетов** использует тот же вытесняющий механизм планирования с приоритетами, что и большинство обычных многозадачных операционных систем. В таких системах для определения приоритетов могут использоваться самые различные факторы. Например, в системе с разделением времени приоритет процесса изменяется в зависимости от того, на что ориентирован этот процесс — на вычисления или на операции ввода-вывода. В системах реального времени назначение приоритетов связано с временными ограничениями каждого из заданий. Примером такого подхода может служить частотно-монотонное планирование, рассматриваемое ниже в данной главе, которое назначает заданиям статические приоритеты на основе информации об их периодах.

В случае использования **динамического планирования на основе расписания** после поступления задания в систему (но до начала его выполнения) предпринимается попытка создать расписание, которое содержит как все имеющиеся в системе задания, так и вновь поступившее. Если удается создать такое расписание, что при выполнении удовлетворяются временные ограничения как нового задания, так и уже имеющихся в системе заданий, оно принимается планировщиком.

**Динамическое планирование наилучшего результата** используется во многих современных коммерческих системах реального времени. При поступлении нового задания система назначает ему приоритет на основе характеристик этого задания. При этом подходе обычно используется планирование, учитывающее предельные сроки, — наподобие планирования наиболее раннего предельного срока. Как правило, поступающие задания непериодические, а потому статический анализ планирования неприменим. При таком типе планирования мы не знаем, будут ли удовлетворены временные ограничения задания до тех пор, пока задание не будет полностью выполнено (или пока не будут нарушены временные ограничения). Именно это и является основным недостатком данной схемы; преимущество же динамического планирования наилучшего результата — в простоте реализации.

## Планирование с предельными сроками

Большинство современных операционных систем реального времени разработаны таким образом, чтобы задания реального времени начинали работу как можно быстрее, а следовательно, особое внимание уделяют вопросам быстрой обработки прерываний и диспетчеризации заданий. На самом деле это не очень практичный критерий оценки операционной системы реального времени. Обычно приложения реального времени не так интересует абсолютная скорость выполнения заданий, как их своевременное завершение (или начало) — не слишком рано и не слишком поздно, несмотря на любые требования к ресурсам и могущие возникать конфликты, перегрузку процессора или аппаратные либо программные сбои. Отсюда следует, что система приоритетов — довольно слабый инструмент, который не может обеспечить гарантированное выполнение заданий в необходимые сроки.

Имеется ряд интересных подходов к планированию, более мощных и лучше приспособленных для работы с заданиями реального времени. Все они основаны на дополнительной информации о заданиях, которая может включать следующее.

- **Время готовности.** Время, когда задание становится доступным для выполнения. В повторяющемся или периодическом задании время готовности представляет собой последовательность заранее известных времен. В непериодическом задании это время может быть известно заранее, но может быть и так, что операционная система узнает его только в тот момент, когда задание станет действительно готовым к выполнению.
- **Предельное время начала выполнения.** Время, когда должно начаться выполнение задания.
- **Предельное время завершения выполнения.** Время, когда задание должно быть полностью завершено. Обычно задания реального времени имеют ограничение — либо по предельному времени начала выполнения, либо по предельному времени завершения выполнения, но не оба ограничения одновременно.
- **Время выполнения.** Время, необходимое заданию для полного выполнения. В некоторых случаях это время известно, а в некоторых система сама оценивает взвешенное среднее значение. В других системах планирования эта величина не используется.
- **Требования к ресурсам.** Множество ресурсов (отличных от процессора), требующихся заданию при его выполнении.
- **Приоритет.** Мера относительной важности задания. Жесткие задания реального времени имеют “абсолютный” приоритет, приводя к сбоям системы при нарушении временных ограничений этих заданий. Если система продолжает работу несмотря ни на что, то как жесткие, так и мягкие задания получают относительные приоритеты, использующиеся в качестве указаний планировщику.
- **Структура подзадач.** Задача может быть разбита на обязательные и необязательные подзадачи. Жесткие предельные сроки при таком разделении имеют только обязательные подзадачи.

Можно показать, что для заданной стратегии вытеснения и использования либо предельного времени начала выполнения, либо предельного времени завершения при-

менение планирования, выбирающего для выполнения задание с наиболее ранним предельным временем, минимизирует долю заданий с нарушенными временными ограничениями [33].

Это заключение справедливо как для однопроцессорной, так и для многопроцессорной конфигураций. Еще одним критическим вопросом является вытеснение. Если определяется предельное время начала работы, то имеет смысл применение невытесняющего планирования. В этом случае желательно, чтобы задания реального времени после завершения обязательной или критической части самостоятельно блокировались, позволяя выполняться другим заданиям реального времени с предельным временем начала работы (см. рис. 10.4, б). Для системы с предельным временем завершения более подходит вытесняющая стратегия (рис. 10.4, в, г). Например, если выполняется задание X, а задание Y находится в состоянии готовности, то может возникнуть ситуация, когда единственным способом удовлетворения ограничений обоих заданий будет вытеснение X, выполнение Y до завершения и возобновление выполнения X.

В качестве примера планирования периодических заданий с предельным временем завершения рассмотрим систему, которая собирает и обрабатывает данные от двух датчиков, А и В. Сроки сбора данных от датчика А — каждые 20 мс, датчика В — каждые 50 мс. Процесс снятия данных, включая накладные расходы операционной системы, занимает для датчика А 10 мс, а для датчика В — 25 мс. В табл. 10.3 приведен профиль выполнения этих двух заданий. На рис. 10.5 сравниваются все три метода планирования с использованием профилей выполнения из табл. 10.3. Первая строка на рис. 10.6 повторяет информацию из табл. 10.3; остальные строки иллюстрируют три метода планирования.

Таблица 10.3. Профиль выполнения двух периодических заданий

Процесс	Время поступления	Время выполнения	Предельное время окончания
A(1)	0	10	20
A(2)	20	10	40
A(3)	40	10	60
A(4)	60	10	80
A(5)	80	10	100
•	•	•	•
•	•	•	•
•	•	•	•
B(1)	0	25	50
B(2)	50	25	100
•	•	•	•
•	•	•	•
•	•	•	•

Компьютер может принимать решение о планировании каждые 10 мс<sup>4</sup>. Предположим, что при этих условиях мы используем схему планирования с приоритетами. Результат показан на первых двух временных диаграммах на рис. 10.5.

Предельные сроки выполнения:

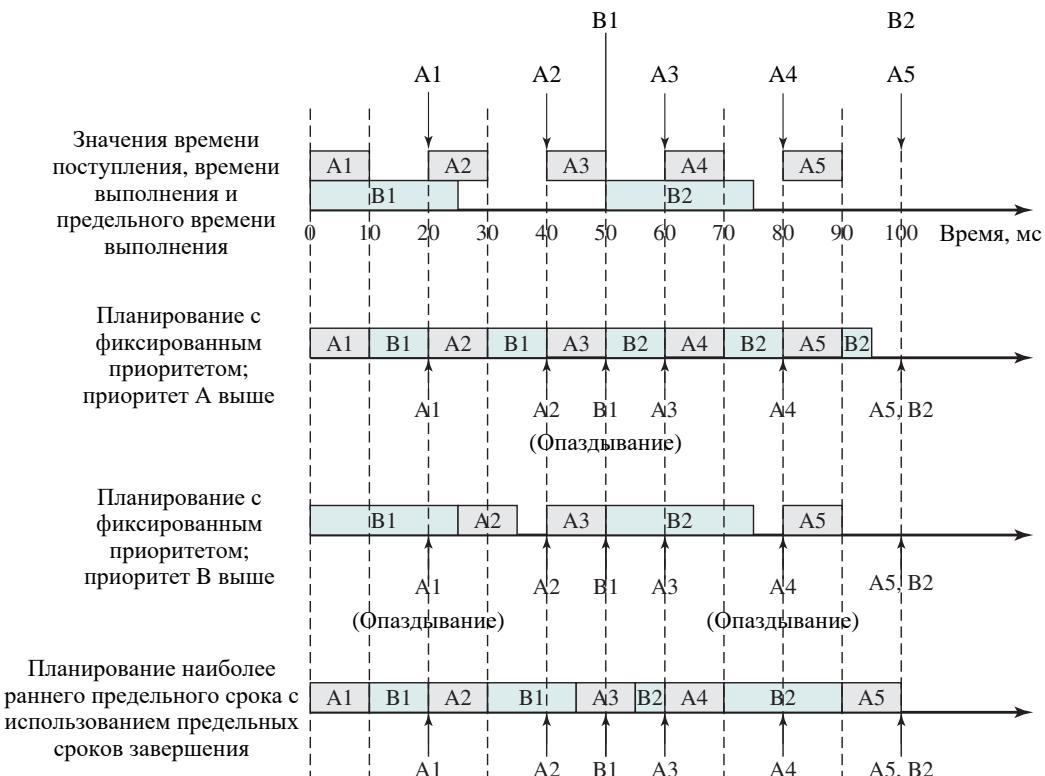


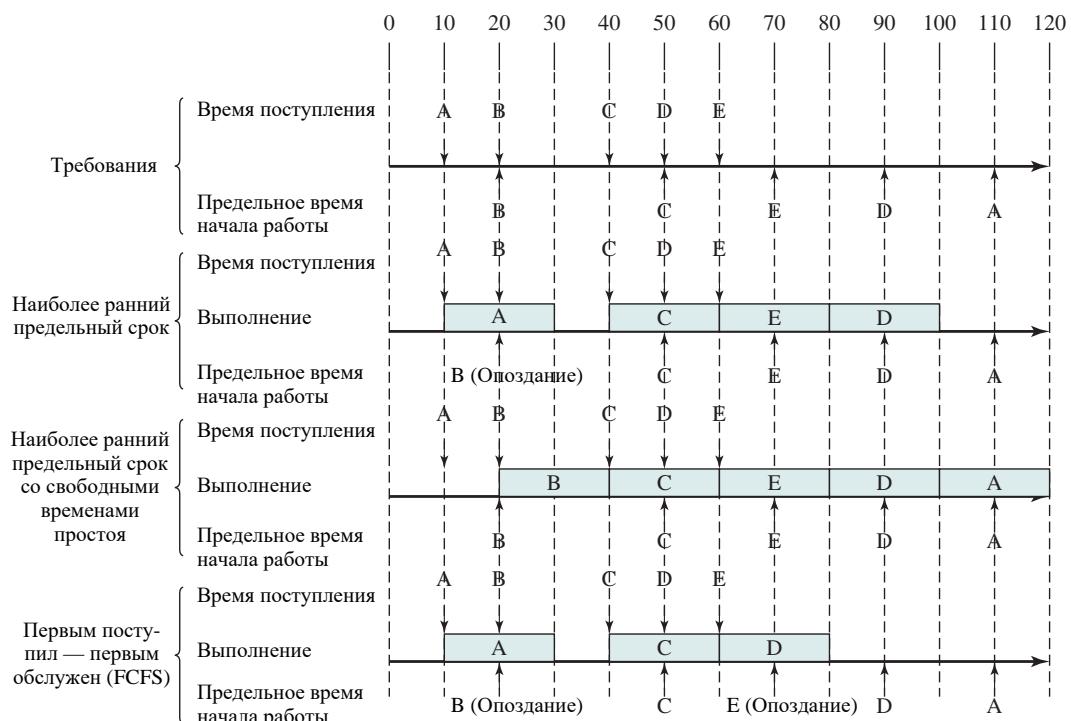
Рис. 10.5. Планирование периодических заданий реального времени с предельными сроками завершения (на основании табл. 10.3)

Если А имеет более высокий приоритет, то первый экземпляр задания В получит только 20 мс процессорного времени в двух смежных интервалах по 10 мс; после этого будет достигнуто предельное время его выполнения, и задание выполнено не будет. Если задание В получит более высокий приоритет, то выполниться в срок не сможет задание А. Последняя временная диаграмма показывает применение в данной ситуации планирования с наиболее ранним предельным сроком. В момент времени  $t = 0$  поступают задания А1 и В1. Поскольку предельный срок А1 наступает раньше предельного срока В1, сначала выполняется задание А1. После его завершения начинается выполнение В1. В момент времени  $t = 20$  в систему поступает задание А2, предельный срок которого наступает раньше предельного срока В1. Соответственно, задание В1 прерывается, и начинается выполнение задания А2. Выполнение В1 продолжается начиная с момента  $t = 30$ . В момент  $t = 40$  в систему поступает задание А3, предельный срок которого, однако, более поздний, чем

<sup>4</sup> Это значение не обязательно должно быть выровнено по границе в 10 мс, если со времени принятия последнего решения о планировании прошло более 10 мс.

пределный срок задания В1, так что задание В1 продолжает выполнение до завершения в момент времени  $t = 45$ . В этот момент начинается выполнение задания А3, завершающееся к моменту  $t = 55$ . В этом примере все требования к системе удовлетворяются там, где в каждой точке вытеснения планировщик дает высший приоритет тому заданию, предельный срок которого наступает раньше. Здесь мы смогли использовать статическое планирование на основе таблиц, поскольку задания периодичны и предсказуемы.

Рассмотрим теперь схему для работы с непериодическими заданиями с предельными сроками начала работы. В верхней части рис. 10.6 приведены значения времени поступления и предельные сроки начала работы для примера, состоящего из пяти заданий, время выполнения каждого из которых составляет 20 мс. Профиль выполнения этих заданий приведен в табл. 10.4.



**Рис. 10.6.** Планирование непериодических заданий реального времени с предельными сроками начала работы

**Таблица 10.4. Профиль выполнения пяти непериодических заданий**

Процесс	Время поступления	Время выполнения	Предельное время начала работы
A	10	20	110
B	20	20	20
C	40	20	50
D	50	20	90
E	60	20	70

Простейшая схема планирования состоит в запуске задания с наиболее ранним предельным сроком и в его выполнении до полного завершения. Обратите внимание на рис. 10.6, где при использовании такого подхода несмотря на то, что задание B требует немедленного выполнения, оно отклоняется системой. В этом заключается основной риск работы с непериодическими заданиями, в особенности с предельным временем начала выполнения. Если предельное время заданий известно до того, как задания становятся готовыми к выполнению, можно усовершенствовать эту схему, повысив тем самым производительность системы. Такая стратегия, именуемая стратегией наиболее раннего предельного срока со свободным временем простоя, работает следующим образом. Планировщик всегда запускает подходящее задание с наиболее ранним предельным сроком, которое выполняется до полного завершения. Подходящее задание может оказаться не готовым, и это может привести к тому, что несмотря на наличие готовых к выполнению заданий процессор простояивает. Обратите внимание, как в нашем примере система воздерживается от выполнения задания A, несмотря на то что это единственное готовое к выполнению задание. В результате, хотя процессор и не используется с максимальной эффективностью, все требования к предельным срокам в нашем примере при такой схеме удовлетворены. И наконец, для сравнения на рис. 10.6 приведен результат использования стратегии “первым поступил — первым обслужен”. Как видно на рисунке, в этом случае отклоненными оказываются два задания — B и E.

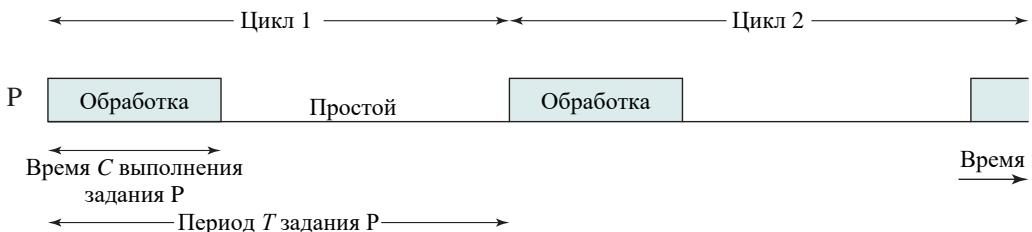
## Частотно-монотонное планирование

Одним из многообещающих методов разрешения конфликтов многозадачного планирования для периодических задач является частотно-монотонное планирование (rate monotonic scheduling — RMS) [157, 30, 222]. Схема RMS назначает приоритеты заданиям на основе их периодов.

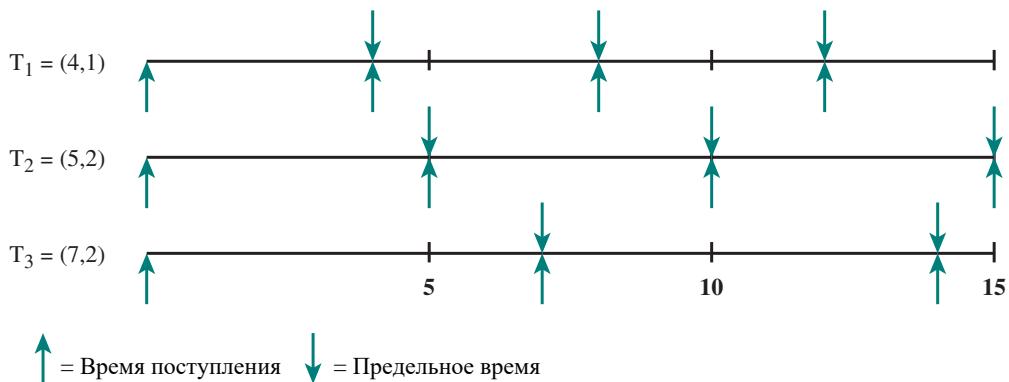
При использовании RMS заданием с наивысшим приоритетом является задание с наименьшим периодом; заданием со вторым по значению приоритетом является задание со вторым по краткости периодом и т.д. Когда для выполнения готово более одного задания, первым обслуживается задание с кратчайшим периодом. Если изобразить приоритет заданий как функцию их частоты, результатом оказывается монотонно растущая функция — откуда и название “частотно-монотонное планирование”.

На рис. 10.7 проиллюстрированы параметры периодических заданий. Период задания  $T$  представляет собой интервал времени между поступлениями двух последовательных экземпляров заданий одного типа. Частота заданий (измеряемая в герцах (Гц)) представляет собой величину, обратную периоду (в секундах). Например, задание с периодом 50 мс имеет частоту 20 Гц. Обычно окончание периода задания является жестким предельным сроком завершения задания, хотя некоторые задания могут иметь и более ранние предельные сроки. Время выполнения (вычисления)  $C$  представляет собой количество процессорного времени, требующегося для каждого задания определенного типа. Очевидно, что в однопроцессорной системе время выполнения не должно превышать период заданий (т.е. должно выполняться условие  $C \leq T$ ). Если периодическое задание всегда выполняется до полного завершения, т.е. если не имеется отклоненных из-за нехватки вычислительного ресурса заданий, то загруженность процессора этим заданием равна  $U = C/T$ . Например, если задание имеет период 80 мс и время выполнения 55 мс, то загруженность им процессора составляет  $55/80 = 0,6875$ .

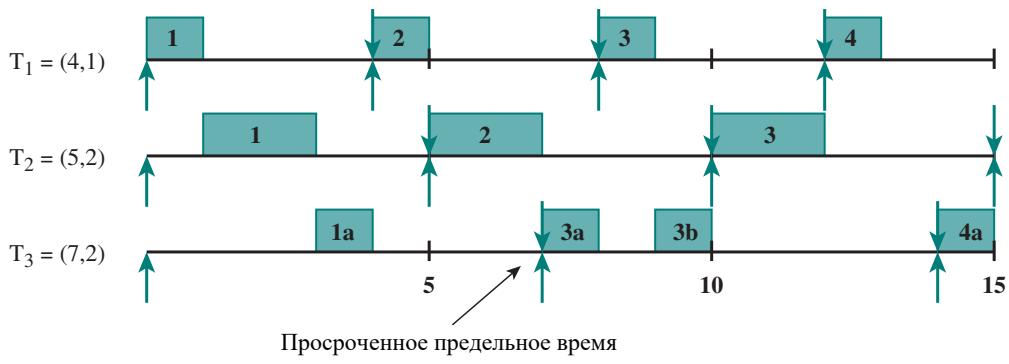
На рис. 10.8 показан простой пример RMS. Экземпляры заданий пронумерованы последовательно. Как можно видеть, для задания 3 второй экземпляр не выполняется, поскольку предельное время просрочено. Третий экземпляр испытывает вытеснение, но все равно остается в состоянии завершиться до предельного срока завершения.



**Рис. 10.7. Временная диаграмма периодического задания**



**(а) Времена поступления и предельные времена завершения задания  $T_i = (P_i, C_i)$ ;  $P_i$  = период,  $C_i$  = время обработки**



**Рис. 10.8. Пример работы RMS**

Одной из характеристик эффективности алгоритма периодического планирования является его гарантия соответствия всем жестким предельным срокам. Предположим, что у нас имеется  $n$  заданий, каждое из которых имеет свое фиксированное время выполнения и период. Тогда необходимым условием соответствия всем жестким предельным срокам является выполнение следующего неравенства:

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \dots + \frac{C_n}{T_n} \leq 1 \quad (10.1)$$

Сумма загруженности процессора разными заданиями не может превышать 1, что соответствует полной загрузке процессора. Неравенство (10.1) определяет верхнюю границу количества заданий, которые может успешно обслуживать идеальный алгоритм планирования. Для конкретного реального алгоритма граница может оказаться ниже. Так, можно показать, что для алгоритма RMS справедливо следующее неравенство:

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \dots + \frac{C_n}{T_n} \leq n(2^{1/n} - 1) \quad (10.2)$$

В табл. 10.5 приведены некоторые значения верхней границы для метода RMS. По мере возрастания количества заданий верхняя граница стремится к значению  $\ln 2 \approx 0,693$ .

**Таблица 10.5. ЗНАЧЕНИЯ ВЕРХНЕЙ ГРАНИЦЫ ЗАГРУЖЕННОСТИ ДЛЯ МЕТОДА RMS**

$n$	$n(2^{1/n} - 1)$
1	1,000
2	0,828
3	0,779
4	0,756
5	0,743
6	0,734
•	•
•	•
•	•
$\infty$	$\ln 2 \approx 0,693$

В качестве примера рассмотрим три периодических задания (здесь  $U_i = C_i/T_i$ ).

- Задание  $P_1$ :  $C_1 = 20$ ;  $T_1 = 100$ ;  $U_1 = 0,2$ .
- Задание  $P_2$ :  $C_2 = 40$ ;  $T_2 = 150$ ;  $U_2 = 0,267$ .
- Задание  $P_3$ :  $C_3 = 100$ ;  $T_3 = 350$ ;  $U_3 = 0,286$ .

Общая загруженность процессора этими тремя заданиями составляет

$$0,2 + 0,267 + 0,286 = 0,753.$$

Верхняя граница загруженности этих трех задач при использовании метода RMS составляет

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \frac{C_3}{T_3} \leq n(2^{1/3} - 1) = 0.779$$

Поскольку общая загруженность процессора по обработке приведенных заданий ниже верхней границы для метода RMS, можно сделать вывод, что при RMS-планировании будут успешно выполнены все задания.

Можно также показать, что верхняя граница из (10.1) справедлива для метода наиболее раннего предельного срока. Таким образом, при применении планирования с наиболее ранним предельным сроком можно достичь более высокой загрузки процессора и, соответственно, обработать большее количество заданий. Тем не менее метод RMS широко распространен и используется во многих промышленных приложениях. В работе [221] это поясняется следующими причинами.

1. На практике отличие производительности невелико. Кроме того, неравенство (10.2) консервативно, и на практике зачастую достигается 90%-ная загрузка процессора.
2. Большинство жестких систем реального времени содержат мягкие компоненты, такие как некритичный вывод на экран или встроенное самотестирование, выполняющееся с низким приоритетом. Эти компоненты используют процессорное время, которое остается после RMS-планирования жестких заданий.
3. При использовании RMS проще обеспечить стабильность. Когда система не в состоянии обеспечить все предельные сроки в силу перегруженности или временных ошибок, необходимо гарантировать выполнение предельных сроков для подмножества обязательных заданий. При статическом назначении приоритетов гарантируется только корректность выполнения основных задач с относительно высокими приоритетами. Это может быть достигнуто и в RMS-планировании путем реструктурирования обязательных задач для повышения их частоты либо посредством изменения приоритетов RMS для подмножества обязательных задач. При планировании с наиболее ранним предельным сроком приоритеты периодических заданий изменяются от одного периода к другому, что усложняет обеспечение корректной работы обязательных заданий с жесткими предельными сроками.

## Инверсия приоритета

Инверсия приоритета представляет собой явление, которое может произойти в любой схеме планирования на основе приоритетов с вытеснением, но особенно актуально в контексте планирования реального времени. Наиболее известный экземпляр инверсии приоритета произошел на Марсе — в миссии Mars Pathfinder. Этот робот совершил посадку на Марс 4 июля 1997 года и начал сбор и передачу объемных данных обратно на Землю. Но через несколько дней начались перегрузки системы, приводящие к потерям данных. После больших усилий команды лаборатории реактивного движения (Jet Propulsion Laboratory — JPL), которая создавала Pathfinder, было определено, что проблема кроется в инверсии приоритетов [121].

В любой схеме планирования на основе приоритетов система всегда должна выполнять задание с наивысшим приоритетом. **Инверсия приоритета** (priority inversion) происходит, когда система заставляет задание с высоким приоритетом ожидать задание с низким приоритетом. Простой пример инверсии приоритета возникает, когда задание с низким приоритетом блокирует ресурс (например, устройство или бинарный семафор), и задание с более высоким приоритетом также пытается заблокировать этот же ресурс. Высокоприоритетное задание оказывается в заблокированном состоянии до тех пор, пока ресурс не станет доступным. Если низкоприоритетное задание вскоре завершает работу с ресурсом и освобождает его, высокоприоритетные задачи могут быстро возобновить работу, и вполне возможно, что никакие ограничения реального времени не будут нарушены.

Более серьезная ситуация — **неограниченная инверсия приоритета** (unbounded priority inversion), в которой продолжительность инверсии приоритетов зависит не только от времени, необходимого для обработки совместно используемого ресурса, но и от непредсказуемых действий других, не связанных задач. Инверсия приоритета в программе Pathfinder оказалась неограниченной и служит хорошим примером этого явления. Наше обсуждение далее следует работе [258]. Программное обеспечение Pathfinder включало следующие три задания в убывающем порядке приоритетности.

$T_1$ : периодическая проверка нормальной работы всех систем и программного обеспечения.

$T_2$ : обработка изображений.

$T_3$ : выполнение времена от времени проверки состояния оборудования.

После выполнения задания  $T_1$  выполнялась повторная инициализация таймера его максимальным значением. Если в какой-то момент значение таймера истекает, предполагается, что целостность программного обеспечения спускаемого устройства нарушена. Процессор останавливается, все устройства сбрасываются, программное обеспечение полностью перезагружается, выполняется тестирование всех систем космического аппарата, и система начинает работу заново. Эта восстанавливающая последовательность не завершается до следующего дня.  $T_1$  и  $T_3$  имеют общую структуру данных, защищенную бинарным семафором  $s$ . На рис. 10.9, а показана последовательность, вызвавшая инверсию приоритета.

$t_1$ :  $T_3$  начинает выполняться.

$t_2$ :  $T_3$  захватывает семафор  $s$  и входит в критический участок кода.

$t_3$ :  $T_1$ , имеющий более высокий приоритет, чем приоритет  $T_3$ , вытесняет  $T_3$  и начинает выполнятся.

$t_4$ :  $T_1$  пытается войти в критический участок кода, но блокируется, поскольку семафор уже захвачен  $T_3$ ;  $T_3$  продолжает выполнение критического участка.

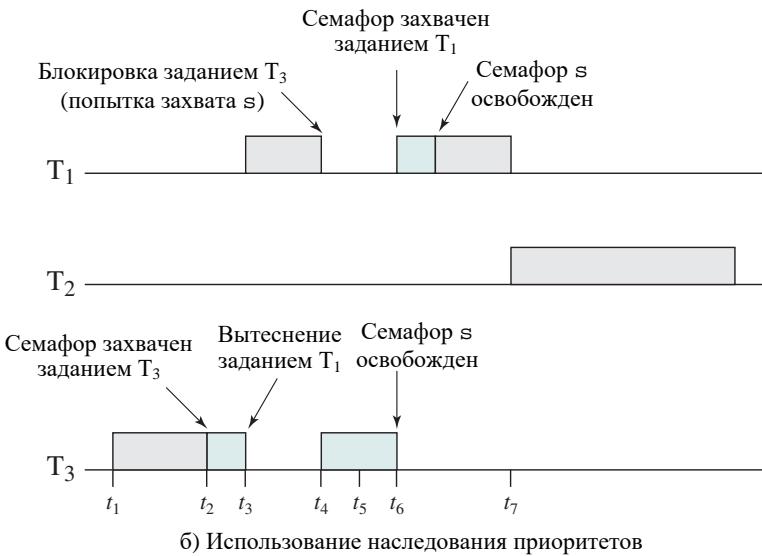
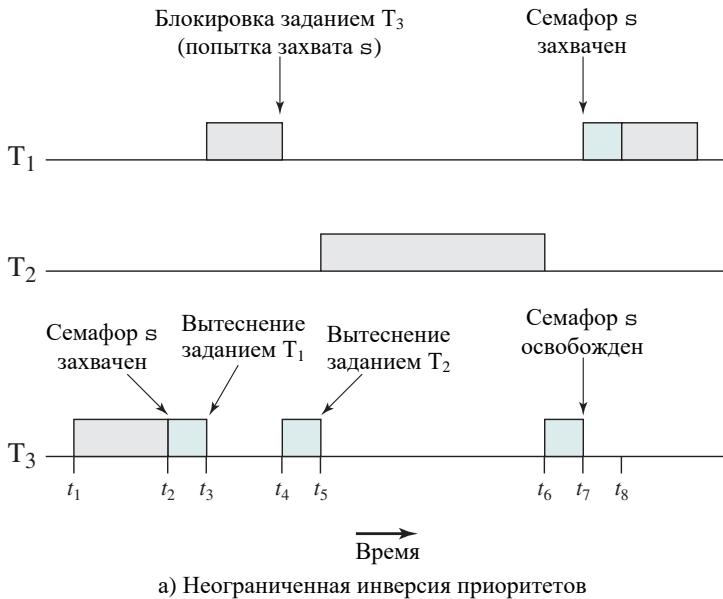
$t_5$ :  $T_2$ , который имеет более высокий приоритет, чем  $T_3$ , вытесняет  $T_3$  и начинает выполнятся.

$t_6$ :  $T_2$  приостанавливается по некоторой причине, не связанной с  $T_1$  и  $T_3$ ;  $T_3$  продолжает работу.

$t_7$ :  $T_3$  покидает критический участок и разблокирует семафор.  $T_1$  вытесняет  $T_3$ , блокирует семафор и входит в критический участок кода.

При таких обстоятельствах  $T_1$  должен дождаться завершения  $T_3$  и  $T_2$  и не сбрасывать таймер до истечения срока его действия.

В практических системах во избежание неограниченной инверсии приоритетов используются два альтернативных подхода: протокол наследования приоритета и протокол потолка приоритета.



■ Нормальное выполнение ■ Выполнение критического участка кода

Рис. 10.9. Инверсия приоритетов

Основная идея **наследования приоритетов** (priority inheritance) заключается в том, что задача с более низким приоритетом наследует приоритет любой высокоприоритетной задачи, ожидающей совместно используемый ресурс. Это изменение приоритета происходит, как только более высокоприоритетное задание блокируется в ожидании ресурса; оно должно заканчиваться, когда ресурс освобождается задачей с более низким приоритетом. На рис. 10.9, б показано, что наследование приоритетов решает проблему неограниченной инверсии приоритетов, показанной на рис. 10.9, а. Соответствующий порядок событий оказывается следующим.

- $t_1$ :  $T_3$  начинает выполняться.
- $t_2$ :  $T_3$  захватывает семафор  $s$  и входит в критический участок кода.
- $t_3$ :  $T_1$ , имеющий более высокий приоритет, чем приоритет  $T_3$ , вытесняет  $T_3$  и начинает выполнять.
- $t_4$ :  $T_1$  пытается войти в критический участок кода, но блокируется, поскольку семафор уже захвачен  $T_3$ .  $T_3$  немедленно (временно) получает тот же приоритет, что и  $T_1$ .  $T_3$  продолжает выполнение критического участка.
- $t_5$ :  $T_2$  готов к выполнению, но поскольку  $T_3$  теперь имеет более высокий приоритет,  $T_2$  не в состоянии вытеснить  $T_3$ .
- $t_6$ :  $T_3$  покидает критический участок и разблокирует семафор: уровень его приоритета опускается до бывшего у него ранее приоритета по умолчанию.  $T_1$  вытесняет  $T_3$ , блокирует семафор и входит в критический участок.
- $t_7$ :  $T_1$  приостанавливается по причине, не связанной с  $T_2$ , и  $T_2$  начинает выполнение.

Именно этот подход был применен для решения проблемы Pathfinder.

В подходе с **потолком приоритета** с каждым ресурсом связан приоритет. Приоритет, назначенный ресурсу, на один уровень выше приоритета его наиболее приоритетного пользователя. Затем планировщик динамически назначает этот приоритет любой задаче, которая обращается к ресурсу. Когда задание заканчивает работу с ресурсом, его приоритет возвращается в нормальное состояние.

## 10.3. ПЛАНИРОВАНИЕ В LINUX

Linux версии 2.4 и более ранних предоставляет возможности планирования реального времени в сочетании с планировщиком для процессов реального времени, который использует традиционный алгоритм планирования UNIX, описанный в разделе 9.3, “Традиционное планирование UNIX”. Linux 2.6 включает в себя, по существу, те же возможности планирования реального времени, что и предыдущие версии, и существенно пересмотренный планировщик для процессов реального времени. Мы по очереди рассмотрим обе указанные темы.

### Планирование реального времени

В Linux имеется три основных класса планирования.

1. **SCHED\_FIFO**. Потоки реального времени с использованием планирования по принципу “первым вошел — первым вышел”.
2. **SCHED\_RR**. Потоки реального времени с использованием кругового планирования.

3. **SCHED\_NORMAL**. Прочие потоки, не являющиеся потоками реального времени (**SCHED\_OTHER** в более старых ядрах).

В пределах каждого класса могут использоваться различные приоритеты, причем приоритеты классов реального времени выше приоритетов класса **SCHED\_NORMAL**. Используются следующие значения по умолчанию: классы приоритетов реального времени используют диапазон от 0 до 99 включительно, а **SCHED\_NORMAL** — диапазон от 100 до 139. Меньшее значение означает более высокий приоритет.

Для потоков FIFO применимы следующие правила.

1. Система не прерывает выполняющийся поток этого класса за исключением следующих ситуаций:
  - a) становится готовым другой поток этого же класса с более высоким приоритетом;
  - б) выполнение потока блокируется ожиданием события (например, выполнения операции ввода-вывода);
  - в) выполняющийся поток добровольно отдает процессор посредством вызова примитива `sched_yield`.
2. При прерывании выполняющегося потока FIFO он помещается в очередь, предназначенную для его уровня приоритета.
3. Если поток FIFO приходит в состояние готовности к выполнению и имеет более высокий приоритет, чем выполняющийся, то текущий поток вытесняется и начинает выполняться готовый поток с высшим приоритетом. Если таких потоков несколько, выбирается поток, находившийся в состоянии ожидания дольше других.

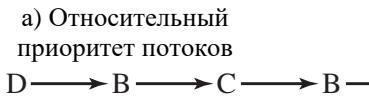
Стратегия **SCHED\_RR** аналогична стратегии **SCHED\_FIFO** за исключением добавления временной квоты, связанной с каждым потоком. При выполнении потока дольше, чем позволяет его квота, он приостанавливается, и для выполнения выбирается поток реального времени с равным или более высоким приоритетом.

На рис. 10.10 приведен пример, иллюстрирующий отличия использования стратегий планирования FIFO и RR. Предположим, что программа содержит четыре потока с относительными приоритетами, назначенными так, как показано на рис. 10.10, а. Предположим также, что все ожидающие потоки готовы к выполнению, когда текущий поток завершается или переходит в состояние ожидания, и что в процессе его выполнения не появляются потоки с более высоким приоритетом. На рис. 10.10, б показана очередность выполнения потоков, если они принадлежат классу **SCHED\_FIFO**. Поток D выполняется до его завершения или состояния ожидания; затем, несмотря на равные приоритеты потоков В и С, выполняется поток В как дольше находившийся в состоянии ожидания. Поток В выполняется до его завершения или состояния ожидания, после чего то же самое происходит с потоком С. И наконец, выполняется поток А.

На рис. 10.10, в показана очередность выполнения потоков в случае их принадлежности классу **SCHED\_RR**. Сначала до перехода в состояние ожидания или завершения выполняется поток D. Затем, чередуясь (поскольку они имеют одинаковый приоритет), выполняются потоки В и С, а после их завершения — поток А.

Последним классом планирования является **SCHED\_NORMAL**. Поток из этого класса может выполняться только в том случае, если не имеется готовых к выполнению потоков реального времени.

A	Минимальный
B	Средний
C	Средний
D	Максимальный



а) Относительный приоритет потоков



б) Выполнение при FIFO-планировании



в) Выполнение при RR-планировании

Рис. 10.10. Пример планирования реального времени в Linux

## Обычное планирование

Планировщик Linux 2.4 для класса SCHED\_OTHER (сейчас он называется SCHED\_NORMAL) не обеспечивает хорошую масштабируемость с увеличением числа процессоров и количества процессов. Недостатки этого планировщика включают следующее.

- Планировщик Linux 2.4 использует единую очередь выполнения для всех процессоров в симметричной многопроцессорной системе (symmetric multiprocessing system — SMP). Это означает, что задание может быть спланировано любому процессору, который может оказаться хорошим с точки зрения балансировки нагрузки, но плохим с точки зрения кешей памяти. Например, предположим, что задание выполнялось на процессоре CPU-1, и его данные находились в кеше данного процессора. Если задание перенесено на CPU-2, это приводит к тому, что его данные становятся недействительными в CPU-1 и будут загружены в CPU-2.
- Планировщик Linux 2.4 использует единую блокировку очереди выполнения. Таким образом, в системе SMP акт выбора выполняемого задания блокирует работу всех других процессоров с очередью выполнения. Бездействие процессоров в ожидании освобождения блокировки очереди выполнения приводит к снижению эффективности.
- Вытеснение в планировщике Linux 2.4 невозможно. Это означает, что низкоприоритетное задание может выполняться в то время, когда более высокоприоритетное задание будет ожидать его завершения.

Чтобы решить эти проблемы, Linux 2.6 использует совершенно новый приоритетный планировщик, известный как O(1)-планировщик<sup>5</sup>. Он разработан так, что время выбора подходящего процесса и назначения его процессору является константным, не зависящим от нагрузки на систему или количества процессоров. Однако O(1)-планировщик оказался слишком громоздким для ядра из-за большого объема кода и сложности используемых алгоритмов.

<sup>5</sup> Термин O(1) — пример записи с большим “O”, применяемой для характеристики сложности алгоритмов (приложение И, “Сложность алгоритмов”).

Как результат недостатков O(1)-планировщика начиная с Linux 2.6.23 появился новый планировщик, именуемый Совершенно Справедливым Планировщиком (Completely Fair Scheduler — CFS) [182]. CFS моделирует идеальный многозадачный процессор на реальном оборудовании, что обеспечивает равный доступ для всех задачий. Для достижения этой цели CFS поддерживает значение *виртуального времени выполнения* для каждой задачи. Виртуальное время выполнения является количеством затраченного к настоящему моменту времени выполнения, нормализованным в соответствии с количеством выполняемых процессов. Чем меньше виртуальное время выполнения (т.е. чем меньше времени заданию был разрешен доступ к процессору), тем выше необходимость в процессоре. CFS включает также понятие справедливости спящего режима, чтобы гарантировать, что задачи, которые в настоящее время не готовы к выполнению (например, в ожидании ввода-вывода), получают сопоставимые доли процессорного времени, когда они в конечном итоге в нем будут нуждаться.

Планировщик CFS реализуется классом планировщика *fair\_sched\_class*. Он основан на использовании красно-черного дерева, в отличие от других планировщиков, которые обычно основаны на очередях выполнения. *Красно-черное дерево* — это тип самобалансирующегося бинарного дерева поиска, которое подчиняется следующим правилам.

1. Узел дерева может быть черным либо красным.
2. Корень дерева — черный.
3. Все листья дерева (NIL) — черные.
4. Если узел красный, оба его дочерних узла — черные.
5. На любом пути от данного узла к любому из его потомков NIL содержится одинаковое количество черных узлов.

Эта схема обеспечивает высокую эффективность при вставке, удалении и поиске задачий благодаря сложности  $O(\log N)$ .

Красно-черное дерево показано на рис. 10.11. В таком дереве в Linux содержится информация о выполняемых процессах. Элементы *rb\_node* дерева встроены в объект *sched\_entity*. Красно-черное дерево упорядочено по виртуальному времени выполнения *vruntime*, где крайний слева узел дерева представляет процесс, который имеет самое низкое значение виртуального времени выполнения и испытывает наибольшую потребность в процессоре; именно этот узел будет первым выбирается процессором, а когда он будет запущен, его *vruntime* обновится в зависимости от потребленного им времени. Поэтому, когда узел будет вставлен обратно в дерево, вероятнее всего, он больше не будет узлом с самым низким *vruntime*, и дерево будет перебалансировано, чтобы отразить текущее состояние системы. Такой процесс перебалансировки будет продолжаться для каждого последующего процесса, который будет выбран планировщиком CFS, и т.д. Таким образом, CFS управляет справедливой политикой планирования.

При выполнении операции вставки в красно-черное дерево крайнее слева дочернее значение для быстрого поиска кешируется в *sched\_entity*.

В ядро 2.6.24 была введена новая функциональная возможность, называемая *групповым планированием* CFS. Эта функциональная возможность позволяет ядру планировать группу задач, как если бы они были одной задачей. Групповое планирование предназначено для обеспечения справедливости, когда задача создает множество других задач.

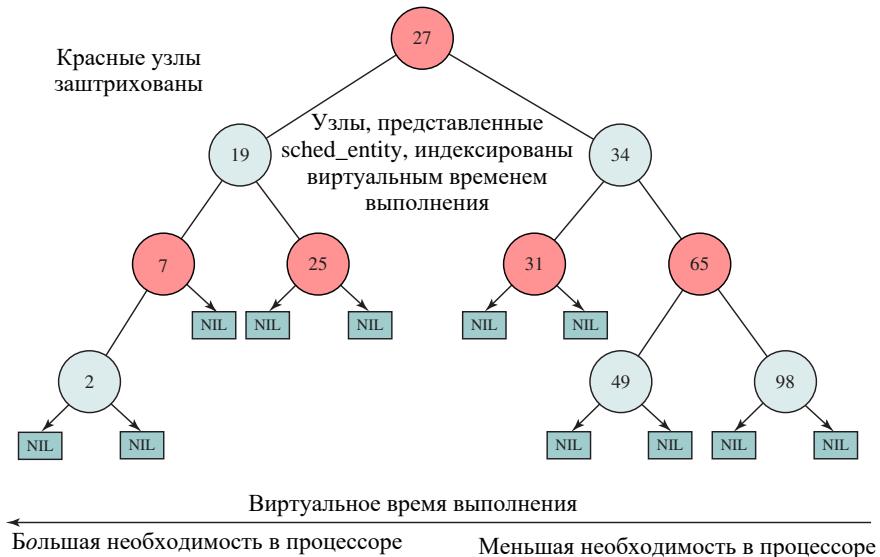


Рис. 10.11. Пример применения красно-чёрного дерева для CFS

## 10.4. ПЛАНИРОВАНИЕ В UNIX SVR4

Алгоритм планирования в UNIX SVR4 представляет собой преобразованный алгоритм из ранних систем UNIX (описанный в разделе 9.3, “Традиционное планирование UNIX”). Новый алгоритм разработан таким образом, что дает наивысший приоритет процессам реального времени, приоритет следующего уровня присваивается процессам ядра, а пользовательские процессы (известные как процессы с разделением времени<sup>6</sup>) получают низший приоритет.

В SVR4 реализованы два основных изменения традиционного алгоритма.

1. Добавлено вытесняющее планирование на основе статических приоритетов; в систему введено множество из 160 уровней приоритетов, разделенных на три класса.
2. Внесены точки вытеснения. Поскольку базовое ядро не вытесняемо, его выполнение может быть разделено на небольшие части, выполняемые до завершения без прерывания. В промежутках между выполнением этих фрагментов размещены точки вытеснения, в которых ядро может быть безопасно прервано и в которых может начаться выполнение нового процесса. Точка вытеснения может быть определена как область кода, в которой все структуры данных ядра либо обновлены и согласованы, либо заблокированы при помощи семафора.

На рис. 10.12 показаны 160 уровней приоритетов, определенные в SVR4. Каждый процесс принадлежит одному из трех классов приоритетов и получает свой уровень приоритета в пределах данного класса.

<sup>6</sup> Процессы с разделением времени — это процессы, соответствующие пользователям в традиционных системах с разделением времени.

Класс приоритета	Глобальное значение	Последовательность планирования
Реальное время	159	Первые
	•	
	•	
	•	
	100	
Ядро	99	
	•	
	•	
	60	
Разделение времени	59	
	•	
	•	
	0	Последние
		↓

Рис. 10.12. Очереди диспетчера SVR4

Вот краткое описание каждого класса приоритетов.

- **Реального времени (159–100).** Процессы этих уровней приоритета гарантированно выбираются для выполнения прежде любых процессов ядра и процессов с разделением времени. Кроме того, процессы реального времени могут использовать точки вытеснения для прерывания выполнения процессов ядра и пользовательских процессов.
- **Ядра (99–60).** Процессы с этими уровнями приоритета гарантированно выбираются для выполнения прежде всех процессов с разделением времени, но уступают процессам реального времени.
- **Разделения времени (59–0).** Процессы с низшим приоритетом, принадлежащие пользовательским приложениям (кроме приложений реального времени).

На рис. 10.13 показано, как реализовано планирование в SVR4. С каждым уровнем приоритета связана своя очередь, и процессы одного уровня приоритета планируются с использованием круговой стратегии. Битовый вектор `dqactmap` содержит по одному биту для каждого уровня приоритета; этот бит устанавливается, если соответствующая данному приоритету очередь не пуста. Когда выполняющийся процесс блокируется или вытесняется, диспетчер обращается к вектору `dqactmap` и запускает готовый к выполнению процесс из непустой очереди с наивысшим приоритетом. Кроме того, по достижении точки вытеснения ядро проверяет состояние флага `kprunrun`. Если данный флаг установлен, это означает, что в состоянии готовности имеется как минимум один процесс реального времени, и в этом случае ядро вытесняет текущий процесс (если его приоритет ниже приоритета готового к выполнению процесса реального времени с наивысшим приоритетом).



**Рис. 10.13.** Классы приоритетов SVR4

Приоритет процесса в классе разделяемого времени является величиной переменной. Планировщик снижает приоритет процесса всякий раз при использовании им очередного кванта времени и повышает — при блокировке процесса в ожидании некоторого события или ресурса. Квант времени, выделяемый процессу с разделением времени, зависит от его приоритета и изменяется от 100 мс для приоритета 0 до 10 мс для приоритета 59. Для сравнения: каждый процесс реального времени имеет фиксированный приоритет и фиксированный квант времени.

## 10.5. ПЛАНИРОВАНИЕ В UNIX FREEBSD

Планировщик UNIX FreeBSD предназначен для обеспечения более эффективной работы, чем предыдущие планировщики UNIX, под высокой нагрузкой и при использовании на многопроцессорных или многоядерных платформах. Этот планировщик довольно сложен, и здесь мы предоставим обзор только наиболее значительных особенностей его дизайна; более подробно этот материал изложен в [168] и [207].

### Классы приоритетов

Механизм приоритетов, лежащий в основе планировщика FreeBSD, подобен механизму UNIX SVR4. В FreeBSD определены пять классов приоритетов (табл. 10.6); первые два класса представляют потоки ядра, а остальные классы — потоки в режиме пользователя. Потоки ядра выполняют код, который компилируется в загрузочный образ ядра, и работают с привилегированным выполняемым кодом ядра.

Потоки с наивысшим приоритетом называются потоками *нижней половины ядра*. Потоки в этом классе выполняются в контексте ядра и планируются на основе приоритетов прерываний. Эти приоритеты устанавливаются при настройке соответствующих устройств и в дальнейшем не изменяются. Потоки *верхней половины ядра* также выполняются в контексте ядра и выполняют различные функции ядра. Их приоритеты устанавливаются на основе предопределенных приоритетов и никогда не изменяются.

Следующий класс с более низким приоритетом называется классом *пользователей реального времени* (real-time user). Поток с приоритетом реального времени не подлежит деградации приоритетов, т.е. поток реального времени поддерживает приоритет, с которым он начал работу, и не опускается до более низкого значения приоритета в результате использования ресурсов. Далее идет класс приоритетов *пользователей в режиме разделения времени* (time-sharing user). Для потоков из этого класса приоритет периодически пересчитывается на основе ряда параметров, включая количество использованного потоком времени процессора, количество использованных ресурсов памяти и другие

параметры потребления ресурсов. Еще более низкий диапазон приоритетов называется *классом простояющих пользователей (idle user)*. Этот класс предназначен для приложений, которые будут потреблять процессорное время только тогда, когда нет других готовых к выполнению потоков.

**Таблица 10.6. Классы планирования потоков FreeBSD**

Класс приоритетов	Тип потока	Описание
0–63	Нижняя половина ядра	Планируются с использованием прерываний. Могут блокироваться в ожидании ресурса
64–127	Верхняя половина ядра	Выполняются до блокирования или завершения. Могут блокироваться в ожидании ресурса
128–159	Потоки реального времени	Могут выполняться до тех пор, пока не будут заблокированы или пока не станет доступным поток с более высоким приоритетом. Вытесняющее планирование
160–223	Потоки разделения времени	Обновление приоритетов на основе использования процессора
224–255	Простаивающие	Работают, только когда нет готовых к выполнению потоков разделения времени или реального времени

*Примечание:* более низкие значения соответствуют более высокому приоритету.

## Поддержка SMP и многоядерности

Последняя версия планировщика FreeBSD, введенная в FreeBSD 5.0, разработана для обеспечения эффективного планирования для SMP или многоядерных систем. Новый планировщик отвечает трем целям проектирования.

1. Разрешает проблемы привязки процессоров в SMP и многоядерных системах. Термин *привязка процессора*, или *средство к процессору* (processor affinity), относится к планировщику, который выполняет перенос потока (его перемещение от одного процессора к другому) только тогда, когда необходимо избегать простоя процессора.
2. Обеспечение лучшей поддержки многопоточных или многоядерных систем.
3. Повышение производительности алгоритма планирования настолько, чтобы она больше не зависела от количества потоков в системе.

В этом подразделе мы рассмотрим три ключевые особенности нового планировщика: структуру очереди, учет интерактивности и миграцию потоков.

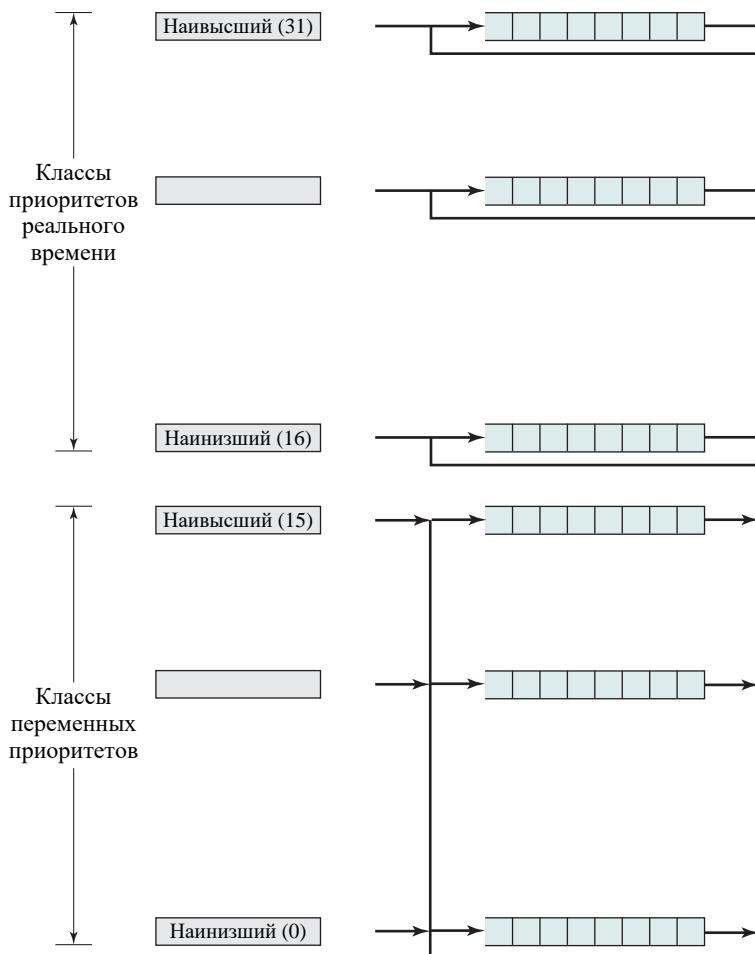
### Структура очереди

В предыдущей версии планировщика FreeBSD использовалась одна глобальная очередь планирования для всех процессоров, для которой раз в секунду выполнялся обход с пересчетом приоритетов. Использование единого списка для всех потоков означает, что

производительность планировщика зависит от количества задач в системе, и по мере роста числа задач все больше времени процессора необходимо тратить на работу планировщика, поддерживающего список.

Новый планировщик выполняет планирование для каждого процессора независимо, поддерживая для каждого процессора три очереди. Каждая из очередей имеет структуру, показанную на рис. 10.14 для SVR4. Две очереди реализуют классы планирования ядра, режима реального времени и режима распределения времени (приоритеты с 0 по 223). Третья очередь предназначена только для класса простаивающих потоков (приоритеты с 224 по 255).

Две очереди обозначены как *текущая* (current) и *следующая* (next). Каждый поток, которому предоставлен квант времени (состояние готовности), помещается либо в одну очередь, либо в другую (как объясняется позже) с подходящим приоритетом для этого потока.



**Рис. 10.14.** Приоритеты потоков Windows

Планировщик для процессора выбирает потоки из текущей очереди в порядке приоритета, пока текущая очередь не станет пустой. Когда текущая очередь пуста, планировщик меняет местами текущую и следующую очередь и начинает планировать потоки из вновь ставшей текущей очереди. Использование двух очередей гарантирует, что каждому потоку будет предоставлено время процессора не реже одного раза за два переключателя очереди независимо от приоритета, что позволяет избежать голодания.

Назначение потока как текущей, так и следующей очереди определяет несколько правил.

1. Потоки ядра реального времени всегда вставляются в текущую очередь.
2. Потоки в режиме распределения времени назначаются текущей очереди, если она является интерактивной (объясняется в следующем подразделе), в противном случае — следующей очереди. Вставка интерактивных потоков в текущую очередь приводит к малому интерактивному времени отклика таких потоков по сравнению с другими потоками с разделением времени, которые не обладают высокой степенью интерактивности.

## Учет интерактивности

Поток рассматривается как **интерактивный**, если отношение времени его не вынужденного ожидания и времени его выполнения ниже определенного порога. Интерактивные потоки обычно имеют высокие значения времени ожидания, так как они ожидают ввода от пользователя. За этими интервалами ожидания следуют всплески активности процессора, когда поток обрабатывает запрос пользователя.

Порог интерактивности определен в коде планировщика и не настраивается. Планировщик использует два уравнения для вычисления показателя интерактивности потока. Во-первых, мы определяем коэффициент масштабирования:

$$\text{Коэффициент масштабирования} = \frac{\text{Максимальный показатель интерактивности}}{2}$$

Для потоков, время сна которых превышает время их работы, используется следующее уравнение:

$$\text{Показатель интерактивности} = \text{Коэффициент масштабирования} \left( \frac{\text{Выполнение}}{\text{Сон}} \right)$$

Когда время выполнения потока превосходит время сна, вместо этого используется следующее соотношение:

$$\text{Показатель интерактивности} = \text{Коэффициент масштабирования} \left( 1 + \frac{\text{Сон}}{\text{Выполнение}} \right)$$

В результате потоки, время сна которых превышает время их выполнения, оказываются в нижней половине диапазона значений интерактивности, а потоки, время выполнения которых превышает время их сна, — в верхней половине диапазона.

## Миграция потоков

В общем случае желательно запланировать готовый поток к выполнению на том процессоре, на котором он работал в последний раз; это явление называется **привязкой к**

процессору (processor affinity). Альтернативой является переход потока для следующего кванта времени выполнения на другой процессор. Привязка к процессору играет важную роль из-за локальных кешей, выделенных одному процессору. Когда поток запускается, его данные все еще могут находиться в кеше последнего процессора. Переход на другой процессор означает, что необходимые данные должны быть загружены в кэши нового процессора, а строки кеша предыдущего процессора становятся недействительными. С другой стороны, переход на другой процессор может обеспечить лучшую балансировку нагрузки и воспрепятствовать периодам простоя на некоторых процессорах, в то время как другие процессоры имеют больше работы, чем могут своевременно обрабатывать.

Планировщик FreeBSD поддерживает два механизма миграции потоков для балансировки нагрузки: pull (тянуть) и push (толкать). С помощью **pull-механизма** проставляющий процессор перехватывает поток у работающего процессора. Когда процессор не имеет работы, он устанавливает бит в глобальной битовой маске, указывающей, что он неактивен. Когда активный процессор собирается добавить работу в собственную очередь выполнения, он сначала проверяет наличие таких битов простоя и, если установленный бит простоя найден, передает поток бездействующему процессору. Это полезно, в первую очередь, тогда, когда есть легкая или спорадическая нагрузка или когда процессы стартуют и завершаются очень часто.

Механизм pull эффективен для предотвращения потерь процессорного времени из-за бездействия. Но это неэффективно и в действительности неактуально в ситуации, когда каждый процессор работает, но нагрузка распределена неравномерно. С помощью **push-механизма** периодическое задание планировщика оценивает текущую ситуацию с загрузкой и выравнивает ее. Дважды в секунду это задание выбирает наиболее и наименее загруженные процессоры в системе и выравнивает их очереди выполнения. Push-миграция обеспечивает справедливость среди запущенных потоков.

## 10.6. ПЛАНИРОВАНИЕ В WINDOWS

Операционная система Windows разработана таким образом, чтобы либо быть по возможности максимально чувствительной к нуждам отдельного пользователя в интерактивной среде, либо выступать в роли сервера. В Windows реализован планировщик с вытеснением и гибкой системой уровней приоритетов, включающий круговое планирование на каждом уровне, а для некоторых уровней — динамическое изменение приоритета на основе текущей активности потоков. В Windows единицей планирования является поток, а не процесс.

### Приоритеты процессов и потоков

Приоритеты в Windows организованы в виде двух групп, или классов: реального времени и переменные. Каждая из этих групп состоит из 16 уровней приоритета. Потоки, требующие немедленного внимания, находятся в классе реального времени, который включает такие функции, как осуществление коммуникаций и задачи реального времени.

В целом, поскольку Windows использует вытесняющий планировщик с учетом приоритетов, потоки с приоритетами реального времени имеют преимущество по отношению к прочим потокам. В однопроцессорной системе, когда становится готовым к выполнению поток с приоритетом, более высоким, чем у выполняющегося в настоящий момент, низкоприоритетный поток вытесняется и начинает выполняться более высокоприоритетный поток.

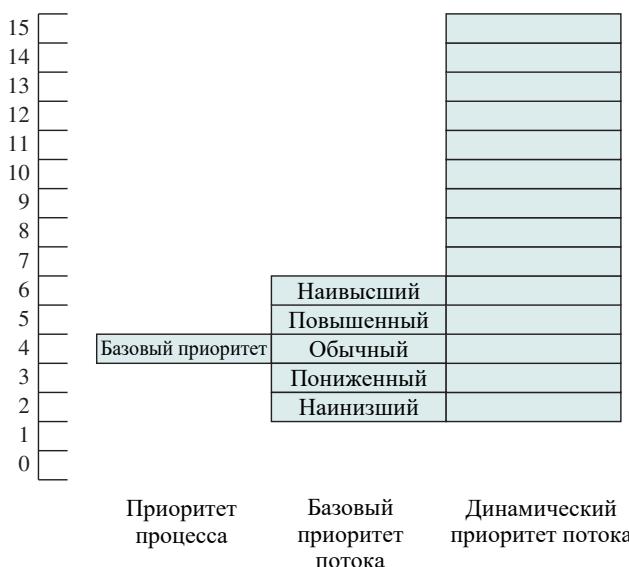
Приоритеты из разных классов обрабатываются несколько по-разному (см. рис. 10.14).

**В классе приоритетов реального времени** все потоки имеют фиксированный приоритет, который никогда не изменяется, и все активные потоки с определенным уровнем приоритета располагаются в круговой очереди данного уровня. В **классе переменных приоритетов** поток начинает работу с некоторым изначально присвоенным ему приоритетом, который затем может изменяться — как в большую, так и в меньшую сторону. Таким образом, на каждом уровне приоритета имеется своя очередь, но процессы могут переходить из одной очереди в другую в пределах одного класса переменных приоритетов. Поток с приоритетом 15 не может получить приоритет 16, как и какой-либо другой приоритет из класса реального времени.

Начальный приоритет потока в классе переменных приоритетов определяется двумя величинами: базовым приоритетом процесса и базовым приоритетом потока. Одним из атрибутов процесса является его базовый приоритет, который может принимать значение от 1 до 15 (приоритет 0 зарезервирован для исполнительных потоков, по одному на процессор). Каждый объект потока, связанный с объектом процесса, имеет собственный базовый приоритет, который указывает базовый приоритет потока по отношению к приоритету процесса и может отличаться от базового приоритета процесса не более чем на 2 уровня в большую или меньшую сторону. Так, например, если базовый приоритет процесса равен 4, а базовый приоритет одного из его потоков равен -1, то начальный приоритет этого потока равен 3.

После создания потока из класса переменных приоритетов его действительный приоритет (именуемый динамическим приоритетом потока) может колебаться в определенных пределах — он не может упасть ниже наименьшего базового приоритета потока или подняться выше максимально возможного значения приоритета данного класса, т.е. 15.

На рис. 10.15 приведен пример процесса с базовым приоритетом, равным 4.



**Рис. 10.15.** Пример отношения приоритетов в Windows

Каждый объект потока, связанный с данным процессом, должен иметь начальный приоритет от 2 до 6. Предположим, что базовый приоритет потока равен 4. Тогда текущий приоритет потока может колебаться в диапазоне от 4 до 15 в зависимости от полученного повышения. Если поток прерывается в ожидании ввода-вывода, ядро повышает его приоритет. Если поток с повышенным приоритетом прерывается из-за исчерпания кванта времени, ядро понижает его приоритет. Таким образом, имеется тенденция повышения приоритетов, ориентированных на ввод-вывод, и снижения приоритетов, ориентированных на вычисления. В потоках, ориентированных на ввод-вывод, приоритет потоков, ожидающих завершения интерактивной операции (например, вывод на экран или ввод с клавиатуры), повышается больше, чем для других операций ввода-вывода (например, дисковых). Следовательно, в пределах класса переменных приоритетов интерактивные потоки имеют тенденцию к получению наивысшего приоритета.

## Многопроцессорное планирование

Windows поддерживает многопроцессорные и многоядерные аппаратные конфигурации. Потоки любого процесса, в том числе исполнительной системы, могут работать на любом процессоре. В отсутствие ограничений привязки к процессору диспетчер ядра назначает готовый к выполнению поток следующему доступному процессору. Это гарантирует, что ни один процессор не простояивает и не выполняет поток с более низким приоритетом при наличии готового потока с более высоким приоритетом. Несколько потоков одного и того же процесса могут выполняться одновременно на нескольких процессорах.

По умолчанию диспетчер ядра при назначении потоков процессорам использует стратегию **мягкой привязки к процессору** (soft affinity), пытаясь назначить потоку тот же процессор, который последним выполнял данный поток. Это способствует повторному использованию данных, если они все еще остаются в кеше процессора после предыдущего выполнения потока. Можно также строго ограничить выполнение потоков приложения только определенными процессорами (**жесткая привязка к процессору** (hard affinity)).

При работе Windows в системе с одним процессором поток с наивысшим приоритетом всегда активен (если только не ожидает наступления какого-либо события). Если имеется несколько потоков с наивысшим приоритетом, то процессор работает с ними с использованием кругового планирования. В многопроцессорной системе с  $N$  процессорами ядро пытается назначить  $N$  процессорам  $N$  потоков с наивысшими приоритетами, готовых к выполнению. Остальные потоки с более низким приоритетом должны ожидать, пока не будет заблокирован какой-то из более высокоприоритетных потоков или пока его приоритет не деградирует. Приоритет низкоприоритетных потоков может также быть повышен до 15 на очень короткое время в случае голода, исключительно для исправления ситуации с инверсией приоритетов.

Описанный принцип управляет привязкой потока к процессору. Если поток готов к выполнению, но доступные процессоры не находятся во множестве процессоров, привязанных к потоку, то такой поток вынужден находиться в состоянии ожидания, в то время как планировщик запускает другой доступный поток.

## 10.7. Резюме

В сильносвязанной многопроцессорной системе доступ к одной и той же основной памяти получают несколько процессоров. В такой конфигурации структура планирования должна быть несколько более сложной, чем в однопроцессорной системе. Например, данный процесс может быть назначен тому же процессору на весь жизненный цикл или диспетчеризоваться новому процессору всякий раз, когда он переходит в состояние готовности к выполнению. Изучение производительности многопроцессорных систем показывает, что различия в производительности при использовании различных стратегий планирования в многопроцессорных системах не так значительны, как в однопроцессорной системе.

Процессами реального времени являются такие процессы, которые выполняются в связи с некоторыми процессами, функциями или множествами событий, внешними по отношению к вычислительной системе, и для которых с целью эффективного и корректного взаимодействия с внешней средой заданы предельные сроки начала или окончания работы. Операционная система реального времени представляет собой операционную систему, которая способна управлять процессами реального времени. В этом контексте традиционные критерии алгоритмов планирования неприменимы, и ключевым фактором для системы планирования становятся предельные сроки, что приводит к использованию специализированных алгоритмов планирования.

## 10.8. Ключевые термины, контрольные вопросы и задачи

### Ключевые термины

pull-механизм	Инверсия приоритета	Планирование потоков
push-механизм	Мягкая привязка к процессору	Планирование реального времени
Бригадное планирование	Мягкое задание реального времени	Планирование с предельными сроками
Детерминизм	Наследование приоритета	Потолок приоритета
Детерминистическая операционная система	Неограниченная инверсия приоритетов	Привязка к процессору
Жесткая привязка к процессору	Непериодическое задание	Разделение загрузки
Жесткое задание реального времени	Операционная система реального времени	Частотно-монотонное планирование
Зернистость	Периодическое задание	Чувствительность

## Контрольные вопросы

- 10.1. Перечислите и кратко опишите пять категорий зернистости синхронизации.
- 10.2. Перечислите и кратко опишите четыре метода планирования потоков.
- 10.3. Перечислите и кратко опишите три версии разделения загрузки.
- 10.4. В чем заключается различие между жесткими и мягкими заданиями реального времени?
- 10.5. В чем заключается различие между периодическими и непериодическими заданиями реального времени?
- 10.6. Перечислите и кратко опишите пять общих типов требований к операционным системам реального времени.
- 10.7. Перечислите и кратко опишите четыре класса алгоритмов планирования реального времени.
- 10.8. Какая информация о задании может использоваться в планировании реального времени?

## Задачи

- 10.1. Рассмотрим множество из трех периодических заданий, профили выполнения которых приведены в табл. 10.7. Разработайте для этого множества диаграмму планирования, аналогичную диаграмме, приведенной на рис. 10.5.

**Таблица 10.7. Профили выполнения процессов к задаче 10.1**

Процесс	Время поступления	Время выполнения	Предельный срок завершения
A(1)	0	10	20
A(2)	20	10	40
•	•	•	•
•	•	•	•
•	•	•	•
B(1)	0	10	50
B(2)	50	10	100
•	•	•	•
•	•	•	•
•	•	•	•
C(1)	0	15	50
C(2)	50	15	100
•	•	•	•
•	•	•	•
•	•	•	•

**10.2.** Рассмотрим множество из пяти непериодических заданий, профили выполнения которых приведены в табл. 10.8. Разработайте для этого множества диаграмму планирования, аналогичную той, которая приведена на рис. 10.6.

**Таблица 10.8. Профили выполнения процессов к задаче 10.2**

Процесс	Время поступления	Время выполнения	Предельный срок завершения
A	10	20	100
B	20	20	30
C	40	20	60
D	50	20	80
E	60	20	70

**10.3.** “Наименьшая неопределенность первой” (Least-laxity-first — LLF) — это алгоритм планирования в реальном времени для периодических задач. Неопределенное время или неопределенность — это время между тем, когда задача завершится, если начнется прямо сейчас, и ее следующий предельный срок. Это размер доступного окна планирования. Неопределенность может быть выражена следующим образом:

Неопределенность =

(Предельное время) – (Текущее время) – (Необходимое время процессора)

LLF-планирование выбирает для выполнения задание с минимальной неопределенностью. Если два или несколько заданий имеют одно и то же минимальное значение неопределенности, они обслуживаются в соответствии с принципом FCFS.

- Предположим, что в настоящий момент задание имеет неопределенность  $t$ . На сколько планировщик может задержать запуск этого задания и при этом не нарушить его предельный срок?
- Предположим, что в настоящий момент задание имеет неопределенность 0. Что это означает?
- Что означает отрицательная неопределенность задания?
- Рассмотрим множество из трех периодических заданий с профилями выполнения, показанными в табл. 10.9, а. Разработайте для данного множества диаграмму планирования, аналогичную той, которая приведена на рис. 10.5, в которой выполнялось бы сравнение частотно-монотонного планирования, планирования наиболее раннего предельного срока и LLF. Считайте, что вытеснение может происходить с интервалом 5 мс. Прокомментируйте полученные результаты.

**ТАБЛИЦА 10.9. ПРОФИЛИ ВЫПОЛНЕНИЯ К ЗАДАЧАМ ОТ 10.3 ДО 10.6****а) Слабая загрузка**

<b>Задание</b>	<b>Период</b>	<b>Время выполнения</b>
A	6	2
B	8	2
C	12	3

**б) Сильная загрузка**

<b>Задание</b>	<b>Период</b>	<b>Время выполнения</b>
A	6	2
B	8	5
C	12	3

**10.4.** Выполните задание 10.3, г для профиля выполнения из табл. 10.9, б. Прокомментируйте полученные результаты.

**10.5.** “Наиболее срочный первым” (maximum-urgency-first — MUF) — это алгоритм планирования в реальном времени для периодических заданий. Каждому заданию назначается срочность, которая определяется как комбинация двух фиксированных приоритетов и одного динамического приоритета. Один из фиксированных приоритетов, критичность, имеет приоритет над динамическим приоритетом. Между тем динамический приоритет имеет приоритет над другим фиксированным приоритетом, называемым приоритетом пользователя. Динамический приоритет обратно пропорционален неопределенности задания. MUF можно объяснить следующим образом. Во-первых, задания упорядочены от самого короткого до самого длительного периода. Определим множество критических заданий как первые N заданий, таких, что использование процессора в худшем случае не превышает 100%. Среди готовых к выполнению критических заданий планировщик выбирает задание с наименьшей неопределенностью. Если нет готового критического задания, планировщик выбирает среди некритических заданий то, которое имеет наименьшую неопределенность. Неоднозначности разрешаются с помощью необязательного пользовательского приоритета, а затем по алгоритму FCFS. Повторите решение задачи 10.3, г, добавив к диаграммам MUF. Считайте, что приоритеты, определенные пользователем, являются для А самым высоким, для В — меньшим и для С — наименьшим. Прокомментируйте полученные результаты.

**10.6.** Повторите решение задачи 10.4, добавив к диаграммам MUF. Прокомментируйте полученные результаты.

**10.7.** Эта задача демонстрирует, что хотя формула (10.2) является для частотно-монотонного планирования условием, достаточным для успешного планирования, оно не

является необходимым, т.е. иногда успешное планирование возможно в ситуации, когда условие (10.2) не выполняется.

а. Рассмотрим множество следующих независимых периодических заданий:

Задание  $P_1$ ;  $C_1 = 20$ ;  $T_1 = 100$ ;

Задание  $P_2$ ;  $C_2 = 30$ ;  $T_2 = 145$ .

Могут ли эти задания быть успешно спланированы с использованием частотно-монотонного планирования?

б. Добавим к рассматриваемому множеству еще одно задание:

Задание  $P_3$ ;  $C_3 = 68$ ;  $T_3 = 150$ .

Удовлетворяется ли при этом условие (10.2)?

в. Предположим, что первые задания рассмотренных трех типов поступают в систему в момент  $t = 0$ , а первые предельные сроки заданий —  $D_1 = 100$ ;  $D_2 = 145$ ;  $D_3 = 150$ . Будут ли выполнены все три ограничения при использовании частотно-монотонного планирования? Что можно сказать о предельных сроках последующих повторений каждого задания?

10.8. Нарисуйте диаграмму, аналогичную диаграмме на рис. 10.9, б, которая показывает события последовательности для этого же примера с использованием потолка приоритета.



ЧАСТЬ V

---

# ВВОД-ВЫВОД И ФАЙЛЫ



# 11

## ГЛАВА

# УПРАВЛЕНИЕ ВВОДОМ-ВЫВОДОМ И ПЛАНИРОВАНИЕ ДИСКОВЫХ ОПЕРАЦИЙ

В ЭТОЙ ГЛАВЕ...

### 11.1. Устройства ввода-вывода

### 11.2. Организация функций ввода-вывода

Эволюция функций ввода-вывода

Прямой доступ к памяти

### 11.3. Вопросы проектирования операционных систем

Цели проектирования

Логическая структура функций ввода-вывода

### 11.4. Буферизация операций ввода-вывода

Двойной буфер

Циклический буфер

Использование буферизации

### 11.5. Дисковое планирование

Параметры производительности диска

Время поиска

Задержка из-за вращения

Время передачи данных

Сравнение времени

Стратегии дискового планирования

FIFO

Приоритеты

Последним вошел — первым вышел

SSTF

SCAN

C-SCAN

N-step-SCAN и FSCAN

## 11.6. RAID

RAID 0

RAID 0 и передача большого объема данных

RAID 0 и высокая частота запросов ввода-вывода

RAID 1

RAID 2

RAID 3

Избыточность

Производительность

RAID 4

RAID 5

RAID 6

## 11.7. Дисковый кеш

Вопросы разработки

Вопросы производительности

## 11.8. Ввод-вывод в UNIX SVR4

Буфер кеша

Очередь символов

Небуферизованный ввод-вывод

Устройства UNIX

## 11.9. Ввод-вывод в Linux

Дисковое планирование

Планировщик Elevator

Планировщик крайнего срока

Упреждающий планировщик ввода-вывода

Планировщик NOOP

Планировщик ввода-вывода CFQ

Страницочный кеш Linux

## 11.10. Ввод-вывод в Windows

Основные средства ввода-вывода

Асинхронный и синхронный ввод-вывод

Программное обеспечение RAID

Теневые копии тома

Шифрование тома

## 11.11. Резюме

## 11.12. Ключевые термины, контрольные вопросы и задачи

Ключевые термины

Контрольные вопросы

Задачи

## УЧЕБНЫЕ ЦЕЛИ

- Ориентироваться в ключевых категориях устройств ввода-вывода на компьютерах.
- Обсуждать организацию функций ввода-вывода.
- Объяснять некоторые ключевые вопросы дизайна поддержки ввода-вывода операционной системой.
- Анализировать влияние на производительность различных альтернатив буферизации ввода-вывода.
- Понимать проблемы производительности, связанные с доступом к магнитному диску.
- Объяснять концепцию RAID и описывать различные уровни RAID.
- Понимать влияние дискового кеша на производительность.
- Описывать механизмы ввода-вывода в UNIX, Linux и Windows.

Ввод-вывод является, пожалуй, самым значимым аспектом при создании операционных систем. Вследствие широкого разнообразия запоминающих устройств и приложений разработать общее согласованное решение, касающееся их организации, очень сложно.

Мы начнем главу с краткого обсуждения устройств и организации функций ввода-вывода. Эти вопросы, обычно относящиеся к области архитектуры компьютера, приводят к необходимости изучения ввода-вывода с точки зрения операционной системы.

В следующем разделе рассматриваются вопросы разработки операционных систем, включающие вопросы организации ввода-вывода. После этого рассматривается буферизация — один из основных предоставляемых операционной системой сервисов ввода-вывода, повышающих общую производительность системы.

Остальные разделы главы посвящены операциям ввода-вывода, связанным с магнитным диском. В современных системах этот вид операций — самый важный, а с пользовательской точки зрения, он является ключом к повышению производительности. Мы начнем с описания разработки модели производительности дисковых операций ввода-вывода, а затем рассмотрим несколько методов повышения этой производительности.

В приложении К, “Дисковые устройства хранения”, представлены характеристики внешних запоминающих устройств, включая магнитный диск и оптические устройства памяти.

## 11.1. УСТРОЙСТВА ВВОДА-ВЫВОДА

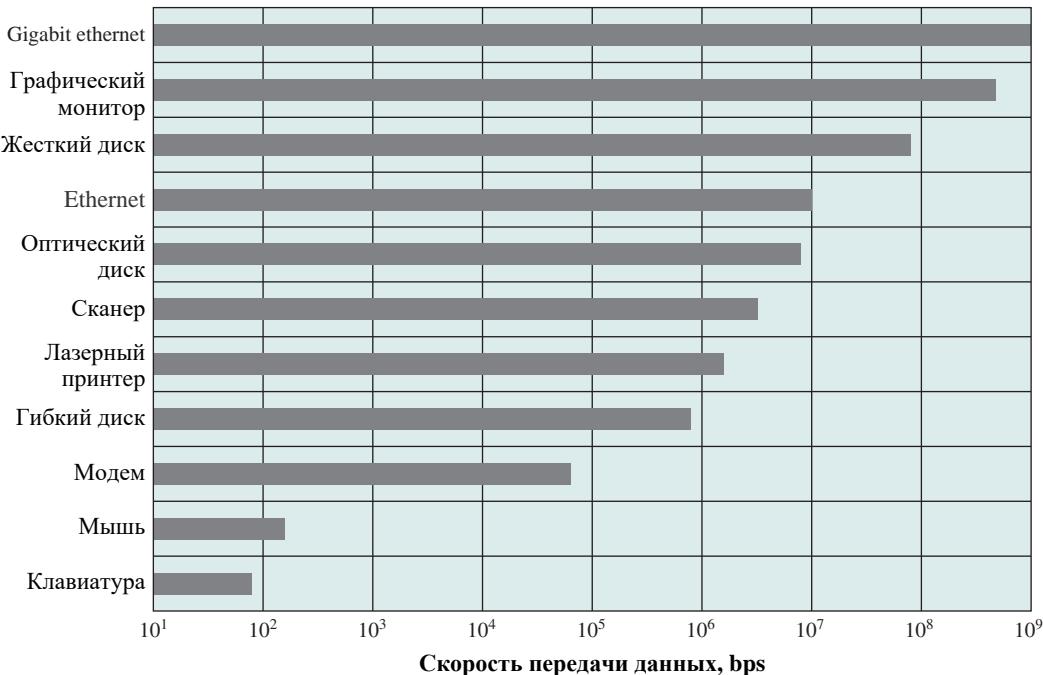
Как упоминалось в главе 1, “Обзор компьютерной системы”, внешние устройства, обеспечивающие операции ввода-вывода в компьютерных системах, в первом приближении могут быть разделены на три группы.

1. **Работающие с пользователем.** Используются для связи с пользователем компьютера. В качестве примера можно привести принтеры и видеотерминалы, состоящие из дисплея, клавиатуры, а также другие устройства, например манипулятор “мышь”.

2. **Работающие с компьютером.** Используются для связи с электронным оборудованием. К ним можно отнести дисковые устройства, USB-ключи, датчики, контроллеры и преобразователи.
3. **Коммуникации.** Используются для связи с удаленными устройствами. К ним относятся модемы и драйверы цифровых линий.

Имеются существенные различия как между устройствами ввода-вывода, как принадлежащими к разным классам, так и в рамках каждого класса. Отметим следующие из этих различий.

- **Скорость передачи данных.** Скорость передачи данных может различаться на несколько порядков (рис. 11.1).
- **Применение.** Каждое действие, поддерживаемое устройством, оказывает влияние на программное обеспечение и стратегии операционной системы. Так, например, использующийся для хранения файлов диск требует программного обеспечения для управления файлами. Диск, используемый в качестве внешнего запоминающего устройства для страниц виртуальной памяти, зависит от программных и аппаратных средств виртуальной памяти. Кроме того, данные приложения оказывают воздействие и на алгоритмы дискового планирования (этот вопрос рассматривается в настоящей главе ниже). В качестве еще одного примера можно привести терминал, который может использоваться как обычным пользователем, так и системным администратором; при этом требуются не только различные уровни привилегий, но и, вероятно, различные уровни приоритетов операционной системы.



**Рис. 11.1.** Скорость передачи данных у типичных устройств ввода-вывода

- **Сложность управления.** Для принтера требуется относительно простой интерфейс управления, диску же необходим намного более сложный интерфейс. Влияние этих различий на операционную систему сглаживается усложнением контроллеров ввода-вывода.
- **Единицы передачи данных.** Данные могут передаваться как поток байтов или символов (например, при терминальном вводе-выводе), и блоками (например, при выполнении дисковых операций ввода-вывода).
- **Представление данных.** Различные устройства используют разные схемы кодирования данных, включая разную кодировку символов и контроль четности.
- **Условия ошибок.** Природа ошибок, способ сообщения о них, их последствия и возможные ответы резко различаются при переходе от одного устройства к другому.

Такое разнообразие приводит тому, что, по сути, невозможна разработка единого и согласованного подхода к проблеме ввода-вывода как с точки зрения операционной системы, так и с точки зрения пользовательских процессов.

## 11.2. ОРГАНИЗАЦИЯ ФУНКЦИЙ ВВОДА-ВЫВОДА

В приложении В, “Дополнительные вопросы параллельности”, рассмотрены три способа ввода-вывода.

1. **Программируемый ввод-вывод.** Процессор посыпает необходимые команды контроллеру ввода-вывода; после этого процесс находится в состоянии ожидания завершения операции ввода-вывода.
2. **Ввод-вывод, управляемый прерываниями.** Процессор посыпает необходимые команды контроллеру ввода-вывода от имени процесса. В этом случае есть две возможности. Если команда ввода-вывода из процесса не блокирующая, процессор продолжает выполнять команды процесса, который выдал команду ввода-вывода. Если же команда ввода-вывода процесса блокирующая, то очередная выполняемая процессором команда поступает от операционной системы, которая помешает текущий процесс в заблокированное состояние и планирует работу другого процесса.
3. **Прямой доступ к памяти (Direct Memory Access — DMA).** Модуль прямого доступа к памяти управляет обменом данных между основной памятью и контроллером ввода-вывода. Процессор посыпает запрос на передачу блока данных модулю прямого доступа к памяти, а прерывание происходит только после передачи всего блока данных.

В табл. 11.1 показана связь между перечисленными способами. В большинстве компьютерных систем основным способом передачи данных, поддерживаемым операционной системой, является прямой доступ к памяти.

ТАБЛИЦА 11.1. Способы ввода-вывода

	Без использования прерываний	С использованием прерываний
<b>Передача данных из устройства ввода-вывода в память с использованием процессора</b>	Программируемый ввод-вывод	Ввод-вывод, управляемый прерыванием
<b>Прямая передача данных из устройства ввода-вывода в память</b>		Прямой доступ к памяти (DMA)

## ЭВОЛЮЦИЯ ФУНКЦИЙ ВВОДА-ВЫВОДА

Параллельно с развитием компьютерных систем возрастают сложность и интеллектуальность их отдельных компонентов, что нигде не очевидно так, как в области ввода-вывода. Этапы развития функциональности устройств ввода-вывода можно охарактеризовать следующим образом.

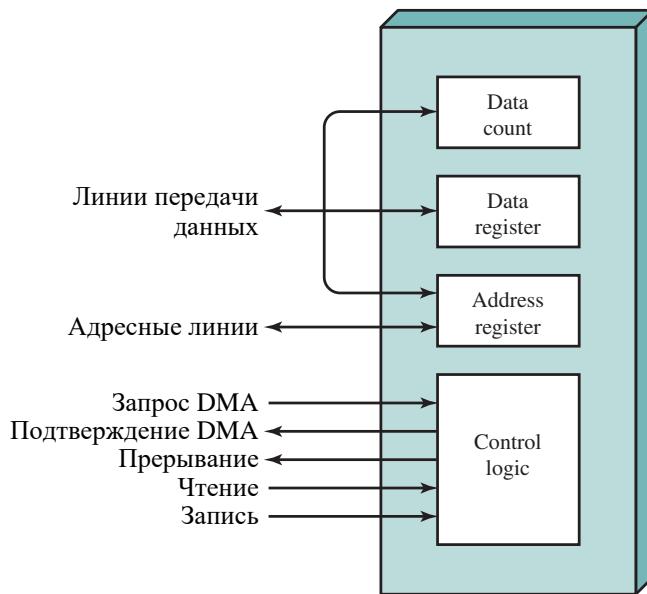
1. Процессор непосредственно управляет периферийным устройством. Это можно увидеть в простых устройствах, контролируемых процессорами.
2. К устройству добавляется контроллер или модуль ввода-вывода. Процессор использует программируемый ввод-вывод без прерываний. На этом этапе процессор становится в некоторой степени отделенным от конкретных деталей интерфейсов внешних устройств.
3. Применяется та же конфигурация, что и в п. 2, только с использованием прерываний. В результате процессору нет необходимости расходовать время на ожидание выполнения операций ввода-вывода, что приводит к увеличению производительности.
4. Модуль ввода-вывода получает возможность непосредственной работы с памятью с использованием DMA. Появляется возможность перемещения блока данных в память или из нее без использования процессора (за исключением моментов начала и окончания передачи данных).
5. Модуль ввода-вывода совершенствуется и становится отдельным процессором, обладающим специализированной системой команд, предназначенных для ввода-вывода. Центральный процессор дает задание процессору ввода-вывода выполнить программу ввода-вывода, находящуюся в основной памяти. Процессор ввода-вывода производит выборку и выполнение соответствующих команд без участия центрального процессора. Это позволяет центральному процессору определить последовательность выполняемых функций ввода-вывода и быть прерваным только при выполнении всей последовательности.
6. Модуль ввода-вывода обладает своей локальной памятью и является, по сути, отдельным компьютером. При такой архитектуре управление многочисленными устройствами ввода-вывода может осуществляться при минимальном вмешательстве центрального процессора. Обычно такая архитектура используется для управления связью с интерактивными терминалами. Процессор ввода-вывода берет на себя большинство задач, связанных с управлением терминалами.

Если проследить описанный выше путь развития устройств ввода-вывода, то можно заметить, что вмешательство процессора в функции ввода-вывода становится все менее и менее заметным. Центральный процессор все больше и больше освобождается от задач, связанных с вводом-выводом, что приводит к повышению общей производительности. Этапы 5 и 6 отражают изменение концепции устройства ввода-вывода — отныне оно способно к самостоятельному выполнению программы.

Обратите внимание на терминологию. Для всех модулей, описанных в пп. 4–6, вполне применим термин *прямой доступ к памяти*, поскольку каждый из них использует непосредственное управление основной памятью модулем ввода-вывода. Модуль ввода-вывода, описанный в п. 5, часто называется также **каналом ввода-вывода**, а модуль, описанный в п. 6, — **процессором ввода-вывода**. Впрочем, иногда в литературе каждый из этих терминов используется и для описания другого типа устройств. В оставшейся части главы мы используем термин *канал ввода-вывода* для обоих типов модулей ввода-вывода.

## Прямой доступ к памяти

На рис. 11.2 представлена обобщенная логическая схема прямого доступа к памяти. Устройство прямого доступа к памяти способно дублировать функции процессора, в частности получать от процессора управление системой. Эта возможность необходима ему для передачи данных по системной шине — как в память, так и из нее.



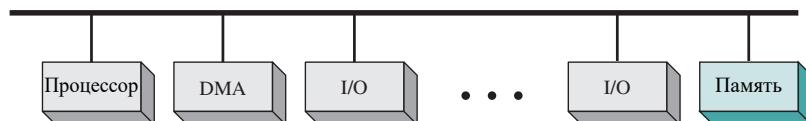
**Рис. 11.2.** Блок-схема прямого доступа к памяти

Рассмотрим работу схемы прямого доступа к памяти. В тот момент, когда процессору необходимо произвести считывание или запись блока данных, он выполняет запрос к модулю DMA, передавая ему следующую информацию.

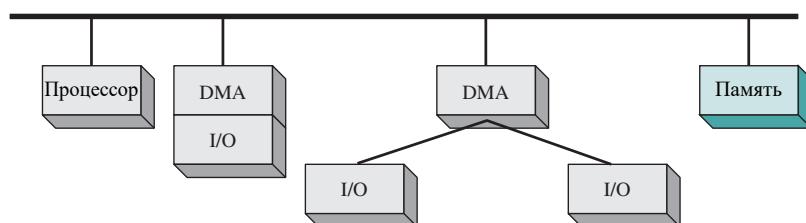
- Какая из операций — чтения или записи — запрашивается. В зависимости от этого будет использоваться либо управляющая линия чтения, либо записи между процессором и модулем DMA.
- Адрес используемого устройства ввода-вывода, подключенного к данным.
- Начальный адрес считываемой (или записываемой) области памяти, хранящийся в адресном регистре модуля DMA.
- Какое количество слов необходимо прочесть или записать. Эта величина хранится в регистре счетчика данных модуля DMA.

После этого процессор продолжает свою работу с другим заданием, передав управление операцией ввода-вывода модулю DMA. В свою очередь, модуль DMA, минуя процессор, передает весь блок данных непосредственно в память (или считывает данные из нее). После выполнения передачи данных модуль DMA посылает процессору сигнал прерывания. Таким образом, процессор включается в этот процесс лишь в начале и в конце передачи данных.

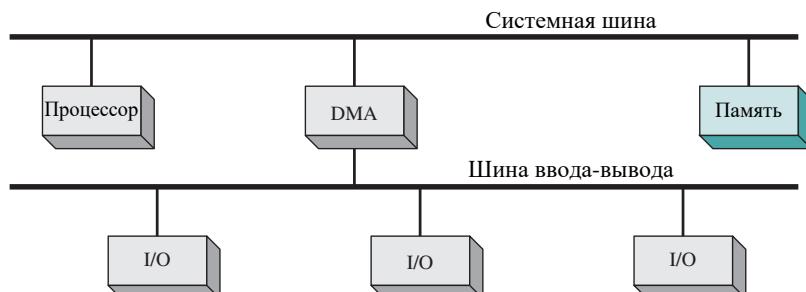
Конфигурирование прямого доступа к памяти может быть выполнено различными способами; некоторые из них представлены на рис. 11.3.



а) Одиночная шина, обособленный DMA



б) Одиночная шина, интегрированный DMA-I/O



в) Шина ввода-вывода

**Рис. 11.3.** Альтернативные конфигурации прямого доступа к памяти

В первом примере все модули подключены к одной и той же системной шине. Модуль DMA, выступающий в качестве дублера процессора, использует программируемый ввод-вывод для обмена данными между памятью и устройством ввода-вывода с участием модуля DMA. Несмотря на достоинство такой конфигурации, заключающееся в относительной дешевизне, она малоэффективна. Поскольку используется программируемый ввод-вывод под управлением процессора, на передачу каждого слова затрачиваются два цикла шины (после запроса на передачу следует передача данных).

Число необходимых циклов шины может быть в значительной степени уменьшено путем интегрирования DMA и функций ввода-вывода. При этом подразумевается (как показано на рис. 11.3, б) наличие магистрали между модулем DMA и одним или несколькими устройствами ввода-вывода без подключения системной шины. Логический узел DMA на самом деле может быть как частью модуля ввода-вывода, так и отдельным модулем, контролирующим один или несколько устройств ввода-вывода. Эту идею можно развивать путем добавления модулей ввода-вывода к модулю DMA с использованием шины ввода-вывода (рис. 11.3, в). Такая схема позволяет уменьшить количество интерфейсов ввода-вывода в модуле DMA до одного и предоставляет возможность легкого расширения данной конфигурации. Во всех представленных случаях (рис. 11.3, б и в) совместно используемая модулем DMA, процессором и основной памятью системная шина используется модулем DMA только для обмена данными с памятью и обмена управляющими сигналами с процессором. Обмен данными между DMA и модулями ввода-вывода происходит вне системной шины.

## 11.3. ВОПРОСЫ ПРОЕКТИРОВАНИЯ ОПЕРАЦИОННЫХ СИСТЕМ

### Цели проектирования

При проектировании средства ввода-вывода разработчики руководствуются двумя целями: достижением высокой эффективности и универсальности. Эффективность очень важна в силу того, что операции ввода-вывода часто способствуют возникновению "заторов" в компьютерной системе. Взгляните еще раз на рис. 11.1: большинство устройств ввода-вывода работают по сравнению с основной памятью и процессором чрезвычайно медленно. Одним из способов решения этой проблемы является многозадачный режим, который позволяет процессору во время выполнения операций ввода-вывода одного процесса работать над выполнением других. Однако даже при наличии большого объема основной памяти в современных компьютерах зачастую будет возникать ситуация, когда операции ввода-вывода отстают от процессора. Для сохранения высокой загруженности процессора применяется подкачка, которая загружает в память готовые к выполнению процессы, но сама она представляет собой не что иное, как операцию ввода-вывода. Таким образом, основное внимание при создании операционной системы направлено на поиск эффективной схемы выполнения операций ввода-вывода. В силу своей важности дисковые операции ввода-вывода являются областью, заслуживающей особого внимания, и большая часть этой главы посвящена именно изучению производительности дисковых операций ввода-вывода.

Другой важной целью является **универсальность**. Чтобы упростить работу с устройствами ввода-вывода и снизить вероятность возникновения ошибок, желательно иметь возможность одинакового управления различными устройствами. Это относится к управлению устройствами ввода-вывода со стороны как пользовательских процессов, так и операционной системы. Достичь реальной универсальности на практике, по сути, невозможно в силу огромного разнообразия характеристик устройств ввода-вывода, и самое большее, что можно сделать в подобной ситуации, — это применить модульный подход при разработке функций ввода-вывода. Такой подход позволяет скрыть большинство низкоуровневых деталей устройства ввода-вывода, так что пользовательские процессы и высокуюровневые операции операционной системы обращаются к устройству только в терминах общих функций, таких как чтение и запись, открытие и закрытие, блокирование и разблокирование. Именно этот подход и будет рассмотрен в данной книге.

## Логическая структура функций ввода-вывода

В главе 2, “Обзор операционных систем”, при обсуждении структуры системы мы придавали особое значение иерархической природе современных операционных систем. Философия иерархии состоит в том, что функции операционной системы следует разделять в соответствии с их сложностью, характерной временной шкалой и уровнем абстракции. Применение этой философии к возможностям ввода-вывода приводит к типу организации, представленной на рис. 11.4. Детали организации будут зависеть от типа устройства и его применения. На рисунке представлены три наиболее важные логические структуры. Естественно, что конкретная операционная система может и не соответствовать в точности этим структурам, но общие положения остаются справедливыми в любом случае, и большинство операционных систем используют ввод-вывод приблизительно таким образом.

Рассмотрим сначала самый простой случай, когда локальное периферийное устройство осуществляет связь посредством потока байтов или записей (рис. 11.4, а). В этом случае уровни будут следующими.

- **Логический ввод-вывод.** Модуль логического ввода-вывода обращается с устройством как с логическим ресурсом и не обращает внимания на детали фактического управления устройством. Логический модуль ввода-вывода работает посредником между пользовательскими процессами и устройством, позволяя им работать с последним с использованием идентификатора устройства и простых команд, таких как открытие, закрытие, чтение и запись.
- **Устройство ввода-вывода.** Запрошенные операции и данные (буферизированные символы, записи и т.п.) конвертируются в соответствующие последовательности инструкций ввода-вывода, команд управления каналом и команд контроллера. Для более эффективного использования устройства может быть применена буферизация.
- **Планирование и контроль.** На этом уровне происходят реальная организация очередей и планирование операций ввода-вывода, а также управление выполнением операций. Таким образом, на этом уровне осуществляются работа с прерываниями, получение и передача информации о состоянии устройства. Это уровень программного обеспечения, которое непосредственно взаимодействует с контроллером ввода-вывода, а следовательно, с аппаратным обеспечением устройства.

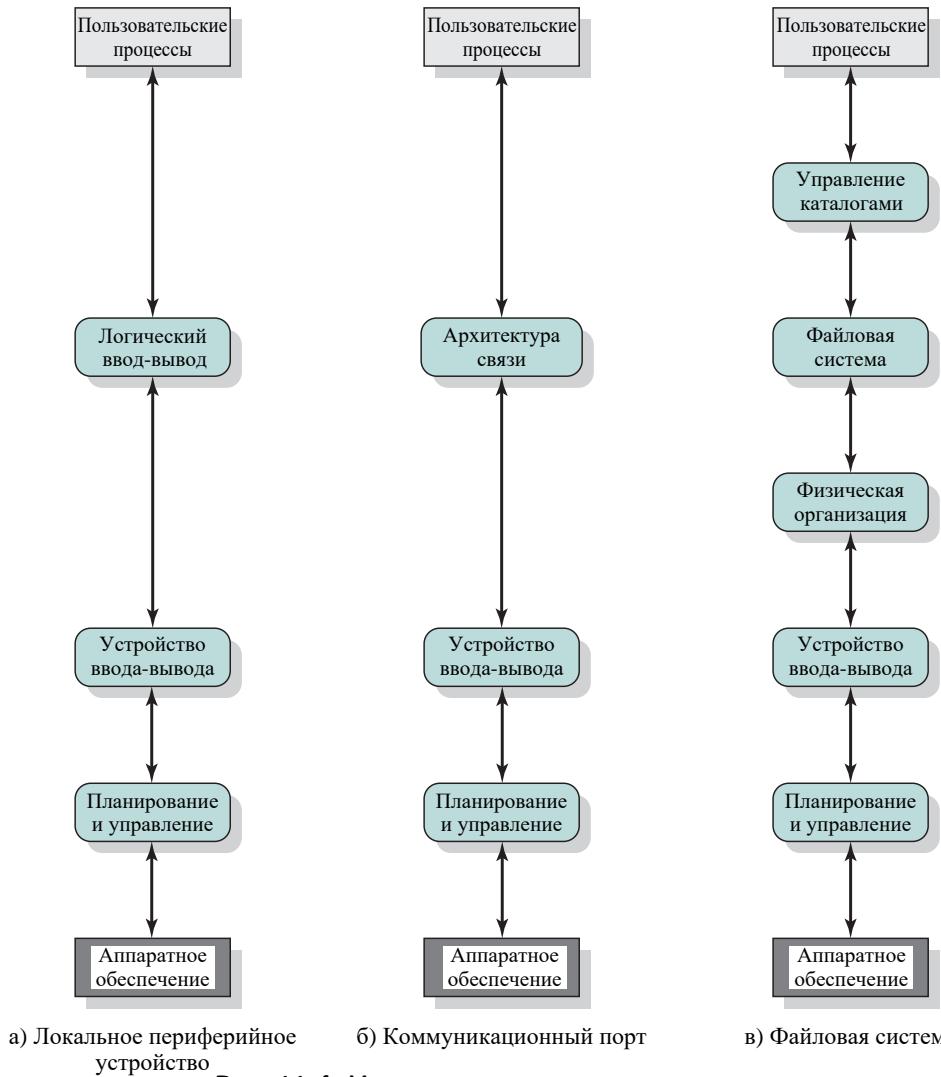


Рис. 11.4. Модель организации ввода-вывода

Для устройств связи структура ввода-вывода (рис. 11.4, б) выглядит почти так же, как и только что описанная. Принципиальное отличие состоит в том, что логический модуль ввода-вывода заменяется коммуникационной архитектурой, которая, в свою очередь, может состоять из некоторого количества уровней. Примером является протокол TCP/IP, рассматриваемый в главе 17, “Сетевые протоколы”.

На рис. 11.4, в представлена характерная структура управления вводом-выводом во вторичное запоминающее устройство, поддерживающее файловую систему. Здесь имеется три уровня, с которыми мы не сталкивались ранее.

1. **Управление каталогами.** На этом уровне происходит преобразование символьных имен файлов в идентификаторы, указывающие на файл, — непосредственно или косвенно, с использованием файлового дескриптора или индексной табли-

цы. Этот уровень связан и с такими пользовательскими операциями с каталогами файлов, как их добавление, удаление или реорганизация.

2. **Файловая система.** Этот уровень работает с логической структурой файлов и операциями, указываемыми пользователями, такими как открытие, закрытие, чтение и запись. Кроме того, управление правами доступа также происходит на этом уровне.
3. **Физическая организация.** Как адреса виртуальной памяти должны быть преобразованы в физические адреса основной памяти с учетом сегментации и страничной организации, так и логические ссылки на файлы и записи должны быть конвертированы в физические адреса конкретного вторичного запоминающего устройства с учетом физической структуры дорожек и секторов этого устройства. На этом же уровне происходит общее управление распределением пространства внешней памяти и буферов основной памяти.

В силу важности файловой системы в этой и следующей главах мы остановимся на рассмотрении ее различных компонентов. В данной главе обсуждаются три нижних уровня; два верхних уровня рассматриваются в главе 12, “Управление файлами”.

## 11.4. БУФЕРИЗАЦИЯ ОПЕРАЦИЙ ВВОДА-ВЫВОДА

Предположим, что пользовательскому процессу необходимо выполнить считывание блоков данных длиной по 512 байт по одному с диска. Данные будут считаны в область внутри адресного пространства пользовательского процесса с виртуальным адресом от 1000 до 1511. Наиболее простой путь решения этой задачи — выполнение команды ввода-вывода (что-то наподобие `Read_Block[1000, disk]`) для диска и ожидание того момента, когда данные станут доступными. Ожидание может быть активным, т.е. будет происходить непрерывное тестирование состояния устройства, либо, что более практически, процесс будет приостановлен до прерывания.

При таком подходе имеются две проблемы. Первая представляет собой приостановку программы для ожидания выполнения относительно медленного ввода-вывода. Вторая проблема состоит в том, что такой подход к вводу-выводу мешает выполнению свопинга операционной системой. Виртуальные адреса с 1000 по 1511 должны находиться в основной памяти при считывании блока (в противном случае часть данных будет потеряна). При использовании страничной организации памяти по крайней мере одна страница (содержащая целевой адрес) должна быть заблокирована в основной памяти. Поэтому, несмотря на то что часть задания может быть выгружена на диск, полный свопинг процесса окажется невозможным, даже если это необходимо для операционной системы. Следует также учесть возможность взаимоблокировки. При выполнении процессом команды ввода-вывода он приостанавливается и выгружается на диск до начала выполнения операции ввода-вывода. Далее процесс ожидает, когда будет выполнена запрошеннная им операция ввода-вывода, которая, в свою очередь, ожидает, когда процесс будет возвращен в основную память, поскольку место в основной памяти для считывания данных попросту отсутствует. Для того чтобы избежать взаимоблокировки, пользовательская память, вовлеченная в операцию ввода-вывода, должна быть заблокирована в основной памяти сразу же после выдачи запроса на ввод-вывод, даже если операция ввода-вывода ставится в очередь и может быть выполнена только через некоторое время.

То же рассуждение применимо и к операции вывода. Если блок пересыпается из адресного пространства пользовательского процесса в модуль ввода-вывода, то на время этой передачи процесс блокируется и не может быть выгружен на диск.

Чтобы уменьшить накладные расходы и увеличить эффективность, иногда удобно выполнить чтение данных заранее, до реального запроса (а запись данных — немного позже реального запроса). Эта технология известна как буферизация. В данном разделе мы рассмотрим некоторые схемы буферизации, поддерживаемые операционными системами для повышения производительности.

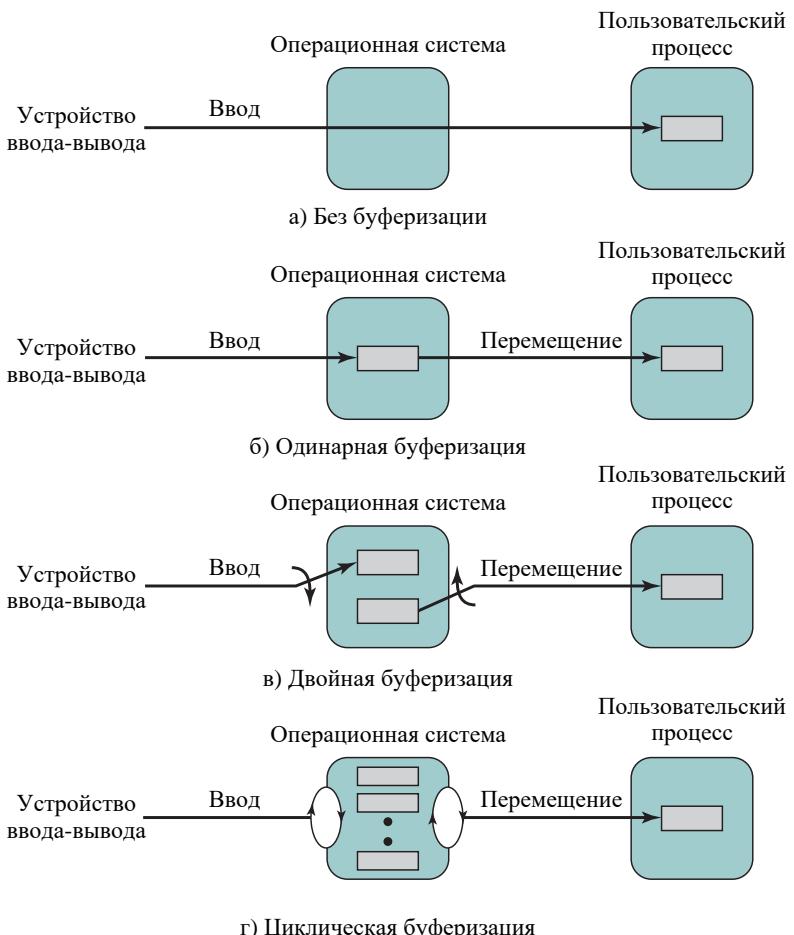
При рассмотрении различных методов буферизации важно учитывать, что существуют устройства ввода-вывода двух типов: блочно-ориентированные и поточно-ориентированные. **Блочно-ориентированные устройства** сохраняют информацию блоками, обычно фиксированного размера, и выполняют передачу данных поблочно. Как правило, при этом можно ссылаться на данные с использованием номера блока. Диски и USB-ключи являются примерами блочно-ориентированных устройств ввода-вывода. **Поточно-ориентированные устройства** выполняют передачу данных в виде неструктурированных потоков байтов, без блочной структуры. К этой группе устройств относятся терминалы, принтеры, коммуникационные порты, манипулятор “мышь” и другие устройства указания, а также большинство устройств, не являющихся внешними запоминающими устройствами.

Простейшим типом поддержки со стороны операционной системы является одинарный буфер (рис. 11.5, б). В тот момент, когда пользовательский процесс выполняет запрос ввода-вывода, операционная система назначает ему буфер в системной части основной памяти.

Схема одинарного буфера для блочно-ориентированных устройств может быть описана следующим образом. Сначала осуществляется передача входных данных в системный буфер. Когда она завершается, процесс перемещает блок в пользовательское пространство и немедленно производит запрос следующего блока. Такая процедура называется опережающим считыванием, или упреждающим вводом; она выполняется в предположении, что этот блок со временем будет затребован. Для многих типов задач этот метод в большинстве случаев неплохо работает, поскольку доступ к данным обычно осуществляется последовательно (только при окончании последовательности обработки считывание блока будет излишним).

В общем случае такой подход, по сравнению с отсутствием буферизации, обеспечивает повышение быстродействия. Пользовательский процесс может обрабатывать один блок данных в то время, когда происходит считывание следующего блока. Операционная система при этом может осуществлять выгрузку процесса, поскольку операция считывания данных выполняется в системную память, а не в память пользовательского процесса. Однако такая технология усложняет логику операционной системы, которая должна следить за назначением системных буферов пользовательским процессам. Влияет буферизация и на схему подкачки: когда операция ввода-вывода работает с тем же диском, который используется и для свопинга, теряется смысл в организации очереди операций записи. Выгрузка процесса и освобождение основной памяти не начнется до тех пор, пока не завершится запрошенная операция ввода-вывода, а тогда выгрузка процесса больше не будет иметь смысла.

Похожие рассуждения применимы и к блочно-ориентированному выводу. Если данные передаются на устройство, то сначала они копируются из пользовательского пространства в системный буфер, из которого они в конечном счете будут записаны на устройство. В этой ситуации выводящий данные процесс может продолжать работу сразу же после передачи данных в системный буфер.



**Рис. 11.5.** Схемы буферизации ввода-вывода (ввод)

В книге [136] приводится грубое, но очень показательное сравнение процессов при использовании одинарной буферизации и при ее отсутствии. Предположим, что  $T$  — это время, необходимое для ввода одного блока, а  $C$  — для вычислений, выполняющихся между запросами на ввод данных. Без буферизации общее время выполнения, приходящееся на один блок, по сути, оказывается равным  $T + C$ . При использовании одинарной буферизации время выполнения равно  $\max[C, T] + M$ , где  $M$  — время, необходимое для перемещения данных из системного буфера в пользовательскую память. В большинстве случаев это время значительно меньше времени работы без буферизации.

Схема одинарного буфера может быть применена и при поточно-ориентированном вводе-выводе — построчно или побайтово. Построчная буферизация применима, например, в неинтеллектуальных терминалах, где пользователь вводит данные построчно, завершая строки символом возврата каретки, сигнализируя этим об окончании строки; вывод на терминал происходит таким же образом — построчно. Другим примером может служить строчный принтер. Побайтовые операции применяются при использовании терминалов в режиме форм, а также многих других периферийных устройств, когда каждое нажатие клавиши является значимым.

В случае построчного ввода-вывода буфер может быть использован для хранения одной строки. Пользовательский процесс приостанавливается на время ввода, ожидая поступления целой строки. При операции вывода пользовательский процесс может разместить строку в буфере и продолжить работу. Необходимость приостановления этого процесса возникает только в том случае, если требуется вывод второй строки, в то время как первая еще не покинула буфер. При побайтовом вводе-выводе взаимодействие операционной системы и пользовательского процесса следует модели производителя/потребителя, рассмотренной в главе 5, “Параллельные вычисления: взаимоисключения и многозадачность”.

## Двойной буфер

Улучшить схему одинарной буферизации можно путем использования двух системных буферов (рис. 11.5, в). Теперь процесс выполняет передачу данных в один буфер (или считывание из него), в то время как операционная система освобождает (или заполняет) другой. Эта технология известна как **двойная буферизация** или **обмен буферов**.

Время выполнения при блочно-ориентированной передаче данных можно грубо оценить как  $\max[C, T]$ . Таким образом, если  $C \leq T$ , то блочно-ориентированное устройство может работать с максимальной скоростью. Если же  $C > T$ , то двойная буферизация гарантирует отсутствие простоя процесса в ожидании завершения ввода-вывода. В любом случае достигается преимущество перед одинарной буферизацией. Это улучшение буферизации осуществляется за счет увеличения ее сложности.

При поточно-ориентированном вводе мы снова обращаемся к двум альтернативным режимам работы. Необходимость приостановления процесса при построчном выводе возникает только в том случае, если при выводе очередной строки оба буфера не пусты. При побайтовых операциях двойной буфер не имеет никакого преимущества перед одинарным буфером двойной длины. В обоих случаях используется модель производителя/потребителя.

## Циклический буфер

Схема двойного буфера призвана выровнять поток данных между устройством ввода-вывода и процессом. Если нас интересует производительность некоторого процесса, то, в первую очередь, требуется, чтобы операции ввода-вывода не тормозили его работу. Двойная буферизация может оказаться недостаточной, если процесс часто выполняет ввод или вывод. Зачастую в таком случае решить проблему помогает увеличение количества буферов.

При использовании более двух буферов схема именуется циклической буферизацией (рис. 11.5, г). В ней каждый индивидуальный буфер представляет собой модуль циклического буфера. Такая буферизация описывается моделью производителя/потребителя с ограниченным буфером, которая рассматривалась в главе 5, “Параллельные вычисления: взаимоисключения и многозадачность”.

## Использование буферизации

Буферизация представляет собой метод сглаживания всплесков количества запросов ввода-вывода. Однако никакое количество буферов не позволит устройству ввода-вывода работать наравне с процессом в течение неограниченного времени в ситуации, когда

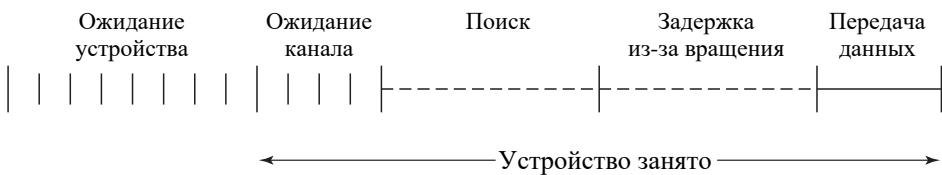
средняя скорость запросов процесса превышает возможности их обработки устройством ввода-вывода. Даже при наличии большого количества буферов в конечном счете все они будут заполнены, и процесс будет вынужден приостановиться в ожидании обработки порции данных устройством ввода-вывода. Однако в многозадачной среде при наличии разнообразных процессов с запросами ввода-вывода и такого же разнообразия устройств буферизация оказывается инструментом, способным увеличить как производительность операционной системы в целом, так и производительность отдельных процессов.

## 11.5. ДИСКОВОЕ ПЛАНИРОВАНИЕ

На протяжении последних 40 лет увеличение скорости процессоров и основной памяти осуществляется с большим опережением по сравнению со скоростью доступа к диску. Приблизительно можно сказать, что рост скорости работы процессора и основной памяти на два порядка соответствует росту скорости работы диска на один порядок. В результате скорость обращения к дискам сейчас как минимум на четыре порядка меньше скорости обращения к основной памяти, и разрыв этот, похоже, в обозримом будущем будет только увеличиваться. Поэтому производительность дисковой системы является жизненно важным вопросом, и множество исследовательских работ направлено на поиск схем ее улучшения. В этом разделе мы рассмотрим некоторые ключевые вопросы и наиболее важные подходы в этой области. Поскольку производительность дисковой системы тесно связана с вопросами файловой системы, рассмотрение этой темы продолжится в главе 12, “Управление файлами”.

### Параметры производительности диска

Конкретные детали дисковой операции ввода-вывода зависят от компьютерной системы, операционной системы, природы канала ввода-вывода и аппаратного обеспечения контроллера диска. Обобщенная временная диаграмма передачи данных дисковым устройством ввода-вывода представлена на рис. 11.6.



**Рис. 11.6.** Временная диаграмма работы диска

При работе диска его скорость вращения постоянна. Для того чтобы выполнить чтение или запись, головка должна находиться над искомой дорожкой, точнее — над началом искомого сектора на этой дорожке<sup>1</sup>. Процедура выбора дорожки включает в себя перемещение головки (в системе с подвижными головками) или электронный выбор нужной головки (в системе с неподвижными головками). В системе с подвижными головками на позиционирование головки над дорожкой затрачивается время, известное

<sup>1</sup> Организация и форматирование диска рассматриваются в приложении К, “Дисковые устройства хранения”.

**как время поиска.** В любом случае после выбора дорожки контроллер диска ожидает момента, когда начало искомого сектора достигнет головки. Время, необходимое для достижения головки началом сектора, известно как **время задержки из-за вращения** или **время ожидания вращения**. Сумма времен поиска (если таковой выполняется) и времени задержки из-за вращения составляет **время доступа** — время, которое требуется для позиционирования для чтения или записи. Как только головка попадает в искомую позицию, выполняется операция чтения или записи, осуществляемая во время движения сектора под головкой, — это и есть непосредственная передача данных при выполнении операции ввода-вывода. Время, требуемое для передачи данных, называется **временем передачи**.

Кроме того, существует ряд других задержек, обычно присутствующих в дисковой операции ввода-вывода. Когда процесс выполняет запрос на ввод-вывод, последний должен быть размещен в очереди в ожидании доступности устройства. После этого выполняется назначение устройства процессу. Если устройство использует каналы ввода-вывода совместно с другими дисками, возможно, потребуется дополнительное ожидание доступности канала. И только после этого осуществляется непосредственный доступ к диску, рассмотренный ранее.

В некоторых высокопроизводительных системах используется методика, известная как вращательное позиционное считывание (rotational positional sensing — RPS), работающая следующим образом. При выполнении команды поиска происходит освобождение канала для обработки других операций ввода-вывода. После выполнения поиска устройство определяет момент, когда данные окажутся под головкой. Как только этот сектор подходит к головке, устройство пытается восстановить связь с узлом. Если либо контроллер, либо канал занят другой операцией ввода-вывода, то попытка восстановления связи оказывается неудачной и диск совершает полный оборот перед повторной попыткой. Это дополнительный элемент задержки, который следует добавить к полному времени ожидания на рис. 11.6.

### Время поиска

Время поиска представляет собой время, необходимое для перемещения головки к нужной дорожке; к сожалению, очень трудно установить этот параметр количественно. Время поиска состоит из двух ключевых компонентов: времени начального запуска и времени, необходимого на пересечение дорожек в процессе поиска. К сожалению, время пересечения дорожек не является линейной функцией от их количества, но при этом включает время начальной установки и принятия решения для каждой пересекаемой дорожки (время, прошедшее после установления головки над искомой дорожкой до подтверждения идентификации последней).

Значительно улучшает характеристики диска уменьшение и облегчение его компонентов. Не так давно типичный диск имел в диаметре 14 дюймов (36 см), в то время как сегодня самый распространенный размер — 3,5 дюйма (8,9 см), что существенно уменьшает расстояния перемещения головок. Типичное среднее время поиска в современных дисках составляет менее 10 мс.

### Задержка из-за вращения

Задержка из-за вращения — это время, необходимое для того, чтобы нужная область диска повернулась в положение, в котором она будет доступна для чтения/записи головкой. Диски врачаются со скоростью от 3600 об/мин (для портативных устройств, таких

как цифровые камеры) до (на момент написания книги) 15 000 об/мин; на этой последней скорости один оборот выполняется за 4 мс. Таким образом, в среднем задержка вращения составляет 2 мс.

### Время передачи данных

Время передачи данных на диск или считывания с него зависит от скорости вращения диска следующим образом:

$$T = \frac{b}{rN}$$

где

- $T$  — время передачи данных;
- $b$  — количество передаваемых байтов;
- $N$  — количество байтов в дорожке;
- $r$  — скорость вращения (об/с).

Таким образом, итоговое среднее время доступа можно выразить как

$$T_a = T_s + \frac{1}{2r} + \frac{b}{rN},$$

где  $T_s$  — среднее время поиска.

### Сравнение времени

Рассмотрим две операции ввода-вывода, показывающие, как опасно полагаться на средние значения. Пусть имеется обычный диск со средним временем поиска 4 мс и скоростью вращения 7 500 об/мин, и диск разбит на сектора по 512 байт, с 500 секторами на одной дорожке. Предположим, что необходимо выполнить чтение файла, состоящего из 2500 секторов общим размером 1,28 Мбайт. Мы хотим оценить общее время передачи данных.

Сначала предположим, что файл сохранен настолько компактно, насколько это возможно на диске, т.е. занимает все секторы на 5 соседних дорожках (5 дорожек  $\times$  500 секторов на дорожке = 2500 секторов). Такое размещение называется *последовательной организацией*. В этом случае время, необходимое для чтения первой дорожки, определяется следующим образом:

Среднее время поиска	4 мс
Задержка на вращение	4 мс
Чтение 500 секторов	8 мс
	_____
	16 мс

Предположим, что остальные дорожки могут быть считаны последовательно, без затраты времени на поиск. Другими словами, операция ввода-вывода не отстает от потока данных с диска. Значит, для каждой последующей дорожки остается только задержка из-за вращения; соответственно, каждая дорожка считывается за  $4 + 8 = 12$  мс. Итак, для чтения всего файла потребуется

$$\text{Общее время} = 16 + 4 \times 12 = 64 \text{ мс} = 0,064 \text{ с}$$

Теперь рассчитаем время, необходимое для чтения тех же данных, но при случайному, а не последовательном доступе, т.е. секторы с содержимым файла распределены на диске случайным образом. Тогда для каждого сектора

Среднее время поиска	4 мс
Задержка на вращение	4 мс
Чтение 1 сектора	0,016 мс
	8,016 мс

$$\text{Общее время} = 2500 \times 8,016 = 20\,040 \text{ мс} = 20,04 \text{ с}$$

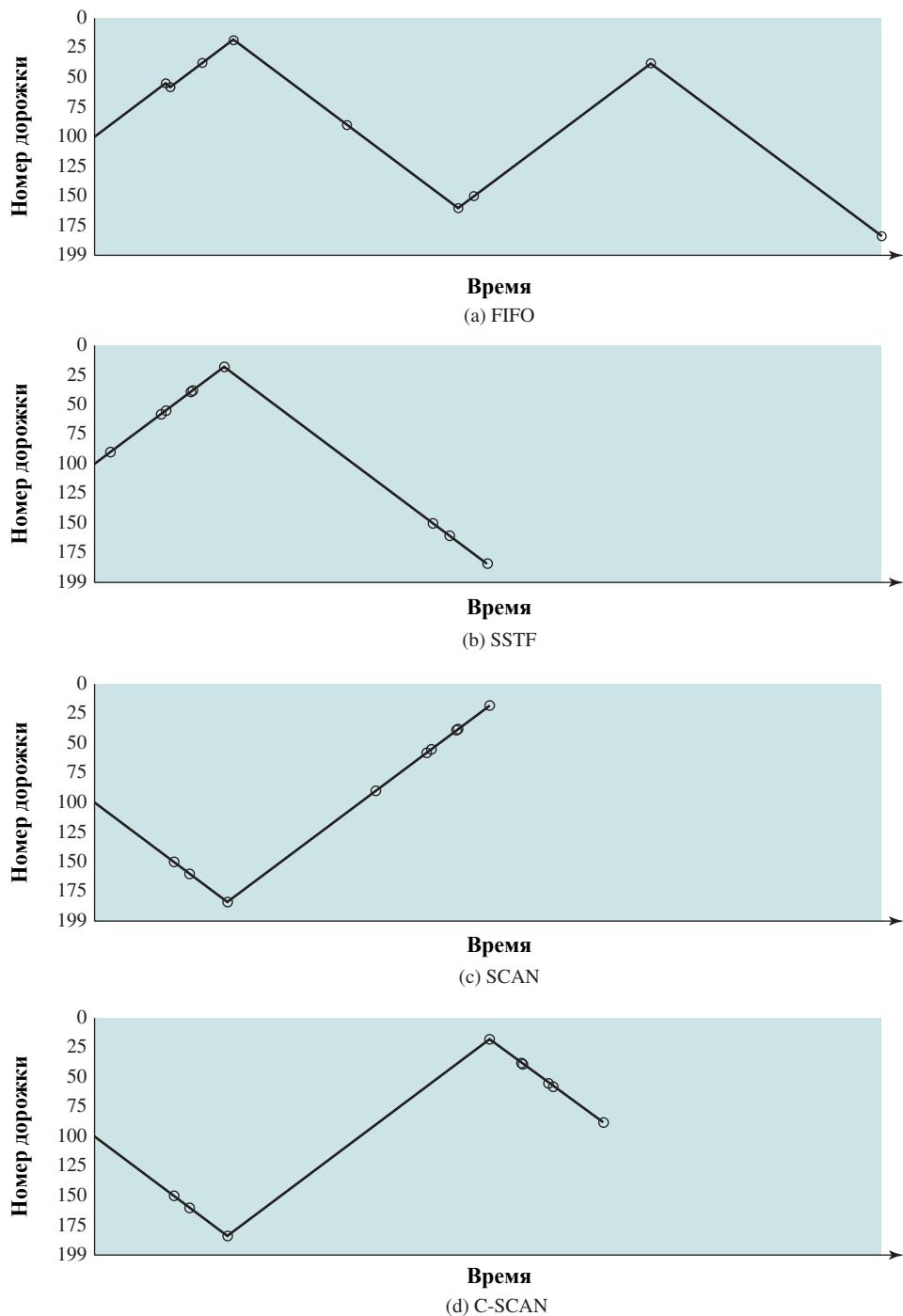
Очевидно, что порядок чтения секторов с диска оказывает огромное влияние на производительность дискового ввода-вывода. Если при доступе к файлу с диска считывается (или записывается на него) несколько секторов, имеется возможность определенного контроля над использованием секторов с данными, но об этом мы поговорим в следующей главе. Однако в многозадачной среде всегда будут в наличии конкурирующие между собой запросы на операции ввода-вывода с одним и тем же диском, так что избежать случайного доступа не удастся. Таким образом, стоит изучить способы повышения производительности дискового ввода-вывода при случайном доступе.

## Стратегии дискового планирования

В только что рассмотренном примере причина разницы в производительности может быть объяснена продолжительностью поиска. Если выполнение обращений к секторам включает выбор дорожек случайным образом, производительность дискового ввода-вывода окажется чрезвычайно низкой. Для ее повышения нам необходимо уменьшить время, затрачиваемое на поиск дорожки.

Рассмотрим типичную ситуацию в многозадачной среде, когда операционная система поддерживает очередь запросов для каждого устройства ввода-вывода. Соответственно, в очереди одного диска будет находиться некоторое количество запросов на ввод-вывод (чтение или запись) от различных процессов. Если выбирать запросы из очереди случайным образом, то следует ожидать, что искомые дорожки будут располагаться в произвольном порядке, и это приведет к очень низкой производительности. Такое **случайное планирование** может служить точкой отсчета для оценки других методик.

На рис. 11.7 сравнивается производительность различных алгоритмов планирования для примера последовательности запросов ввода-вывода. Вертикальная ось соответствует дорожкам диска; горизонтальная ось соответствует времени, или, что эквивалентно, количеству пройденных дорожек. На этом рисунке предполагается, что изначально головка диска расположена на дорожке 100; диск имеет 200 дорожек, а в очереди запросов к диску содержатся случайные запросы (запрошенные дорожки, в порядке поступления планировщику диска — 55, 58, 39, 18, 90, 160, 150, 38, 184). В табл. 11.2 приведены количественные результаты.



**Рис. 11.7.** Сравнение алгоритмов дискового планирования (см. табл. 11.2)

**Таблица 11.2. СРАВНЕНИЕ АЛГОРИТМОВ ДИСКОВОГО ПЛАНИРОВАНИЯ**

а) FIFO (стартовая дорожка 100)		б) SSTF (стартовая дорожка 100)		в) SCAN (стартовая дорожка 100; направление в сторону уменьшения номера дорожки )		г) C-SCAN (стартовая дорожка 100; направление в сторону уменьшения номера дорожки )	
Следующая дорожка	Количество пересеченных дорожек	Следующая дорожка	Количество пересечен- ных дорожек	Следующая дорожка	Количество пересечен- ных дорожек	Следующая дорожка	Количество пересеченных дорожек
55	45	90	10	150	50	150	50
58	3	58	32	160	10	160	10
39	19	55	3	184	24	184	24
18	21	39	16	90	94	18	166
90	72	38	1	58	32	38	20
160	70	18	20	55	3	39	1
150	10	150	132	39	16	55	16
38	112	160	10	38	1	58	3
184	<u>146</u>	184	<u>24</u>	18	<u>20</u>	90	<u>32</u>
<b>Средняя продолжи- тельность поиска</b>		<b>55,3</b>	<b>27,5</b>	<b>Средняя продолжи- тельность поиска</b>	<b>27,8</b>	<b>Средняя продолжи- тельность поиска</b>	<b>35,8</b>

**FIFO**

Самая простая форма планирования — планирование “первым пришел — первым вышел” (FIFO), при котором элементы из очереди обрабатываются последовательно. Преимущество этой стратегии — в справедливости, потому что каждый запрос выполняется, причем все запросы выполняются в порядке получения. На рис. 11.7, а показано движение головки диска при использовании этой стратегии. Этот график получен непосредственно из данных табл. 11.2, а. Как можно видеть, доступ к диску осуществляется в том же порядке, в котором первоначально были получены запросы.

При использовании стратегии FIFO надеяться на высокую производительность можно только при небольшом количестве процессов и запросах в основном к близким группам секторов. Однако при работе большого количества процессов производительность будет почти такой же, как и при случайном планировании. Таким образом, может быть выгодно рассмотреть более сложную стратегию планирования. Некоторые из таких стратегий перечислены в табл. 11.3 и будут вскоре рассмотрены.

**Таблица 11.3. Алгоритмы дискового планирования**

Название	Описание	Примечания
<b>Выбор в соответствии с источником запроса</b>		
Random	Случайное планирование	Для анализа и моделирования
FIFO	“Первым вошел — первым вышел”	Наиболее беспристрастный метод
PRI	Приоритет процесса	Очередь запросов к диску управляется извне
LIFO	“Последним вошел — первым вышел”	Максимизация локальности и использования ресурса
<b>Выбор в соответствии с содержимым запроса</b>		
SSTF	Выбор самого короткого времени обслуживания	Высокая степень использования, малые очереди
SCAN	Перемещение вперед и назад по диску	Лучшее распределение обслуживания
C-SCAN	Однонаправленное перемещение с быстрым возвратом	Низкая изменчивость обслуживания
N-step-SCAN	SCAN с $N$ записями в одном пакете	Гарантия обслуживания
FSCAN	$N$ -step-SCAN, где $N$ — размер очереди в начале цикла SCAN	Чувствительный к загрузке

**Приоритеты**

В системе с использованием приоритета (PRI) управление планированием является внешним по отношению к программному обеспечению управления диском. Такой подход не имеет отношения к оптимизации использования диска, но зато удовлетворяет некоторым другим целям операционной системы. Зачастую коротким пакетным заданиям, а также интерактивным заданиям присваивается более высокий приоритет, чем длинным заданиям, требующим более длительных вычислений. Эта схема позволяет быстро

завершить большое количество коротких заданий в системе и обеспечивает малое время отклика. Однако при использовании этого метода у больших заданий оказывается слишком длительным ожидание выполнения дисковых операций. Кроме того, такая стратегия может привести к противодействию со стороны пользователей, которые будут разделять свои задания на малые подзадания. Не подходит эта стратегия и для работы с базами данных.

### **Последним вошел — первым вышел**

Как это ни удивительно, стратегия выполнения первым последнего запроса имеет свои преимущества. В системах обработки транзакций при предоставлении устройства для последнего пользователя должно выполняться лишь небольшое перемещение указателя последовательного файла. Использование преимуществ локальности позволяет повысить пропускную способность и уменьшить длину очереди. К сожалению, если нагрузка на диск велика, существует очевидная возможность голодаания процесса. Когда задание размещает запрос ввода-вывода в очереди, и оно попадает в ее начало, это задание не сможет продолжать работу, пока вся очередь перед ним не опустеет.

Три рассмотренные стратегии планирования — FIFO, PRI и LIFO — основаны исключительно на атрибутах очереди или запрашивающего процесса. Однако если планировщику известна текущая дорожка, то появляется возможность использования стратегии планирования, основанной на содержимом запроса.

### **SSTF**

Стратегия выбора наименьшего времени обслуживания (Shortest Service Time First — SSTF) заключается в выборе того дискового запроса на ввод-вывод, который требует наименьшего перемещения головок из текущей позиции. Следовательно, мы минимизируем время поиска. Естественно, постоянный выбор минимального времени поиска не дает гарантии, что среднее время поиска при всех перемещениях будет минимальным; тем не менее эта стратегия обеспечивает лучшую по сравнению с FIFO производительность дисковой системы. Поскольку головки могут перемещаться в двух направлениях, при равных расстояниях для принятия решения может быть использован случайный выбор направления.

На рис. 11.7, б и в табл. 11.2, б показана производительность стратегии SSTF для той же последовательности запросов, что и при рассмотрении стратегии FIFO. Первое обращение выполняется к дорожке 90, потому что это самая близкая к начальной позиции запрошенная дорожка. Следующей дорожкой, к которой будет доступ, является 58, потому что она самая близкая из оставшихся запрошенных дорожек к текущей позиции 90. Последующие дорожки выбираются аналогично.

### **SCAN**

Все стратегии, описанные к настоящему времени (за исключением FIFO), могут оставить некоторый запрос невыполненным до тех пор, пока не освободится вся очередь, т.е. в ходе работы всегда могут иметься новые запросы, которые будут выбраны до уже имеющегося в очереди. Избежать такого рода голодаания можно при использовании стратегии SCAN, известной также как алгоритм лифта (Elevator) из-за работы, напоминающей работу лифта.

При использовании этого алгоритма перемещение головки происходит только в одном направлении, удовлетворяя те запросы, которые соответствуют выбранному направ-

лению. После достижения последней дорожки в выбранном направлении (или когда исчерпаются возможные запросы), направление изменится на противоположное. Это последнее уточнение иногда называют стратегией LOOK. Затем направление обслуживания меняется на противоположное, и сканирование продолжается в противоположном направлении, снова собирая все запросы по порядку.

Стратегия SCAN представлена на рис. 11.7, в и в табл. 11.2, в. В предположении, что начальным направлением является увеличение номера дорожки, первая выбранная дорожка равна 150, поскольку эта дорожка — самая близкая к начальной дорожке номер 100 в направлении увеличения.

Как видите, стратегия SCAN ведет себя почти так же, как и стратегия SSTF. Фактически, если предположить, что изначально головка перемещается в сторону меньших номеров дорожек, схема планирования окажется идентичной для SSTF и SCAN. Однако это статический пример, в котором в очередь не добавляется ни один запрос. Однако даже при динамическом изменении очереди стратегии SCAN и SSTF выглядят, как правило, очень похоже.

Обратите внимание, что стратегия SCAN имеет отрицательный “перекос” в отношении недавно пройденной области. Таким образом, данная стратегия использует имеющуюся локальность не так эффективно, как SSTF.

Нетрудно увидеть, что стратегия SCAN оказывает предпочтение тем заданиям, запросы которых относятся к дорожкам, находящимся ближе всего к центру либо наиболее удаленным от него, а также отдает предпочтение запросам, поступившим последними. Первой проблемы можно избежать путем применения стратегии C-SCAN; вторая же проблема решается с помощью стратегии N-step-SCAN.

### C-SCAN

Стратегия C-SCAN (циклическое сканирование) ограничивает сканирование только одним направлением. Когда обнаруживается последняя дорожка в заданном направлении, головка возвращается в противоположный конец диска, и сканирование начинается снова. Это уменьшает максимальную задержку, вызванную новыми запросами. Если при использовании стратегии SCAN ожидаемое время сканирования от внутренней к внешней дорожке равно  $t$ , то ожидаемый интервал обслуживания секторов, находящихся на периферии, будет равен  $2t$ . При использовании стратегии C-SCAN этот интервал будет порядка  $t+s_{\max}$ , где  $s_{\max}$  — максимальное время поиска.

Поведение стратегии C-SCAN показано на рис. 11.7, г и в табл. 11.2, г.

В этом случае первые три запрошенные дорожки — 150, 160 и 184. Затем начинается сканирование с самого малого номера дорожки, так что следующая запрошенная дорожка — 18.

### N-step-SCAN и FSCAN

При использовании стратегий SSTF, SCAN и C-SCAN может возникнуть ситуация, когда один или несколько процессов с высокой частотой обращений к одной дорожке монополизируют устройство за счет многочисленных повторений запросов к одной и той же дорожке. Наиболее характерна эта особенность для многоповерхностных дисков с большой плотностью записи. Для предотвращения такого “залипания головки” очередь дисковых запросов может быть сегментирована, причем за один прием полностью выполняется весь сегмент заданий. Примерами такого подхода являются стратегии N-step-SCAN и FSCAN.

Стратегия  $N$ -step-SCAN разделяет очередь дисковых запросов на подочереди длиной  $N$ . Каждая подочередь обрабатывается за один прием с использованием стратегии SCAN. В ходе обработки очереди к некоторой другой очереди могут добавляться новые запросы. Если в конце текущего сканирования доступными оказываются менее  $N$  запросов, то все они обрабатываются в следующем цикле сканирования. При больших значениях  $N$  производительность алгоритма  $N$ -step-SCAN достигает таковой у алгоритма SCAN; предельный случай  $N = 1$  соответствует стратегии FIFO.

FSCAN — стратегия, использующая две подочереди. С началом сканирования все запросы находятся в одной из очередей; другая при этом остается пустой. Во время сканирования первой очереди все новые запросы попадают во вторую очередь. Таким образом, обслуживание новых очередей откладывается, пока не будут обработаны все старые запросы.

## 11.6. RAID

Как упоминалось ранее, рост производительности вторичных запоминающих устройств значительно отстает от роста производительности процессоров и основной памяти. Такое несоответствие вынуждает обращать особое внимание на диковую систему при повышении уровня общей производительности.

Как и в других областях, дополнительное повышение эффективности может быть достигнуто путем параллельного использования нескольких устройств. В случае с дисками это означает использование массивов независимо и параллельно работающих дисков. При наличии множества дисков различные запросы ввода-вывода могут обрабатываться параллельно, если блок данных, к которому производится обращение, распределен по множеству дисков. Кроме того, даже единственный запрос ввода-вывода может быть выполнен параллельно, если блок данных, к которому осуществляется доступ, распределен по нескольким дискам.

В случае применения нескольких дисков имеется большое количество вариантов организации данных и добавления избыточности для повышения надежности (что может создать трудности при разработке схем баз данных, способных работать на разных платформах под управлением разных операционных систем). К счастью, имеется промышленный стандарт RAID (Redundant Array of Independent Disks — избыточный массив независимых дисков). RAID-схема состоит из 7 уровней<sup>2</sup> — от нулевого до шестого. Эти уровни не имеют иерархической структуры, но определяют различные архитектуры со следующими общими характеристиками.

1. RAID — это набор физических дисков, рассматриваемых операционной системой как единый логический диск.
2. Данные распределены по физическим дискам массива.
3. Избыточная емкость дисков используется для хранения контрольной информации, гарантирующей восстановление данных в случае отказа одного из дисков.

Вторая и третья характеристики различны для разных уровней RAID. RAID нулевого и первого уровней не поддерживает третью характеристику вовсе.

---

<sup>2</sup> Некоторыми исследователями и производителями определены дополнительные уровни, однако семь описанных в этом разделе уровней наиболее распространены и универсальны.

Термин *RAID* первоначально был употреблен в научном докладе группы разработчиков Университета Калифорнии в Беркли [186]<sup>3</sup>. В докладе в общих чертах были рассмотрены различные конфигурации и применение RAID, а также определения уровней RAID. Эта стратегия заменяет диски с большой плотностью записи множеством дисков с малой плотностью и распределяет данные таким образом, что обеспечивает возможность одновременного доступа к данным из разных дисков. Это существенно повышает эффективность ввода-вывода и дает возможность постепенного наращивания емкости массива.

Уникальность предложенной технологии заключается в эффективном использовании избыточности. Благодаря наличию большого количества дисков повышается производительность, но увеличивается вероятность сбоев. В связи с этим RAID предусматривает хранение дополнительной информации, позволяющей восстанавливать данные, утерянные вследствие сбойной ситуации.

Сейчас мы рассмотрим каждый из уровней RAID. В табл. 11.4 содержится краткое руководство по всем семи уровням. В таблице производительность ввода-вывода показана как с точки зрения пропускной способности передачи данных, так и с точки зрения скорости запросов ввода-вывода, поскольку уровни RAID по своей природе по-разному работают в смысле этих двух метрик. Сильные стороны каждого уровня RAID выделены цветом. На рис. 11.8 показан пример, иллюстрирующий использование семи схем RAID для поддержки емкости данных, которой требуются четыре диска без избыточности. На рисунке показана структура пользовательских данных и избыточных данных и указаны относительные требования к хранилищу на разных уровнях. Мы не раз будем обращаться к этому рисунку на протяжении всего обсуждения RAID.

Из семи описанных уровней RAID обычно используются только четыре: уровни RAID 0, 1, 5 и 6.

## RAID 0

Уровень 0 не является настоящим RAID-уровнем, поскольку он не использует избыточность для повышения эффективности. Тем не менее существует ряд применений, таких как некоторые суперкомпьютеры, в которых доминируют вопросы производительности и емкости, а снижение стоимости более важно, чем надежность.

В схеме RAID 0 пользовательские и системные данные распределяются по всем дискам массива. Это дает заметное преимущество перед использованием одного большого диска: если два различных запроса ввода-вывода обращаются к двум различным блокам данных, то имеется немалая вероятность того, что эти блоки размещены на различных дисках, и два запроса могут быть обработаны, уменьшая тем самым время ожидания в очереди ввода-вывода.

Заметим, однако, что RAID 0 идет дальше простого распределения данных по массиву дисков: данные *расщеплены* (*stripped*) по всем имеющимся дискам (см. рис. 11.8).

---

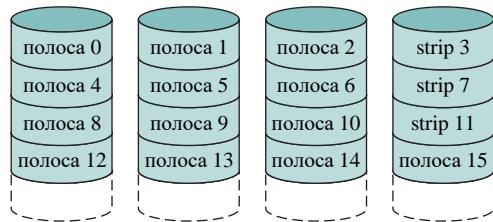
<sup>3</sup> В этом докладе аббревиатура "RAID" расшифровывается как "избыточный массив недорогих дисков". Термин "недорогой" был использован для различия малых и относительно недорогих дисков, составляющих массив, и их альтернативы — одиночного большого и дорогого диска (*single large expensive disk* — SLED). Поскольку теперь в разных дисках используются одни и те же технологии, естественно, что термин "недорогие" был заменен термином "независимые", который подчеркивает повышение производительности и надежности при использовании RAID.

**Таблица 11.4. Уровни RAID**

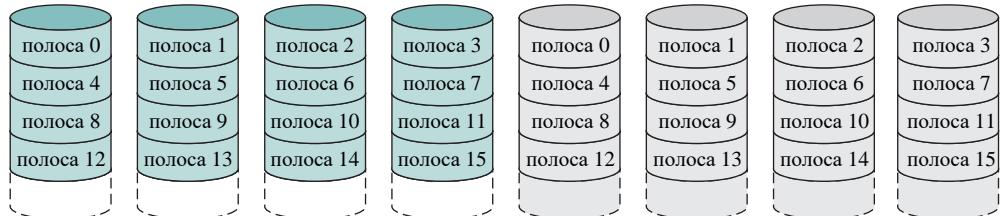
Категория	Уровень	Описание	Требуемое количество дисков <sup>1</sup>	Доступность данных	Пропускная способность передачи больших данных	Скорость запросов малого ввода-вывода
Расщепление	0	Без избыточности	N	Меньше, чем у одного диска	Очень высокая	Очень высокая для чтения и записи
Отражение <sup>2</sup>	1	Отражение	2N	Больше, чем у RAID 2, 3, 4 и 5, но меньше, чем у RAID 6	Для чтения больше, чем у одного диска; для записи сравнимо с одним диском	Для чтения почти вдвое больше, чем у одного диска; для записи сравнимо с одним диском
Параллельный доступ	2	Избыточность с кодами Хэмминга	N+m	Гораздо выше, чем у одного диска, сравнимо с RAID 3, 4 и 5	Наибольшая среди всех перечисленных альтернатив	Почти вдвое больше, чем у одного диска
	3	Четность с чередующимися битами	N+1	Гораздо выше, чем у одного диска, сравнимо с RAID 2, 4 и 5	Наибольшая среди всех перечисленных альтернатив	Почти вдвое больше, чем у одного диска
Независимый доступ	4	Четность с чередующимися блоками	N+1	Гораздо выше, чем у одного диска, сравнимо с RAID 2, 3 и 5	Для чтения аналогична RAID 0; для записи значительно меньше, чем у одного диска	Для чтения аналогична RAID 0; для записи в общем случае меньше, чем у одного диска
	5	Распределенная четность с чередующимися блоками	N+1	Гораздо выше, чем у одного диска, сравнимо с RAID 2, 3 и 4	Для чтения аналогична RAID 0; для записи меньше, чем у одного диска	Для чтения аналогична RAID 0; для записи меньше, чем у RAID 5
	6	Двойная распределенная четность с чередующимися блоками	N+2	Наибольшая среди всех перечисленных альтернатив	Для чтения аналогична RAID 0; для записи меньше, чем у RAID 5	Для чтения аналогична RAID 0; для записи значительно меньше, чем у RAID 5

<sup>1</sup> N — количество дисков данных; m — пропорционально  $\log N$ .

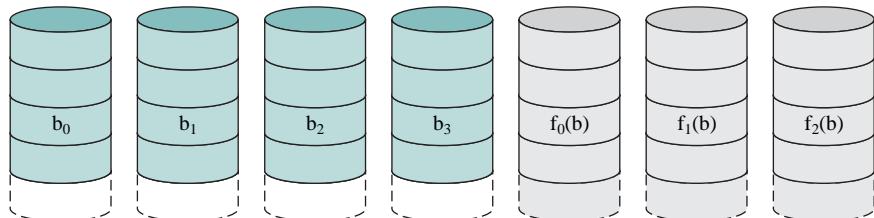
<sup>2</sup> Иногда в литературе можно встретить перевод “mirroring” как “зеркалирование”. — Примеч. пер.



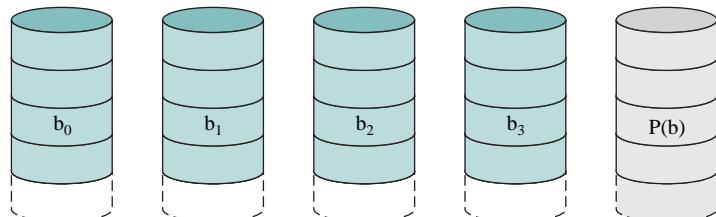
а) RAID 0 (без избыточности)



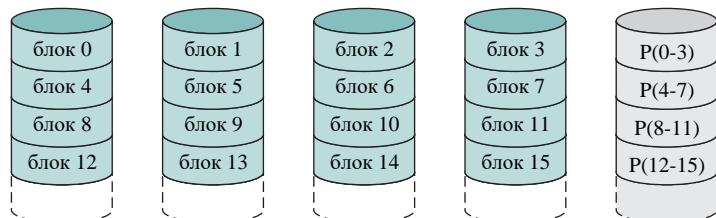
б) RAID 1 (отражение)



в) RAID 2 (избыточность с кодами Хэмминга)

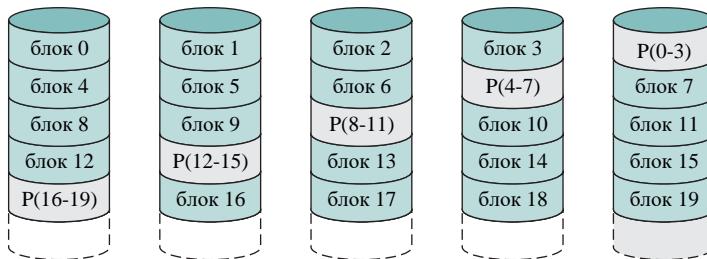


г) RAID 3 (четность с чередующимися битами)

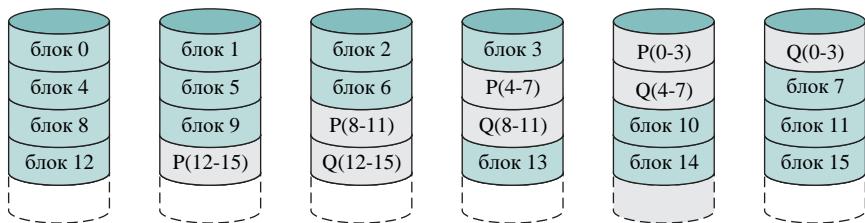


д) RAID 4 (четность с чередующимися блоками)

Рис. 11.8. Уровни RAID



e) RAID 5 (распределенная четность с чередующимися блоками)



ж) RAID 6 (двойная распределенная четность с чередующимися блоками)

Рис. 11.8. Уровни RAID (продолжение)

Все пользовательские и системные данные рассматриваются как хранящиеся на одном логическом диске. Диск делится на полосы, которые могут быть физическими блоками, секторами или другими единицами хранения. Полосы циклически размещаются на последовательных дисках RAID-массива. В  $n$ -дисковом массиве первые  $n$  логических полос физически располагаются как первые полосы каждого из  $n$  дисков; вторые  $n$  полос расположаются как вторые полосы каждого из дисков и т.д. Преимущество такой компоновки состоит в том, что если один запрос ввода-вывода обращается к множеству логически последовательных полос, то параллельно может быть обработано до  $n$  полос, и намного уменьшается тем самым время обработки запроса.

### **RAID 0 и передача большого объема данных**

Производительность любого из уровней RAID критически зависит от схемы запросов компьютера и способа размещения данных. Наиболее очевидна адресация этих запросов в RAID 0, где анализу не препятствует избыточность. Прежде всего рассмотрим использование RAID 0 для получения высокой скорости передачи данных. Приложения с высокими требованиями к скорости передачи данных должны удовлетворять двум условиям. Первое состоит в том, что на протяжении всего маршрута между главной памятью компьютера и отдельными дисководами должна быть большая пропускная способность передачи данных. Сюда включаются внутренние шины контроллера, шины ввода-вывода компьютера, адаптеры ввода-вывода, шины основной памяти.

Второе условие заключается в создании приложением таких запросов ввода-вывода, которые эффективно управляли бы дисковым массивом. Это условие выполняется, если типичный запрос адресован большому по сравнению с размером полосы объему логически непрерывных данных. В этом случае один запрос ввода-вывода включает параллельную передачу данных из нескольких дисков, увеличивая производительность передачи данных по сравнению с передачей при одном диске.

## RAID 0 и высокая частота запросов ввода-вывода

В среде, ориентированной на транзакции, пользователя обычно больше интересует время отклика, чем скорость передачи данных. При индивидуальном запросе ввода-вывода для небольшого объема данных преобладающее время операции затрачивается на перемещение дисковых головок (время поиска) и на вращение диска.

В транзакционной среде могут происходить сотни запросов ввода-вывода в секунду. Дисковый массив может обеспечить высокую скорость выполнения операций ввода-вывода путем выравнивания загрузки нескольких дисков. Эффективное выравнивание загрузки достигается только при наличии большого множества ожидающих обработки запросов ввода-вывода. Под этим, по сути, подразумевается существование нескольких независимых приложений (или одного, ориентированного на выполнение транзакций, способного выполнять множество асинхронных запросов ввода-вывода). На эффективность будет оказывать влияние и размер полосы. Если размер полосы сравнительно большой, такой, что один запрос ввода-вывода требует доступа только к одному диску, то множество находящихся в режиме ожидания запросов может быть обработано параллельно, и тем самым уменьшается время ожидания в очереди каждого запроса.

## RAID 1

RAID 1 отличается от RAID 2–6 способом достижения избыточности. Во всех остальных RAID-схемах используется какой-либо способ вычислений, в то время как в RAID 1 избыточность достигается простым дублированием всех данных. Как показано на рис. 11.8, б, в этой схеме используется то же расщепление данных, что и в RAID 0, но каждая логическая полоса размещается на двух разных физических дисках, так что для каждого диска массива имеется зеркальный диск, содержащий точно такие же данные.

Организация RAID 1 обладает следующими положительными характеристиками.

1. Запрос на чтение может быть обслужен любым из двух дисков, содержащих необходимые данные; для обслуживания выбирается тот диск, у которого минимальное время поиска и минимальная задержка из-за вращения.
2. Для запроса на запись необходимо обновление обеих полос, что может быть выполнено в параллельном режиме. Поэтому скорость записи определяется более медленной из них (т.е. той, для которой время поиска оказывается большим). Однако никаких дополнительных расходов на запись при применении RAID 1 не требуется. На уровнях 2–6 операция записи требует вычисления контрольных битов. Поэтому, когда обновляется одна полоса, управляющее массивом программное обеспечение должно сначала вычислить и обновить биты четности, а также обновить фактическую полосу, о которой идет речь.
3. Простота восстановления данных в случае сбоя — при сбое одного диска данные могут быть доступны из второго.

Принципиальным недостатком RAID 1 является стоимость, связанная с необходимостью двойного дискового пространства для логического диска. По этой причине использование RAID 1 ограничено дисками с системным программным обеспечением и данными, а также другими критически важными файлами. В этих случаях RAID 1 обеспечивает создание резервных копий всех файлов в режиме реального времени, так что в случае аварийной ситуации на диске все критические данные могут быть немедленно извлечены.

В среде, ориентированной на транзакции, RAID 1 может достичь высокой частоты запросов ввода-вывода, если основная масса запросов — на чтение диска. В этой ситуации производительность RAID 1 может приблизиться к двойной производительности RAID 0. Тем не менее, если большая часть запросов — на запись, существенного повышения производительности по сравнению с RAID 0 достичь не удастся. RAID 1 может также обеспечить повышенную производительность для приложений с интенсивным считыванием с диска. Улучшение наблюдается, если приложение в состоянии так разделять каждый запрос на чтение, чтобы в его обработке могли одновременно участвовать оба диска-члена.

## RAID 2

На уровнях 2 и 3 используется технология параллельного доступа. В таком массиве все диски, являющиеся элементами массива, участвуют в выполнении каждого запроса ввода-вывода. Обычно шпинделы индивидуальных дисководов синхронизируются таким образом, что все головки дисков в любой момент времени располагаются в одной и той же позиции.

Как и в других схемах, здесь используется разделение данных на полосы. В схемах RAID 2 и RAID 3 полосы оказываются очень малыми; нередко они соответствуют одному байту или слову. В схеме RAID 2 код с коррекцией ошибок рассчитывается по соответствующим битам каждого диска и хранится в соответствующих местах дискового массива. Обычно в этом случае используется код Хэмминга (Hamming), который способен исправлять одинарные и выявлять двойные ошибки.

Несмотря на то что для RAID 2 необходимо меньшее количество дисков, чем для RAID 1, эта схема все еще весьма дорогостоящая. Количество резервных дисков пропорционально логарифму количества дисков данных. При одиночном считывании осуществляется одновременный доступ ко всем дискам. Данные запроса и код коррекции ошибок передаются контроллеру массива. При наличии однобитовой ошибки контроллер способен быстро ее распознать и откорректировать, так что доступ для чтения в этой схеме не замедляется. При одиночной записи происходит одновременное обращение ко всем дискам массива.

Схема RAID 2 могла бы использоваться в среде с многочисленными ошибками дисков. Однако в силу высокой надежности дисков схема RAID 2 оказалась излишней и не была реализована.

## RAID 3

Схема RAID 3 организована аналогично схеме RAID 2. Отличие состоит в том, что для RAID 3 требуется только один резервный диск, независимо от размера дискового массива. В RAID 3 применяется параллельный доступ с распределенными по небольшим полосам данными. Вместо кода с исправлением ошибок для всех битов в одной и той же позиции на всех дисках, размещается рассчитанный простой бит четности.

### Избыточность

При сбое дисковода происходит обращение к дисководу четности, и данные восстанавливаются на основе информации из оставшихся устройств. Как только сбойный диск будет заменен, отсутствующие данные могут быть заново сохранены на новом диске, после чего продолжается штатная работа системы.

Восстановить данные довольно просто. Рассмотрим массив из пяти дисков, в которых  $X_0$ – $X_3$  — данные на дисках 0–3, а  $X_4$  — данные диска четности. Четность для  $i$ -го бита вычисляется следующим образом<sup>4</sup>:

$$X_4(i) = X_3(i) \oplus X_2(i) \oplus X_1(i) \oplus X_0(i)$$

Предположим, что произошел сбой диска  $X_1$ . Если мы добавим  $X_4(i) \oplus X_1(i)$  к обеим частям предыдущего уравнения, то получим

$$X_1(i) = X_4(i) \oplus X_3(i) \oplus X_2(i) \oplus X_0(i)$$

Таким образом, содержимое каждой полосы данных  $X_1$  может быть восстановлено по содержимому соответствующих полос остальных дисков массива. Этот принцип работает во всех RAID-уровнях 3–6.

В случае сбоя диска все данные остаются доступными в так называемом сокращенном режиме. В этом режиме для операций чтения отсутствующие данные восстанавливаются “на лету”, с применением описанного способа. При сокращенной записи данных должна поддерживаться согласованность по четности для позднейшего восстановления информации. Возврат к штатному функционированию требует замены сбояного диска и полного восстановления его содержимого.

### Производительность

Поскольку данные разбиваются на очень малые полосы, RAID 3 может обеспечить высокую скорость передачи данных. Любой запрос ввода-вывода включает параллельную передачу данных из всех дисков массива. Особенно заметна повышенная производительность при передаче большого объема данных. Однако за один раз может быть выполнен только один запрос ввода-вывода, поэтому в ориентированной на транзакции среде производительность падает.

## RAID 4

RAID-уровни 4–6 используют технологию независимого доступа. В массиве с независимым доступом каждый диск функционирует независимо от других, так что отдельные запросы ввода-вывода могут выполняться параллельно. Соответственно, массивы с независимым доступом могут использоваться в тех приложениях, которым необходима высокая частота запросов ввода-вывода, и менее пригодны для приложений, требующих большой скорости передачи данных.

Как и в других RAID-схемах, здесь применяется расщепление данных на полосы. В схемах RAID 4–6 полосы сравнительно большие. В RAID 4 по соответствующим полосам на каждом диске данных вычисляется полоса четности, хранящаяся на дополнительном избыточном диске.

В схеме RAID 4 имеются дополнительные расходы при выполнении операции записи небольшого блока данных. При каждой записи программное обеспечение управления массивом должно обновить не только пользовательские данные, но и соответствующие биты четности. Рассмотрим массив, состоящий из пяти дисков, в котором устройства  $X_0$ – $X_3$  содержат данные, а  $X_4$  представляет собой диск четности. Предположим, что выполняется запись, которая включает только полосу на диске  $X_1$ .

---

<sup>4</sup> Здесь  $\oplus$  обозначает операцию “исключающее или”. — Примеч. пер.

Изначально для каждого  $i$ -го бита выполняется следующее соотношение:

$$X4(i) = X3(i) \oplus X2(i) \oplus X1(i) \oplus X0(i) \quad (11.1)$$

После обновления (измененные биты отмечены штрихом) получаем:

$$\begin{aligned} X4'(i) &= X3(i) \oplus X2(i) \oplus X1'(i) \oplus X0(i) \\ &= X3(i) \oplus X2(i) \oplus X1'(i) \oplus X0(i) \oplus X1(i) \oplus X1(i) \\ &= X3(i) \oplus X2(i) \oplus X1(i) \oplus X0(i) \oplus X1(i) \oplus X1'(i) \\ &= X4(i) \oplus X1(i) \oplus X1'(i) \end{aligned}$$

Этот набор уравнений получается следующим образом. В строке 1 демонстрируются изменения в  $X1$ , которые действуют на диск четности  $X4$ . В строке 2 мы добавляем члены  $[ \oplus X1(i) \oplus X1(i) ]$ . Поскольку исключающее ИЛИ любой величины с самой собой равно 0, на уравнение это добавление не влияет, но позволяет, путем переупорядочения, получить строку 3. Затем уравнение (11.1) позволяет заменить первые четыре члена одним —  $X4(i)$ .

Итак, для вычисления новой четности программное обеспечение управления массивом должно прочитать старую пользовательскую полосу и старую полосу четности. После этого программное обеспечение может обновить эти две полосы новыми данными и вновь рассчитанной четностью. Таким образом, запись каждой полосы включает два чтения и две записи.

При большом размере записи ввода-вывода, которая включает полосы на всех дисковых накопителях, четность легко вычисляется путем расчета с использованием только новых битов данных. Таким образом, информация на диске четности может быть обновлена параллельно с пользовательскими данными, без лишних операций чтения и записи.

Тем не менее в любом случае каждая операция записи должна обновлять информацию на диске четности, что может стать узким местом системы.

## RAID 5

RAID 5 организован подобно RAID 4, но с тем отличием, что RAID 5 распределяет полосы четности по всем дискам. Распространенное размещение полос четности — в соответствии с циклической схемой, как показано на рис. 11.8, e. Для массива из  $n$  дисков полоса четности находится на другом диске для первых  $n$  полос; затем шаблон повторяется.

Распределение полос четности по всем накопителям позволяет избежать снижения производительности, связанного с операциями ввода-вывода с одним диском четности (с чем мы столкнулись при рассмотрении RAID 4). Кроме того, RAID 5 обладает тем свойством, что потеря любого одного диска не приводит к потере данных.

## RAID 6

Схема RAID 6 была представлена в работе [125] разработчиками из Беркли. В этой схеме выполняются два различных расчета четности, результаты которых хранятся в разных блоках на разных дисках. Поэтому массивы RAID 6 с объемом пользовательских данных, требующих  $N$  дисков, состоят из  $N+2$  дисков.

На рис. 11.8, ж показана схема RAID 6. На этом рисунке Р и Q представляют результаты применения двух различных алгоритмов проверки данных. Один из них — применение операции “исключающего ИЛИ”, используемой в RAID 4 и RAID 5, другой представляет собой более сложный алгоритм проверки данных. Это дает возможность восстановления данных даже в случае сбоя двух дисков массива.

Преимущество RAID 6 состоит в том, что эта схема обеспечивает чрезвычайно высокую надежность хранения данных. Потери данных возможны лишь при одновременном выходе из строя трех дисков массива. С другой стороны, у RAID 6 высокие накладные расходы при операциях записи, поскольку каждая запись затрагивает два блока четности.

Тесты производительности [73] показали, что контроллер RAID 6 может претерпеть более чем 30%-ное снижение общей производительности записи по сравнению с реализацией RAID 5; производительность чтения RAID 5 и RAID 6 различается мало.

## 11.7. ДИСКОВЫЙ КЕШ

В разделе 1.6 и приложении 1.А, “Характеристики производительности двухуровневой памяти”, мы рассмотрели принципы работы кеш-памяти. Термин *кеш-память* обычно применяется к памяти, которая меньше и быстрее основной памяти и которая располагается между основной памятью и процессором. Кеш-память уменьшает среднее время доступа к памяти благодаря принципу локальности.

Этот же принцип может быть применен и к дисковой памяти. Кеш диска представляет собой буфер в основной памяти для содержимого некоторых секторов диска. Если запрос ввода-вывода обращается к отдельному сектору, то сначала производится проверка на наличие этого сектора в кеше. Если сектор имеется в кеше, то запрос удовлетворяется из кеша. В противном случае запрошенный сектор считывается в кеш с диска. Если блок данных потребовался для выполнения одиночного запроса ввода-вывода, то, исходя из принципа локальности, весьма вероятно, что он вновь потребуется в ближайшем будущем.

### Вопросы разработки

Некоторые вопросы разработки представляют особый интерес. Первый из них связан с тем, что если запрос ввода-вывода удовлетворяется дисковым кешем, то данные из кеша должны быть переданы запрошившему процессу. Это может быть выполнено как путем пересылки блока данных в основной памяти из кеша в область пользовательского процесса, так и посредством совместного использования памяти кеша (в этом случае запрошившему процессу можно передать указатель на соответствующий слот дискового кеша). Последний подход экономит время пересылки данных из памяти в память и разрешает совместный доступ к кешу другим процессам (в соответствии с рассмотренной в главе 5, “Параллельные вычисления: взаимоисключения и многозадачность”, моделью читателей/писателей).

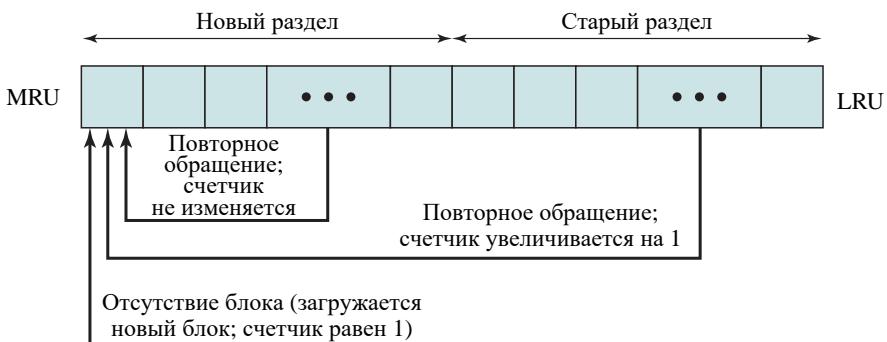
Второй вопрос связан со стратегией замещения. Когда в кеш поступает новый сектор, он должен заменить один из содержащихся в кеше секторов. Эта проблема аналогична рассмотренной в главе 8, “Виртуальная память”, проблеме замещения страниц. При изучении этого вопроса был опробован ряд алгоритмов, и в результате наиболее подходящим (и, соответственно, наиболее распространенным) оказался алгоритм замены блока, к которому дольше всего не было обращений (*least recently used* — LRU). Логически кеш состоит из стека блоков, причем на вершине стека располагается блок, к которому было последнее обращение. При обращении к блоку в кеше он перемещается

из текущей позиции на вершину стека. Если блок поступает с диска, то самый нижний блок стека удаляется, а вновь поступивший блок размещается на вершине стека. Естественно, необходимости в реальном перемещении блоков по основной памяти нет — с кешем может быть связан стек указателей на блоки.

Другой возможностью является алгоритм замены тех блоков, обращение к которым происходит наименее часто (least frequently used — LFU). Этот алгоритм может быть реализован посредством назначения счетчика каждому блоку кеша. При поступлении блока его счетчику присваивается значение 1; с каждым новым обращением значение счетчика увеличивается на 1. При необходимости замены выбирается блок с наименьшим значением счетчика. Может показаться, что LFU является более подходящей по сравнению с LRU стратегией, поскольку LFU использует более существенную информацию о блоках.

Однако у простого LFU-алгоритма имеется следующая проблема. Может возникнуть ситуация, когда обращение к некоторым блокам выполняется относительно редко, но зато при обращении интервалы между повторными обращениями оказываются короткими вследствие локальности. Тем самым значение счетчика резко увеличивается и не отражает реальной вероятности использования данного блока в ближайшее время. Так эффект локальности может стать причиной того, что алгоритм LFU произведет неверный выбор замещаемого блока.

Для преодоления этого недостатка алгоритма LFU предлагается технология, известная как замещение, основанное на частоте обращений [208]. Для ясности рассмотрим упрощенную версию, представленную на рис. 11.9, а. Блоки логически организованы в виде стека, как в алгоритме LRU. Ряд блоков в верхней части стека отделяется как новый раздел. При успешном обращении к кешу соответствующий блок перемещается на вершину стека.



(a) FIFO



б) Использование трех разделов

Рис. 11.9. Замещение, основанное на частоте обращений

Если блок к этому времени уже находился в новом разделе, то его счетчик обращений не увеличивается; в противном случае его значение увеличивается на 1. Из-за того что размер нового раздела достаточно большой, счетчики блоков, к которым происходит многократное обращение за короткий период времени, останутся неизменными. Если же выполняется обращение к блоку, отсутствующему в кеше, для замещения выбирается блок с наименьшим значением счетчика, расположенный вне нового раздела; в случае наличия нескольких таких блоков выбирается тот, к которому дольше всего не было обращений.

Авторы сообщают, что такая стратегия приводит лишь к незначительному улучшению стратегии LRU. Проблема заключается в следующем.

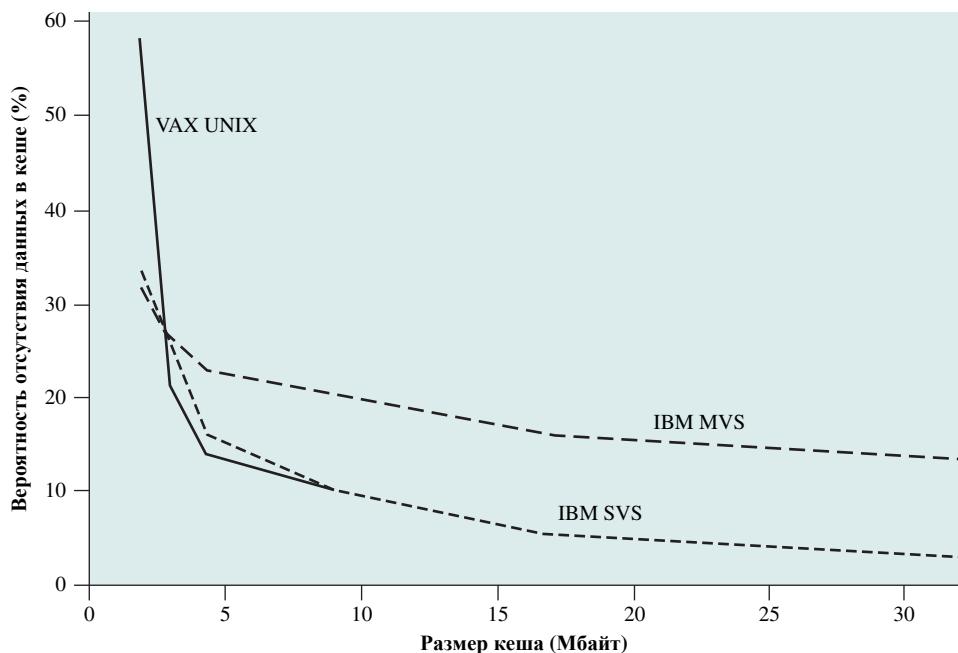
1. При отсутствии блока в кеше блок загружается в новый раздел со значением счетчика, равным 1.
2. Значение счетчика остается равным 1 все время пребывания блока в новом разделе.
3. В конечном счете блоки покидают новый раздел, значение счетчика при этом остается равным 1.
4. Если к такому блоку не происходит обращения за достаточно короткий промежуток времени, то вполне вероятно, что он будет заменен, поскольку значение его счетчика — наименьшее среди всех блоков вне нового раздела.

Внесение еще одного улучшения в алгоритм позволяет решить эту проблему. Разобьем стек на три раздела — новый, средний и старый (рис. 11.9, б). Как и прежде, счетчик обращений у блоков нового раздела не увеличивается. Для замещения, как и ранее, выбираются только блоки из старого раздела. Если средний раздел имеет достаточно большой размер, то это позволяет блокам с относительно частыми обращениями успеть увеличить значение счетчика до того, как они могут оказаться замещенными. При имитационном моделировании авторы обнаружили, что такая усовершенствованная стратегия работает значительно лучше, чем простой алгоритм LRU или LFU.

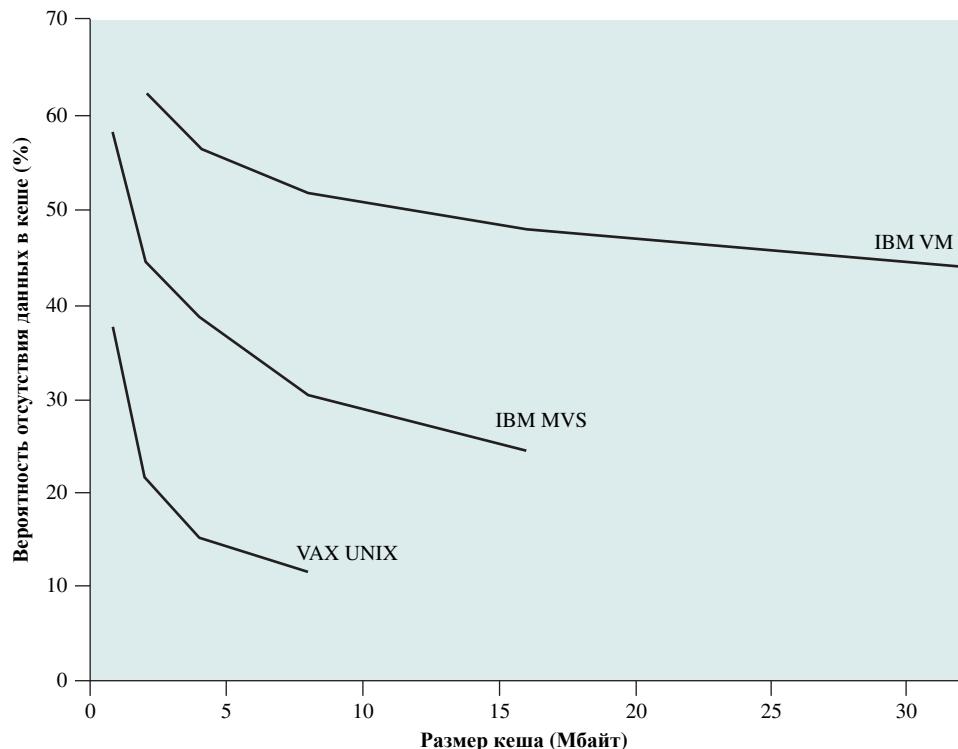
Независимо от конкретной стратегии замещение может выполняться по требованию или предварительно. В первом случае сектор замещается только тогда, когда требуется свободный слот; при втором подходе одновременно освобождается несколько слотов. Причина такого подхода заключается в необходимости записи секторов из кеша на диск. Если сектор, загруженный в кеш, использовался только для чтения, то в его записи на диск нет необходимости; однако если содержимое сектора было изменено, то перед замещением сектор необходимо записать обратно на диск. В этом случае имеет смысл кластеризация для достижения минимального времени поиска.

## Вопросы производительности

Здесь применимы все рассуждения из приложения 1.А, “Характеристики производительности двухуровневой памяти”. Проблема производительности кеша сводится к вопросу о том, можно ли достичь заданной результативности поиска. Ответ на этот вопрос зависит от локальности дисковых обращений, алгоритма замещения и других факторов проектирования. В первую очередь, результативность поиска, конечно, зависит от размера дискового кеша. На рис. 11.10 приведены итоги ряда исследований с использованием LRU (одно — для операционной системы UNIX, работающей на платформе VAX [180], а два других — для операционных систем мейнфреймов IBM [235]). На рис. 11.11 показаны результаты имитационного моделирования алгоритма замещения на основе частоты обращений. Сравнение этих двух графиков указывает на одну из опасностей такой оценки эффективности.



**Рис. 11.10.** Производительность дискового кеша при использовании LRU



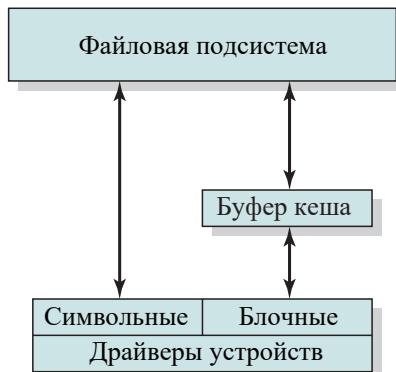
**Рис. 11.11.** Производительность дискового кеша при использовании алгоритма замещения с учетом частоты обращений

Из графиков следует, что LRU выигрывает у алгоритма замещения на основе частоты обращений. Однако при сравнении идентичных последовательностей обращений с использованием одной и той же структуры кеша выясняется, что лучшим оказывается алгоритм замещения с учетом частоты обращений. Следовательно, влияние на достигаемую производительность, кроме прочего, оказывает конкретная последовательность обращений.

## 11.8. Ввод-вывод в UNIX SVR4

В UNIX каждое устройство ввода-вывода рассматривается как специальный файл. Эти файлы управляются файловой системой, и чтение и запись осуществляются так же, как и чтение и запись обычных пользовательских файлов данных. Тем самым обеспечивается ясный и однородный интерфейс для пользователей и процессов. Для осуществления чтения из устройства или записи в него соответствующие запросы направляются к специальному файлу, связанному с устройством.

На рис. 11.12 показана логическая схема средств обслуживания ввода-вывода. Файловая подсистема управляет файлами на устройствах вторичной памяти, а кроме того, она служит интерфейсом процесса к устройствам, поскольку они рассматриваются как файлы.



**Рис. 11.12.** Структура ввода-вывода в UNIX

В UNIX существуют два вида ввода-вывода: с буферизацией и без нее. Буферизированный ввод-вывод выполняется через системные буферы, в то время как небуферизированный ввод-вывод обычно включает средства DMA, которые обеспечивают передачу данных между устройством ввода-вывода и областью ввода-вывода процесса. При буферизированном вводе-выводе используются два вида буферов: системные буферы и очереди символов.

### Буфер кеша

Буфер кеша в UNIX является, по сути, кешем диска. Дисковые операции ввода-вывода работают через этот буфер. Передача данных между буфером и пространством пользователяского процесса всегда происходит с использованием DMA. Поскольку и буфер, и область ввода-вывода процесса размещены в основной памяти, DMA используется для копирования “память–память”. В этом случае процессор не используется, но расходуются циклы шины.

Для управления буфером кеша поддерживаются три списка.

- Список свободных слотов.** Список всех слотов кеша (в UNIX слот рассматривается как буфер; каждый слот хранит один сектор диска), доступных для распределения.
- Список устройств.** Список всех буферов, связанных в данный момент с каждым диском.
- Очередь драйвера ввода-вывода.** Список буферов, участвующих в операциях ввода-вывода конкретного устройства (или находящихся в состоянии ожидания операций ввода-вывода).

Каждый буфер должен находиться либо в списке свободных слотов, либо в списке очереди драйвера ввода-вывода. Буфер, однажды назначенный устройству, остается назначенным ему, даже если попадает в свободный список, — до тех пор, пока не будет реально востребован и назначен другому устройству. Реально эти списки представляют собой списки указателей на буфера, а не физически отдельные списки.

При обращении к номеру физического блока определенного устройства операционная система прежде всего выполняет проверку наличия этого блока в буфере кеша. Для минимизации времени поиска список устройства организован в виде хеш-таблицы с использованием методики, аналогичной методу переполнения с цепочками, рассматривающейся в приложении Е, “Хеш-таблицы”. Общая организация буфера кеша показана на рис. 11.13.

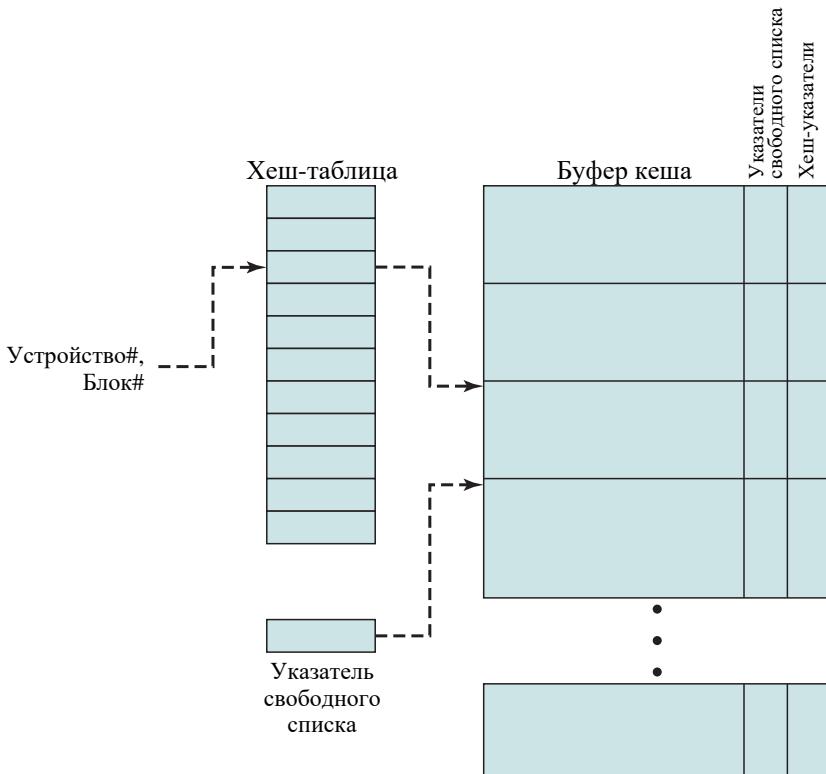


Рис. 11.13. Организация буфера кеша в UNIX

Хеш-таблица фиксированной длины содержит указатели на буфер кеша. Каждая ссылка на (*устройство#*, *блок#*) отображается на определенную запись хеш-таблицы. Указатель в этой записи указывает на первый буфер в цепочке. Указатель, связанный с каждым буфером, указывает на следующий буфер в цепочке. Следовательно, для всех обращений вида (*устройство#*, *блок#*), отображающихся в одну запись хеш-таблицы, искомый блок окажется в цепочке этой записи поля хеш-таблицы, если, конечно, данный блок имеется в кеше. Таким образом, при использовании хеш-таблицы длиной  $N$  длительность поиска в кеше снижается в  $N$  раз.

Для замещения блоков используется алгоритм LRU. После того как дисковому блоку выделяется буфер, он не может быть использован для другого блока до тех пор, пока все остальные буфера не окажутся занятыми, причем позднее рассматриваемого. Список свободных слотов сохраняет этот порядок.

## Очередь символов

Кеш способен эффективно обслуживать такие блочно-ориентированные устройства, как диски и USB-ключи. Для символьно-ориентированных устройств, таких как терминалы и принтеры, требуется иная форма буферизации. Информация либо записывается в очередь символов устройством ввода-вывода и считывается процессом, либо записывается процессом и считывается устройством. В обоих случаях используется модель производителя/потребителя, изучавшаяся нами в главе 5, “Параллельные вычисления: взаимоисключения и многозадачность”. Таким образом, символы из очереди могут быть считаны только один раз: прочитанный из очереди символ уничтожается. В этом и состоит отличие от буфера кеша, в котором процедура чтения может выполняться неоднократно и, следовательно, применима модель читателей/писателей (которая также рассматривалась в главе 5).

## Небуферизованный ввод-вывод

Небуферизованный ввод-вывод, представляющий собой простой DMA-обмен между устройством и областью памяти процесса, всегда оказывается самым быстрым методом выполнения ввода-вывода. Процесс, выполняющий небуферизованный ввод-вывод, блокируется в основной памяти и не может быть выгруженным. Тем самым снижается возможность выполнения выгрузки процесса на диск и как следствие уменьшается общая производительность системы. Кроме того, устройство ввода-вывода оказывается связанным с процессом на все время выполнения данных, делая его недоступным для других процессов.

## Устройства UNIX

UNIX распознает 5 типов устройств.

- Дисководы
- Лентопротяжные устройства
- Терминалы
- Линии связи
- Принтеры

В табл. 11.5 показаны типы ввода-вывода, соответствующие каждому виду устройств. Широко используемые UNIX дисковые накопители являются блочно-ориентированными устройствами и обладают высокой пропускной способностью. Таким образом, ввод-вывод для этих устройств обычно либо небуферизированный, либо осуществляется через буфер кеша. Лентопротяжные устройства функционально подобны дискам и используют похожие схемы ввода-вывода.

**Таблица 11.5. Устройства ввода-вывода UNIX**

	Небуферизированный ввод-вывод	Буфер кеша	Очередь символов
<b>Дисковод</b>	×	×	
<b>Лентопротяжное устройство</b>	×	×	
<b>Терминалы</b>			×
<b>Линии связи</b>			×
<b>Принтеры</b>	×		×

Поскольку обмен информацией у терминалов относительно медленный, они обычно используют очередь символов. Линии связи также требуют последовательной передачи байтов данных и лучше всего обрабатываются с использованием очередей символов. И наконец, тип ввода-вывода для принтера в общем случае зависит от его скорости. Медленно действующие принтеры обычно используют очередь символов, в то время как быстро действующий принтер может воспользоваться небуферизированным вводом-выводом. Для скоростных принтеров в принципе возможно использование кеша. Однако в силу того, что данные, поступающие на принтер, никогда не используются повторно, в использовании кеша нет никакого смысла.

## 11.9. Ввод-вывод в Linux

В общих чертах ввод-вывод в Linux очень похож на ввод-вывод в других реализациях UNIX, таких как SVR4. В Linux распознаются блочные и символьные устройства. В этом разделе мы рассмотрим несколько особенностей средств ввода-вывода Linux.

### Дисковое планирование

Дисковый планировщик по умолчанию в Linux 2.4, известный как Linux Elevator, является вариацией алгоритма LOOK, рассматривавшегося в разделе 11.5. В Linux 2.6 алгоритм Elevator расширен двумя дополнительными алгоритмами: планировщиком ввода-вывода крайнего срока и упреждающим планировщиком ввода-вывода [158]. Мы рассмотрим их оба.

#### *Планировщик Elevator*

Планировщик на основе алгоритма лифта поддерживает единую очередь запросов чтения и записи дисков и выполняет сортировку и слияние функций в очереди. В об-

щих чертах этот планировщик поддерживает список запросов отсортированным по номерам блоков. Таким образом, по мере обработки дисковых запросов перемещение по диску выполняется в единственном направлении, удовлетворяя каждый встречающийся запрос. Эта общая стратегия усовершенствована следующим образом. Когда новый запрос добавляется к очереди, рассматриваются четыре операции.

1. Если запрос выполняется к тому же сектору на диске или к сектору, смежному со следующим запросом в очереди, то существующий и новый запросы сливаются в один запрос.
2. Если запрос в очереди достаточно старый, то новый запрос вставляется в конец очереди.
3. Если имеется подходящее местоположение, новый запрос вставляется в очередь в порядке сортировки.
4. Если подходящего местоположения нет, новый запрос вставляется в конец очереди.

### **Планировщик крайнего срока**

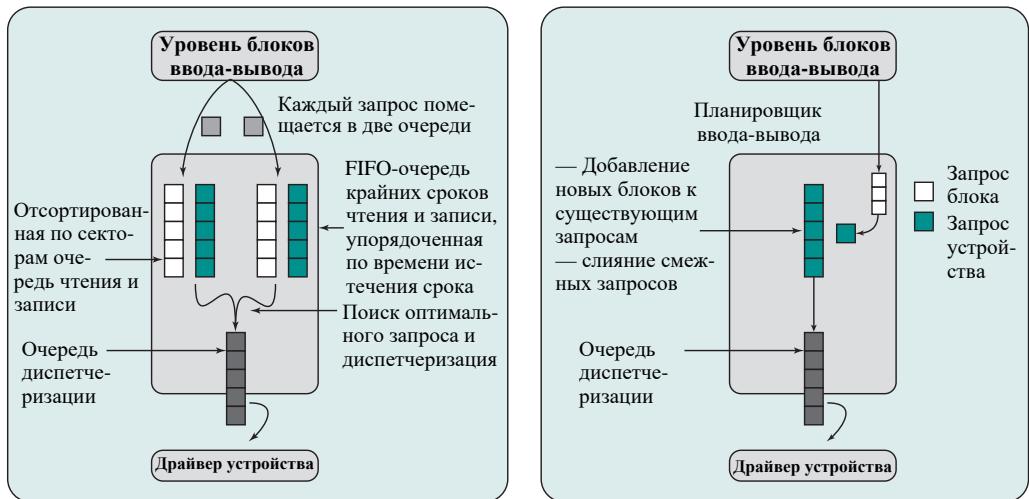
Предназначение операции 2 из предыдущего списка — предотвращение голодания (хотя и не очень эффективное [158]). Она не пытается обслужить запросы в заданный период времени, а просто останавливает сортирующую вставку запросов после соответствующей задержки. В схеме лифта проявляются две проблемы. Первая из них — запрос удаленного блока, который может оказаться отложенным на существенное время из-за динамического обновления очереди. Рассмотрите, например, следующий поток запросов к дисковым блокам: 20, 30, 700, 25. Планировщик на основе схемы лифта переупорядочивает их, так что в очередь запросы помещаются как 20, 25, 30, 700, где 20 оказывается в начале очереди. Если поступает непрерывная последовательность запросов блоков с небольшими номерами, то обработка запроса блока 700 будет существенно задерживаться.

Еще более серьезная проблема касается различия между запросами чтения и записи. Как правило, запрос записи асинхронен, т.е. как только процесс выдает запрос на запись, он не должен ждать, когда он в действительности будет выполнен. Когда приложению требуется запись, ядро копирует данные в соответствующий буфер, чтобы выполнить их запись, как только позволит время. Как только данные собраны в буфере ядра, приложение может продолжать работу. Однако для многих операций чтения процесс должен ждать, пока запрошенные данные будут доставлены в приложение, прежде чем продолжить работу. Таким образом, поток запросов на запись (например, для записи на диск большого файла) может в течение значительного времени блокировать запросы на чтение и, таким образом, блокировать процесс.

Чтобы преодолеть эти проблемы, в 2002 году был разработан новый планировщик ввода-вывода с крайним сроком. В этом планировщике используются две пары очередей (см. рис. 11.14). Каждый входящий запрос (на чтение или запись), как и ранее, помещается в отсортированную очередь лифта. Кроме того, тот же запрос помещается в хвост FIFO-очереди чтения (в случае запросов чтения; в случае запроса на запись — очереди записи). Таким образом, очереди чтения и записи поддерживают список запросов в той очередности, в которой были сделаны запросы. С каждым запросом связано время прекращения срока действия, со значением по умолчанию, равным 0,5 секунды для запроса на чтение и 5 секунд — для запроса на запись. Обычно планировщик выполняет диспет-

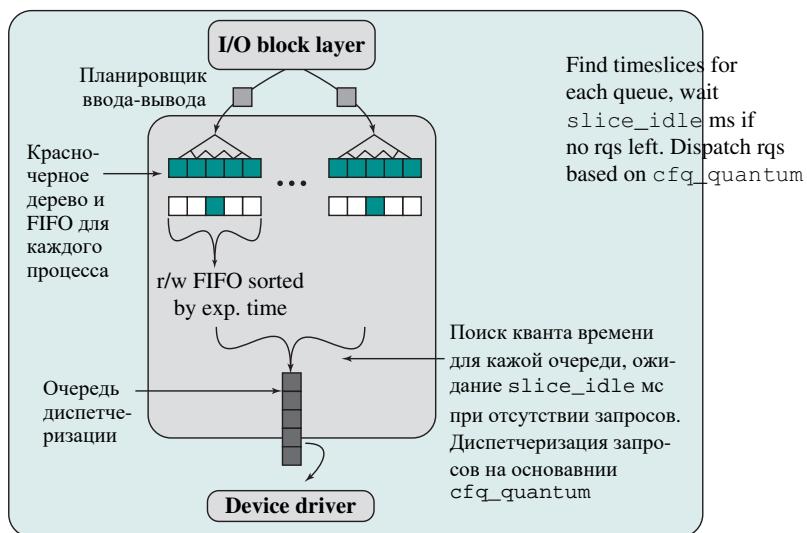
чериацию отсортированной очереди. Когда запрос удовлетворен, он удаляется из головы отсортированной очереди, а также из соответствующей FIFO-очереди. Однако если элемент во главе одной из FIFO-очередей становится старше указанного срока действия, то следующим диспетчеризуется запрос из этой очереди (плюс несколько следующих запросов из очереди). Как только запрос диспетчеризуется, он также удаляется и из отсортированной очереди.

Схема планировщика ввода-вывода с крайним сроком решает как проблему голодания, так и проблемы чтения и записи.



Планировщик с крайним сроком

Планировщик NOOP



Планировщик CFQ

Рис. 11.14. Планировщики ввода-вывода в Linux

## Упреждающий планировщик ввода-вывода

Исходный планировщик лифта и планировщик крайнего срока разработаны так, что диспетчериизуют новый запрос, как только удовлетворен текущий запрос, и таким образом поддерживают диск как можно более занятым. Эта же стратегия применяется всеми алгоритмами планирования, рассмотренными в разделе 11.5. Однако она может оказаться контрпродуктивной, если имеется множество синхронных запросов на чтение. Как правило, прежде чем отправлять следующий запрос, приложение будет ждать, пока будет удовлетворен запрос на чтение и данные станут доступными. Небольшая задержка между получением данных последнего чтения и выдачей запроса на следующее чтение позволяет планировщику рассмотреть и диспетчериизовать запрос, находящийся в режиме ожидания.

Из-за принципа локальности достаточно вероятно, что последовательные чтения одного и того же процесса будут обращаться к дисковым блокам, расположенным рядом один с другим. Если планировщик недолго приостановится после выполнения запроса на чтение, чтобы увидеть, не сделан ли новый запрос на чтение из соседнего блока, это может улучшить общую производительность системы. Данная философия лежит в основе упреждающего планировщика, предложенного в [116] и реализованного в Linux 2.6.

В Linux упреждающий планировщик накладывается на планировщик крайнего срока. Когда отправляется запрос на чтение, упреждающий планировщик приводит к задержке системы планирования до 6 мс в зависимости от конкретной конфигурации. Во время этой небольшой задержки имеется высокая вероятность того, что приложение, отправившее последний запрос на чтение, отправит еще один запрос на чтение в ту же область диска. Если это так, такой запрос будет обработан немедленно. Если же такого запроса на чтение нет, планировщик возобновляет работу, используя алгоритм планирования крайнего срока.

В [158] сообщается о двух тестах алгоритмов планирования Linux. Первый тест включал чтение файла размером 200 Мбайт при длительной потоковой записи в фоновом режиме. Второй тест включал чтение всех файлов в дереве исходного кода ядра при чтении большого файла в фоновом режиме. Результаты тестов приведены в следующей таблице.

Планировщик ввода-вывода и версия ядра	Тест 1	Тест 2
Планировщик лифта в Linux 2.4	45 с	30 мин 28 с
Планировщик крайнего срока в Linux 2.6	40 с	3 мин 30 с
Упреждающий планировщик в Linux 2.6	4.6 с	15 с

Как видите, улучшение производительности зависит от характера рабочей нагрузки. Но в обоих случаях упреждающий планировщик обеспечивает значительное повышение производительности. Упреждающий планировщик был удален из ядра 2.6.33 из-за принятия планировщика CFQ (описан ниже).

## Планировщик *NOOP*

Это самый простой среди планировщиков ввода-вывода Linux. Этот минимальный планировщик ставит запросы ввода-вывода в FIFO-очередь и использует слияние. Его основное использование — блочные устройства, не являющиеся дисками, такие как запоминающие устройства, а также специализированные программные или аппаратные среды, которые выполняют собственное планирование и нуждаются в минимальной поддержке в ядре.

## Планировщик ввода-вывода *CFQ*

Планировщик ввода-вывода “Совершенно справедливая очередь” (Completely Fair Queueing — CFQ) был разработан в 2003 году и является планировщиком ввода-вывода по умолчанию в Linux. Планировщик CFQ гарантирует справедливое распределение пропускной способности дискового ввода-вывода среди всех процессов. Он поддерживает очереди ввода-вывода для каждого процесса; каждому процессу назначается единственная очередь. Каждая очередь имеет выделенный временной интервал. Запросы отправляются в эти очереди и обрабатываются в циклическом порядке.

Когда планировщик обслуживает определенную очередь и в этой очереди больше нет запросов, он находится в режиме ожидания новых запросов в течение предопределенного интервала времени и, если запросов нет, переходит к следующей очереди. Такая оптимизация повышает производительность в ситуации, когда в этот промежуток времени поступает больше запросов.

Следует отметить, что планировщик ввода-вывода может быть установлен с помощью параметра загрузки в `grub` или во время выполнения, например, путем вывода `noop`, `deadline` или `cfq` в `/sys/class/block/sda/queue/scheduler`. Есть также дополнительные оптимизации планировщиков, описанные в документации ядра Linux.

## Страницочный кеш Linux

В Linux 2.2 и более ранних версиях ядро поддерживало страницочный кеш для чтения и записи из файлов обычных файловых систем и страниц виртуальной памяти, а также отдельный буферный кеш для блочного ввода-вывода. В Linux 2.4 и более поздних версиях имеется единый страницочный кеш, который участвует во всем трафике между диском и основной памятью.

Страницочный кеш обеспечивает два преимущества. Во-первых, когда приходит время записывать “грязные” страницы обратно на диск, их коллекцию можно правильно упорядочить и эффективно записать. Во-вторых, из-за принципа временной локальности к страницам в кеше, скорее всего, будут новые обращения, прежде чем они будут выгружены из кеша, что приводит к экономии дисковых операций ввода-вывода.

“Грязные” страницы записываются обратно на диск в двух случаях.

1. Когда объем свободной памяти падает ниже указанного порога, ядро уменьшает размер страницочного кеша, чтобы освободить память и добавить ее в пул свободной памяти.
2. Когда грязные страницы становятся старше указанного порогового возраста, ряд грязных страниц записывается обратно на диск.

## 11.10. Ввод-вывод в Windows

На рис. 11.15 показаны ключевые компоненты ядра, связанные с диспетчером ввода-вывода операционной системы Windows. Диспетчер отвечает за весь ввод и вывод операционной системы и обеспечивает однородный интерфейс, который может быть вызван драйвером любого типа.

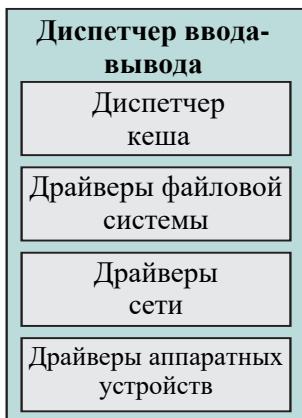


Рис. 11.15. Диспетчер ввода-вывода Windows

### Основные средства ввода-вывода

Диспетчер ввода-вывода тесно связан с четырьмя типами компонентов ядра.

- Диспетчер кеша.** Управляет кешированием для всех файловых систем. Он способен динамически увеличивать и уменьшать размер кеша, связанного с определенным файлом, в соответствии с изменением объема доступной физической памяти. Системные записи обновляются только в кеше, но не на диске. Поток отложенной записи ядра периодически собирает обновления в пакеты для записи на диск. Такая пакетная запись обеспечивает более высокую эффективность ввода-вывода. Диспетчер кеша работает с помощью отображения областей файлов в виртуальную память ядра, опираясь на менеджер виртуальной памяти, который выполняет большую часть работы по копированию страниц в файлы на диске и из них.
- Драйверы файловой системы.** Диспетчер ввода-вывода рассматривает драйвер файловой системы как обычный драйвер устройства и перенаправляет запросы к томам файловой системы соответствующему программному драйверу этого тома. Файловая система, в свою очередь, отправляет запросы ввода-вывода программным драйверам, управляющим аппаратными устройствами.
- Драйверы сети.** Windows включает интегрированные сетевые возможности и поддержку удаленных файловых систем. Эти возможности реализованы как программные драйверы, а не как часть исполнительной системы Windows.
- Драйверы аппаратных устройств.** Эти драйверы обращаются к аппаратным регистрам периферийных устройств через точки входа на уровне аппаратных

абстракций (HAL). Множество этих подпрограмм разработано для каждой платформы, поддерживаемой Windows. Поскольку имена подпрограмм одинаковы для всех платформ, исходные тексты драйверов устройств Windows переносимы на различные типы процессоров.

## АСИНХРОННЫЙ И СИНХРОННЫЙ ВВОД-ВЫВОД

Windows предоставляет два режима выполнения ввода-вывода — асинхронный и синхронный. Асинхронный ввод-вывод используется там, где можно оптимизировать производительность приложения. При асинхронном вводе-выводе приложение инициирует операцию ввода-вывода, а затем может продолжить свою работу (во время выполнения этого запроса). При синхронном вводе-выводе приложение блокируется до завершения выполнения операции ввода-вывода.

С точки зрения вызывающего потока асинхронный ввод-вывод более эффективен, поскольку позволяет продолжать выполнение, в то время как операция ввода-вывода ставится диспетчером ввода-вывода в очередь и впоследствии выполняется. Однако приложение, использующее асинхронный ввод-вывод, требует механизма определения завершенности этой операции. Windows предоставляет пять различных способов извещения о завершении ввода-вывода.

- Сигнал объекту файла.** При завершении операции ввода-вывода устанавливается индикатор, связанный с файловым объектом. Поток, вызвавший операцию ввода-вывода, может продолжить свое выполнение до тех пор, пока достигнет точки, в которой он должен дождаться завершения выполнения операции ввода-вывода. В этой точке поток может находиться в состоянии ожидания до завершения операции ввода-вывода, после чего продолжить свою работу. Эта технология проста и легка в использовании, но не подходит для обработки множественных запросов ввода-вывода. Например, если потоку необходимо выполнить множество одновременных операций над одним файлом (таких, как чтение одной и запись другой части в файл), то при описанной методике поток будет не в состоянии отличить завершение операции чтения от завершения операции записи. Он будет просто знать о том, что завершена некоторая операция ввода-вывода для этого файла.
- Сигнал объекту события.** Эта методика допускает одновременные запросы ввода-вывода к единственному устройству или файлу. Для каждого запроса поток создает событие; позже поток может ожидать завершения одного из этих запросов (или завершения серии запросов).
- Асинхронный вызов процедуры.** При этом методе используется очередь, связанная с потоком и известная как очередь асинхронных вызовов процедур (asynchronous procedure call — APC). В этом случае поток выполняет запросы ввода-вывода, указывая подпрограмму пользовательского режима, которая должна быть вызвана, когда завершится ввод-вывод. Менеджер ввода-вывода помещает результаты каждого запроса в очередь APC вызывающего потока. В следующий раз, когда поток будет заблокирован в ядре, будут выполнены APC, каждый из которых заставит поток вернуться в режим пользователя и выполнить указанную подпрограмму.

4. **Порты завершения ввода-вывода.** Эта технология используется в серверах Windows для оптимизации использования потоков. Приложение создает пул потоков для обработки завершения запросов ввода-вывода. Каждый поток ожидает порта завершения; ядро пробуждает потоки для обработки каждого завершения ввода-вывода. Одно из преимуществ этого подхода заключается в том, что приложение может указать ограничение на количество одновременно работающих потоков.
5. **Опрос.** Когда операция завершается, асинхронные запросы ввода-вывода записывают состояние и количество передач в пользовательскую виртуальную память процесса. Поток может просто проверять эти значения, чтобы увидеть, завершена ли операция.

## Программное обеспечение RAID

Windows поддерживает две разновидности конфигураций RAID, определенные в [171] следующим образом.

1. **Аппаратный RAID.** Раздельные физические диски комбинируются контроллером диска в один или несколько логических дисков.
2. **Программный RAID.** Дисковое пространство, состоящее из несмежных участков, скомбинировано в один или несколько логических разделов посредством отказоустойчивого программного драйвера диска (fault-tolerance disk driver — FTDISK).

В аппаратном RAID созданием и восстановлением резервной информации руководит интерфейс контроллера. Программный RAID, доступный в Windows Server, реализует функциональность RAID как часть операционной системы и может использоваться с любым множеством дисков. Программный RAID реализует уровни RAID 1 и RAID 5. При RAID 1 (зеркальное дублирование дисков) два диска, содержащие первичный и зеркальный разделы, могут располагаться как на одном, так и на разных контроллерах дисков (последний случай известен как *дуплексирование диска*).

## Теневые копии тома

Теневые копии — это эффективное средство создания согласованных снимков томов, чтобы их можно было архивировать. Они также полезны для архивирования файлов каждого тома. Если пользователь удаляет файл, он может получить более раннюю копию из любой доступной теневой копии, созданной системным администратором. Теневые копии реализуются программным драйвером, который создает копии данных на томе перед его перезаписью.

## Шифрование тома

Windows поддерживает шифрование целых томов, используя функцию под названием “BitLocker”. Это более безопасно, чем шифрование отдельных файлов, поскольку, чтобы гарантировать безопасность данных, работает вся система. Может быть предусмотрено до трех различных способов предоставления криптографического ключа, что позволяет использовать несколько взаимосвязанных уровней безопасности.

## 11.11. Резюме

По отношению к внешнему окружению интерфейс компьютерной системы представляет собой архитектуру ввода-вывода. Эта архитектура разработана для обеспечения систематических средств контролируемого взаимодействия с окружающим миром и снабжает операционную систему информацией, необходимой ей для эффективного управления вводом-выводом.

В общем случае ввод-вывод разделяется на уровни, причем нижние уровни имеют дело с деталями физического функционирования устройств, а верхние работают с вводом-выводом на логическом уровне. В результате изменения параметров аппаратных средств не влияют на большинство программ, использующих ввод-вывод.

Ключевым аспектом ввода-вывода является использование буферов, управляемых утилитами ввода-вывода, а не прикладными процессами. Буферизация сглаживает различия между внутренними скоростями компьютерной системы и скоростями устройств ввода-вывода. Использование буферов, кроме того, отделяет реальную передачу данных от адресного пространства прикладного процесса, что придает операционной системе большую гибкость при управлении памятью.

Наибольшее влияние на общую производительность системы оказывает дисковый ввод-вывод. Соответственно, в этой области сосредоточены гораздо большие исследовательские усилия, чем в какой-либо другой, связанной с вводом-выводом. Два основных подхода, направленных на повышение производительности дискового ввода-вывода, — это дисковое планирование и кеширование.

В любой момент времени может существовать очередь запросов ввода-вывода к одному и тому же диску. Целью дискового планирования является удовлетворение всех запросов таким образом, чтобы минимизировать время механического поиска дорожек на диске и тем самым повысить производительность диска. Здесь учитываются физическое размещение целей отложенных запросов и принцип локализации.

Дисковый кеш представляет собой буфер, обычно содержащийся в основной памяти и функционирующий как кеш блоков диска между дисковой памятью и остальной частью основной памяти. В соответствии с принципом локализации использование дискового кеша существенно уменьшает количество блоков, передаваемых от диска в основную память.

## 11.12. КЛЮЧЕВЫЕ ТЕРМИНЫ, КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАЧИ

### Ключевые термины

Блок	Двойная буферизация	Обмен буферов
Блочно-ориентированное устройство	Дисковый кеш	Полоса
Буфер ввода-вывода	Дорожка	Поточно-ориентированное устройство
Ввод-вывод	Задержка из-за вращения	Программный ввод-вывод
Ввод-вывод, управляемый прерываниями	Избыточный массив независимых дисков	Процессор ввода-вывода
Время доступа к диску	Канал ввода-вывода	Прямой доступ к памяти
Время передачи	Логический ввод-вывод	Сектор
Время поиска	Магнитный диск	Устройство ввода-вывода
Головка чтения-записи	Наименее часто используемый	Циклический буфер

### Контрольные вопросы

- 11.1. Перечислите и кратко охарактеризуйте три способа выполнения ввода-вывода.
- 11.2. В чем состоит различие между логическим вводом-выводом и устройством ввода-вывода?
- 11.3. В чем состоит различие между блочно-ориентированными и поточно-ориентированными устройствами? Приведите несколько примеров каждого из них.
- 11.4. Почему при использовании двойного буфера ожидается большая производительность, чем при использовании одинарного буфера ввода-вывода?
- 11.5. Какие элементы задержки сопутствуют процедурам дискового считывания или записи?
- 11.6. Вкратце опишите стратегии дискового планирования, приведенные на рис. 11.7.
- 11.7. Вкратце опишите семь уровней RAID.
- 11.8. Каков типичный размер сектора диска?

## Задачи

- 11.1.** Проанализируйте программу, обращающуюся к единственному устройству ввода-вывода, и сравните небуферизированный ввод-вывод с использованием буфера. Покажите, что использование буфера позволяет уменьшить время выполнения не более чем в два раза.
- 11.2.** Обобщите результат задачи 11.1 на случай обращения программы к *n* устройствам.
- 11.3.** а. Выполните такой же анализ, что и в табл. 11.2, для указанной последовательности запросов к дорожкам диска: 27, 129, 110, 186, 147, 41, 10, 64, 120. Предположим, что головка диска изначально расположена над дорожкой 100 и перемещение головки происходит в направлении уменьшения номеров дорожек.  
б. Произведите тот же анализ, предполагая, что головка диска перемещается по направлению увеличения номеров дорожек.
- 11.4.** Предположим, что *N* дорожек диска пронумерованы от 0 до *N*-1 и что запрашиваемые сектора распределены по диску случайно и равномерно. Необходимо вычислить среднее количество дорожек, которые головка пересекает при поиске.
- Сначала рассчитайте вероятность поиска длиной *j*, если головка находится над дорожкой *t*. *Указание:* определите общее количество комбинаций с учетом предположения, что все целевые дорожки равновероятны.
  - Рассчитайте вероятность поиска длиной *K* для произвольной текущей позиции головки. *Указание:* следует учесть суммирование всех возможных комбинаций перемещений через *K* дорожек.
  - Рассчитайте среднее количество дорожек, пересеченных при поиске, используя для ожидаемого значения формулу

$$E[x] = \sum_{i=0}^{N-1} i \times \Pr[x = i]$$

*Указание:* используйте равенства  $\sum_{i=1}^n i = \frac{n(n+1)}{2}$  и  $\sum_{i=1}^n i^2 = \frac{n(n+1)(2n+1)}{6}$ .

- г. Покажите, что для больших значений *N* среднее количество пересекаемых при поиске дорожек приблизительно равно *N*/3.

- 11.5.** Приведенное ниже уравнение предложено как для кеш-памяти, так и для дискового кеша:

$$T_S = T_C + M \times T_D$$

Обобщите это уравнение для иерархической памяти с *N* уровнями вместо двух.

- 11.6.** Для алгоритма замещения, основанного на частоте обращений (см. рис. 11.9), определите  $F_{\text{нов}}$ ,  $F_{\text{сред}}$ ,  $F_{\text{стар}}$  как доли кеша, входящие в новый, средний и старый разделы соответственно. Очевидно, что  $F_{\text{нов}} + F_{\text{сред}} + F_{\text{стар}} = 1$ . Опишите стратегию, если

- a.  $F_{\text{стар}} = 1 - F_{\text{нов}}$   
 б.  $F_{\text{стар}} = 1 / (\text{размер кеша})$

**11.7.** Рассчитайте количество дискового пространства (в секторах, дорожках и поверхностях), необходимого для хранения 300 000 120-байтных логических записей, если диск разбит на секторы размером 512 байт, с 96 секторами на дорожке, 110 дорожками на поверхности и 8 используемыми поверхностями. Служебные записи о файле во внимание не принимайте; считайте также, что запись не может быть разбита и размещена на двух секторах.

**11.8.** Рассмотрим дисковую систему, описанную в задаче 11.7, и предположим, что диск вращается со скоростью 3600 об/мин. Процессор производит чтение одного сектора диска с использованием ввода-вывода, управляемого прерыванием, причем на каждый байт приходится одно прерывание. Если для обработки каждого прерывания требуется 2,5 мкс, то какую часть времени процессор затратит на ввод-вывод (временем поиска пренебрегаем).

**11.9.** Повторите задание 11.8 при использовании DMA, при условии, что одно прерывание приходится на один сектор.

**11.10.** 32-разрядный компьютер имеет два селекторных и один мультиплексный канал. Каждый селекторный канал поддерживает два магнитных диска и два накопителя на магнитной ленте. К мультиплексному каналу подключены два линейных принтера, два устройства считывания карт и десять терминалов VDT. Предположим, что имеются следующие скорости передачи (в килобайтах в секунду).

<b>Диск</b>	800
<b>Магнитная лента</b>	200
<b>Принтер</b>	6,6
<b>Кард-ридер</b>	1,2
<b>VDT</b>	1

Оцените максимальную суммарную скорость передачи данных в этой системе.

**11.11.** Должно быть очевидно, что разбивка диска на полосы может привести к повышению скорости передачи данных, если размер полосы мал по сравнению с размером запросов ввода-вывода. Должно быть ясно также, что RAID 0 обеспечивает повышенную производительность по сравнению с одним большим диском, поскольку множественные запросы могут обрабатываться параллельно. Однако есть ли необходимость в разбивке диска на полосы в последнем случае? Действительно ли разбивка диска повышает производительность по сравнению с таким же дисковым массивом, но без разбивки?

**11.12.** Рассмотрим RAID-массив из четырех 200-гигабайтных дисков. Какова доступная для хранения данных емкость в случае использования каждого из уровней RAID — 0, 1, 3, 4, 5, 6?

# ГЛАВА 12

---

## УПРАВЛЕНИЕ ФАЙЛАМИ

В ЭТОЙ ГЛАВЕ...

### 12.1. Обзор

- Файлы и файловые системы
- Структура файла
- Системы управления файлами
  - Архитектура файловой системы
  - Функции управления файлами

### 12.2. Организация файлов и доступ к ним

- Смешанный файл
- Последовательный файл
- Индексно-последовательный файл
- Индексированный файл
- Файл прямого доступа

### 12.3. В-деревья

### 12.4. Каталоги файлов

- Содержимое
- Структура
- Именование

### 12.5. Совместное использование файлов

- Права доступа
- Одновременный доступ

### 12.6. Записи и блоки

### 12.7. Управление вторичной памятью

- Размещение файлов
  - Предварительное и динамическое размещение
  - Размер порции
  - Методы размещения файлов

Управление свободным пространством

Битовые таблицы

Цепочки свободных порций

Индексирование

Список свободных блоков

Тома

Надежность

## 12.8. Управление файлами в UNIX

Индексные узлы

Размещение файлов

Каталоги

Структура тома

## 12.9. Виртуальная файловая система Linux

Суперблок

Индексный узел

Запись каталога

Файл

Кеши

## 12.10. Файловая система Windows

Ключевые возможности NTFS

Том NTFS и файловая структура

Схема тома NTFS

Главная файловая таблица

Способность восстановления данных

## 12.11. Управление файлами в Android

Файловая система

SQLite

## 12.12. Резюме

## 12.13. Ключевые термины, контрольные вопросы и задачи

Ключевые термины

Контрольные вопросы

Задачи

## УЧЕБНЫЕ ЦЕЛИ

- Описывать основные концепции файлов и файловых систем.
- Понимать фундаментальные методы организации файлов и доступа к ним.
- Знать, что такое В-деревья.
- Пояснять систему каталогов файлов.
- Знать требования к совместной работе с файлами.
- Понимать концепцию блокировки записей.
- Описывать основные проблемы проектирования управления вторичной памятью.
- Понимать проблемы проектирования безопасности файловой системы.
- Описывать файловые системы разных операционных систем, используемых в Linux, UNIX и Windows.

Для большинства приложений центральным элементом является файл. За исключением приложений реального времени и некоторых других специализированных приложений, ввод данных в приложение осуществляется посредством файлов; выходные данные почти всех приложений также сохраняются в виде файлов для долговременного хранения и последующего обращения к ним пользователей и других программ.

Файлы существуют отдельно от индивидуальных приложений, использующих их для ввода и/или вывода. Пользователям необходима возможность доступа к файлам, их сохранения, поддержки целостности их содержимого. Для выполнения этих требований практически все операционные системы обеспечивают системы управления файлами. Обычно система управления файлами состоит из системных утилит, функционирующих как привилегированные приложения. Как минимум система управления файлами нуждается в специальных службах операционной системы, а как максимум — вся система управления файлами рассматривается как составная часть операционной системы. Следовательно, вопросы управления файлами вполне можно рассматривать в данной книге.

Мы начнем с небольшого обзора, после чего остановимся на различных способах организации файлов. Несмотря на то что этот вопрос не рассматривается в рамках операционной системы, все же необходимо иметь общее представление об основных вариантах организации файлов для понимания природы определенных компромиссов, на которые приходится идти при разработке систем управления файлами. Оставшаяся часть главы посвящена прочим вопросам, связанным с управлением файлами.

## 12.1. Обзор

### Файлы и файловые системы

С точки зрения пользователя, одной из самых важных частей операционной системы является файловая система, которая позволяет пользователям создавать коллекции данных, именуемые файлами, со следующими желательными свойствами.

- **Долгосрочное существование.** Файлы хранятся на диске или в другой вторичной памяти и не исчезают при выходе пользователя из системы.

- **Совместное использование процессами.** Файлы имеют имена и могут иметь связанные с ними права доступа, которые обеспечивают управляемый совместный доступ.
- **Структура.** В зависимости от файловой системы файл может иметь внутреннюю структуру, удобную для определенных приложений. Кроме того, файлы могут быть организованы в иерархическую или более сложную структуру, отражающую отношения между файлами.

Любая файловая система предоставляет не только средства для хранения данных, организованных в виде файлов, но и набор функций, которые можно выполнять над файлами. Типичные операции включают в себя следующее.

- **Создание.** Новый файл определяется и размещается в файловой системе.
- **Удаление.** Файл удаляется из файловой системы и уничтожается.
- **Открытие.** Существующий файл объявляется “открытым” определенным процессом, что позволяет этому процессу выполнять над файлом различные действия.
- **Закрытие.** Файл для процесса закрывается, после чего процесс больше не сможет выполнять действия над файлом, пока не откроет файл заново.
- **Чтение.** Процесс считывает все данные из файла или их часть.
- **Запись.** Процесс обновляет файл, либо добавляя новые данные, которые увеличивают размер файла, либо изменяя значения существующих элементов данных в файле.

Обычно файловая система поддерживает набор атрибутов, связанных с файлом. К ним относятся владелец файла, время его создания, время последнего изменения и права доступа.

## Структура файла

При рассмотрении файлов мы используем четыре термина.

- Поле
- Запись
- Файл
- База данных

**Поле (field)** является основным элементом данных. Индивидуальное поле содержит в себе единственное значение, такое как имя служащего или дата, или значение, полученное от некоторого датчика. Поле характеризуется длиной и типом данных (например, строка ASCII, десятичное число и т.п.). В зависимости от структуры файла поля могут быть либо фиксированной, либо переменной длины. В последнем случае поле часто состоит из двух или трех подполей: действительного значения, имени поля и, иногда, длины поля (поля переменной длины могут также отделяться одно от другого специальными разграничительными символами).

**Запись (record)** является набором связанных между собой полей, которые могут быть обработаны как единое целое некоторой прикладной программой. Например, запись данных о служащем может содержать такие поля, как имя, номер социального страхования,

оклад, дата принятия на работу и т.п. В зависимости от структуры записи могут быть фиксированной или переменной длины. Запись имеет переменную длину, если некоторые из ее полей — переменной длины или если переменно количество полей в записи. В любом случае вся запись обычно включает длину полей.

**Файл (file)** представляет собой набор однородных записей. Файл рассматривается пользователями и приложениями как единое целое, и обращение к нему осуществляется по его имени. Файлы можно создавать и удалять; каждый из них имеет собственное уникальное имя. Ограничения доступа обычно осуществляются на уровне файла. Другими словами, пользователи и программы, работающие в системах общего пользования, могут обладать правом доступа (или лишены его) к файлу как единому целому. В некоторых более сложных системах управление доступом осуществляется на уровне записи, а иногда даже и на уровне поля.

Некоторые файловые системы структурированы только в терминах полей, но не записей. В таком случае файл представляет собой набор полей.

**База данных (database)** представляет собой набор связанных между собой данных. К наиболее существенным аспектам базы данных можно отнести то, что отношения между данными выражены явно, а сама база данных спроектирована специально для использования большим количеством разных приложений. База данных может содержать в себе всю информацию, связанную с организацией или проектом, например результаты бизнес-проекта или научного исследования. База данных состоит из файлов одного или нескольких типов. Обычно существует отдельная система управления базой данных (СУБД), независимая от операционной системы, тем не менее она почти всегда использует некоторые программы управления файл пользователем и приложениям, поддерживаются следующие операции.

- Выбрать \_Все. Выборка всех записей из файла. Эта операция необходима тем приложениям, которые обрабатывают всю содержащуюся в файле информацию целиком. Например, приложение, производящее анализ информации в файле, должно выбрать для этого все записи. Этой операции часто соответствует термин *последовательная обработка*, поскольку доступ ко всем записям осуществляется в последовательном порядке.
- Выбрать \_Одну. Выборка только одной записи. Используется диалоговыми приложениями, ориентированными на выполнение транзакций.
- Выбрать \_Следующую. Выборка записи, которая является “следующей” в некоторой логической последовательности по отношению к последней ранее выбранной записи. Эта операция будет необходима, например, приложениям с заполнением форм. Ее могут использовать и приложения, выполняющие поиск.
- Выбрать \_Предыдущую. Подобна только что рассмотренной операции, однако в этом случае выбирается запись, являющаяся “предыдущей” по отношению к доступной в данный момент записи.
- Вставить \_Одну. Вставка записи в файл. Может быть необходима, если новая запись помещается в определенную позицию для сохранения логики связей между записями файла.
- Удалить \_Одну. Удаление существующей записи. Для сохранения логики связей между записями файла может возникнуть необходимость в обновлении некоторых связей или других структур данных.

- Обновить\_Одну. Выборка записи, обновление одного или более ее полей и перезапись обновленной записи обратно в файл. Эта операция также может потребовать выполнения дополнительных действий для сохранения последовательности файла. Если длина записи изменилась, то процедура обновления становится существенно более сложной.
- Выбрать\_Несколько. Выборка некоторого количества записей. Выполняется, например, когда приложению или пользователю необходимо выбрать записи, удовлетворяющие определенному установленному набору критерии.

Вид операций, которые чаще всего будут выполняться с файлом, влияет на способ организации файла, что обсуждается в разделе 12.2.

Следует отметить, что не все файловые системы имеют обсуждаемую в этом разделе структуру. В UNIX и UNIX-подобных системах базовая файловая структура представляет собой поток байтов. Например, программа на языке программирования С хранится в виде файла, но не имеет физических полей или записей.

## Системы управления файлами

Систему управления файлами составляет программное обеспечение, предоставляющее служебные функции при работе пользователей и приложений с файлами. Обычно единственным способом работы с файлами является использование системы управления файлами. Ее применение позволяет избавиться от необходимости разработки специальных программ работы с файлами для каждого отдельного приложения, а также обеспечивает систему средствами управления ее наиболее важными компонентами. В [95] перечислены следующие цели системы управления файлами.

- Соответствие требованиям управления данными и требованиям со стороны пользователей, включающим возможность хранения данных и выполнения рассмотренных ранее операций с ними.
- Гарантия корректности данных, содержащихся в файле.
- Оптимизация производительности как с точки зрения системы (пропускная способность), так и с точки зрения пользователя (время отклика).
- Поддержка ввода-вывода для различных типов устройств хранения информации.
- Минимизация или исключение потенциальных потерь или повреждений данных.
- Обеспечение стандартизированного набора подпрограмм интерфейса ввода-вывода.
- Обеспечение поддержки коллективного использования несколькими пользователями многопользовательской системы.

Что касается первого пункта — соответствия пользовательским требованиям, — то спектр этих требований зависит от разнообразия приложений и от среды, в которой используется компьютерная система. Ниже приведен минимальный набор требований для диалоговой системы общего назначения. Каждый пользователь должен иметь следующие возможности.

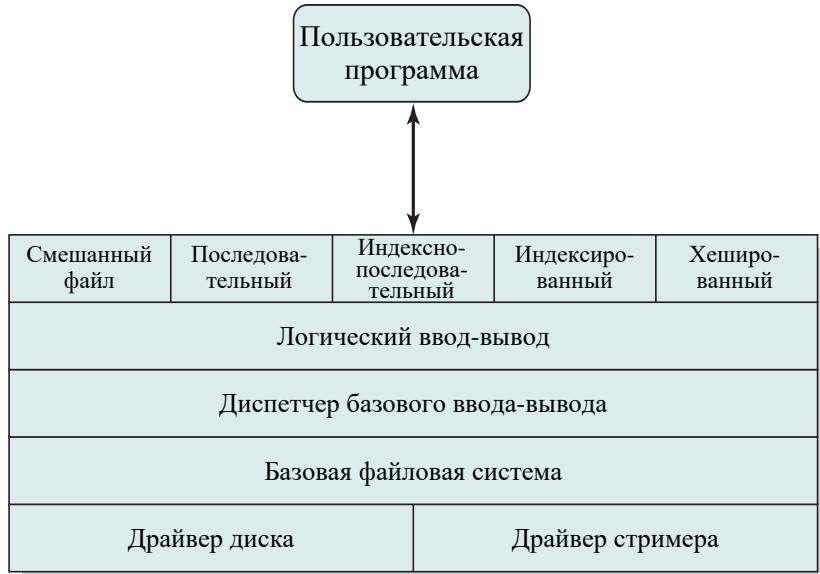
1. Создавать, удалять, читать и изменять файлы.
2. Иметь управляемый доступ к файлам других пользователей.

3. Управлять доступом к своим файлам со стороны других пользователей.
4. Перемещать данные между файлами.
5. Выполнять резервное копирование и восстанавливать файлы в случае повреждения.
6. Иметь доступ к файлам по символьным именам.

Мы будем учитывать эти задачи при нашем дальнейшем рассмотрении систем управления файлами.

### **Архитектура файловой системы**

Чтобы хоть как-то представить себе спектр действий при управлении файлами, можно обратиться к рис. 12.1, на котором представлена типичная схема организации программного обеспечения. Конечно, различные системы организованы по-разному, однако приведенная организация типична. На нижнем уровне **драйверы устройств** взаимодействуют непосредственно с периферийными устройствами или их контроллерами либо каналами. Драйвер устройства отвечает за начальные операции ввода-вывода устройства и за обработку завершения запроса ввода-вывода. При файловых операциях контролируемыми устройствами являются дисковод и накопитель на магнитной ленте. Драйверы устройств обычно рассматриваются как часть операционной системы.



**Рис. 12.1.** Архитектура программного обеспечения файловой системы [95]

Следующий уровень называется **базовой файловой системой** или **уровнем физического ввода-вывода**. Это первичный интерфейс с окружением компьютерной системы. Он оперирует блоками данных, которыми обменивается с дисковой системой, магнитной лентой или другим носителем, поэтому он связан с размещением и буферизацией блоков в оперативной памяти. На этом уровне не выполняется работа с содержимым блоков данных или структурой файлов. Базовая файловая система часто рассматривается как часть операционной системы.

**Супервизор базового ввода-вывода** отвечает за начало и завершение файлового ввода-вывода. На этом уровне поддерживаются управляющие структуры, связанные с устройством ввода-вывода, планированием и статусом файлов. Супервизор базового ввода-вывода осуществляет выбор устройства, на котором будет выполняться операция файлового ввода-вывода, исходя из выбранного файла. Кроме того, он осуществляет планирование обращения к носителю (лентам, диску и др.) с целью повышения производительности. На этом уровне происходит назначение буферов ввода-вывода и распределение внешней памяти. Супервизор базового ввода-вывода является частью операционной системы.

**Логический ввод-вывод** предоставляет пользователям и приложениям доступ к записям. Таким образом, базовая файловая система оперирует блоками данных, а модуль логического ввода-вывода — файловыми записями. Логический ввод-вывод обеспечивает возможности общего назначения по вводу-выводу записей и поддерживает информацию о файлах.

Наиболее близкий к пользователю уровень файловой системы часто называется **методом доступа**. Он обеспечивает стандартный интерфейс между приложениями и файловыми системами и устройствами, содержащими данные. Различные методы доступа отражают различные структуры файлов и различные пути доступа и обработки данных. Некоторые из наиболее общих методов доступа представлены на рис. 12.1 и вкратце описаны в разделе 12.2.

## ФУНКЦИИ УПРАВЛЕНИЯ ФАЙЛАМИ

Еще одно представление о функциях файловой системы дает рис. 12.2. Рассмотрим эту диаграмму слева направо. Пользователи и прикладные программы взаимодействуют с файловой системой посредством команд для создания и удаления файлов, а также для выполнения операций над ними. Перед выполнением любой операции файловая система должна идентифицировать выбранный файл и определить его местоположение. Это требует использования каталогов определенного типа, которые служат для описания местоположения файлов и их характеристик. Кроме того, большинство систем с совместным использованием включают управление доступом пользователей (только авторизованным пользователям разрешен доступ к определенным файлам определенными способами). Базовые операции, которые могут быть выполнены пользователем или приложением над файлами, выполняются на уровне записей. С точки зрения пользователя или прикладной программы, файл имеет некоторую последовательную структуру, организующую записи (например, записи о персонале хранятся в алфавитном порядке). Поэтому для преобразования команд пользователя в конкретные команды управления файлом должен использоваться метод доступа, соответствующий структуре данного файла.

В то время как пользователи и приложения оперируют записями, низкоуровневый ввод-вывод выполняется блоками. Следовательно, записи файла требуется сгруппировать для вывода и разгруппировать после ввода. Для поддержки блочного ввода-вывода файлов необходимо несколько функций. Сохранение на внешних запоминающих устройствах должно быть управляемым. К функциям управления относятся распределение файлов по свободным блокам при сохранении на внешних устройствах и управление свободной памятью таким образом, чтобы знать, какие блоки являются доступными для создания новых файлов и наращивания существующих. Кроме того, индивидуальные

запросы блочного ввода-вывода должны планироваться (этот вопрос был рассмотрен в главе 11, “Управление вводом-выводом и планирование дисковых операций”). И распределение файлов, и дисковое планирование тесно связаны с оптимизацией производительности системы. Поэтому, как и следовало ожидать, эти функции необходимо рассматривать вместе. Более того, оптимизация будет зависеть от структуры файлов и схем доступа. Соответственно, разработка оптимальной системы управления файлами с точки зрения эффективности является чрезвычайно сложной задачей.

На рис. 12.2 предложено разделение между системой управления файлами и операционной системой; точка пересечения представляет собой обработку записей. Такое разделение достаточно произвольно, поскольку различные системы применяют различные подходы.

В оставшейся части этой главы мы остановимся на некоторых вопросах проектирования, предложенных на рис. 12.2, и начнем с описания организации файлов и методов доступа к ним. Хотя эта тема обычно не рассматривается при изучении операционных систем, невозможно перейти к другим вопросам проектирования, связанного с файлами, не разобравшись в файловой организации и предоставлении доступа. Далее мы познакомимся с концепцией файловых каталогов. Зачастую управление ими осуществляется операционной системой от имени системы управления файлами. Остальные вопросы относятся к физическим аспектам управления файловым вводом-выводом и рассматриваются как аспекты проектирования операционной системы. Одним из вопросов является способ организации логических записей в физические блоки. В заключение остановимся на вопросах распределения файлов при сохранении данных на внешних устройствах и вопросах управления свободной вторичной памятью.

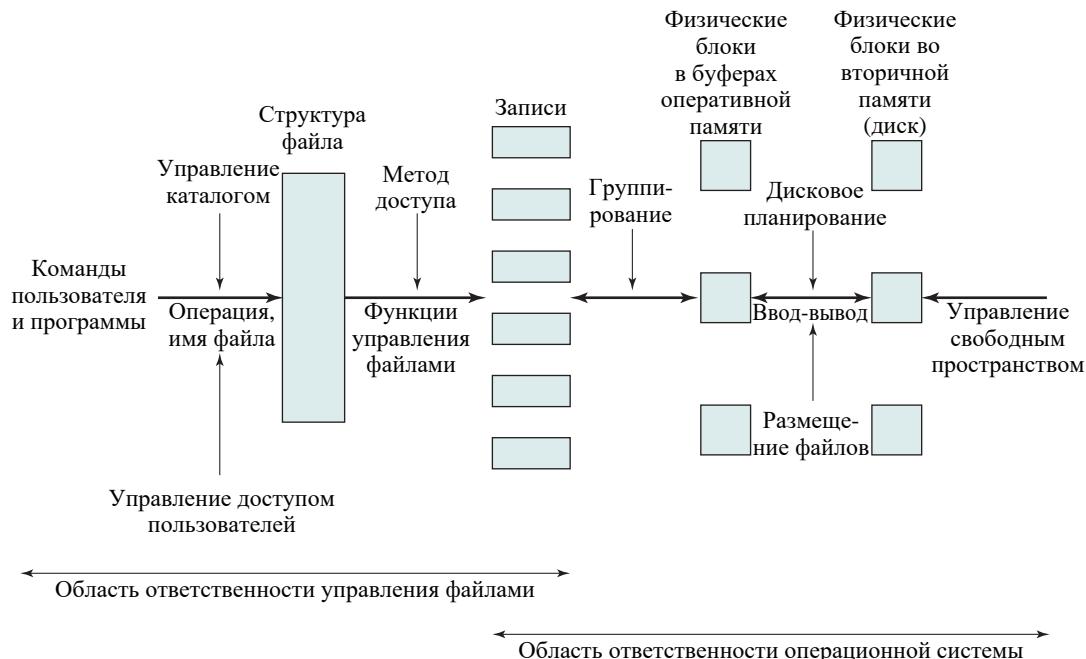


Рис. 12.2. Элементы управления файлами

## 12.2. ОРГАНИЗАЦИЯ ФАЙЛОВ И ДОСТУП К НИМ

В этом разделе используемый нами термин *организация файла* относится к логической структуре записей, определяющей способ доступа к ним. Физическая организация файла на устройстве вторичной памяти зависит от стратегий объединения записей в блоки и размещения файла, что будет рассмотрено в этой главе ниже.

При выборе способа организации файла важно учитывать несколько критериев.

- Быстрота доступа
- Легкость обновления
- Экономность хранения
- Простота обслуживания
- Надежность

Относительный уровень приоритета этих критериев зависит от приложений, которые будут работать с файлом. Например, если файл предназначен только для обработки в пакетном режиме, когда одновременно выбираются несколько записей, быстрый доступ для выборки одной записи заинтересует нас менее всего. Файл, сохраненный на CD-ROM, никогда не будет обновлен, поэтому вопрос легкости в обновлении не имеет смысла.

Перечисленные критерии могут конфликтовать между собой. Так, для экономии при сохранении необходима минимальная избыточность данных. С другой стороны, избыточность является главным фактором увеличения скорости доступа к данным. В качестве примера можно привести использование индексов.

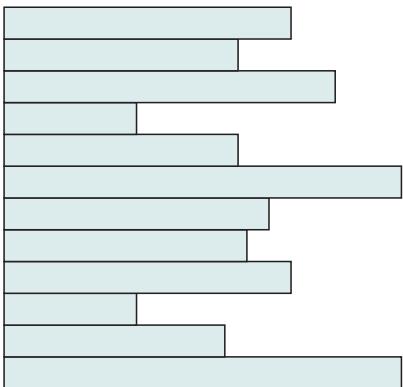
Количество разных способов организации файлов, которые уже реализованы на практике или только предложены, слишком велико и их рассмотрение даже в книге, посвященной файловым системам, невозможно. В нашем кратком обзоре в общих чертах описаны пять фундаментальных способов организации. Большинство структур, используемых в реальных системах, либо подпадают под одну из этих категорий, либо могут быть реализованы как комбинация этих способов организации. Ниже приводятся пять способов организации файла (первые четыре из них изображены на рис. 12.3).

- Смешанный файл
- Последовательный файл
- Индексно-последовательный файл
- Индексированный файл
- Файл прямого доступа (хешированный)

### Смешанный файл

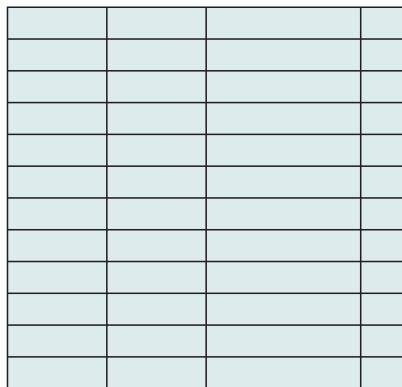
Наименее сложная форма организации файла может быть названа *смешанный* (*pile*). Данные накапливаются в порядке своего поступления. Каждая запись состоит из одного пакета данных. Такая форма упрощает накопление всей массы данных и их хранение. Записи могут иметь как различные поля, так и одинаковые поля, расположенные в различном порядке. Поэтому каждое поле должно описывать само себя, включая как значение, так и имя. Длина каждого поля должна быть либо указана неявным образом посредством применения разделителя, либо явно включена как подполе (или известна для данного типа файла заранее).

Поскольку смешанный файл не имеет никакой структуры, доступ к записи осуществляется путем полного перебора всех записей файла. То есть, если необходимо найти запись, содержащую определенное поле с определенным значением, проверяется каждая запись до тех пор, пока не будет найдена искомая или пока весь файл не будет просмотрен. Если необходимо найти все записи, содержащие определенное поле, или все записи, содержащие поле с определенным значением, то поиск должен производиться по всему файлу.



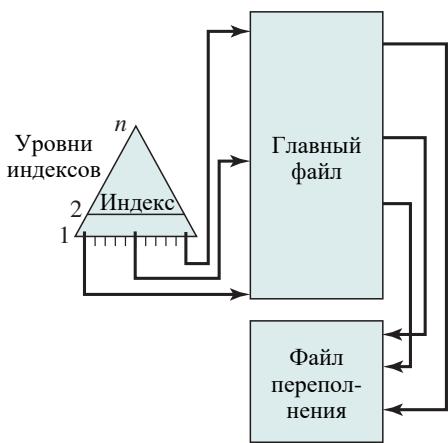
Записи переменной длины  
Переменный набор полей  
Хронологический порядок

а) Смешанный файл

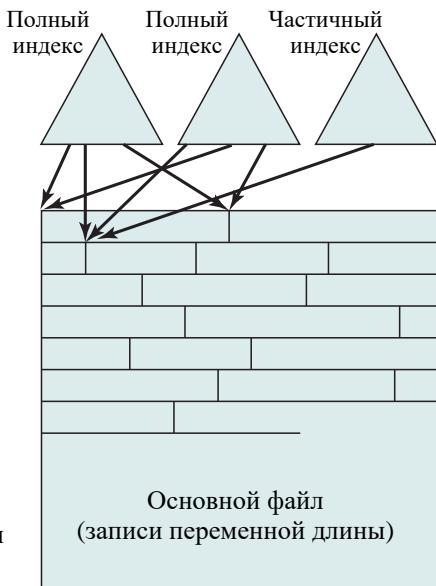


Записи постоянной длины  
Фиксированный набор полей в фиксированном порядке  
Последовательный порядок, основанный на ключевом поле

б) Последовательный файл



в) Индексно-последовательный файл



г) Индексированный файл

**Рис. 12.3.** Основные способы организации файлов

Смешанные файлы встречаются тогда, когда данные накапливаются и сохраняются перед обработкой или если они неудобны для организации. Файлы этого типа рационально используют дисковое пространство при работе с данными различного размера и структуры; он хорошо подходит для полного перебора, но недостаточно прост при обновлении данных. В большинстве других случаев этот тип файла непригоден.

## Последовательный файл

Наиболее распространенным видом файловой структуры является последовательный файл. В нем для записей используется фиксированный формат. Все записи имеют одинаковую длину и состоят из одинакового количества полей фиксированной длины, организованных в определенном порядке. Поскольку длина и позиция каждого поля известны, сохранению подлежат только значения полей; атрибутами файловой структуры являются имя и длина каждого поля.

Одно определенное поле, обычно — первое поле в каждой записи, называется **ключевым полем**, которое идентифицирует запись уникальным образом, так что ключевые значения различных записей всегда различны. Более того, записи сохраняются в “ключевой” последовательности: в алфавитном порядке для текстового ключа и в числовом порядке для числового ключа.

Последовательные файлы часто используются пакетными приложениями и обычно являются оптимальным вариантом, если эти приложения выполняют обработку всех записей (например, программы составления счетов или расчета платежных ведомостей). Только файл с последовательной организацией может быть одинаково легко сохранен как на магнитной ленте, так и на диске или другом носителе.

Для интерактивных приложений, которые работают с запросами и/или выполняют обновление индивидуальных записей, последовательный файл является малоэффективным. Для нахождения необходимой записи нужно произвести последовательный перебор записей файла. Если в основную память может быть внесен весь файл целиком или большая его часть, то возможно использование более эффективных методов поиска. Тем не менее обращение к записи в большом последовательном файле отнимает, как правило, относительно много времени. Дополнения к файлу также создают проблемы. Обычно последовательный файл сохраняется с последовательной организацией записей внутри блоков. Другими словами, физическая организация файла на магнитной ленте или на диске в точности соответствует логической организации файла. В этом случае обычно выполняется размещение новых записей в отдельном смешанном файле, называемом журнальным файлом или файлом транзакции. Периодически в пакетном режиме выполняется слияние основного и журнального файлов в новый файл с корректной последовательностью ключей.

Альтернативной организацией может служить физическая организация последовательного файла в виде списка с использованием указателей. В каждом физическом блоке сохраняется одна или несколько записей. Каждый блок на диске содержит указатель на следующий блок. Для вставки новых записей достаточно изменить указатели, и нет необходимости в том, чтобы новые записи занимали определенную физическую позицию. Это удобство достигается за счет определенных накладных расходов и дополнительной работы.

## Индексно-последовательный файл

Популярным методом преодоления недостатков последовательного файла является индексно-последовательная организация файла. Индексно-последовательный файл сохраняет главную особенность последовательного файла: записи организованы последовательно на основании значений ключевого поля. Но при описываемой организации добавлены две особенности: индекс файла для поддержки произвольного доступа и файл (или область) переполнения. Индекс обеспечивает возможность быстрого поиска требуемой записи. Файл переполнения подобен журнальному файлу, используемому файлом последовательного доступа, однако организован таким образом, что записи в нем размещаются, следуя указателю предшествующей записи.

В простейшей индексированно-последовательной структуре используется единственный уровень индексации, и индекс в этом случае представляет собой простой последовательный файл. Каждая запись в индексном файле состоит из двух полей: ключевого поля, идентичного ключевому полю в основном файле, и указателя в основной файл с ключами. Для обнаружения определенного поля сперва выполняется поиск в индексном файле. После того как в нем найдено наибольшее значение ключа, которое не превышает искомое, продолжается поиск в основном файле в позиции, определенной указателем из индексного файла.

Чтобы убедиться в эффективности этого подхода, можно в качестве примера рассмотреть последовательный файл, содержащий 1 миллион записей. Для поиска определенного ключевого значения необходимо в среднем полмиллиона операций доступа к записям. Предположим, что создается индекс, содержащий 1000 элементов, со значениями ключей индекса, более или менее равномерно распределенными в главном файле. Теперь потребуется около 500 операций доступа к индексному файлу, после чего нужно еще около 500 операций доступа к главному файлу для нахождения требуемой записи. Итак, средняя длина поиска уменьшилась с 500 000 до 1000.

Дополнения к файлу обрабатываются следующим образом. В каждой записи главного файла содержится дополнительное поле, невидимое для приложения и являющееся указателем на файл переполнения. Если в файл производится вставка новой записи, она добавляется в файл переполнения. Запись в главном файле, непосредственно предшествующая новой записи в логической последовательности, обновляется и указывает на новую запись в файле переполнения. Если запись, непосредственно предшествующая новой, сама оказывается в файле переполнения, то обновляется указатель этой записи. Как и в случае с последовательным файлом, время от времени выполняется слияние индексно-последовательного файла с файлом переполнения.

Индексно-последовательная организация намного сокращает время, необходимое для доступа к записи, не изменяя при этом последовательную природу файла. Для последовательной обработки всего файла записи в главном файле обрабатываются до тех пор, пока не будет обнаружена ссылка на файл переполнения. После этого выполняются операции с цепочкой записей в файле переполнения до тех пор, пока не встретится нулевой указатель, свидетельствующий о завершении цепочки. Затем продолжится обработка записей основного файла с того места, на котором произошло обращение к файлу переполнения.

Для обеспечения большей производительности при обращениях может использоваться многоуровневая индексация. При этом нижний уровень индексного файла рассматривается как последовательный файл, для которого создается индексный файл верхнего

уровня. Рассмотрим снова файл с 1 миллионом записей. Для него создается индекс нижнего уровня с 10 000 элементов, после чего может быть создан индекс верхнего уровня из 100 элементов для индекса нижнего уровня. Поиск начинается с индекса верхнего уровня (средняя длина равна 50 обращениям) для нахождения точки входа в индексе нижнего уровня. Затем производится поиск в этом индексе (средняя длина равна 50 обращениям) для нахождения точки входа в главный файл, поиск в котором также требует в среднем 50 обращений. Таким образом, средняя длина поиска уменьшилась с 500 000 до 1000 при одноуровневом индексе и до 150 — при двухуровневом.

## Индексированный файл

Индексно-последовательный файл сохраняет одно ограничение последовательного файла: эффективная работа с файлом ограничена работой с ключевым полем. Если необходимо производить поиск записи по какой-либо иной характеристики, отличной от ключевого поля, то оказываются непригодными обе организации последовательного файла, в то время как в некоторых приложениях эта гибкость крайне желательна.

Для достижения гибкости необходимо использование большего количества индексов, по одному для каждого типа поля, которое может быть объектом поиска. В обобщенном индексированном файле доступ к записям осуществляется только по их индексам. В результате в размещении записей нет никаких ограничений до тех пор, пока указатель по крайней мере в одном индексе ссылается на эту запись. Кроме того, в таком файле легко реализуются записи переменной длины.

Используются два типа индексов. Полный индекс содержит по одному элементу для каждой записи главного файла. Сам по себе индекс организовывается в виде последовательного файла для облегчения поиска. Частный индекс содержит элементы для записей, в которых имеется интересующее нас поле. (Некоторые записи переменной длины могут не содержать всех полей.) При добавлении новой записи в главный файл необходимо обновлять все индексные файлы.

Индексированные файлы используются прежде всего теми приложениями, в которых время доступа к информации является критической характеристикой, а обработка всех записей в файле требуется редко. Примерами могут служить системы заказа авиабилетов или системы инвентаризации.

## Файл прямого доступа

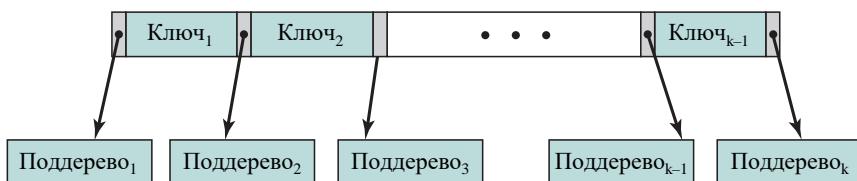
Файл прямого доступа, или хешированный файл, использует возможность прямого доступа к блоку с известным адресом при хранении файлов на диске. Как в последовательных файлах и в файлах индексно-последовательного доступа, в каждой записи должно иметься ключевое поле. Однако концепция последовательного размещения данных здесь не используется.

Файл прямого доступа применяет хеширование ключевых значений. Подробнее хеширование описано в приложении Е, “Хеш-таблицы”, где вы найдете схему хеширования с таблицей переполнения, которое обычно используется при организации хеш-файла.

Файлы прямого доступа используются, когда необходим очень быстрый доступ, при записях фиксированной длины, а также в случаях, когда доступ осуществляется ко всем записям. Примерами могут служить каталоги, прайс-листы, расписания и списки имен.

## 12.3. В-ДЕРЕВЬЯ

В предыдущем разделе говорилось об использовании индексного файла для доступа к отдельным записям в файле или базе данных. Для большого файла или базы данных один последовательный файл индексов первичного ключа не обеспечивает быстрый доступ. Для обеспечения более эффективного доступа обычно используется структурированный индексный файл. Простейшей такой структурой является двухуровневая организация, в которой исходный файл разбит на разделы, а верхний уровень состоит из последовательного набора указателей на разделы нижнего уровня. Эта структура может быть расширена до более чем двух уровней, в результате чего получается древовидная структура. Если на построение древовидного индекса не наложено некоторое ограничение, то, скорее всего, он будет иметь неравномерную структуру с рядом коротких и рядом длинных ветвей, так что время поиска в индексе будет неравномерным. Наилучшую среднюю производительность демонстрирует сбалансированная древовидная структура, все ветви которой имеют одинаковую длину. Такой структурой является В-дерево, которое стало стандартным методом организации индексов баз данных и обычно используется в файловых системах операционных систем, в том числе поддерживаемых в Mac OS X, Windows и некоторых файловых системах Linux. Структура В-дерева обеспечивает эффективный поиск, добавление и удаление элементов.



**Рис. 12.4.** Узел В-дерева с  $k$  дочерними узлами

Прежде чем проиллюстрировать концепцию В-дерева, давайте определим В-дерево и его характеристики более точно. В-дерево — это древовидная структура (не имеющая замкнутых циклов) со следующими характеристиками (рис. 12.4).

1. Дерево состоит из ряда узлов и листьев.
2. Каждый узел содержит как минимум один ключ, который однозначно идентифицирует файловую запись, и более одного указателя на дочерние узлы или листья. Количество ключей и указателей, содержащихся в узле, может варьироваться в описанных ниже пределах.
3. Каждый узел ограничен одинаковым максимальным количеством ключей.
4. Ключи в узле хранятся в порядке неубывания. Каждый ключ имеет связанный с ним дочерний узел, который является корнем поддерева, содержащего все узлы с ключами, которые меньше или равны данному ключу, но больше предыдущего ключа. У узла также есть дополнительный крайний справа дочерний элемент, который является корнем поддерева, содержащего все ключи, большие, чем любые ключи в узле. Таким образом, каждый узел имеет указателей на один больше, чем ключей.

В-дерево характеризуется минимальной степенью  $d$  и удовлетворяет следующим свойствам.

1. Каждый узел имеет не более  $2d-1$  ключей и  $2d$  дочерних узлов, или, что то же самое,  $2d$  указателей<sup>1</sup>.
2. Каждый узел, кроме корневого, имеет не менее  $d-1$  ключей и  $d$  указателей. В результате каждый внутренний узел, кроме корневого, заполнен как минимум наполовину и имеет как минимум  $d$  дочерних узлов.
3. Корневой узел имеет как минимум один ключ и два дочерних узла.
4. Все листья находятся на одном уровне и не содержат информации. Это логическая конструкция для завершения дерева; фактическая реализация может отличаться. Например, каждый узел на нижнем уровне может содержать ключи, чередующиеся с нулевыми указателями.
5. Внутренний узел с  $k$  указателями содержит  $k-1$  ключей.

Как правило, В-дерево имеет относительно большой коэффициент ветвления (большое количество дочерних узлов), что приводит к деревьям небольшой высоты.

На рис. 12.4 показаны два уровня В-дерева. Верхний уровень имеет  $(k-1)$  ключ и  $k$  указателей и удовлетворяет следующему соотношению:

$$\text{Ключ}_1 < \text{Ключ}_2 < \dots < \text{Ключ}_{k-1}$$

Каждый указатель указывает на узел, который является верхним уровнем поддерева этого узла верхнего уровня. Каждый из этих узлов поддерева содержит некоторое количество ключей и указателей, если только это не лист. Поддерживаются следующие соотношения.

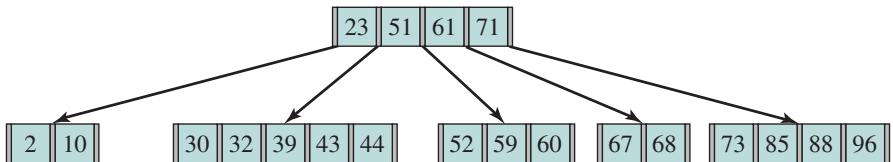
Все ключи Поддерева <sub>1</sub>	меньше Ключа <sub>1</sub>	
Все ключи Поддерева <sub>2</sub>	больше Ключа <sub>1</sub>	и меньше Ключа <sub>2</sub>
Все ключи Поддерева <sub>3</sub>	больше Ключа <sub>2</sub>	и меньше Ключа <sub>3</sub>
•	•	
Все ключи Поддерева <sub><math>k-1</math></sub>	больше Ключа <sub><math>k-2</math></sub>	и меньше Ключа <sub><math>k-1</math></sub>
Все ключи Поддерева <sub><math>k</math></sub>	больше Ключа <sub><math>k-1</math></sub>	

Поиск ключа начинается с корневого узла. Если нужный ключ находится в узле, поиск успешно завершен. Если нет, вы переходите один уровень ниже. Имеется три возможности.

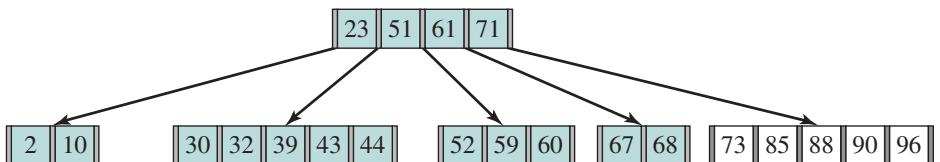
1. Ключ, который вам нужен, меньше самого маленького ключа в этом узле. В этом случае на следующий уровень необходимо перейти по крайнему слева указателю.
2. Ключ, который вам нужен, больше самого большого ключа в этом узле. В этом случае на следующий уровень необходимо перейти по крайнему справа указателю.
3. Значение ключа находится между значениями двух смежных ключей в этом узле. В этом случае на следующий уровень необходимо перейти по указателю между этими ключами.

<sup>1</sup> В одних деревьях требуется, чтобы максимальное количество ключей было нечетным (например, [54]), в других — четным ([49]); есть версии, допускающие как четное, так и нечетное количество [137]. Существенного влияния на производительность В-деревьев данный выбор не оказывает.

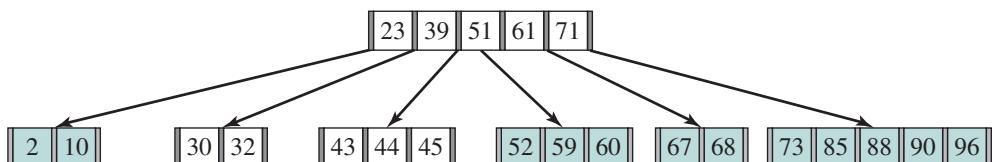
Например, рассмотрим поиск ключа 84 в дереве на рис. 12.5, г. На уровне корня  $84 > 51$ , поэтому берем крайнюю справа ветвь для перехода вниз на следующий уровень. Здесь у нас есть  $71 < 84 < 88$ , поэтому следует перейти по указателю между 71 и 88 на следующий уровень, где и находится ключ 84. С этим ключом связан указатель на нужную вам запись. Преимущество данной древовидной структуры перед другими древовидными структурами состоит в том, что она широкая и неглубокая, поэтому поиск в ней завершается очень быстро. Кроме того, в силу сбалансированности (все ветви от корня до листьев имеют одинаковую длину) нет поисков, длинных по сравнению с другими поисками в том же дереве.



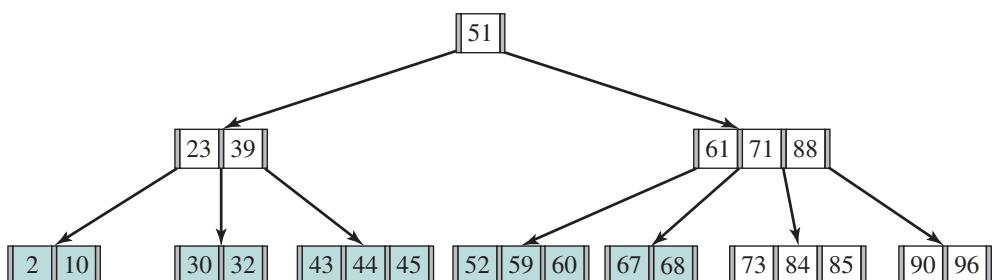
а) В-дерево с минимальной степенью  $d=3$



б) Вставлен ключ 90. Это простая вставка в узел



в) Вставлен ключ 45. Вставка требует разделения узла на две части и повышения одного ключа в корневой узел



г) Вставлен ключ 84. Вставка требует разделения узла на две части и повышения одного ключа в корневой узел, что приводит к разделению корневого узла и созданию нового корня

**Рис. 12.5.** Вставка узлов в В-дерево

Правила вставки нового ключа в В-дерево должны поддерживать сбалансированность дерева. Это достигается следующим образом.

1. Выполняем поиск ключа в дереве. Если ключ в дереве отсутствует, значит, достигнут узел на самом низком уровне.
2. Если этот узел имеет менее  $2d-1$  ключей, вставляем ключ в узел в правильное местоположение.
3. Если узел заполнен (имеет  $2d-1$  ключей), делим этот узел вокруг медианного ключа на два новых узла с  $d-1$  ключами каждый и переводим медианный ключ на более высокий уровень, как описано в шаге 4. Если новый ключ имеет значение, меньшее, чем медианный ключ, вставляем его в новый левый узел; в противном случае вставляем его в новый правый узел. В результате исходный узел разделяется на два узла: один — с  $d-1$  ключами, другой — с  $d$  ключами.
4. Поднимаемый узел вставляется в родительский, следуя правилам шага 3. Поэтому, если родительский узел заполнен, он также должен быть разделен, а его медианный ключ поднят на следующий, более высокий уровень.
5. Если процесс подъема достигает корневого узла и корневой узел оказывается заполненным, то вставка вновь выполняется по правилам шага 3. Однако в этом случае медианный ключ становится новым корневым узлом, а высота дерева увеличивается на 1.

На рис. 12.5 показан процесс вставки в В-дерево степени  $d = 3$ . В каждой из частей рисунка узлы, затронутые процессом вставки, оставлены не заштрихованными.

## 12.4. КАТАЛОГИ ФАЙЛОВ

### Содержимое

Связующим звеном между системой управления файлами и набором файлов служит файловый каталог. Каталог содержит информацию о файлах, включая атрибуты, местоположение и принадлежность. Большая часть этой информации, особенно та, которая связана с хранением, находится под управлением операционной системы. Сам по себе каталог является файлом (владельцем которого является операционная система), доступным для различных подпрограмм управления файлами. Хотя некоторая часть информации в каталогах доступна пользователям и приложениям, эта доступность косвенная (посредством системных подпрограмм). Таким образом, прямого доступа к каталогу у пользователей нет (даже в режиме только для чтения).

В табл. 12.1 приведена информация о файлах, которая обычно хранится в каталоге. С точки зрения пользователя, каталог отображает имена файлов, известные пользователям и приложениям, на собственно файлы. Таким образом, каждая запись в каталоге содержит имя файла. Практически все системы работают с файлами различных типов и способов организации, и эта информация также предоставляется каталогом. Важной категорией информации о каждом файле является информация, связанная с его хранением на диске, включая размер файла и его местоположение. В системах с совместным использованием очень важно обеспечить информацию для управления доступом к файлам. Обычно у файла имеется один владелец, а остальным пользователям могут быть предоставлены определенные права доступа. И наконец, информация по использованию файла необходима как для текущего управления файлом, так и для записи истории работы с ним.

**Таблица 12.1. Элементы информации файлового каталога**

<b>Основная информация</b>	
<b>Имя файла</b>	Имя, выбранное создателем (пользователем или программой). Должно быть единственным внутри определенного каталога
<b>Тип файла</b>	Например, текстовый, бинарный, загружаемый модуль и т.п.
<b>Организация файла</b>	Для систем, поддерживающих различные типы организации
<b>Адресная информация</b>	
<b>Том (носитель)</b>	Определяет устройство, на котором хранится файл
<b>Начальный адрес</b>	Начальный физический адрес в устройстве внешней памяти (например, цилиндр, дорожка, номер блока на диске)
<b>Занимаемый размер</b>	Текущий размер файла в байтах, словах или блоках
<b>Выделенный размер</b>	Максимальный размер файла
<b>Информация об управлении доступом</b>	
<b>Владелец</b>	Пользователь, которому передано управление этим файлом. Пользователь может разрешить доступ или отказать в нем остальным пользователям, а также изменить их привилегии доступа
<b>Информация о доступе</b>	В простейшем варианте может включать имя и пароль каждого авторизованного пользователя
<b>Допустимые действия</b>	Управление чтением, записью, пересылкой данных по сети
<b>Информация об использовании</b>	
<b>Дата создания</b>	Дата первоначального размещения файла в каталоге
<b>Создатель</b>	Обычно (но не обязательно) текущий владелец
<b>Дата последнего чтения</b>	Дата последнего чтения данных из файла
<b>Последний читатель</b>	Пользователь, который последним обращался к файлу для чтения
<b>Дата последнего изменения</b>	Дата последних процедур обновления, вставки или удаления
<b>Последний редактор файла</b>	Пользователь, который последним обращался к файлу для обновления, вставки или удаления
<b>Дата последнего резервного хранения</b>	Дата последнего резервного хранения файла на другом устройстве
<b>Текущее использование</b>	Информация о текущих действиях с файлом, например о процессе или процессах, открывших файл, о том, заблокирован ли файл процессом, обновлен ли файл в оперативной памяти, но еще не обновлен на диске

## Структура

Способы хранения информации, предложенной в табл. 12.1, широко варьируются в разных системах. Часть информации может храниться в заголовочной записи, связанной с файлом. Это снижает количество информации в каталоге, вследствие чего каталог

может быть полностью загружен в основную память (и, соответственно, при этом резко вырастает скорость работы с ним).

Простейшей структурой каталога является список записей, по одной для каждого файла. Эту структуру можно представить в виде простого последовательного файла, в котором ключевым полем служит имя файла. В некоторых ранних однопользовательских системах применялась именно такая технология, но она не подходит ни для многопользовательской системы, ни даже для одиночных пользователей со многими файлами.

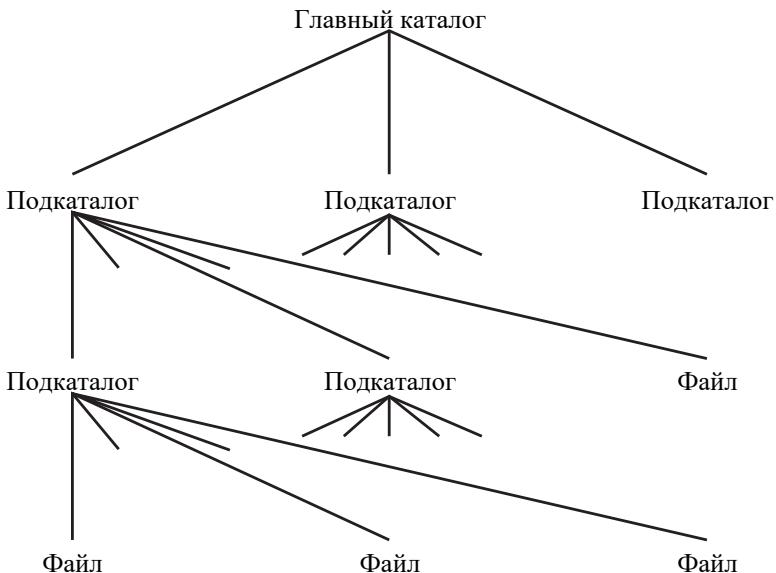
Чтобы понять требования, предъявляемые к файловой структуре, полезно рассмотреть типы операций, которые могут быть выполнены над каталогом.

- **Поиск.** При обращении пользователя или приложения к файлу требуется выполнение поиска записи об этом файле в каталоге.
- **Создание файла.** При создании нового файла необходимо добавить соответствующий элемент в каталог.
- **Удаление файла.** При удалении файла из каталога должен быть удален соответствующий элемент.
- **Список файлов в каталоге.** В ходе работы может оказаться необходимым запрос всего содержимого каталога (или его части). Обычно на этот запрос пользователя возвращаются список всех файлов, принадлежащих данному пользователю, а также некоторые атрибуты каждого файла (например, его тип, информация о доступе, информация об использовании).
- **Обновление каталога.** Поскольку в каталоге хранятся некоторые атрибуты файла, изменение хотя бы одного из них требует внесения изменения в соответствующий элемент каталога.

Для поддержки всех этих операций простой список не подходит. Рассмотрим требования одного пользователя. Пользователь может иметь несколько типов файлов, таких как текстовые файлы, графические файлы, электронные таблицы и т.д. Пользователю может понадобиться хранение файлов, организованных в соответствии с проектом, над которым он работает, в соответствии с типом или какими-то другими характеристиками. Если каталог представлен простым последовательным списком, он не способен оказать помощь в организации файлов, и пользователю нужно быть предельно осторожным и не применять одинаковые имена для разных файлов. Ситуация усложняется в многопользовательских системах, в которых необходимость присвоения уникальных имен становится серьезной проблемой. Кроме того, если каталог не структурирован, сложно скрывать части общего каталога от пользователей.

Решением этих проблем в первом приближении может служить двухуровневая схема, в которой, помимо главного каталога, имеются каталоги для каждого пользователя. Главный каталог содержит записи для каждого каталога пользователя, обеспечивая его информацией об адресе и управлении доступом. Каждый пользовательский каталог представляет собой простой список файлов этого пользователя. Такая структура означает, что имена должны быть уникальными только внутри набора файлов одного пользователя и что файловая система может легко обеспечить ограничение доступа к каталогам. Однако эта система все еще не способна оказать помощь пользователю при структурировании наборов файлов.

Более мощным и гибким (и широко распространенным) подходом является использование иерархической, или древовидной, структуры (рис. 12.6). Как и ранее, в ней имеется главный каталог, внутри которого размещены каталоги пользователей. В свою очередь, каждый из этих пользовательских каталогов (любого уровня) может содержать подкаталоги и файлы. Это справедливо для любого уровня — каталог на любом уровне может состоять из записей для подкаталогов и/или файлов.



**Рис. 12.6.** Древовидная структура каталогов

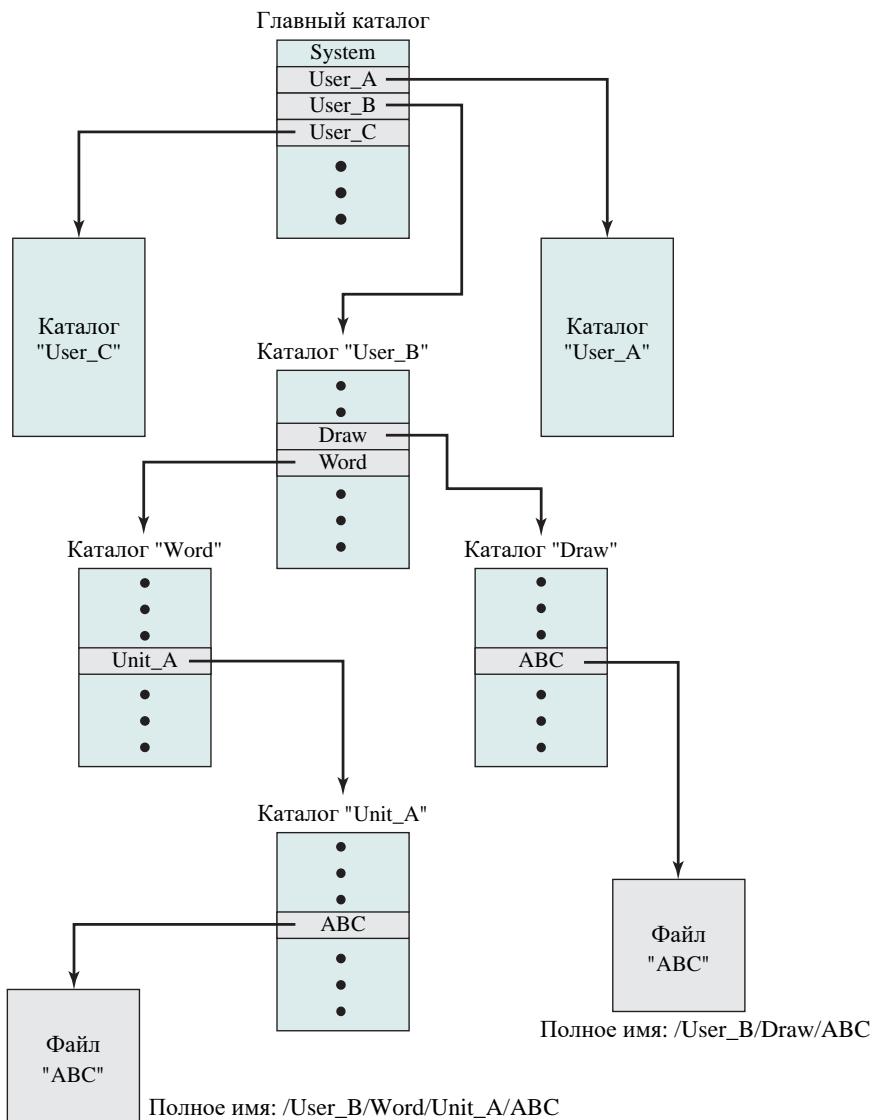
Остается сказать о том, как организованы каталоги и подкаталоги. Естественно, самым простым решением является хранение каждого каталога в виде последовательного файла. Однако если каталоги могут содержать очень большое количество элементов, то такая организация может привести к непозволительно продолжительному времени поиска. В этом случае предпочтительной оказывается хеш-структура.

## Именование

Пользователю необходима возможность обращения к файлу по его символьному имени. Очевидно, что каждый файл в системе должен иметь уникальное имя, для того чтобы обращение к файлу не оказалось неоднозначным.

Использование древовидного каталога минимизирует сложность назначения уникальных имен. К любому файлу в системе можно добраться по пути от корневого каталога вниз по ветвям подкаталогов. Серия имен подкаталогов, завершающаяся собственно именем файла, определяет **полное имя** (путь) файла. Например, файл в левом нижнем углу на рис. 12.7 имеет полное имя /User\_B/Word/Unit\_A/ABC. Наклонная черта используется для разделения имен каталогов. Имя главного каталога указано неявно, поскольку все пути начинаются именно от него. Заметим, что в связи с этим в системе допускается наличие файлов с одинаковыми именами, поскольку каждый из них имеет уникальное полное имя. В нашем примере в системе есть еще один файл с именем ABC, но его полное имя — /User\_B/Draw/ABC.

Хотя полное имя и облегчает выбор имен файлов, указывать его каждый раз при обращении к файлу не совсем удобно. Обычно интерактивный пользователь или процесс связан с текущим каталогом, называемым также **рабочим** (или **текущим**). Таким образом, обращение к файлам можно осуществлять, указывая их имена относительно рабочего каталога. Например, если рабочий каталог пользователя В — “Word”, то имени Unit\_A/ABC достаточно для идентификации файла, показанного в нижнем левом углу рис. 12.7. При входе интерактивного пользователя в систему, как и при создании процесса, рабочим каталогом по умолчанию является пользовательский каталог. Во время работы пользователь может перемещаться по дереву каталогов вниз или вверх, определяя иной рабочий каталог.



**Рис. 12.7.** Пример древовидной структуры каталога

## 12.5. СОВМЕСТНОЕ ИСПОЛЬЗОВАНИЕ ФАЙЛОВ

В многопользовательской системе практически всегда необходима возможность совместного использования файлов пользователями. При этом возникают два вопроса: права доступа и управление одновременным доступом.

### Права доступа

Файловая система должна обеспечить гибкую возможность управляемого доступа к файлу со стороны множества пользователей. Она должна предоставлять различные способы управления доступом к конкретному файлу. Обычно права доступа к файлу представляются пользователям или группам пользователей. Используется широкий диапазон прав доступа. В приведенном списке указаны права доступа, которые могут быть предоставлены определенному пользователю по отношению к некоторому файлу.

- **Отсутствие.** Пользователь не может обнаружить даже существование файла, не говоря уже о доступе к нему. Для обеспечения такого ограничения нужно запретить пользователю чтение пользовательского каталога, содержащего этот файл.
- **Знание.** Пользователь может обнаружить существование файла и установить его владельца. После этого пользователь может обратиться к владельцу для получения дополнительных прав доступа к файлу.
- **Выполнение.** Пользователь может загрузить и выполнить программу, однако выполнить копирование файла не может. Пользовательские программы часто бывают доступными с таким ограничением.
- **Чтение.** Пользователь может осуществить чтение файла для любой цели, включая копирование и выполнение. Некоторые системы могут различать чтение и копирование. В таком случае содержимое файла может быть выведено на дисплей, однако выполнить копирование файла пользователю не удастся.
- **Добавление.** Пользователь может добавить данные в файл, часто только в его конец, но произвести изменение или удаление содержимого файла ему не удастся. Это право полезно при накоплении данных из различных источников.
- **Обновление.** Пользователь может выполнять изменение, удаление или добавление данных в файле. Обычно сюда относятся операции начальной записи в файл, полной или частичной перезаписи, полного или частичного удаления данных. Некоторые системы различают разные степени обновления.
- **Изменение защиты.** Пользователь может изменять права доступа, предоставленные другим пользователям. Обычно это право принадлежит только владельцу файла. В некоторых системах пользователь может распространить это право на других пользователей. Для защиты от злоупотребления этим механизмом владелец файла может определять, какие права могут быть изменены.
- **Удаление.** Пользователь может удалить файл из файловой системы.

Эти права могут рассматриваться как определенная иерархия, в которой обладание одним правом доступа влечет за собой обладание всеми предшествующими ему в иерархии правами. Так, если конкретный пользователь обладает правом обновления файла, то этому пользователю принадлежат также права знания, выполнения, чтения и добавления.

Один из пользователей определяется как владелец файла; обычно это пользователь, создавший файл. Владелец обладает всеми перечисленными правами и может предоставлять права остальным пользователям. Доступ может быть предоставлен различным классам пользователей.

- **Конкретный пользователь.** Индивидуальные пользователи, определяемые посредством своих идентификаторов.
- **Группы пользователей.** Множество пользователей, не определенных индивидуально. Система должна обладать некоторым способом отслеживания членства пользователей в группе.
- **Все.** Все пользователи, имеющие доступ к системе. Файлы общего пользования доступны для всех пользователей.

## Одновременный доступ

Если права доступа позволяют добавлять или обновлять файл более чем одному пользователю, то в этом случае операционной системой или системой управления файлами должен быть организован определенный порядок работы пользователей с файлом. Грубый подход дает возможность пользователю заблокировать весь файл на время его обновления. Более тонкий подход управления заключается в блокировании индивидуальных записей при обновлении. Эта задача читателей/писателей рассматривалась в главе 5, “Параллельные вычисления: взаимоисключения и многозадачность”. На этапе проектирования совместно используемого доступа необходимо решить вопросы взаимного исключения и взаимоблокировки.

## 12.6. ЗАПИСИ И БЛОКИ

Как показано на рис. 12.2, записи представляют собой логическую единицу доступа к структурированному файлу<sup>2</sup>, в то время как единицами ввода-вывода при сохранении во внешней памяти являются блоки. Для выполнения операций ввода-вывода необходимо организовать записи в блоки.

При этом возникает несколько вопросов. Первый из них: какой длины должны быть блоки — фиксированной или переменной. В большинстве систем блоки имеют фиксированную длину. Это упрощает ввод-вывод, размещение буфера в основной памяти и организацию блоков при хранении на внешних носителях. Второй вопрос: каким должен быть размер блоков относительно среднего размера записи. Дело в том, что чем больше размер блока, тем больше записей будет передано одной операцией ввода-вывода. Это является преимуществом при последовательной обработке или последовательном поиске в файле, поскольку с использованием блоков большого размера количество операций ввода-вывода уменьшается, что приводит к увеличению скорости обработки. С другой стороны, если доступ к записям происходит произвольно, использование блоков больших размеров приведет к нежелательной пересылке неиспользуемых записей. Однако в целом можно утверждать, что использование блоков большего размера уменьшает время передачи данных при вводе-выводе. Тем не менее блокам большого размера необходимы большие буферы ввода-вывода, что приводит к усложнению управления ими.

<sup>2</sup> В отличие от файла, рассматриваемого как поток байтов, как, например, в файловой системе Unix.

Для данного размера блока могут использоваться три метода группирования.

- Фиксированное группирование.** Используются записи фиксированной длины, и в блоке хранится целое количество записей. При этом в конце каждого блока может появиться неиспользуемое пространство (внутренняя фрагментация).
- Сцепленное группирование переменной длины.** Используются записи переменной длины, причем их упаковка в блоки происходит без наличия неиспользуемого пространства. Так, некоторые записи должны сцеплять два блока.
- Группирование переменной длины без сцепления.** Используются записи переменной длины без сцепления. В большинстве блоков имеются пустые пространства, что вызвано невозможностью использования части блока, когда размер следующей записи больше, чем оставшееся неиспользуемое пространство.

Эти методы показаны на рис. 12.8. Предполагается, что файл сохраняется на диске в виде последовательности блоков и что файл достаточно большой для сцепления двух дорожек<sup>3</sup>. При использовании иной схемы размещения файлов эффект группирования не изменится (см. раздел 12.7).

Запись 1	Запись 2	Запись 3	Запись 4	Дорожка 1
----------	----------	----------	----------	-----------

Запись 5	Запись 6	Запись 7	Запись 8	Дорожка 2
----------	----------	----------	----------	-----------

а) Фиксированное группирование



б) Группирование переменной длины: сцепленное

Запись 1	Запись 2	Запись 3		Дорожка 1
Запись 4	Запись 5	Запись 6		Дорожка 2

б) Группирование переменной длины: без сцепления

**Рис. 12.8.** Методы группирования записей

<sup>3</sup> Данные на диске организованы в виде множества концентрированных колец, именуемых дорожками. Все дорожки имеют одну и ту же ширину, равную размеру головки чтения/записи.

Группирование фиксированной длины — самый распространенный режим для последовательных файлов с записями фиксированной длины. Группирование переменной длины со сцеплением эффективно в плане экономии места при хранении и не ограничивает размер записей. Однако эта технология сложна в реализации. Записи, сцепляющие два блока, требуют две операции ввода-вывода, что вызывает сложности при обновлении независимо от используемой организации файла. Группирование переменной длины без сцепления приводит к появлению пустого пространства и ограничению размера записи размером блока.

Технология группирования записей может взаимодействовать с аппаратным обеспечением виртуальной памяти, если таковое применяется. В среде с виртуальной памятью желательно, чтобы единицей пересылки данных была страница. Страницы обычно довольно малы, поэтому практически нет возможности рассматривать страницу в качестве блока для группирования без сцепления. Соответственно, некоторые системы объединяют несколько страниц для создания блока большего размера, предназначенного для файлового ввода-вывода. Такой подход используется в файлах VSAM мейнфреймов IBM.

## 12.7. УПРАВЛЕНИЕ ВТОРИЧНОЙ ПАМЯТЬЮ

Файл, хранящийся в устройстве внешней памяти, состоит из набора блоков. За выделение блоков файлам отвечает либо операционная система, либо система управления файлами, что порождает две проблемы, связанные с управлением. Первая проблема — пространство вторичной памяти должно распределяться между файлами, а вторая — при этом необходим контроль за доступным для распределения пространством. Само собой разумеется, что эти задачи тесно связаны между собой. Другими словами, метод, используемый при размещении файла, может оказывать влияние на метод управления свободным пространством.

Этот раздел мы начнем с описания различных способов размещения файла на диске. После этого мы рассмотрим вопрос управления свободным пространством, а в завершение обсудим вопросы надежности.

### Размещение файлов

При размещении файлов возникает несколько вопросов.

1. Необходимо ли выделять файлу максимальное пространство сразу при его создании?
2. Для файла отводится пространство в виде одного или нескольких непрерывных единиц, которые мы будем называть порциями. Размер такой порции может варьироваться от одного блока до целого файла. Какой размер порции следует использовать при размещении файла?
3. Какой тип структур данных или таблиц используется для учета порций файла? Такая таблица обычно называется **таблицей размещения файлов** (file allocation table — FAT) и имеется в DOS и некоторых других операционных системах.

Теперь рассмотрим эти вопросы подробнее.

## Предварительное и динамическое размещение

Стратегия предварительного размещения файла при запросе на его создание требует, чтобы заранее был известен максимальный размер файла. В ряде случаев, таких как компиляция программ, получение файлов сводных данных или пересылка файла из другой системы по сети, эта величина может быть надежно оценена. Однако для большинства приложений такая оценка оказывается сложной, а то и попросту невозможной. В подобных случаях пользователям и разработчикам прикладных программ приходится преувеличивать оцениваемый размер файла, чтобы не оказаться в ситуации, когда заказанного размера недостаточно. Очевидно, что такая стратегия приводит к перерасходу дискового пространства, и предпочтительнее использовать динамическое размещение, при котором выделение пространства под порции файлов происходит по мере необходимости.

### Размер порции

Другой вопрос касается размера порции, выделяемой файлу. Один предельный случай — выделение порции, достаточно большой для размещения всего файла. Другой предельный случай — распределение пространства на диске происходит по одному блоку. При выборе размера порции требуется компромисс между эффективностью работы с одним файлом и общей эффективностью системы. В ходе анализа альтернативных вариантов рассматриваются следующие соображения [268].

1. Непрерывность пространства увеличивает производительность, в частности при выполнении операций типа *Выбрать\_Следующую*, а особенно транзакций, выполняемых в транзакционно-ориентированной операционной системе.
2. Наличие большого количества маленьких порций увеличивает размер таблиц, необходимых для управления информацией о размещении.
3. Наличие порций фиксированного размера (например, блоков) упрощает перераспределение пространства.
4. Наличие порций переменной длины или малого размера минимизирует потери неиспользуемого пространства из-за излишнего распределения.

Естественно, что эти условия взаимосвязаны и должны рассматриваться вместе. В итоге мы получаем два основных варианта.

1. **Большие непрерывные порции переменной длины.** Обеспечивают лучшую производительность. Переменный размер помогает избежать наличия неиспользуемых участков, а таблицы размещения файлов имеют небольшой размер. Однако при этом трудно организовать повторное использование дискового пространства.
2. **Блоки.** Небольшие порции фиксированного размера обеспечивают большую гибкость. При этом могут потребоваться таблицы, имеющие больший размер или сложную структуру. Непрерывность отсутствует, а блоки для файла выделяются по необходимости.

Любой из вариантов совместим как с предварительным, так и с динамическим размещением. При использовании первого варианта файл размещается предварительно, как одна непрерывная группа блоков. Необходимость в таблице размещения файлов отпадает. Нужно лишь указать на первый блок и количество размещаемых блоков. При выборе блочной структуры все необходимые порции размещаются одновременно. Это означает, что FAT файла имеет фиксированный размер.

Используя порции переменного размера, придется иметь дело с фрагментацией свободного пространства. Этот вопрос был затронут при рассмотрении распределения основной памяти в главе 7, “Управление памятью”. Возможны следующие стратегии распределения.

- Первый подходящий.** Выбор первой неиспользуемой непрерывной группы блоков подходящего размера из списка свободных блоков.
- Наилучший подходящий.** Выбор наименьшей неиспользуемой группы, размер которой является достаточным.
- Ближайший подходящий.** Выбор неиспользуемой группы подходящего размера, которая ближе всех расположена к последнему размещенному файлу для увеличения локализации.

Какая же из приведенных стратегий лучше? Сложность моделирования этих стратегий заключается в том, что необходимо учитывать множество факторов, таких как типы файлов, схемы доступа к ним, степень многозадачности, кеширование диска, дисковое планирование и др.

### Методы размещения файлов

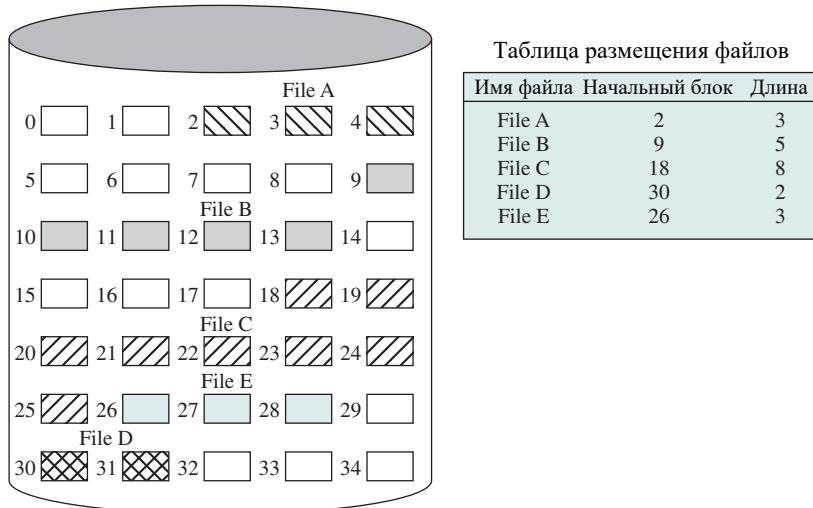
Рассмотрев вопросы предварительного и динамического размещения файлов, мы можем провести сравнительный анализ различных методов размещения файлов. Наиболее широко используются три метода: непрерывный, цепочечный и индексированный. В табл. 12.2 приведены некоторые характеристики каждого метода.

Таблица 12.2. Методы размещения файлов

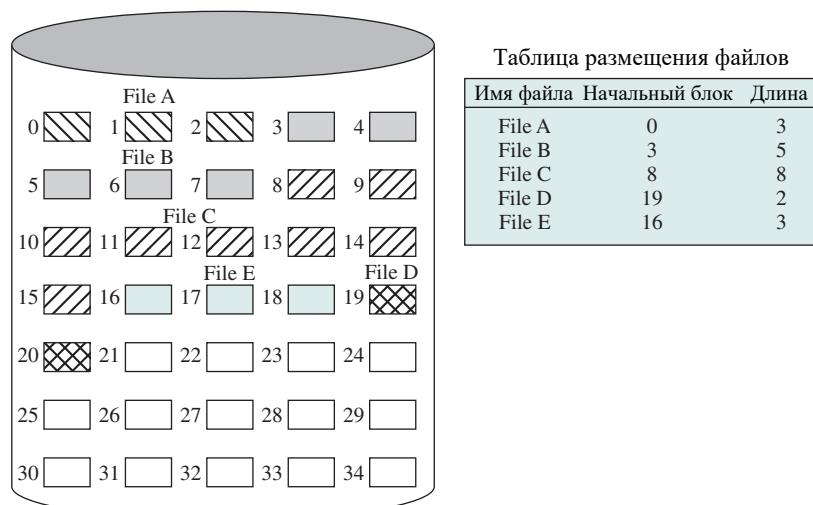
	Непрерывный	Цепочечный	Индексированный	
<b>Предварительное размещение</b>	Необходимо	Возможно	Возможно	
<b>Фиксированный или переменный размер порции</b>	Переменный	Фиксированные блоки	Фиксированные блоки	Переменный
<b>Размер порции</b>	Большой	Малый	Малый	Средний
<b>Частота размещения</b>	Одинарное размещение	От низкой до высокой	Высокая	Низкая
<b>Время размещения</b>	Среднее	Длительное	Короткое	Среднее
<b>Размер таблицы размещения файла</b>	Одна запись	Одна запись	Большой	Средний

При **непрерывном размещении** создаваемому файлу выделяется отдельное непрерывное множество блоков (рис. 12.9). Таким образом, это стратегия предварительного размещения с порциями переменного размера. Для каждого файла таблице размещения необходим только один элемент, определяющий начальный блок и длину файла. Непрерывное размещение является наилучшим с точки зрения индивидуального файла последовательного доступа. Для повышения производительности ввода-вывода при последовательной обработке одновременно может обрабатываться большое количество блоков. Выборка одиночного блока осуществляется очень просто. Например, если файл начинается с блока  $b$ , а искомый блок —  $i$ -й, то его местоположение на внешнем запо-

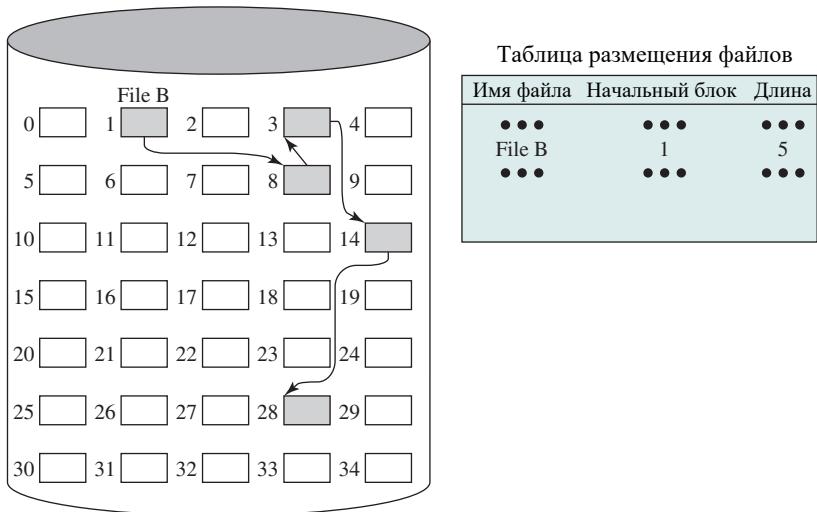
минающем устройстве определяется как  $b+i-1$ . Однако непрерывное размещение файла вызывает некоторые проблемы. Так, появляется внешняя фрагментация, что усложняет поиск непрерывных блоков подходящего размера. Время от времени возникает потребность в выполнении упаковки для освобождения необходимого пространства на диске (рис. 12.10). Кроме того, как уже упоминалось, при предварительном размещении в момент создания файла следует объявлять его размер.



**Рис. 12.9.** Непрерывное размещение файлов



**Рис. 12.10.** Непрерывное размещение файлов (после уплотнения)



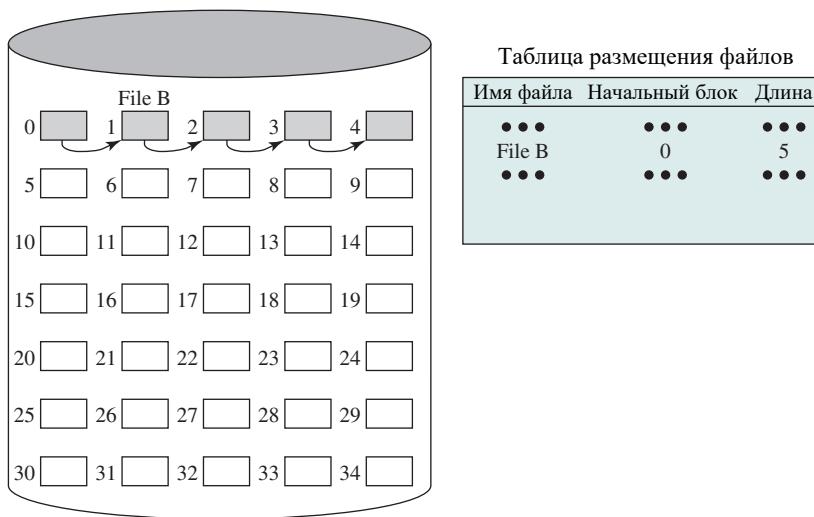
**Рис. 12.11.** Цепочечное размещение файлов

Как противоположность непрерывному размещению разработан метод **цепочечного размещения файла** (рис. 12.11). Обычно размещение выполняется по одному блоку. Каждый блок содержит указатель на следующий блок в цепочке. В этой схеме таблице размещения файлов необходим только один элемент для каждого файла, указывающий начальный блок и длину файла. Несмотря на то что в этой схеме возможно использование предварительного распределения блоков, все же обычно проще выделять блоки для файла при необходимости. Выбор блоков происходит очень просто — в цепочку может быть добавлен любой свободный блок. Заботиться о внешней фрагментации при этом не надо, поскольку блоки выделяются по одному. Такой тип физической организации лучше всего подходит для файлов последовательного доступа, обработка данных в которых выполняется последовательно. Для выбора отдельного блока файла необходимо отследить всю цепочку до достижения искомого блока.

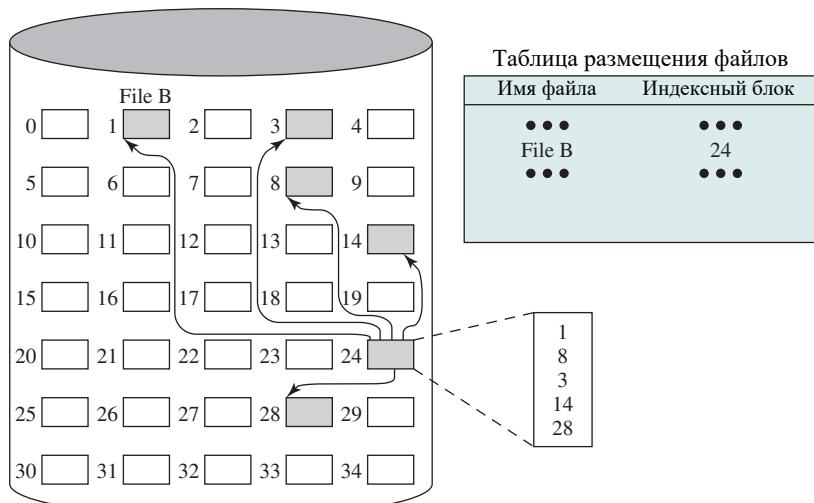
Приведенная цепочечная методика имеет одну особенность — к ней неприменим принцип локализации. Поэтому при необходимости одновременной загрузки нескольких блоков файла (довольно обычная операция при последовательной обработке) требуется серия обращений к различным участкам диска. Пожалуй, наиболее ощутим этот эффект в однопользовательской системе, но может существенно влиять и на совместно используемую систему. Для решения данной проблемы некоторые системы периодически производят уплотнение файлов (рис. 12.12).

**Индексированное размещение** решает ряд проблем непрерывного и цепочечного размещения файлов. В этом случае в таблице размещения файлов имеется отдельный одноуровневый индекс для каждого файла. Индекс содержит по одной записи для каждой порции файла. Обычно индексы файла физически не хранятся в виде части таблицы размещения файлов; вместо этого индексы файла сохраняются в отдельном блоке, и в элементе таблицы размещения файла имеется указатель на этот блок. Размещение может основываться как на блоках фиксированного размера (рис. 12.13), так и на блоках переменного размера (рис. 12.14).

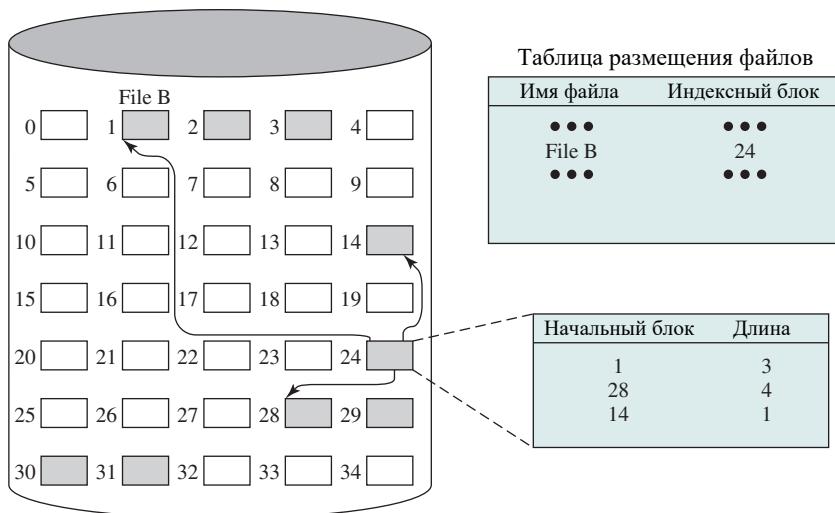
Поблочное расположение исключает внешнюю фрагментацию, в то время как размещение по порциям переменного размера повышает степень локализации. В любом случае время от времени может выполняться уплотнение файлов, которое позволяет уменьшить размер индекса при размещении с порциями переменной длины (но не при блочном распределении). Индексированное размещение поддерживает как последовательный, так и прямой доступ к файлу и потому является наиболее популярным видом файлового размещения.



**Рис. 12.12.** Цепочечное размещение файлов (после уплотнения)



**Рис. 12.13.** Индексированное размещение с порциями блоков



**Рис. 12.14.** Индексированное размещение с порциями переменного размера

## Управление свободным пространством

Так же, как требуется управление пространством, выделяемым файлам, должно быть управляемым и пространство, которое в текущий момент не выделено ни одному файлу. Применяя любую из описанных выше технологий размещения файлов, нужно знать, какие блоки на диске доступны. Для этого в дополнение к таблице размещения файлов нам необходима **таблица дискового размещения**. Рассмотрим ее возможные реализации.

### Битовые таблицы

Этот метод использует вектор, содержащий по одному биту для каждого блока диска. Каждый нуль соответствует свободному блоку, а каждая единица — используемому. Например, для представленной на рис. 12.9 схемы необходим вектор длиной 35 бит, имеющий значение

001110000111110000111111111011000

Одно из преимуществ битовой таблицы состоит в относительной простоте поиска одного свободного блока или непрерывной группы свободных блоков. Второе преимущество — ее наименьший размер из всех возможных представлений.

Однако даже такая таблица может оказаться внушительного размера. Объем памяти в байтах, необходимой для битовой карты, определяется следующим образом:

$$\frac{\text{Размер диска в байтах}}{8 \times \text{Размер блока файловой системы}}$$

Так, для диска объемом 16 Гбайт с блоками размером 512 байт битовая таблица займет около 4 Мбайт. Можем ли мы выделить под битовую таблицу 4 Мбайт основной памяти? Если да, то поиск в таблице может осуществляться без обращений к диску. Однако даже при современных объемах памяти 4 Мбайт — это слишком много для одной функции. Альтернативой является перенесение битовой таблицы на диск. Однако на диске для 4-мегабайтовой битовой таблицы понадобится около 8000 дисковых блоков.

Мы не можем затрачивать усилия на поиск в таком объеме дискового пространства всякий раз при необходимости найти один блок.

Но даже если битовая таблица размещена в основной памяти, то полный поиск в таблице может снизить производительность файловой системы до недопустимого уровня. Это особенно проявляется в случаях, когда диск практически заполнен, и свободными на нем остаются лишь несколько блоков. Соответственно, большинство файловых систем, использующих битовые таблицы, поддерживают вспомогательные структуры данных, резюмирующие содержимое поддиапазонов битовой таблицы. Например, таблицу можно логически разделить на несколько поддиапазонов одинакового размера. Сводная таблица может включать информацию о количестве свободных блоков для каждого поддиапазона и наибольшем непрерывном блоке. При потребности в некотором количестве последовательных блоков файловая система может просканировать сводную таблицу для поиска соответствующего поддиапазона, а затем произвести конкретный поиск в нем.

### **Цепочки свободных порций**

Свободные порции могут быть связаны в цепочки с использованием указателей и значений длины каждой свободной порции. Этот метод приводит к незначительным наездным расходам, поскольку нет необходимости в таблице дискового размещения — требуется только указать начало цепочки и длину первой порции. Этот способ подходит для всех методов файлового размещения. Если размещение происходит поблочно, то осуществляется выбор свободного блока в вершине цепочки и соответствующее изменение указателя на начало цепочки и значения длины. При размещении порциями переменной длины можно использовать алгоритм первого подходящего — извлекая по одному из цепочки заголовки порций до нахождения подходящей для нас. Значения указателя и длины корректируются соответствующим образом.

Описанный метод обладает и недостатками. После использования в течение некоторого времени диск становится довольно фрагментированным и большое число порций может иметь длину, сравнимую с одиночным блоком. Заметьте также, что при каждом размещении блока до того, как записывать в него данные, требуется произвести чтение блока для определения указателя на следующий за ним свободный блок, который теперь будет первым в цепочке. Если при файловой операции необходимо сразу разместить большое количество отдельных блоков, это значительно замедлит процедуру создания файла. Точно так же расходуется очень много времени и при удалении высокофрагментированных файлов.

### **Индексирование**

При индексировании свободное пространство рассматривается как файл (как и при размещении файлов, здесь используется индексная таблица). Для эффективной работы индексы должны базироваться на порциях переменного размера, а не на блоках. Поэтому каждой свободной порции на диске соответствует один элемент в таблице. Такой подход обеспечивает эффективную поддержку всех методов размещения файлов.

### **Список свободных блоков**

В этом методе каждому блоку присваивается порядковый номер, а список номеров всех свободных блоков содержится в зарезервированной области диска. В зависимости от размера диска для сохранения одного номера блока требуется 24 или 32 бит. Поэтому размер списка свободного блока в 24 или 32 раза больше соответствующей би-

товой таблицы и, таким образом, он должен храниться на диске, а не содержаться в основной памяти. Несмотря на это, предложенный метод довольно неплох. Рассмотрим следующие положения.

1. Дисковое пространство, отводимое под список свободных блоков, занимает меньше 1% общего пространства диска. При использовании 32-битовой нумерации блоков расход составляет 4 байт на каждый 512-байтовый блок.
2. Несмотря на то что список свободных блоков слишком велик для хранения в основной памяти, есть два эффективных метода хранения небольшой части списка в основной памяти.
  - Список может рассматриваться как стек, первые несколько тысяч элементов которого содержатся в основной памяти. При размещении нового блока он извлекается из вершины стека (содержащейся в основной памяти). Аналогично при освобождении блока его номер возвращается в стек. Передача данных между частями стека в основной памяти и на диске происходит только тогда, когда часть стека в основной памяти становится либо переполненной, либо пустой. Поэтому такая методика в большинстве случаев дает почти нулевое время обращения.
  - Список может быть организован в виде очереди; в основной памяти находится несколько тысяч ее элементов — как из вершины, так и из хвоста очереди. Блок размещается путем извлечения первой записи из вершины очереди, а освобождается путем добавления его номера в конец очереди. Передача данных между диском и основной памятью происходит только в том случае, когда размещенная в основной памяти часть вершины очереди освобождается или когда хвост очереди в основной памяти переполняется.

В любой из описанных стратегий можно использовать фоновый поток, который будет заниматься сортировкой списка в основной памяти с целью облегчения непрерывного распределения.

## Тома

Термин *том* (*volume*) используется несколько по-разному в разных операционных системах и системах управления файлами, но по сути том — это логический диск. [38] определяет том следующим образом.

**Том:** коллекция адресуемых секторов во вторичной памяти, которую операционная система или приложение может использовать для хранения данных. Сектора в томе не обязательно должны быть последовательными на физическом устройстве хранения; вместо этого они должны только выглядеть таковыми для операционной системы или приложения. Том может быть результатом сборки и объединения меньших томов.

В простейшем случае один диск равен одному тому. Часто диск делится на разделы, причем каждый раздел функционирует как отдельный том. Также часто несколько дисков, или разделы на нескольких дисках, рассматриваются как один том.

## Надежность

Рассмотрим следующий сценарий.

1. Пользователь А производит запрос на размещение для добавления информации к существующему файлу.
2. Запрос принимается, и таблицы дискового и файлового размещения записей обновляются в основной памяти, но не на диске.
3. В системе происходит сбой с последующей перезагрузкой.
4. Пользователь Б производит запрос на размещение файла, и файл размещается в пространстве, перекрывающемся с последним размещением по запросу пользователя А.
5. Пользователь А обращается к выделенной ему ранее (а теперь занятой пользователем Б) порции с использованием ссылки, сохраненной в файле пользователя А.

Проблема возникает из-за того, что для повышения эффективности система содержала копию таблицы дискового размещения в основной памяти. Чтобы предотвратить ошибки такого типа, при запросе на размещение файла следует выполнить ряд действий.

1. Заблокировать таблицу дискового размещения на диске. Это предотвратит внесение изменений в таблицу другими пользователями до завершения обработки данного запроса.
2. Выполнить поиск доступного пространства в таблице дискового размещения. Предполагается, что копия таблицы дискового размещения всегда содержится в основной памяти. Если это не так, то сначала таблица должна быть считана в основную память.
3. Распределить пространство, обновить таблицу дискового размещения в памяти и на диске. При цепочечном дисковом размещении следует также обновить указатели на диске.
4. Обновить таблицу размещения файлов в основной памяти и на диске.
5. Деблокировать таблицу дискового размещения.

Этот метод предохраняет от ошибок, но при частом распределении маленьких порций приводит к существенному снижению производительности. Чтобы уменьшить этот эффект, можно использовать пакетную схему размещения. В этом случае для размещения выделяется пакет свободных порций на диске. Соответствующие порции на диске помечаются как используемые. Пакетное размещение может происходить в основной памяти. При исчерпании пакета обновляется таблица размещения файлов на диске и происходит выделение нового пакета. При сбое системы порции диска, помеченные как используемые, должны быть очищены перед их повторным размещением (метод очистки зависит от конкретных характеристик файловой системы).

## 12.8. УПРАВЛЕНИЕ ФАЙЛАМИ В UNIX

Файловая система UNIX различает шесть типов файлов.

1. **Обычные.** Файлы, содержащие произвольную информацию, в нулевом или большем количестве блоков. Такие файлы содержат данные, введенные пользователем, прикладной программой или системной утилитой. Файловая система не навязы-

вает какую-либо внутреннюю структуру обычного файла, рассматривая его как поток байтов.

2. **Каталоги.** Содержат списки имен файлов и указатели на индексные узлы (*index node* — *inode*), рассматривающиеся далее. Каталоги имеют иерархическую структуру (см. рис. 12.6). Каталоги файлов представляют собой обычные файлы, обладающие особыми привилегиями защиты от записи, так что запись в них может произвести только файловая система, в то время как программам пользователя разрешен доступ для чтения.
3. **Специальные.** Используются для доступа к периферийным устройствам, таким как терминалы или принтеры. Как упоминалось в разделе 11.8, каждое устройство ввода-вывода связано с определенным файлом.
4. **Именованные каналы.** Именованные каналы, рассматривавшиеся в разделе 6.7. Файл канала буферизует данные, полученные на входе, так что процесс, который читает выход канала, получает данные в порядке поступления.
5. **Ссылки.** По сути, ссылка представляет собой альтернативное имя существующего файла.
6. **Символические ссылки.** Файл данных, содержащий имя файла, на который выполняется ссылка.

В этом разделе мы рассмотрим работу с обычными файлами.

## Индексные узлы

Современные операционные системы UNIX поддерживают несколько файловых систем, но все они отображаются в единую базовую систему для поддержки файловых систем и выделения дискового пространства для файлов. Все типы файлов UNIX управляются операционной системой посредством индексных узлов — *inode* (*index node*). Такой узел представляет собой структуру управления, которая содержит ключевую информацию, необходимую операционной системе для конкретного файла. С одним индексным узлом могут быть связаны несколько имен файлов, но активный индексный узел связан только с одним файлом, и каждый файл управляется только одним индексным узлом.

Атрибуты файла, а также разрешения доступа к нему и другая управляющая информация хранятся в индексном узле. Точная структура индексного узла варьируется от одной реализации UNIX к другой. Структура индексного узла FreeBSD, показанная на рис. 12.15, включает в себя следующие элементы данных.

- Тип и режим доступа к файлу.
- Идентификаторы владельца файла и доступа для групп.
- Время создания файла, последних чтения и записи, а также последнего обновления индексного узла системой.
- Размер файла в байтах.
- Последовательность указателей блоков (рассматривается далее).
- Количество использованных файлом физических блоков, включая блоки, использованные для хранения косвенных указателей и атрибутов.
- Количество записей каталога, ссылающихся на файл.

- Флаги, устанавливаемые ядром и пользователем и описывающие характеристики файла.
- Номер генерации файла (случайно выбранный номер, назначаемый индексному узлу при его выделении для нового файла; номер генерации используется для обнаружения ссылок на удаленные файлы).
- Размер блоков данных, на которые ссылается индексный узел (обычно такой же, как и размер блока файловой системы, но иногда — больший).
- Размер информации в расширенных атрибутах.
- Нуль или более записей расширенных атрибутов.

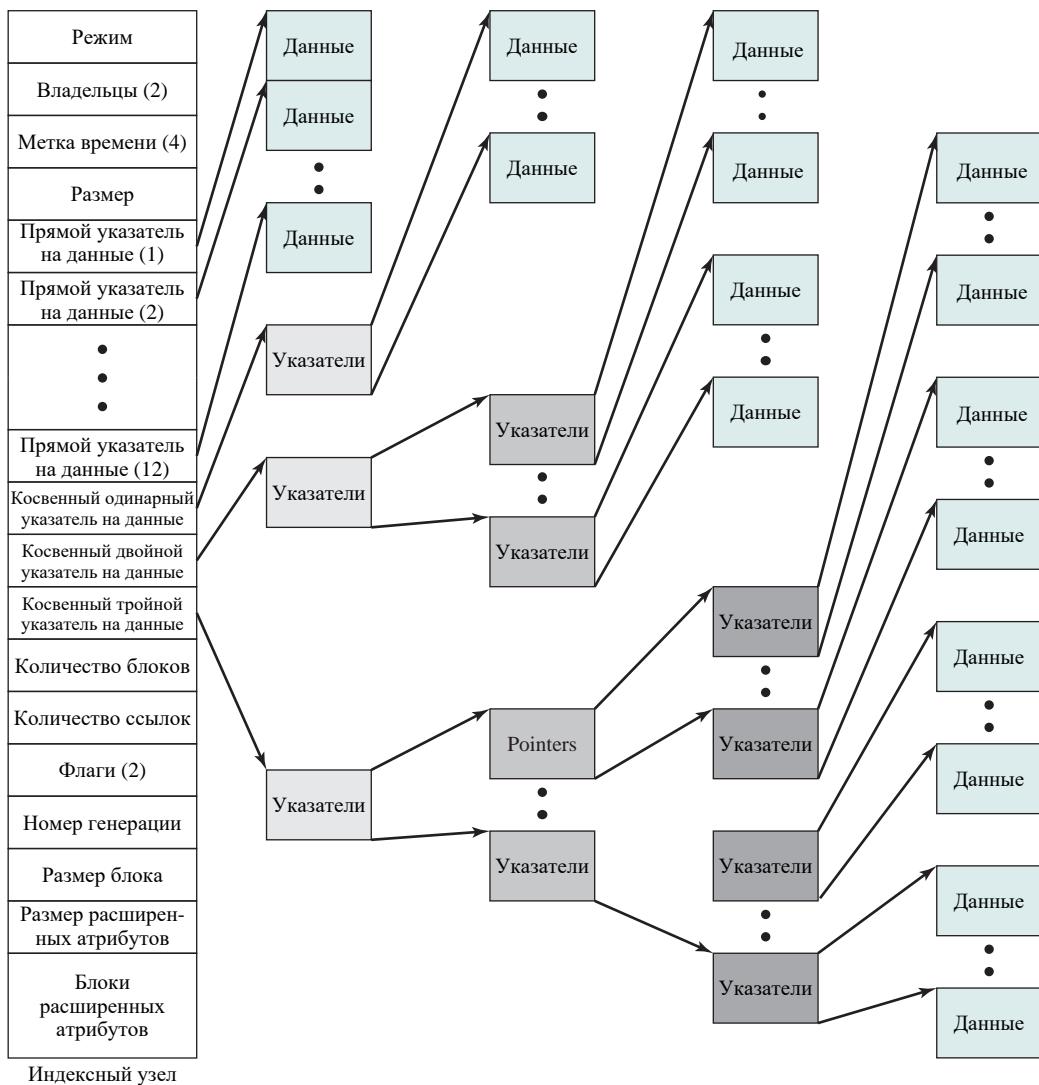


Рис. 12.15. Структура индексных узлов и файлов в FreeBSD

Значение размера блока обычно такое же, как и блока файловой системы (но иногда может быть большим). В традиционных системах UNIX использовался фиксированный размер блока 512 байт. FreeBSD имеет минимальный размер блока 4096 байт (4 Кбайт); размер блока может быть любой степенью 2, большей или равной 4096. Для типичных файловых систем размер блока составляет 8 или 16 Кбайт. Размер блока FreeBSD по умолчанию составляет 16 Кбайт.

Записи расширенных атрибутов — это записи переменной длины, используемые для хранения вспомогательных данных, которые отделены от содержимого файла. Первые два расширенных атрибута, определенных для FreeBSD, связаны с безопасностью. Первый из них поддерживает списки контроля доступа; они будут описаны в главе 15, “Безопасность операционных систем”. Второй определенный расширенный атрибут поддерживает использование меток безопасности, которые являются частью так называемой схемы обязательного контроля доступа, определенной в той же главе.

На диске есть таблица (список индексных узлов), которая содержит индексные узлы всех файлов файловой системы. Когда файл открывается, его индексный узел переносится в основную память и сохраняется в резидентной таблице индексных узлов.

## Размещение файлов

Размещение файлов выполняется поблочно, динамически, по мере необходимости. Следовательно, блоки файла на диске не обязательно образуют непрерывную область. Для отслеживания файлов используется индексный метод; часть индекса хранится в узле файла. Во всех реализациях UNIX индексный узел включает ряд прямых указателей и три косвенных (одинарный, двойной и тройной).

Индексный узел FreeBSD включает 120 байт адресной информации, организованной как пятнадцать 64-битных адресов, или указателей. Первые 12 адресов указывают на первые 12 блоков данных файла. Если файл длиннее 12 блоков, то используется один или несколько уровней косвенности.

- Тринадцатый адрес индексного узла указывает на блок на диске (блок первого уровня косвенности), содержащий следующую часть индекса. Этот блок содержит указатели на последующие блоки файла.
- Если файл содержит еще большее количество блоков, то четырнадцатый адрес узла укажет блок второго уровня косвенности. Этот блок содержит список адресов дополнительных блоков первого уровня косвенности. В свою очередь, каждый из них содержит указатели на файловые блоки.
- Если файл содержит еще больше блоков, для которых не хватает косвенности второго уровня, то пятнадцатый адрес в узле указывает на блок третьего уровня косвенности. Этот блок указывает на дополнительные косвенные блоки второго уровня.

Эта схема показана на рис. 12.15. Общее количество блоков данных в файле зависит от емкости блоков фиксированного размера в системе. В UNIX System V длина блоков равна 1 Кбайт, и каждый блок может содержать до 256 адресов блоков. Следовательно, максимальный размер файла в этой схеме превышает 500 Гбайт (табл. 12.3).

**ТАБЛИЦА 12.3. ЕМКОСТЬ ФАЙЛА FreeBSD С БЛОКАМИ РАЗМЕРОМ 4 КБАЙТА**

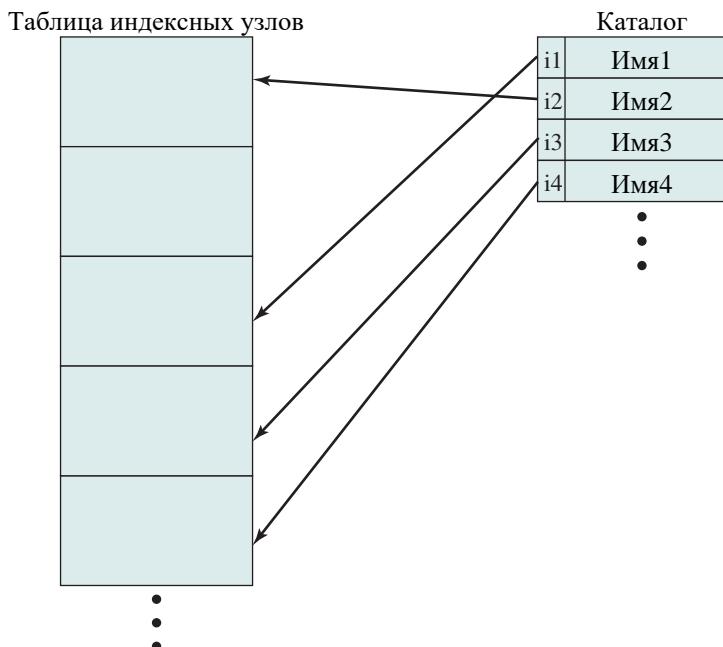
Уровень	Количество блоков	Количество байтов
Прямой	12	48 Кбайт
Один уровень косвенности	542	2 Мбайт
Два уровня косвенности	512×512=256 К	1 Гбайт
Три уровня косвенности	512×256К=128 М	512 Гбайт

Такая схема обладает рядом преимуществ.

1. Узел имеет фиксированный относительно небольшой размер и потому может длительное время содержаться в основной памяти.
2. Доступ к небольшим файлам осуществляется с малым уровнем косвенности (или и вовсе без него), что сокращает время обработки запроса и доступа к диску.
3. Теоретический максимальный размер файла достаточно велик для удовлетворения практических всех приложений.

## Каталоги

Каталоги структурированы в иерархическое дерево. Каждый каталог может содержать файлы и/или другие каталоги. Каталог внутри другого каталога называется подкаталогом. Как уже говорилось, каталог — это просто файл, который содержит список имен файлов плюс указатели на связанные с ним индексные узлы. На рис. 12.16 показана общая структура каталога. Каждая запись в каталоге содержит имя связанного файла или подкаталога плюс целое число, называемое i-номером (номером индекса). При обращении к файлу или каталогу его i-номер используется в качестве индекса в таблице индексных узлов.

**Рис. 12.16. Каталоги и индексные узлы UNIX**

## Структура тома

Файловая система UNIX находится на одном логическом диске или разделе диска и имеет следующие элементы.

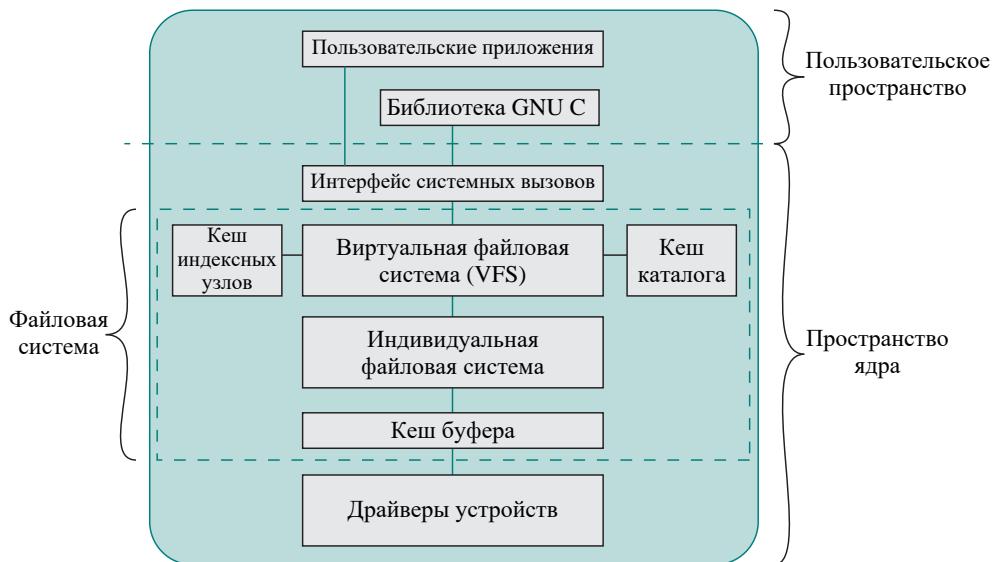
- **Загрузочный блок.** Содержит код, требуемый при загрузке операционной системы.
- **Суперблок.** Содержит атрибуты и информацию о файловой системе, такую как размер раздела и размер таблицы индексных узлов.
- **Таблица индексных узлов.** Набор индексных узлов для каждого файла.
- **Блок данных.** Место для хранения файлов данных и подкаталогов.

## 12.9. ВИРТУАЛЬНАЯ ФАЙЛОВАЯ СИСТЕМА LINUX

Linux включает универсальное и мощное средство работы с файлами, разработанное для поддержки широкого спектра систем управления файлами и структур файлов. Подход, принятый в Linux, заключается в использовании **виртуальной файловой системы** (virtual file system — VFS), которая предоставляет единый, унифицированный интерфейс файловой системы для пользовательских процессов. VFS определяет общую файловую модель, которая способна представлять общие возможности и поведение любой мыслимой файловой системы. VFS предполагает, что файлы — это объекты в запоминающем устройстве компьютера, которые имеют общие свойства независимо от целевой файловой системы или аппаратного обеспечения. Файлы имеют символические имена, позволяющие уникально их идентифицировать в пределах определенного каталога файловой системы. Файл имеет владельца, защиту от несанкционированного доступа или изменения, а также ряд других свойств. Файл может быть создан, прочитан, записан или удален. Для любой конкретной файловой системы необходим модуль преобразования, конвертирующий характеристики реальной файловой системы в характеристики, ожидаемые виртуальной файловой системой.

На рис. 12.17 показаны ключевые компоненты стратегии файловой системы Linux. Пользовательский процесс выдает системный вызов (например, чтения), используя схему VFS. VFS преобразует его во внутренний (для ядра) вызов файловой системы, который передается в функцию преобразования для конкретной файловой системы (например, файловой системы ext2). В большинстве случаев функция преобразования представляет собой простое отображение одной схемы функциональных вызовов файловой системы на другую. В некоторых случаях функция преобразования оказывается более сложной.

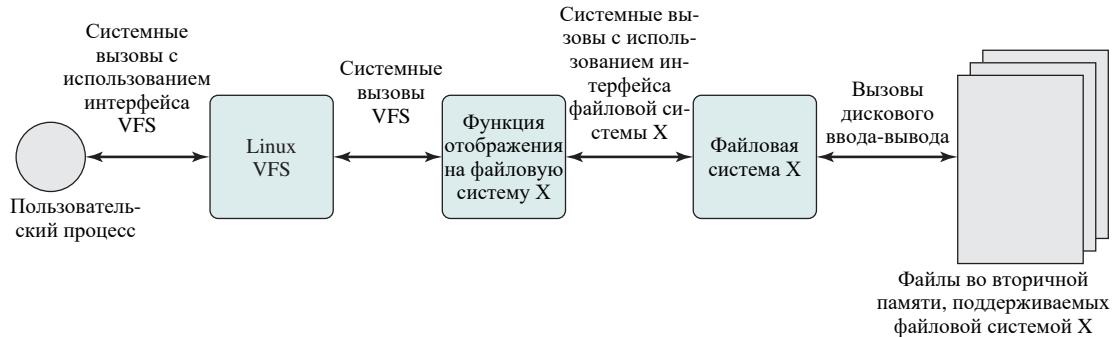
Например, некоторые файловые системы используют таблицу размещения файлов (file allocation table — FAT), в которой хранится положение каждого файла в дереве каталогов. В этих файловых системах каталоги не являются файлами. Для таких файловых систем функция отображения должна иметь возможность при необходимости динамически создавать файлы, соответствующие каталогам. В любом случае исходный пользовательский вызов файловой системы преобразуется в вызов, который является “родным” для целевой файловой системы. Затем вызывается программное обеспечение целевой файловой системы для выполнения запрошеннной функции над файлом или каталогом, находящимся во вторичной памяти под ее управлением. Результаты операции передаются обратно пользователю аналогичным образом.



**Рис. 12.17.** Виртуальная файловая система Linux

На рис. 12.18 показана роль, которую VFS играет в ядре Linux. Когда процесс инициирует файловый системный вызов (например, чтение), ядро вызывает функцию VFS. Эта функция обрабатывает запрос и инициирует вызов функции целевой файловой системы. Этот вызов проходит через функцию отображения, которая преобразует вызов VFS в вызов целевой файловой системы. VFS не зависит ни от какой файловой системы, поэтому реализация функции отображения должна быть частью реализации файловой системы Linux. Целевая файловая система преобразует запрос файловой системы в команды, ориентированные на устройство, которые передаются соответствующему драйверу устройства.

VFS является объектно-ориентированной схемой. Поскольку она написана на С, а не на языке, который поддерживает объектно-ориентированное программирование (например, C++ или Java), объекты VFS реализуются просто как структуры данных С. Каждый объект содержит как данные, так и указатели на функции для работы с данными, реализованные в файловой системе.



**Рис. 12.18.** Концепция VFS Linux

Четырьмя основными типами объектов в VFS являются следующие.

- **Суперблок.** Представляет конкретную смонтированную файловую систему.
- **Индексный узел.** Представляет конкретный файл.
- **Запись каталога.** Представляет конкретную запись каталога.
- **Файл.** Представляет открытый файл, связанный с процессом.

Эта схема основана на концепциях, используемых в файловых системах UNIX, описанных в разделе 12.7. Ключевые концепции файловой системы UNIX, которые следует запомнить, следующие. Файловая система состоит из иерархической организации каталогов. Каталог — это то же, что называется папкой на многих платформах, отличных от UNIX, и он может содержать файлы и/или другие каталоги. Поскольку каталог может содержать другие каталоги, таким образом формируется древовидная структура. Путь через древовидную структуру из корня состоит из последовательности записей каталога, заканчивающейся либо записью каталога, либо именем файла. В UNIX каталог реализован в виде файла, в котором перечислены файлы и каталоги, содержащиеся в нем. Таким образом, файловые операции могут выполняться как с файлами, так и с каталогами.

## Суперблок

Объект суперблока хранит информацию, описывающую конкретную файловую систему. Как правило, суперблок соответствует суперблоку файловой системы или управляемому блоку файловой системы, который хранится в специальном секторе на диске.

Суперблок состоит из нескольких элементов данных, включая следующее.

- Устройство, на котором смонтирована файловая система
- Базовый размер блока файловой системы
- Флаг, указывающий, что суперблок был изменен, но не записан обратно на диск
- Тип файловой системы
- Флаги, такие как флаг “только для чтения”
- Указатель на корень каталога файловой системы
- Список открытых файлов
- Семафор для управляемого доступа к файловой системе
- Список операций суперблока

Последний элемент предыдущего списка относится к объекту операций, содержащемуся в объекте суперблока. Объект операций (`super_operations`) определяет методы (функции) объекта, которые ядро может вызывать для объекта суперблока. Методы, определенные для объекта суперблока, включают следующие.

- `alloc_inode`. Выделение памяти для индексного узла.
- `write_inode`. Запись данного индексного узла на диск.
- `put_super`. Вызывается VFS при размонтировании для освобождения данного суперблока.
- `statfs`. Получение статистики файловой системы.
- `remount_fs`. Вызывается VFS, когда файловая система перемонтируется с новыми параметрами.

## Индексный узел

Индексный узел связан с каждым файлом. Соответствующий объект содержит всю информацию об именованном файле, кроме его имени и фактического содержимого файла данных. Информация, содержащаяся в объекте индексного узла, включает владельца, группу, разрешения, значения времени обращений к файлу, размер хранящихся в нем данных и количество ссылок.

Объект индексного узла включает в себя также объект операций индексного узла. Последний описывает реализованные файловой системой функции, которые VFS может вызывать для индексного узла. Методы, определенные для объекта индексного узла, включают следующие.

- `create`. Создает новый индексный узел для обычного файла, связанного с объектом записи каталога в некотором каталоге.
- `lookup`. Ищет в каталоге индексный узел, соответствующий имени файла.
- `mkdir`. Создает новый индексный узел для каталога, связанного с объектом записи каталога в некотором каталоге.

## Запись каталога

Запись каталога (*directory entry, dentry*) является определенным компонентом пути. Компонент может быть либо именем каталога, либо именем файла. Объекты записи каталога облегчают быстрый поиск файлов и каталогов и используются для этой цели в кеше записей каталога. Объект записи каталога содержит указатель на индексный узел и суперблок. Он также включает указатель на запись родительского каталога и указатель на все подчиненные записи каталогов.

## Файл

Объект файла (файловый объект) используется для представления файла, открытого процессом. Объект создается в ответ на системный вызов `open()` и уничтожается в ответ на системный вызов `close()`. Файловый объект состоит из нескольких элементов, включая следующие.

- Объект записи каталога, связанный с файлом
- Файловая система, содержащая файл
- Счетчик использования файлового объекта
- Пользовательский идентификатор пользователя
- Пользовательский идентификатор группы
- Файловый указатель, который представляет собой текущую позицию в файле, для которой будет выполняться следующая операция

Файловый объект также включает в себя объект операций индексного узла, описывающий реализованные файловой системой функции, которые VFS может вызывать для файлового объекта. Методы, определенные для объекта файла, включают чтение, запись, открытие, освобождение и блокировку.

## Кеши

VFS использует три кеша для повышения производительности.

1. **Кеш индексных узлов.** Поскольку каждый файл и каталог представлен индексным узлом VFS, команда просмотра каталога или команда доступа к файлу приводит к обращению к нескольким индексным узлам. Кеш индексных узлов хранит недавно посещенные узлы для ускорения обращений.
2. **Кеш каталогов.** Кеш каталогов хранит соответствие между полными именами каталогов и номерами их индексных узлов. Это ускоряет процесс чтения каталога.
3. **Буферный кеш.** Буферный кеш не зависит от файловых систем и интегрирован в механизмы, которые ядро Linux использует для выделения, а также чтения и записи буферов данных. Поскольку реальные файловые системы читают данные с базовых физических дисков, это приводит к запросам к драйверам блочных устройств на чтение физических блоков с устройств, которыми они управляют. Таким образом, если часто нужны одни и те же данные, они будут извлекаться из буферного кеша, а не физически считываться с диска.

## 12.10. ФАЙЛОВАЯ СИСТЕМА WINDOWS

Разработчики Windows NT спроектировали новую файловую систему, New Technology File System (NTFS), предназначенную для удовлетворения высоких требований рабочих станций и серверов. Примерами приложений такого уровня могут служить:

- клиент/серверные приложения, такие как файловые серверы, вычислительные серверы и серверы баз данных;
- инженерные и научные приложения с интенсивным использованием ресурсов;
- сетевые приложения больших корпоративных систем.

В этом разделе приведен обзор NTFS.

### Ключевые возможности NTFS

NTFS представляет собой гибкую и мощную файловую систему, которая, как мы увидим, построена на простой и элегантной модели. К наиболее достойным внимания особенностям NTFS относятся следующие.

- **Способность восстановления данных.** Первой в списке требований к новой файловой системе Windows была способность восстановления данных при полном отказе системы и сбоях дисков. В таких случаях NTFS способна реконструировать дисковые тома и вернуть их в согласованное состояние. Это достигается посредством использования модели обработки транзакций для операций обмена в файловой системе; каждый значительный обмен рассматривается как атомарное действие, которое либо выполняется полностью, либо не выполняется вовсе. Каждая транзакция, которая обрабатывалась в момент сбоя, впоследствии либо отменяется, либо доводится до завершения. Кроме того, NTFS использует избыточное хранение критических данных файловой системы, поэтому сбой сектора диска не приводит к потере данных, описывающих структуру и состояние файловой системы.

- **Безопасность.** Для обеспечения безопасности NTFS использует объектную модель Windows. Открытый файл реализуется как файловый объект с дескриптором, определяющим атрибуты безопасности. Дескриптор безопасности сохраняется как атрибут для каждого файла на диске.
- **Диски и файлы больших объемов.** NTFS поддерживает очень большие диски и файлы более эффективно по сравнению с другими файловыми системами, такими как FAT.
- **Множественные потоки данных.** Содержимое файла рассматривается как поток байтов. В NTFS можно определить несколько потоков данных для одного файла. Примером использования этой особенности служит использование Windows удаленных системами Macintosh для хранения и получения файлов. В компьютерах Macintosh каждый файл состоит из двух компонентов: данных файла и ветви ресурса, которая содержит информацию о файле. NTFS рассматривает эти компоненты как два потока данных.
- **Журнилизирование.** NTFS ведет журнал всех изменений, внесенных в файлы в томах. Программы, такие как поиск на рабочем столе, могут читать журнал, чтобы определить, какие файлы были изменены.
- **Сжатие и шифрование.** Отдельные файлы и целые каталоги могут быть прозрачно сжаты и/или зашифрованы.
- **Жесткие и символические ссылки.** Чтобы поддерживать POSIX, Windows всегда поддерживала “жесткие ссылки”, которые позволяют одному файлу быть доступным по нескольким именам путей в одном томе. Начиная с Windows Vista поддерживаются “символические ссылки”, которые позволяют файлу или каталогу быть доступными по нескольким путям, даже если они находятся на разных томах. Windows также поддерживает “точки монтирования”, которые позволяют томам появляться в точках соединения других томов, а не именоваться буквами диска, такими как “D:”.

## Том NTFS и файловая структура

NTFS использует следующие концепции дискового хранения.

- **Сектор.** Наименьшая единица физического хранения на диске. Размер данных в байтах является степенью двойки и почти всегда равен 512 байт.
- **Кластер.** Один или несколько последовательных секторов на одной дорожке. Размер кластера в секторах является степенью двойки.
- **Том.** Логический раздел диска, состоящий из некоторого количества кластеров и используемый файловой системой для распределения пространства. В любой момент времени том состоит из информации файловой системы, набора файлов и нераспределенного пространства, оставшегося в томе, которое может быть выделено файлам. Том может занимать как весь диск, так и его часть или охватывать несколько дисков. При использовании RAID 5 том состоит из полос, охватывающих несколько дисков. Максимальный размер тома NTFS составляет  $2^{64}$  кластеров.

Кластер является фундаментальной единицей размещения в файловой системе NTFS, которая не распознает секторы. Предположим, например, что размер каждого сектора

составляет 512 байт и что система настроена так, что в одном кластере содержатся по два сектора (один кластер равен 1 Кбайт). При создании пользователем файла размером 1600 байт файлу отводятся два кластера. Если впоследствии пользователь обновляет файл и он увеличивается до 3200 байт, то ему выделяются еще два кластера. Кластеры, выделяемые файлу, не обязательно должны образовывать непрерывный блок; в NTFS допускается фрагментация файла на диске. В настоящее время максимальный размер файла, поддерживаемый NTFS, составляет  $2^{32}$  кластеров, что эквивалентно максимум  $2^{48}$  байт. Максимальный размер кластера —  $2^{16}$  байт.

Использование кластеров для размещения файлов делает NTFS независимой от размеров физических секторов. Это позволяет NTFS без препятствий поддерживать нестандартные диски с размером сектора, не равным 512 байт, а также эффективно поддерживать диски очень большой емкости и файлы больших размеров посредством большего размера кластера. Эффективность обеспечивается тем, что файловая система должна отслеживать каждый кластер, выделенный файлу; использование кластеров большего размера облегчает эту задачу.

В табл. 12.4 приведены размеры по умолчанию кластеров системы NTFS. Эти размеры зависят от размера тома. Размер кластера, используемого в конкретном томе, устанавливается системой NTFS при форматировании.

**Таблица 12.4. РАЗДЕЛЫ NTFS И РАЗМЕРЫ КЛАСТЕРОВ**

Размер тома	Количество секторов в кластере	Размер кластера
≤ 512 Мбайт	1	512 байт
512 Мбайт – 1 Гбайт	2	1 Кбайт
1 Гбайт – 2 Гбайт	4	2 Кбайт
2 Гбайт – 4 Гбайт	8	4 Кбайт
4 Гбайт – 8 Гбайт	16	8 Кбайт
8 Гбайт – 16 Гбайт	32	16 Кбайт
16 Гбайт – 32 Гбайт	64	32 Кбайт
> 32 Гбайт	128	64 Кбайт

### Схема тома NTFS

NTFS использует простой и в то же время мощный подход к организации информации на томе диска. Каждый элемент тома представляет собой файл, и каждый файл состоит из набора атрибутов (даже данные, хранящиеся в файле, рассматриваются как атрибут). При такой простой структуре достаточно небольшого количества функций общего назначения для организации и управления файловой системой.



**Рис. 12.19. Схема тома NTFS**

На рис. 12.19 показана схема тома NTFS, состоящего из четырех областей. Первые несколько секторов любого тома занимает **загрузочный сектор раздела** (несмотря на название, размер этой области может быть до 16 секторов), содержащий информацию о схеме тома и структурах файловой системы, а также начальная загрузочная информация и код загрузки. За этой областью следует **главная файловая таблица** (master file table — MFT), содержащая информацию обо всех файлах и папках (каталогах) этого тома NTFS, а также информацию о свободном пространстве. По сути, MFT представляет собой список всего содержимого тома NTFS, организованный в виде множества строк в табличной структуре.

За областью MFT следует область, содержащая **системные файлы**. Среди файлов в этой области находятся следующие.

- **MFT2.** Зеркальное отображение первых трех строк MFT, используемых для гарантированного доступа к MFT в случае сбоя одного сектора.
- **Журнальный файл.** Список шагов транзакций, используемый при восстановлении данных в NTFS.
- **Битовая карта кластеров.** Представление тома, указывающее используемые кластеры.
- **Таблица определения атрибутов.** Определяет типы атрибутов, поддерживаемых этим томом, и показывает, могут ли они быть проиндексированы, а также могут ли они быть восстановлены во время системной операции восстановления.

### Главная файловая таблица

Сердцем файловой системы в Windows является MFT. MFT организована в виде таблицы строк длиной 1024 байта, именуемых записями. Каждая строка описывает файл или папку этого тома, включая саму MFT, которая рассматривается как файл. Если содержимое файла достаточно мало, то он полностью размещается в строке MFT. В противном случае строка для этого файла будет содержать частичную информацию, а оставшаяся часть файла будет распределена среди доступных кластеров тома с указателями на эти кластеры в строке MFT для данного файла.

Каждая запись MFT состоит из набора атрибутов, служащих для определения характеристик файла (или папки), а также для определения содержимого файла. В табл. 12.5 перечислены атрибуты, которые могут находиться в строке MFT (обязательные атрибуты выделены темным фоном).

**Таблица 12.5. Типы атрибутов файлов и каталогов в Windows NTFS**

Тип атрибута	Описание
<b>Стандартная информация</b>	Включает атрибуты доступа (только для чтения, чтение/запись и т.д.); временные метки, включая время создания и последней модификации файла; количество каталогов, указывающих на файл (счетчик связей)
<b>Список атрибутов</b>	Список атрибутов, составляющих файл, и ссылка на другую запись MFT, в которой размещены атрибуты. Используется, когда все атрибуты не помещаются в одну запись MFT
<b>Имя файла</b>	Файл (или каталог) должен иметь одно или несколько имен

Тип атрибута	Описание
<b>Дескриптор безопасности</b>	Определяет владельца файла и пользователей, которым разрешен доступ к файлу
<b>Данные</b>	Содержимое файла. Файл содержит один неименованный атрибут данных по умолчанию и может иметь один или несколько именованных атрибутов данных
<b>Корневой индекс</b>	Используется для реализации папок
<b>Размещение индекса</b>	Используется для реализации папок
<b>Информация о томе</b>	Включает информацию, относящуюся к тому, например версию и имя тома
<b>Битовая карта</b>	Карта, предоставляющая записи, используемые MFT или каталогом

## Способность восстановления данных

NTFS позволяет восстанавливать согласованное состояние файловой системы после краха системы или сбоя диска. Ключевыми элементами, поддерживающими восстановление, являются следующие (рис. 12.20).

- **Диспетчер ввода-вывода.** Включает драйвер NTFS, обрабатывающий основные функции NTFS — открытие и закрытие файла, чтение, запись. Кроме того, может использоваться программный модуль RAID (FTDISK).
- **Сервис системного журнала.** Обеспечивает регистрацию дисковых записей. Системный журнал используется для восстановления тома NTFS при сбое в системе (без запуска утилиты проверки файловой системы).
- **Диспетчер кеш-памяти.** Отвечает за кеширование чтения и записи файлов для улучшения производительности. Диспетчер кеша оптимизирует дисковый ввод-вывод.
- **Диспетчер виртуальной памяти.** NTFS обращается к кешированным файлам путем отображения файловых ссылок в ссылки виртуальной памяти и чтения и записи виртуальной памяти.

Важно обратить внимание на то, что используемые системой NTFS процедуры восстановления созданы для восстановления метаданных файловой системы, а не содержащего файлов. Поэтому пользователь не должен потерять том или структуру каталогов и файлов после сбоя системы. Тем не менее полное восстановление данных пользователей файловой системой не гарантируется. Для полного восстановления, включая данные пользователей, необходимы более мощные и ресурсоемкие средства восстановления.

Суть процедуры восстановления данных в NFTS заключается в протоколировании. Каждая операция, изменяющая файловую систему, обрабатывается как транзакция. Каждая подоперация транзакции, видоизменяющая важные структуры данных файловой системы, перед записью на диск протоколируется в системном журнале.

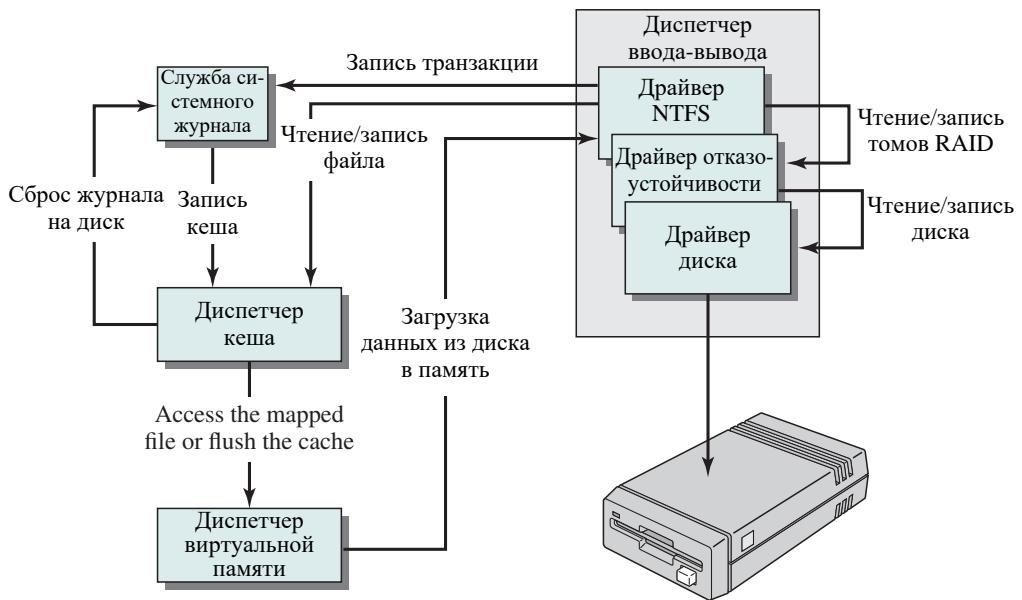


Рис. 12.20. Компоненты NTFS

При сбое в системе он позволяет частично выполненной транзакции при восстановлении быть либо выполненной повторно, либо отмененной.

В общих чертах эти этапы описаны в [212].

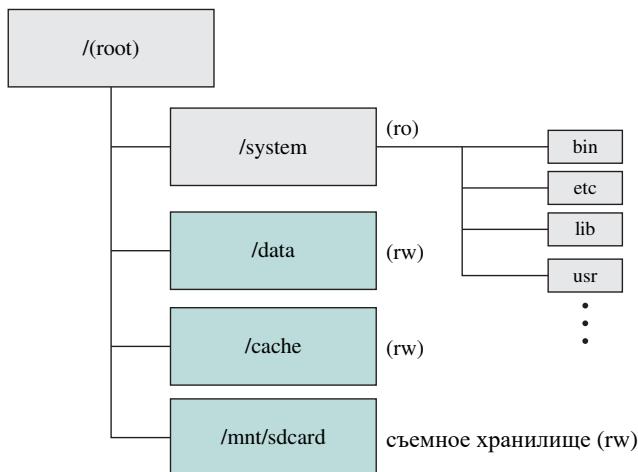
1. NTFS сначала вызывает системный журнал для записи в него (в кеш-памяти) всех транзакций, модифицирующих структуру тома.
2. NTFS модифицирует том (в кеш-памяти).
3. Диспетчер кеш-памяти сбрасывает системный журнал на диск.
4. После того как обновленный системный журнал успешно сброшен на диск, диспетчер кеш-памяти сбрасывает на диск изменения тома.

## 12.11. УПРАВЛЕНИЕ ФАЙЛАМИ В ANDROID

### Файловая система

Android использует возможности управления файлами, встроенные в Linux. Каталог файловой системы Android похож на то, что мы видим в типичной установке Linux, с некоторыми функциями, специфичными для Android.

На рис. 12.21 показаны верхние уровни типичного каталога файловой системы Android. **Системный каталог** содержит основные части операционной системы, включая системные бинарные файлы, системные библиотеки и файлы конфигурации. Он также включает в себя базовый набор приложений для Android, таких как Будильник, Калькулятор или Камера. Образ системы заблокирован, и пользователям файловой системы предоставляется доступ только для чтения. Остальные каталоги, показанные на рис. 12.21, предназначены для чтения и записи.



ro: смонтировано только для чтения

rw: смонтировано для чтения и записи

**Рис. 12.21.** Типичное дерево каталогов Android

**Каталог данных** предоставляет основное местоположение, используемое приложениями для хранения своих закрытых данных. Этот раздел содержит данные пользователя, такие как контакты, SMS, настройки и все приложения Android, которые вы установили. Когда пользователь выполняет сброс устройства до заводских настроек, этот раздел стирается. После этого устройство будет в состоянии первого включения или в том состоянии, в каком оно было после последней официальной или пользовательской прошивки ПЗУ. При установке в системе нового приложения в отношении каталога данных среди прочего предпринимаются следующие действия.

- Файл .apk (Android package) помещается в каталог /data/app.
- Ориентированные на приложение библиотеки устанавливаются в каталог /data/data/<имя\_приложения>. Это область изолированной программной среды приложения (“песочница”), недоступная для других приложений.
- Устанавливаются файловые базы данных приложения.

**Каталог кеша** используется операционной системой для временного хранения. Это раздел, где Android хранит часто используемые данные и компоненты приложений. Очистка кеша не влияет на личные данные пользователя, а просто избавляет от имеющихся там данных, которые автоматически восстанавливаются, когда пользователь продолжает использовать устройство.

Каталог /mnt/sdcard — это не раздел внутренней памяти устройства, а SD-карта, представляющая собой энергонезависимую карту памяти, которая может быть вставлена в устройство Android. SD-карта — это съемная карта памяти, которую пользователь может извлечь и подключить к своему компьютеру. С точки зрения использования, это пространство хранения для чтения/записи данных, аудио- и видеофайлов пользователя. На устройствах с внутренней и внешней SD-картой раздел /sdcard всегда используется для обозначения внутренней SD-карты. Для внешней SD-карты, если таковая имеется в наличии, используется альтернативный раздел, который отличается от устройства к устройству.

## SQLite

Особого упоминания заслуживает SQLite, библиотека для работы с данными на основе SQL. Язык структурированных запросов (Structured Query Language — SQL) представляется в качестве стандартного средства для определения реляционной базы данных и доступа к ней со стороны локального или удаленного пользователя или приложения. Язык структурированных запросов (SQL), первоначально разработанный IBM в середине 1970-х годов, является стандартизованным языком, который можно использовать для определения схемы, работы и запроса данных в реляционной базе данных. Существует несколько версий стандарта ANSI/ISO и множество различных реализаций, но все они используют один и тот же базовый синтаксис и семантику.

SQLite — это наиболее широко используемый в мире механизм баз данных SQL. Он призван предоставлять оптимизированные СУБД на основе SQL, пригодные для встраиваемых систем и других систем с ограниченной памятью. Полная библиотека SQLite может быть реализована с размером менее 400 Кбайт. Во время компиляции можно отключить ненужные функциональные возможности, что при желании позволяет уменьшить размер библиотеки до 190 Кбайт.

В отличие от других систем управления базами данных, SQLite не является отдельным процессом, доступ к которому осуществляется из клиентского приложения. Вместо этого библиотека SQLite компонуется с прикладной программой и, таким образом, становится ее неотъемлемой частью.

## 12.12. Резюме

Система управления файлами представляет собой набор системных программ, которые предоставляют пользователям и прикладным программам возможность использования файлов, включая работу с файлами, обслуживание каталогов и управление правами доступа. Система управления файлами обычно рассматривается как системный сервис, обслуживаемый операционной системой, а не как часть операционной системы. Тем не менее в любой системе как минимум часть функций управления файлами выполняется операционной системой.

Файл представляет собой набор записей. Способ доступа к записям файла определяется его логической организацией и в некоторой степени — физической организацией на диске. Когда изначально предполагается обрабатывать файл целиком, наиболее простым и подходящим способом оказывается последовательная организация файла. Если необходим как последовательный, так и произвольный доступ к файлу, то в этом случае наилучшим решением будет индексированный последовательный файл. Если же доступ к файлу предполагается осуществлять исключительно случайным образом, то наиболее подходящей организацией файла является индексированная или хешированная организация.

Какой бы тип файлов ни был выбран, для полноценной работы необходима позволяющая иерархически организовывать файлы возможность использовать каталоги. Такая организация облегчает управление как файлами, так и правами доступа и другими сервисами.

Файловые записи, даже если они имеют фиксированный размер, обычно не соответствуют размерам физических блоков диска. Поэтому, естественно, необходима некоторая стратегия группирования записей. Компромисс между сложностью, производитель-

ностью и использованием дискового пространства определяет применяемую стратегию группирования.

Ключевой функцией любой схемы управления файлами является управление дисковым пространством. Частью этой функции является стратегия выделения дисковых блоков файлам. Для этого используются различные методы, а для отслеживания распределения каждого файла применяются различные структуры данных. Кроме того, требуется управление свободным пространством диска. Эта функция главным образом состоит в ведении таблицы распределения диска, указывающей, какие блоки в настоящий момент свободны.

## 12.13. КЛЮЧЕВЫЕ ТЕРМИНЫ, КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАЧИ

### Ключевые термины

/mnt/sdcard	Индексный узел	Распределение
База данных	Каталог данных	Система управления файлами
Битовая таблица	Каталог кеша	Системные файлы
Блок	Каталог файлов	Системный каталог
Виртуальная файловая система (VFS)	Ключевое поле	Смешанная организация файла
Главная таблица файлов (MFT)	Логический ввод-вывод	Таблица размещения файлов (FAT)
Драйвер устройства	Метод доступа	Таблица распределения диска
Загрузочный сектор раздела	Непрерывный файл	Текущий каталог
Запись	Поле	Файл
Имя файла	Последовательный файл	Файловая система
Индексированный файл	Путь	Физический ввод-вывод
Индексное размещение файлов	Рабочий каталог	Хешированный файл
Индексно-последовательный файл	Размещение файла	Цепочечное размещение файла

### Контрольные вопросы

- 12.1. В чем различие между полем и записью?
- 12.2. В чем различие между файлом и базой данных?
- 12.3. Что представляет собой система управления файлами?
- 12.4. Какие критерии следует учитывать при выборе организации файла?
- 12.5. Перечислите и кратко опишите пять типов организации файлов.
- 12.6. Почему среднее время поиска записи в индексно-последовательном файле меньше, чем в последовательном файле?

- 12.7. Какие типичные операции могут выполняться с каталогом?
- 12.8. Как связаны полное имя файла и рабочий каталог?
- 12.9. Какие типичные права доступа могут быть предоставлены (или в которых может быть отказано) определенному пользователю по отношению к некоторому файлу?
- 12.10. Перечислите и кратко опишите три метода группирования.
- 12.11. Перечислите и кратко опишите три метода размещения файлов.

## Задачи

12.1. Обозначим:

$B$  — размер блока;

$R$  — размер записи;

$P$  — размер указателя блока;

$F$  — коэффициент группирования (ожидаемое количество записей в блоке).

Выведите формулу для  $F$  для всех методов группирования, показанных на рис. 12.8.

- 12.2. Одно из решений проблемы потери непрерывности расположения файла на диске при распределении по требованию заключается в увеличении размера выделяемых порций при увеличении файла. Например, начнем с размера порции, равного одному блоку, и для каждого последующего размещения будем удваивать этот размер. Рассмотрим файл с  $n$  записями с коэффициентом группирования  $F$  и предположим, что в качестве таблицы размещения файлов используется простой одноуровневый индекс.
  - а. Определите верхний предел количества элементов в таблице размещения файлов как функцию  $F$  и  $n$ .
  - б. Чему равно максимальное количество никогда не используемого пространства, выделенного файлу?
- 12.3. Какую организацию файла следует выбрать для получения максимальной эффективности (в плане скорости доступа, использования дискового пространства и простоты обновления), если данные:
  - а. обновляются не часто, при частом обращении в произвольном порядке;
  - б. обновляются часто, при относительно частом доступе;
  - в. обновляются часто, при частом обращении в произвольном порядке.
- 12.4. Рассмотрите В-дерево на рис. 12.4, в. Покажите, каким будет результат вставки в него ключа 97.
- 12.5. Альтернативный алгоритм вставки в В-дерево состоит в следующем: по мере перемещения алгоритма вставки вниз по дереву каждый обнаруженный полный узел немедленно разделяется, даже если может оказаться, что это разделение не было необходимым.
  - а. Каково преимущество данного алгоритма?
  - б. В чем заключаются его недостатки?

- 12.6.** И время поиска, и время вставки для В-дерева зависят от его высоты. Мы хотим выяснить меру времени поиска или вставки в худшем случае. Рассмотрите В-дерево степени  $d$ , с общим количеством  $n$  ключей. Запишите неравенство, которое показывает верхнюю границу высоты  $h$  дерева в зависимости от  $d$  и  $n$ .
- 12.7.** Игнорируя накладные расходы на дескрипторы каталогов и файлов, рассмотрите файловую систему, в которой файлы хранятся в блоках размером 16 Кбайт. Для каждого из следующих размеров файлов рассчитайте процент потерянного файлового пространства из-за неполного заполнения последнего блока: 41 600 байт; 640 000 байт; 4 064 000 байт.
- 12.8.** Каковы недостатки использования каталогов?
- 12.9.** Каталоги могут быть реализованы либо как “специальные файлы” с ограниченным количеством способов доступа, либо как обычные файлы данных. В чем достоинства и недостатки каждого подхода?
- 12.10.** Некоторые операционные системы имеют древовидную файловую структуру с сильно ограниченной высотой дерева. Как это влияет на работу пользователей? Насколько это упрощает проектирование файловой системы (если такое упрощение имеет место)?
- 12.11.** Рассмотрим иерархическую файловую систему, в которой свободное дисковое пространство содержится в списке свободного пространства.
- Предположим, что указатель на свободное пространство утерян. Способна ли система воспроизвести список свободного пространства?
  - Предложите схему, гарантирующую, что при однократном сбое памяти этот указатель никогда не будет утерян.
- 12.12.** В UNIX System V длина блока составляет 1 Кбайт, а каждый блок может содержать до 256 адресов блоков. Каков максимальный размер файла при использовании схемы индексных узлов?
- 12.13.** Рассмотрим организацию файлов UNIX, представленную на рис. 12.15. Пусть в каждом узле содержится 12 прямых указателей блоков, а также одинарный, двойной и тройной указатели. Далее положим, что размер системного блока и размер дискового сектора равны 8 Кбайт. Допустим, что размер указателя дискового блока — 32 бита (8 бит для указания физического диска и 24 бита для указания физического блока).
- Какой максимальный размер файла, поддерживаемый в этой системе?
  - Какой максимальный размер раздела файловой системы, поддерживаемого в этой системе?
  - Предположим, что в основной памяти не содержится ничего, кроме индексного узла. Сколько обращений к диску потребуется для доступа к байту в позиции 13 423 956?

ЧАСТЬ VI

---

Дополнительные  
темы

Tlgm: @it\_boooks

# ГЛАВА 13

# ВСТРОЕННЫЕ ОПЕРАЦИОННЫЕ СИСТЕМЫ

В ЭТОЙ ГЛАВЕ...

## 13.1. Встроенные системы

- Концепции встроенных систем
- Прикладные и специализированные процессоры
- Микропроцессоры
- Микроконтроллеры
- Глубоко встроенные системы

## 13.2. Характеристики встроенных операционных систем

- Исходные и целевые среды
- Начальный загрузчик
- Ядро
- Корневая файловая система
- Подходы к разработке
- Адаптация существующей коммерческой операционной системы
- Специально разработанная встроенная операционная система

## 13.3. Встроенная система Linux

- Характеристики встроенной системы Linux
- Размер ядра
- Объем памяти
- Прочие характеристики
- Файловые системы встроенного Linux
- Преимущества встроенных систем Linux
- μLinux
- Сравнение с полномасштабной системой Linux
- μClibc
- Android

### 13.4. TinyOS

- Беспроводные сети датчиков
- Цели TinyOS
- Компоненты TinyOS
- Планировщик в TinyOS
- Пример конфигурации
- Интерфейс ресурсов TinyOS

### 13.5. Ключевые термины, контрольные вопросы и задачи

- Ключевые термины
- Контрольные вопросы
- Задачи

## УЧЕБНЫЕ ЦЕЛИ

- Разъяснить понятие встроенной системы.
- Уяснить характеристики встроенных операционных систем.
- Объяснить отличия обычной ОС Linux от встроенной.
- Описать архитектуру и основные функциональные возможности операционной системы TinyOS.

В этой главе рассматриваются встроенные операционные системы — одна из самых важных и широко употребляемых категорий операционных систем. Среда встроенной системы накладывает на операционную систему особые жесткие требования; кроме того, предусматриваются совсем иные стратегии проектирования, чем в случае обычных операционных систем.

Начнем эту главу с краткого обзора понятия встроенных систем, а затем перейдем к рассмотрению принципов действия встроенных операционных систем. И наконец в этой главе будут представлены два самых разных подхода к проектированию встроенных операционных систем: встроенной операционной системы Linux и операционной системы TinyOS. Еще один подход к проектированию встроенных операционных систем рассматривается в приложении Р, “eCos”, на примере eCos — еще одной очень важной встроенной ОС.

## 13.1. ВСТРОЕННЫЕ СИСТЕМЫ

В этом разделе представлено понятие встроенной системы. Для понимания этой концепции требуется также пояснить, чем микропроцессор отличается от микроконтроллера.

### Концепции встроенных систем

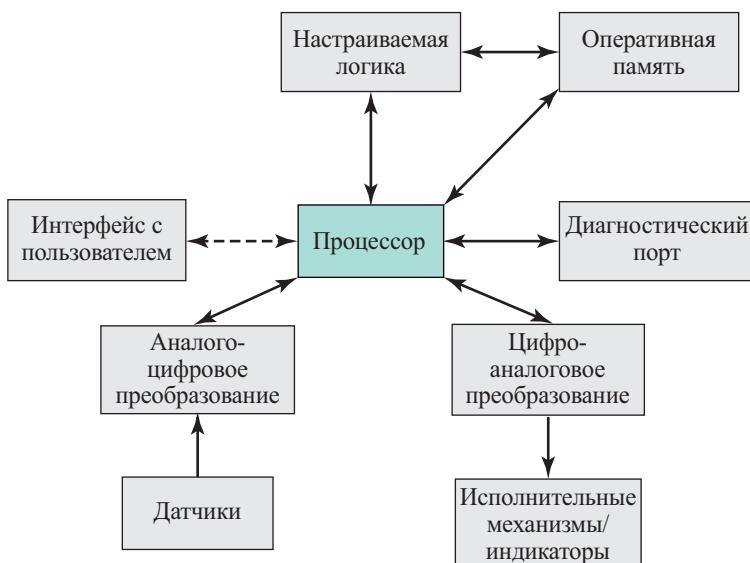
Термин *встроенная система* означает применение в конкретном продукте аппаратных и программных средств, выполняющих особую функцию или ряд функций, в отличие от универсального (например, переносного или настольного) компьютера. Встроенную систему можно также определить как любое устройство, в состав которого входит компьютерная микросхема, но не являющееся универсальной рабочей станцией, настольным или переносным компьютером. С одной стороны, ежегодно продаются сотни миллионов компьютеров, в том числе переносных и персональных, рабочих станций, серверов, больших и суперЭВМ. С другой стороны, ежегодно производятся десятки миллиардов микроконтроллеров, встраиваемых в более крупные устройства. Ныне большинство устройств, работающих на электрической энергии, содержат встроенную вычислительную систему. И, вероятнее всего, в ближайшем будущем практически все подобные устройства будут иметь в своем составе встроенные вычислительные системы.

Типы устройств со встроенными системами слишком многочисленны, чтобы пытаться их перечислить. К их числу относятся сотовые телефоны, цифровые фото- и видеокамеры, калькуляторы, микроволновые печи, домашние системы безопасности, стиральные и посудомоечные машины, системы освещения, термостаты, печатающие уст-

тряйства, различные автомобильные системы (например, управления коробкой передач, регулирования скорости движения, впрыскивания топлива, незаклинивающих тормозов и подвесок), теннисные ракетки, зубные щетки и многочисленные типы датчиков и исполнительных механизмов в автоматизированных системах.

Нередко встроенные системы тесно связаны со своим окружением. А это может привести к ограничениям реального времени, накладываемым в связи с необходимостью взаимодействовать с окружающей средой. И такие ограничения (например, накладываемые скоростью движения, точностью измерения или временными промежутками) требуют синхронизации программных операций. Если же под контролем одновременно должны выполняться многие действия, то это накладывает еще более сложные ограничения — реального времени.

На рис. 13.1 наглядно показана организация встроенных систем с использованием наиболее общей терминологии.



**Рис. 13.1.** Возможная организация встроенной системы

Типичный настольный компьютер отличается от переносного не только процессором и оперативной памятью, но и, как поясняется ниже, рядом других элементов.

- Для целей измерения, манипулирования или иного взаимодействия с внешним окружением в системе могут присутствовать самые разные интерфейсы. Встроенные системы нередко взаимодействуют (реагируют, манипулируют и сообщаются) с внешним миром через датчики и исполнительные механизмы, а следовательно, они, как правило, являются реагирующими системами. Реагирующая система находится в постоянном взаимодействии со своим окружением, выполняя действия в темпе, который задается данным окружением.
- Интерфейс пользователя может быть как простым (наподобие сигнальных лампочек), так и сложным (типа робототехнического зрения реального времени). Зачастую интерфейс с пользователем может просто отсутствовать.

- Диагностический порт может быть использован для диагностирования контролируемой системы (а не просто для работы компьютера).
- Для повышения производительности или надежности может использоваться интегральная схема специального назначения — программируемая (FPGA) или созданная для конкретного приложения (ASIC) или даже нецифровая аппаратура.
- Программное обеспечение нередко выполняет фиксированную функцию и предназначено для конкретного приложения.
- Первостепенное значение для встроенных систем имеет эффективность. Такие системы оптимизированы по потребляемой энергии, объему кода, времени выполнения, весу, габаритам и стоимости.

Имеется несколько заслуживающих внимания областей применения встроенных систем, сходных с универсальными вычислительными системами, как описывается ниже.

- Несмотря на фиксированную функциональность программного обеспечения, возможность обновления для устранения программных ошибок, повышения безопасности и расширения функциональных возможностей стала очень важным свойством встроенных систем, и не только в бытовой технике.
- Одной относительно недавней разработкой встроенных систем стали платформы, поддерживающие обширный ряд приложений. Характерными тому примерами служат смартфоны и аудиовизуальные устройства вроде “интеллектуальных” телевизоров.

## Прикладные и специализированные процессоры

**Прикладные процессоры** (application processors) определяются как способные выполнять сложные операционные системы, подобные Linux, Android и Chrome. Таким образом, прикладной процессор имеет универсальный характер. Соответствующая встроенная система разработана для поддержки многочисленных приложений и выполнения обширного ряда функций.

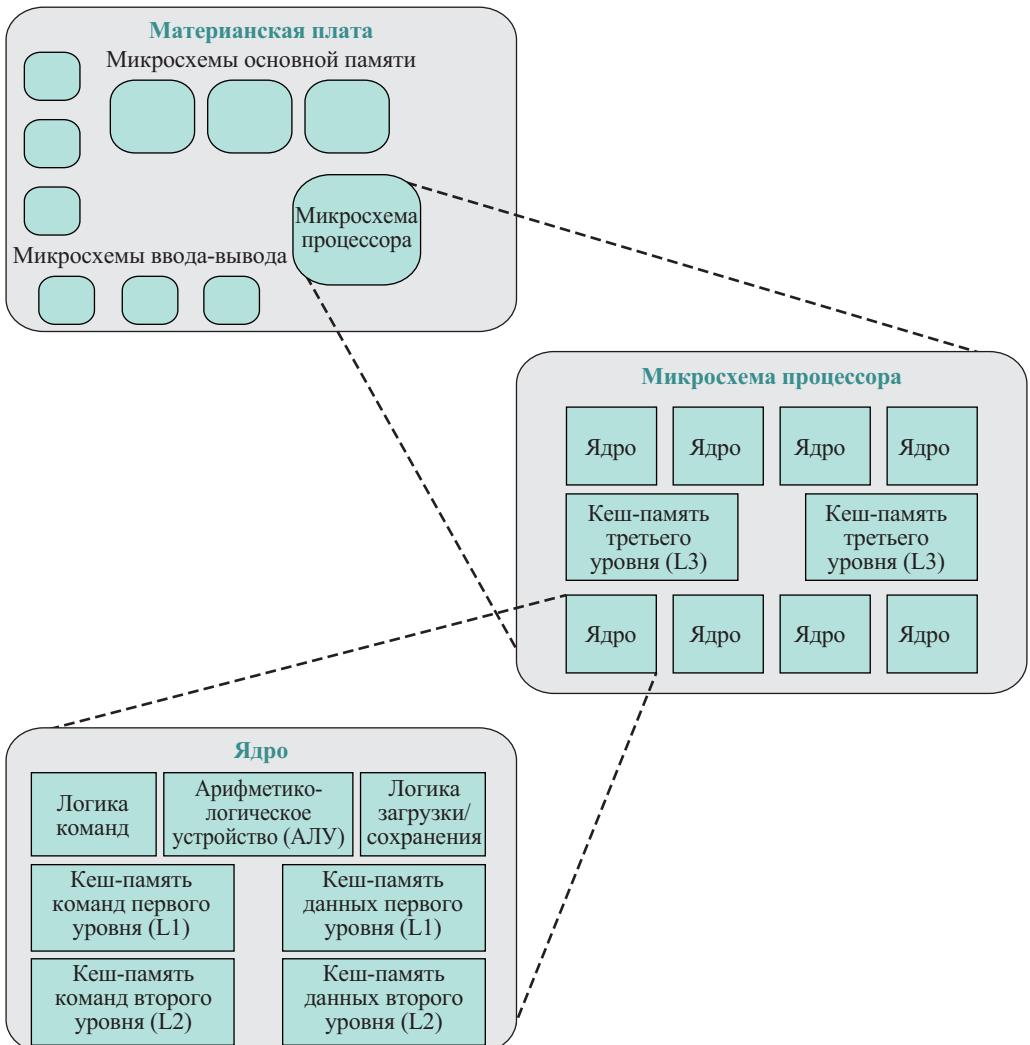
В большинстве встроенных систем применяется **специализированный процессор** (dedicated processor), который, как подразумевает его название, специально предназначен для решения одной конкретной задачи или небольшого числа подобных задач по требованию главного устройства. А поскольку такая встроенная система изначально предназначена для решения конкретной задачи или нескольких задач, то процессор и связанные с ним компоненты могут быть спроектированы специальным образом, позволяющим сократить его габариты и стоимость.

## Микропроцессоры

Микропроцессор — это процессор, элементы которого сведены в одну или несколько интегральных схем (ИС). Первоначально процессорные ИС включали в себя регистры, арифметико-логическое устройство (АЛУ) и некоторый блок управления или логику обработки команд. По мере увеличения плотности размещения транзисторов на кристалле появилась возможность усложнить архитектуру набора команд, а в конечном счете — добавить больше оперативной памяти и процессоров. Современные микропроцессорные ИС содержат несколько процессоров, иначе называемых *ядрами*, а также значительный

объем сверхоперативной или так называемой *кеш-памяти*. Но, как показано на рис. 13.2, микропроцессорная ИС включает в себя лишь некоторые элементы, образующие вычислительную систему.

Большинство компьютеров, включая встроенные компьютеры в смартфонах и планшетах, а также персональные и переносные компьютеры и рабочие станции, располагаются на материнской плате. Но, прежде чем рассматривать это размещение, необходимо определить ряд терминов. В частности, **печатная плата** (printed circuit board — PCB) представляет собой жесткую монтажную плоскость для установки и соединения ИС и прочих электронных компонентов. Такая плата состоит из нескольких слоев (как правило, от двух до десяти), соединяющих электронные компоненты с помощью медных дорожек, вытравленных на плате.



**Рис. 13.2.** Упрощенное схематическое представление основных элементов многоядерного компьютера

Основная печатная плата в компьютере называется **системной** или **материнской**, тогда как более мелкие печатные платы, вставляемые в разъемы на основной плате, называются платами **расширения**.

Самыми заметными элементами на системной плате являются микросхемы. **Микросхема** (чип) — это единое полупроводниковое (как правило, кремниевое) изделие с электронными и логическими схемами. Получающееся в конечном итоге изделие называется **интегральной схемой** (ИС).

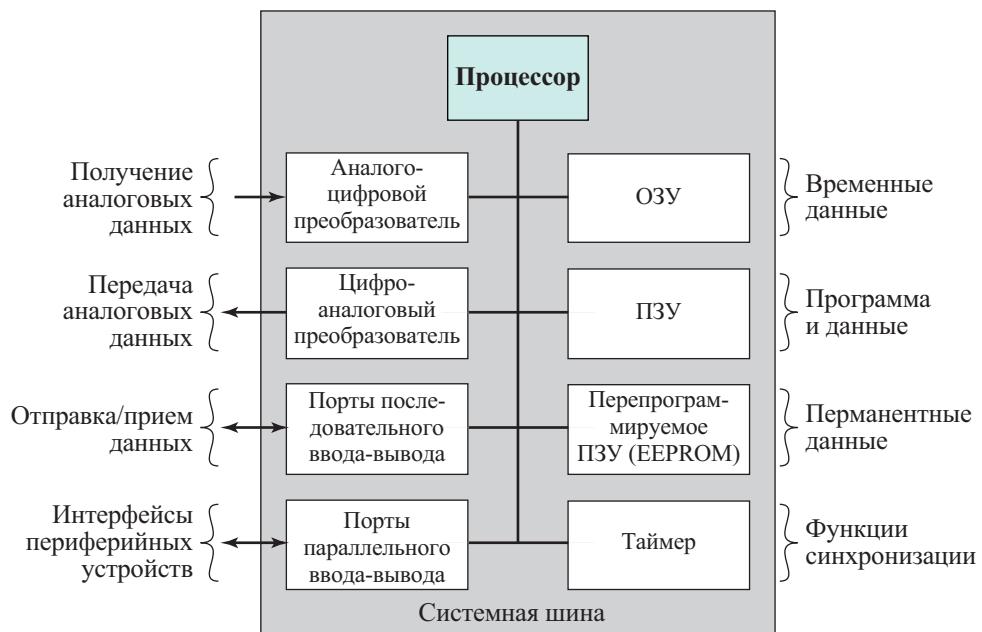
На системной плате находится разъем или панель для установки процессора, который обычно состоит из нескольких отдельных ядер, и поэтому он называется **многоядерным процессором**. Там же находятся разъемы для установки интегральных схем оперативной памяти, контроллеров ввода-вывода и других базовых компонентов компьютера. В соответствующие разъемы системных плат настольных компьютеров можно установить дополнительные компоненты и платы расширения. Таким образом, современная материнская плата соединяет лишь некоторые отдельные микросхемы, причем каждая из них содержит от нескольких тысяч до сотен миллионов транзисторов.

## Микроконтроллеры

**Микроконтроллер** — это однокристальная интегральная схема, состоящая из процессора, постоянной памяти для хранения программы (ПЗУ или флеш-памяти), кратковременной памяти (ОЗУ) для ввода-вывода данных, генератора тактовых сигналов и блока управления вводом-выводом. Иногда она называется “однокристальным компьютером”. Схемы микроконтроллеров используют доступное логическое пространство совершенно иначе. На рис. 13.3 представлена общая терминология для обозначения тех элементов, которые обычно находятся в интегральной схеме микроконтроллера. В частности, процессорная часть микроконтроллера занимает намного меньше места на кристалле, чем у других микропроцессоров, и обладает намного более высокой энергетической эффективностью.

Ежегодно миллиарды микроконтроллерных устройств встраиваются в миллиарды изделий: от игрушек до бытовых приборов и автомобилей. Например, в одном автомобиле может быть использовано 70 и более микроконтроллеров. Как правило, микроконтроллеры (особенно более мелкие и менее дорогие их разновидности) применяются в качестве специализированных процессоров, предназначенных для решения конкретных задач. Например, микроконтроллеры широко применяются в автоматизации производственных процессов. Обеспечивая простую реакцию на ввод, они способны управлять механизмами, включать и выключать вентиляторы, открывать и закрывать клапаны и т.д. Они являются неотъемлемой частью современной промышленной технологии и относятся к числу самых недорогих средств производства механизмов, способных выполнять чрезвычайно сложные функции.

Микроконтроллеры отличаются самыми разными габаритами и производительностью, а процессоры — архитектурами: от 4- до 32-разрядных. Микроконтроллеры обычно работают намного медленнее, чем микропроцессоры: первые, как правило, — в диапазоне тактовых частот порядка мегагерц, тогда как вторые — в диапазоне тактовых частот порядка гигагерц. Для микроконтроллеров также характерно, что они не обеспечивают взаимодействие с человеком. Микроконтроллер, встраиваемый в устройство, программируется для решения конкретной задачи, выполняемой данным устройством, а следовательно, вступает в действие, когда в этом возникает потребность.



**Рис. 13.3.** Типичные элементы в ИС микроконтроллеров

## Глубоко встроенные системы

Большая часть из общего числа встроенных систем называется **глубоко встроенным системами**. И хотя этот термин широко употребляется в технической и коммерческой литературе, искать ясное его определение в Интернете бесполезно (по крайней мере, автору этих строк найти такое определение не удалось). Как правило, можно сказать, что глубоко встроенная система содержит процессор, поведение которого трудно наблюдать как программисту, так и пользователю. В глубоко встроенной системе применяется микроконтроллер, а не микропроцессор, она программируется лишь один раз, когда программная логика работы конкретного устройства “зашивается” в ПЗУ, т.е. в доступную только для чтения память, а кроме того, она никак не взаимодействует с пользователем.

Глубоко встроенные системы являются специализированными устройствами узкого назначения, обнаруживающими нечто в своем окружении, выполняющими обработку сначала на элементарном уровне, а затем и получаемых результатов тем или иным образом. Глубоко встроенные системы нередко обладают возможностями беспроводной связи и присутствуют в таких сетевых конфигурациях, как сети датчиков, разворачиваемые на большой площади (например, на фабрике или сельскохозяйственном поле). В частности, Интернет вещей сильно зависит от глубоко встроенных систем. Как правило, в глубоко встроенных системах накладываются крайне жесткие ограничения на доступные ресурсы, включая оперативную память, габариты процессора и потребляемую мощность.

## 13.2. ХАРАКТЕРИСТИКИ ВСТРОЕННЫХ ОПЕРАЦИОННЫХ СИСТЕМ

Управлять простой встроенной системой, обладающей несложной функциональностью, можно из одной специализированной программы или ряда программ, не прибегая к услугам другого программного обеспечения. Как правило, более сложные встроенные системы включают операционную систему. И хотя для встроенной системы в принципе можно воспользоваться универсальной операционной системой (например, Linux), ограничения, накладываемые на объем оперативной памяти и потребляемую мощность, а также требования к режиму работы в реальном времени обычно предписывают пользоваться специализированной операционной системой, предназначенной для применения в окружении конкретной встроенной системы.

Ниже перечислен ряд особых характеристик и требований к проектированию встроенных операционных систем.

- **Работа в реальном времени.** Во многих встроенных системах правильность вычисления отчасти зависит от времени получения его результата. Зачастую ограничения реального времени диктуются требованиями к внешнему интерфейсу и устойчивости управления.
- **Реагирующее действие.** Встроенное программное обеспечение может выполняться в ответ на внешние события. Если эти события не наступают периодически или через прогнозируемые промежутки времени, встроенное программное обеспечение, возможно, должно принять во внимание наихудшие условия и задать приоритеты выполнения процедур.
- **Конфигурируемость.** Вследствие большого разнообразия встроенных систем к функциональным возможностям встроенной операционной системы предъявляются самые разные (как количественные, так и качественные) требования. Следовательно, встраиваемая операционная система, предназначенная для применения в разнообразных встроенных системах, должна допускать гибкое конфигурирование, чтобы предоставлять лишь те функциональные возможности, которые требуются для конкретного приложения или комплекта оборудования. Характерные тому примеры приведены в [164]: применение функций компоновки и загрузки для выбора только тех модулей операционной системы, которые требуется загрузить; условная компиляция; определение соответствующих подклассов, если применяется объектно-ориентированная структура. Но при проектировании встроенных систем с большим количеством производных специально настраиваемых операционных систем могут возникнуть трудности, связанные с верификацией. На подобную потенциальную трудность для eCos указывает Такада в [250].
- **Гибкость устройств ввода-вывода.** Практически не существует такого устройства ввода-вывода, которое требовалось бы поддерживать во всех версиях операционных систем, и имеется большое разнообразие таких устройств. Поэтому в [164] предлагается поддержка относительно медленных устройств ввода-вывода (например, жестких дисков и сетевых интерфейсов) с помощью специальных задач вместо того, чтобы интегрировать их драйверы в ядро операционной системы.

- **Рационализированные механизмы защиты.** Как правило, встроенные системы предназначены для выполнения ограниченных, вполне определенных функций. Непротестированные программы редко внедряются в программное обеспечение. Следовательно, после настройки и тестирования программное обеспечение можно рассматривать как надежное. Таким образом, кроме мер безопасности, у встроенных систем могут быть ограниченные механизмы защиты. Например, команды ввода-вывода не обязаны быть привилегированными, чтобы операционная система могла перехватывать управление; задачи могут самостоятельно выполнять свой ввод-вывод. Аналогично могут быть сокращены до минимума механизмы защиты памяти. В [164] приведен следующий тому пример: пусть `switch` соответствует отображаемому в память адресу ввода-вывода значения, которое требуется проверять в ходе операции ввода-вывода. Мы можем позволить программе ввода-вывода выполнить отдельную команду наподобие загрузки этого `switch` в регистр для определения текущего значения. Такой подход предпочтителен, чем использование вызова служб операционной системы, которые приводят к накладным расходам на сохранение и восстановление контекста задачи.
- **Непосредственное применение прерываний.** В универсальных операционных системах любому пользовательскому процессу обычно не разрешается пользоваться прерываниями непосредственно. В [164] перечислены причины, по которым прерывания могут непосредственно запускать и останавливать задачи (например, сохранив адрес запуска задачи в таблице адресов векторов прерываний) вместо того, чтобы проходить обычные для операционных систем процедуры обслуживания прерываний: 1) встроенные системы можно рассматривать как тщательно проверенные, с весьма редкими модификациями операционной системы или прикладного кода; 2) защита не является обязательной, как пояснялось в предыдущем абзаце; 3) требуется эффективное управление самыми разнообразными устройствами.

## Исходные и целевые среды

Главное отличие настольных и серверных дистрибутивов операционной системы Linux от встроенных заключается в том, что настольное и серверное программное обеспечение, как правило, компилируется или конфигурируется на той платформе, на которой оно будет выполняться, тогда как встроенные дистрибутивы Linux обычно компилируются или конфигурируются на одной платформе, называемой исходной (*host*), а предназначаются для выполнения на другой платформе, называемой целевой (*target*, рис. 13.4). Основными компонентами, которые сначала разрабатываются на исходной платформе, а затем переносятся в целевую систему, являются начальный загрузчик, ядро и корневая файловая система.

### Начальный загрузчик

Это небольшая программа, загружающая операционную систему в оперативную память после включения электропитания. Она отвечает за процесс первоначальной загрузки системы и загрузку ее ядра в основную память.



**Рис. 13.4.** Исходная и целевая платформы

Ниже приведена типичная последовательность начальной загрузки во встроенной системе.

- Процессор во встроенной системе выполняет код, хранящийся в ПЗУ, чтобы загрузить начальный загрузчик первой стадии из внутренней флеш-памяти, SD-карты памяти или порта последовательного ввода-вывода.
- Начальный загрузчик первой стадии инициализирует контроллер памяти и несколько периферийных устройств, а также загружает начальный загрузчик второй стадии в ОЗУ. Никакое взаимодействие с ним невозможно; обычно этот начальный загрузчик предоставляется поставщиком процессора в ПЗУ.
- Начальный загрузчик второй стадии загружает ядро и корневую файловую систему из флеш-памяти в основную память (ОЗУ). Ядро и корневая файловая система обычно хранятся во флеш-памяти в виде упакованных файлов, так что процесс начальной загрузки отчасти состоит в распаковке файлов в бинарные образы ядра и корневой файловой системы. Затем начальный загрузчик передает управление ядру. Как правило, на второй стадии данного процесса применяется начальный загрузчик с открытым кодом.

## Ядро

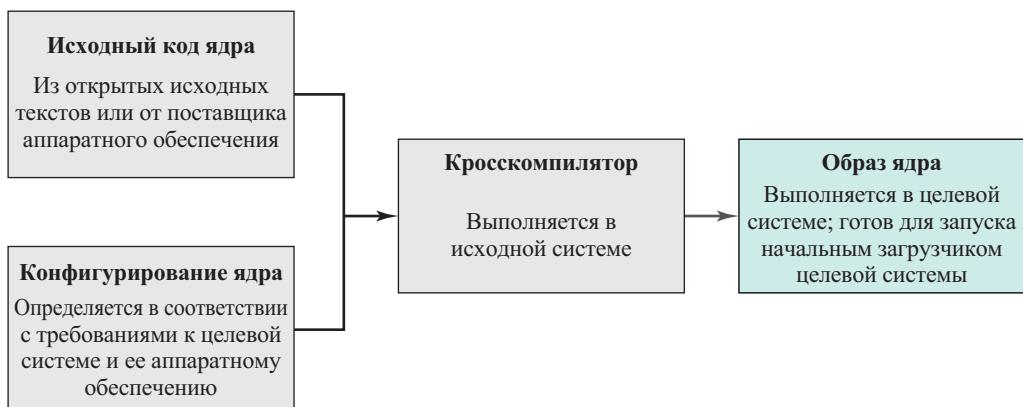
Полное ядро включает в себя целый ряд отдельных модулей, выполняющих среди прочего следующие функции.

- Управление памятью
- Управление процессами и потоками выполнения
- Межпроцессное взаимодействие; таймеры
- Ввод-вывод, поддержка сети, звука, хранения данных, графики и прочего с помощью драйверов
- Организация различных файловых систем

- Организация сети
- Управление электропитанием

Из программного обеспечения полного ядра данной операционной системы для встроенной системы опускается целый ряд дополнительных компонентов. Так, если аппаратные средства встроенной системы не поддерживают страничный обмен, то подсистему управления памятью можно удалить. Полное ядро будет включать в себя несколько файловых систем, драйверов устройств и так далее, из которых могут потребоваться лишь некоторые.

Как упоминалось ранее, главное отличие настольных и серверных дистрибутивов Linux от встроенных заключается в том, что настольное и серверное программное обеспечение, как правило, компилируется на той платформе, на которой оно будет выполняться, тогда как встраиваемые дистрибутивы Linux обычно компилируются на одной платформе, а предназначаются для выполнения на другой. Программное обеспечение, предназначенное для этой цели, называется *кросскомпилятором* (*межплатформенным компилятором*). Его применение наглядно показано на рис. 13.5.



**Рис. 13.5.** Компиляция ядра

### Корневая файловая система

Во встроенной или любой другой операционной системе имеется единая глобальная иерархия каталогов и файлов, предназначенная для представления всех файлов в системе. На вершине такой иерархии (в ее корне) находится корневая файловая система, в которой содержатся все файлы, требующиеся для нормальной работы системы. Корневая файловая система встроенной операционной системы подобна используемой на рабочей станции или сервере, с тем отличием, что она содержит лишь минимальный набор приложений, библиотек и файлов, требующихся для функционирования системы.

### Подходы к разработке

Имеются два общих подхода к разработке встроенной операционной системы. Первый подход состоит в том, чтобы взять существующую операционную систему и приспособить ее для применения во встроенном варианте. Другой подход состоит в разработке и реализации операционной системы, предназначеннной исключительно для встроенного применения.

## Адаптация существующей коммерческой операционной системы

Существующая коммерческая операционная система может быть использована для разработки встроенной системы; для этого необходимо дополнить ее возможностями функционирования в реальном времени, рационализировать ее работу и добавить необходимые функциональные средства. При таком подходе к разработке встроенной системы обычно применяется не только Linux, но и FreeBSD, Windows и другие универсальные операционные системы, хотя зачастую они работают медленнее и менее предсказуемо, чем специализированные встроенные операционные системы. Преимущество такого подхода заключается в том, что встроенная операционная система, производная от коммерческой универсальной операционной системы, основывается на ряде знакомых интерфейсов, что упрощает ее переносимость.

Недостаток же применения универсальной операционной системы для разработки встроенной заключается в том, что она не оптимизирована для применения в реальном времени и во встроенным варианте. Следовательно, для достижения необходимой производительности может потребоваться значительная ее модификация. В частности, типичная операционная система оптимизируется для среднего, а не наихудшего случая планирования заданий, обычно назначая ресурсы по требованию и пренебрегая большинством, если не всеми семантическими сведениями о приложении.

## Специально разработанная встроенная операционная система

Значительное количество операционных систем разработаны с самого начала для применения во встроенным варианте. Двумя характерными примерами такого подхода к разработке встроенных систем являются операционные системы eCos и TinyOS, обсуждаемые далее в этой главе.

Ниже перечислены типичные характеристики специализированной встроенной операционной системы.

- Наличие переключателя быстрых и упрощенных процессов или потоков исполнения.
- Стратегия планирования, реализуемая в виде модуля диспетчеризации в реальном времени как часть планировщика, а не отдельного компонента.
- Малые размеры.
- Быстрое реагирование на внешние прерывания. Типичным требованием к встроенной операционной системе является время отклика менее 10 мс.
- Минимизация промежутков времени, в течение которых запрещены прерывания.
- Предоставление для управления памятью разделов фиксированного или переменного размера, а также возможности блокировать код и данные в памяти.
- Предоставление специальных последовательных файлов, в которых можно накапливать данные с большой скоростью.

Чтобы каким-то образом удовлетворять временными ограничениям, ядро

- предоставляет ограниченное время для выполнения большинства примитивов;
- поддерживает часы реального времени;
- выдает специальные предупреждающие сигналы и блокировки по времени;
- поддерживает правила организации очередей (например, первоочередное обслуживание запросов с самым ранним сроком выполнения), а также примитивов для сжатия сообщения в начале очереди;
- предоставляет примитивы для задержки обработки на фиксированное время, а также для приостановки и возобновления процесса выполнения.

Перечисленные выше характеристики имеют немало общего с требованиями к функционированию встроенных операционных систем в реальном времени. Тем не менее в требованиях к сложным встроенным системам основной акцент может быть сделан на предсказуемое, а не на быстрое действие, а для этого придется принимать совсем другие проектные решения, особенно в области планирования заданий.

### 13.3. ВСТРОЕННАЯ СИСТЕМА LINUX

Термин *встроенная система Linux* просто означает версию Linux, выполняемую во встроенной системе. Как правило, во встроенной системе Linux применяется один из официальных выпусков ядра, несмотря на то что в некоторых системах используется модифицированное ядро, приспособленное к конкретной конфигурации оборудования или для поддержки определенного класса приложений. Прежде всего, ядро встроенной системы Linux отличается от ядра обычной операционной системы Linux, установленной на рабочей станции или сервере, конфигурацией сборки и средой разработки.

В этом разделе сначала выделяются некоторые из основных отличий встроенной Linux от ее версии, работающей на настольном компьютере или сервере, а затем описывается распространенное программное обеспечение под названием “μLinux”.

## ХАРАКТЕРИСТИКИ ВСТРОЕННОЙ СИСТЕМЫ Linux

### Размер ядра

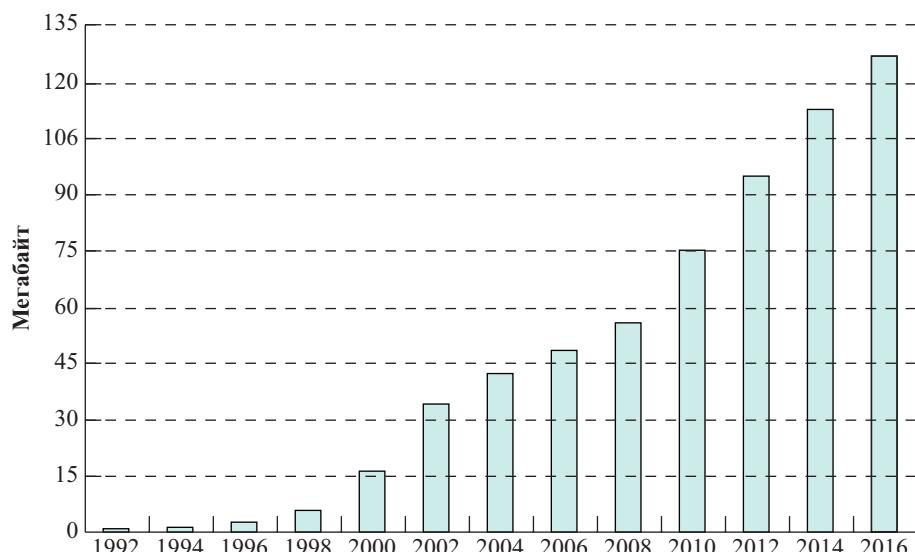
Настольные и серверные системы Linux должны поддерживать большое число устройств, поскольку в Linux применяются самые разные конфигурации. Такие системы должны поддерживать и целый ряд протоколов передачи и обмена данными, а следовательно, их можно применять для самых разных целей. Встроенным устройствам, как правило, требуется поддержка определенного ряда конкретных устройств, периферийных устройств и протоколов в зависимости от установленного оборудования и предназначения. К счастью, ядро Linux в высшей степени поддается конфигурированию в плане архитектуры, для которой оно скомпилировано, а также поддерживаемых в нем процессоров и устройств.

Дистрибутив встроенной системы Linux является версией Linux, специально приспособленной под ограничения, накладываемые на размер и оборудование встроенных устройств. В его состав входят программные пакеты, поддерживающие различные службы и приложения, действующие на подобных устройствах. Таким образом, ядро встроенной системы Linux оказывается намного меньше, чем ядро обычной системы Linux.

## Объем памяти

Размер встроенной системы Linux классифицируется в [76] по объему доступного ПЗУ или ОЗУ в трех обширных категориях малых, средних и крупных систем. В частности, малые системы характеризуются наличием маломощного процессора, ПЗУ объемом минимум 2 Мбайт и ОЗУ объемом 4 Мбайт. Средние по размеру системы отличаются наличием среднемощного процессора, ПЗУ объемом около 32 Мбайт и ОЗУ объемом 64 Мбайт. А крупные системы характеризуются наличием мощного процессора или даже нескольких процессоров и большими объемами как оперативной, так и постоянной памяти.

В системе без постоянной памяти все ядро Linux должно вмещаться в ОЗУ и ПЗУ, чего нельзя сказать о полномасштабной современной системе Linux. В качестве иллюстрации данного положения на рис. 13.6 показано увеличение размера ядра полномасштабной системы Linux со временем. Безусловно, любую систему Linux можно сконфигурировать лишь с некоторыми компонентами из полного выпуска. Тем не менее на диаграмме, приведенной на рис. 13.6, наглядно показано, что, в особенности для мелких и средних по размеру встроенных систем, могут быть опущены значительные объемы ядра.



**Рис. 13.6.** Размер ядра Linux, показанный в сжатом формате файлов GZIP

## Прочие характеристики

К другим характеристикам встроенных систем Linux относятся следующие.

- **Временные ограничения.** Жесткие временные ограничения требуют от системы реакции на внешние воздействия в течение указанного промежутка времени. А мягкие временные ограничения пригодны для тех систем, в которых медленная реакция системы является некритичной.

- Подключаемость к сети.** Означает способность системы работать в сети. Буквально все современные встроенные устройства обладают такой способностью, подключаясь, как правило, к беспроводной сети.
- Степень взаимодействия с пользователем.** Одни устройства сосредоточены на взаимодействии с пользователем, тогда как другие, в том числе управляющие промышленными процессами, могут предоставлять для взаимодействия с пользователем очень простой интерфейс (например, через светодиоды и кнопки). Имеются и такие устройства, у которых взаимодействие с конечным пользователем отсутствует (например, датчики, действующие в Интернете вещей, собирающие данные и передающие их в облако).

В табл. 13.1, взятой из [76], приведены характеристики некоторых коммерчески доступных встроенных систем, в которых применяется ядро Linux.

**Таблица 13.1. ХАРАКТЕРИСТИКИ ПРИМЕРОВ ВСТРОЕННЫХ СИСТЕМ LINUX**

Описание	Тип	Размер	Времен- ные ограни- чения	Подклю- чаемость к сети	Степень взаимодей- ствия с поль- зователем
Устройства управления акселератором	Управление промышленными процессами	Средний	Жесткие	Да	Низкая
Автоматизированная обучающая система	Авиакосмическая отрасль	Крупный	Жесткие	Нет	Высокая
Устройство типа Bluetooth для доступа к местным данным	Организация сети	Мелкий	Мягкие	Да	Очень низкая
Преобразователь протоколов для управления системой сбора данных	Управление промышленными процессами	Средний	Жесткие	Нет	Очень низкая
Карманный персональный компьютер	Бытовая электроника	Средний	Мягкие	Да	Очень высокая
Устройство управления двигателем, применяемое в системе управления космическим аппаратом	Авиакосмическая отрасль	Крупный	Жесткие	Да	Высокая

## Файловые системы встроенного Linux

В некоторых приложениях могут создаваться относительно небольшие файловые системы, которые предназначены для применения только во время работы самого приложения и могут храниться в основной памяти. Но в целом файловая система должна храниться в долговременной памяти (например, во флеш-памяти или на традиционных

дисковых запоминающих устройствах). Впрочем, для большинства встроенных систем внутренний или внешний диск не подходит, и поэтому в качестве долговременной памяти обычно применяется флеш-память.

Что же касается остальных свойств встроенной системы Linux, то ее файловая система должна быть как можно более компактной. Для применения во встроенных системах был разработан целый ряд компактных файловых систем. В качестве примера ниже перечислены некоторые из наиболее употребительных систем данной категории.

- **cramfs.** Упакованная файловая система в ОЗУ (Compressed RAM File System), простая и доступная только для чтения. Она служит для того, чтобы минимизировать размер, максимально эффективно используя емкость исходного запоминающего устройства. В файловых системах cramfs отдельные файлы упаковываются в блоки, совпадающие по размеру со страницами в Linux (как правило, 4096 байт или 4 Мбайт в зависимости от версии ядра и конфигурации), чтобы обеспечить эффективный произвольный доступ к содержимому файлов.
- **squashfs.** Как и cramfs, squashfs является упакованной доступной только для чтения файловой системой, предназначенней для применения в средах с малой или ограниченной емкостью запоминающего устройства (например, во встроенных системах Linux).
- **jffs2.** Вторая версия файловой системы для флеш-памяти с журналированием (Journaling Flash File System). Она предназначена для применения в устройствах флеш-памяти типа NOR и NAND с особым акцентом на такие вопросы эксплуатации флеш-памяти, как выравнивание износа.
- **ubifs.** Файловая система с несортированными образами блоков (Unsorted Block Image File System), которая обычно обеспечивает лучшую производительность, чем файловая система jffs2 в крупных устройствах флеш-памяти. Для дополнительного повышения производительности в ней также поддерживается кеширование записи.
- **yaffs2.** Еще одна версия файловой системы (Yet another Flash File System) для флеш-памяти, обеспечивающая быстрое и надежное хранение файлов во флеш-памяти. Для хранения данных состояния файловой системы yaffs2 требуется меньше места в ОЗУ, чем в таких файловых системах, как jffs2. И, как правило, она обеспечивает лучшую производительность, если запись данных в файловой системе выполняется слишком часто.

## Преимущества встроенных систем Linux

Встроенные версии Linux стали появляться еще в 1999 году. В целом ряде компаний были разработаны свои версии, предназначенные для конкретных платформ. Ниже перечислены преимущества применения Linux в качестве основания для разработки встроенной операционной системы.

- **Независимость поставщиков.** Поставщик платформы не зависит от поставщика конкретной встроенной системы, чтобы предоставить необходимые функциональные средства и уложиться в крайние сроки развертывания.

- **Разнообразная аппаратная поддержка.** В системе Linux поддерживается обширный ряд архитектур процессоров и периферийных устройств, и благодаря этому она оказывается вполне пригодной для разработки буквально каждой встроенной системы.
- **Малые затраты.** Применение Linux позволяет свести к минимуму затраты на разработку и обучение.
- **Открытость исходного кода.** Применение Linux дает все преимущества программного обеспечения с открытым исходным кодом.

## µLinux

µLinux (т.е. микроконтроллерная Linux) является весьма распространенной разновидностью ядра Linux с открытым исходным кодом, предназначенного для микроконтроллеров и прочих очень малых встроенных систем. В силу модульного характера Linux совсем не трудно сократить операционную среду, исключив из нее служебные программы, инструментальные средства и прочие системные службы, которые не нужны во встроенной среде. В этом, собственно, состоит основной принцип проектирования µLinux.

Чтобы дать ясное представление о размере загружаемого образа µLinux (ядра и корневой файловой системы), обратимся к опыту компании EmCraft Systems, разрабатывающей системы на уровне печатных плат с помощью микроконтроллеров Cortex-M и микропроцессоров Cortex-A [74]. Это едва ли не самые малые встроенные системы, в которых применяется µLinux. Так, минимальная конфигурация µLinux может занимать не больше 0,5 Мбайт, хотя поставщик посчитал вполне практическим размер загружаемого образа, включая Ethernet, TCP/IP, приемлемый набор инструментальных средств из пользовательского пространства и сконфигурированных приложений, в пределах от 1,5 до 2 Мбайт. А объем памяти, требующийся для µLinux во время выполнения, должен находиться в пределах от 8 до 32 Мбайт. Эти числовые показатели значительно меньше, чем у типичной системы Linux.

### *Сравнение с полномасштабной системой Linux*

Главные отличия µLinux от Linux для крупных систем (подробнее об этом см. в [165]) заключаются в следующем.

- Linux является многопользовательской операционной системой, основанной на UNIX, а µLinux — версией Linux, предназначеннной для разработки встроенных систем (как правило, без взаимодействия с пользователем).
- В отличие от Linux, в µLinux не поддерживается управление памятью. Следовательно, виртуальные адресные пространства в µLinux не требуются, а приложения должны быть привязаны к абсолютным адресам.
- В ядре Linux поддерживается отдельное пространство виртуальных адресов для каждого процесса, а в µLinux — единое адресное пространство, общее для всех процессов.
- Адресное пространство в Linux восстанавливается при переключении контекста, чего нельзя сказать о µLinux.

- В отличие от Linux, в µClinix не обеспечивается системный вызов `fork()`, а вместо этого применяется функция `vfork()`. По существу, системный вызов с помощью функции `fork()` приводит к созданию дубликата, во многом похожего на вызывающий процесс (однако при этом копируется не все (например ресурсы в некоторых реализациях ограничены), хотя основной замысел состоит в том, чтобы создать как можно более полную копию). Новый (порожденный) процесс получает другой идентификатор процесса и имеет тот же идентификатор родительского процесса. Главное отличие функции `vfork()` от функции `fork()` заключается в следующем: когда новый процесс создается с помощью функции `vfork()`, родительский процесс временно приостанавливается, а порожденный процесс может позаимствовать адресное пространство своего родителя. И так продолжается до тех пор, пока завершится порожденный процесс или же им будет вызвана функция `execve()`, после чего родительский процесс продолжится.
- µClinix модифицирует драйверы устройств для использования локальной системной шины вместо ISA или PCI.

Самое значительное отличие полномасштабной системы Linux от µClinix относится к области управления памятью. Отсутствие поддержки управления памятью в µClinix может иметь целый ряд последствий, в том числе следующие.

- Основная память, выделяемая для процесса, в общем случае должна быть непрерывной. Если же выполняется свопинг целого ряда процессов, это может привести к фрагментации памяти (см. рис. 7.4). Тем не менее во встроенных системах, как правило, имеется фиксированное множество процессов, загружаемых во время начальной загрузки и работающих до следующего сброса. Следовательно, такая функциональная возможность обычно не требуется.
- µClinix не может расширить память для работающих процессов, поскольку с ней могут быть смежными другие процессы. Следовательно, вызовы функций `brk()` и `sbrk()`, динамически изменяющих объем используемого пространства, выделяемого для сегмента данных из вызывающего процесса, недоступны. Но в µClinix все же предоставлена реализация функции `malloc()`, предназначенная для выделения блока памяти из глобального пула памяти.
- В µClinix отсутствует стек динамических приложений. Это может привести к переполнению стека, а значит, и к повреждению памяти. Во избежание этого на стадии разработки и конфигурирования приложений необходимо принять соответствующие меры.
- В µClinix не обеспечивается защита памяти, что представляет риск повреждения одним приложением части другого приложения или даже ядра. Впрочем, этот недостаток все же исправлен в некоторых реализациях. Например, архитектура Cortex-M3/M4 предоставляет механизм защиты памяти под названием “MPU” (Memory Protection Unit — блок защиты памяти). Используя механизм MPU, компания Emcraft Systems внедрила в ядро дополнительное функциональное средство, реализующее взаимную защиту как самих процессов, так и ядра и процессов, сопоставимую с механизмами защиты памяти, реализованными в Linux с помощью блока управления памятью (MMU) [130].

## *μClibc*

Это системная библиотека на C, первоначально разработанная для поддержки μLinux и обычно применяемая вместе с μLinux, хотя библиотекой μClibc можно пользоваться и вместе с другими ядрами Linux. Основное назначение μClibc — представить системную библиотеку, написанную на языке C и пригодную для разработки встроенных систем Linux. Она намного меньше библиотеки GNU C (glibc), широко применяемой в системах Linux, хотя практически все приложения, поддерживаемые в библиотеке glibc, идеально взаимодействуют и с μClibc. Перенос приложений из библиотеки glibc в библиотеку μClibc, как правило, предусматривает лишь перекомпиляцию исходного кода. В библиотеке μClibc поддерживаются также совместно используемые библиотеки и поточная обработка.

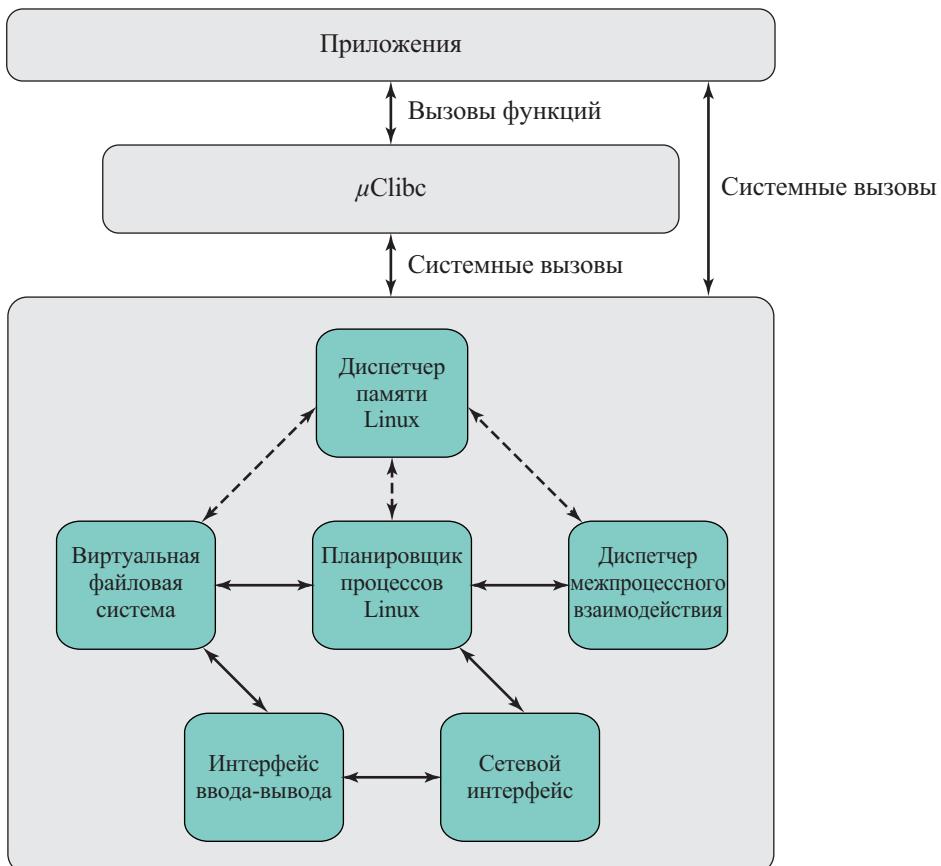
В табл. 13.2, составленной по материалам, взятым из [4], сравниваются размеры функций в обеих упомянутых выше библиотеках. Как видите, экономия занимаемого места вполне очевидна. Такая экономия достигается благодаря запрету некоторых функциональных средств по умолчанию и радикальной переделке исходного кода с целью исключить его избыточность. А на рис. 13.7 показана высокогородневая программная архитектура встроенной системы, построенной с помощью μLinux и μClibc.

**ТАБЛИЦА 13.2. РАЗМЕРЫ НЕКОТОРЫХ ФУНКЦИЙ ИЗ БИБЛИОТЕК μCLIBC И GLIBC**

Имя функции из glibc	Размер функции glibc, Кбайт	Имя функции из μClibc	Размер функции μClibc, Кбайт
libc-2.3.2.so	1200	libuClibc-0.9.2.7.so	284
ld-2.3.2.so	92	libcrypt-0.9.2.7.so	20
libcrypt-2.3.2.so	20	libdl-0.9.2.7.so	12
libdl-2.3.2.so	12	libm-0.9.2.7.so	8
libm-2.3.2.so	136	libnsl-0.9.2.7.so	56
libnsl-2.3.2.so	76	libpthread-0.9.2.7.so	4
libpthread-2.3.2.so	84	libresolv-0.9.2.7.so	84
libresolv-2.3.2.so	68	libutil-0.9.2.7.so	4
libutil-2.3.2.so	8	libcrypt-0.9.2.7.so	8

## Android

Как подробно обсуждалось ранее в этой книге, Android является встроенной операционной системой, основанной на ядре Linux. Следовательно, Android можно с полным основанием рассматривать как пример встроенной системы Linux. Тем не менее многие разработчики встроенных систем Linux не считают Android примером системы данной категории [47]. С их точки зрения, у классического встроенного устройства имеется фиксированный набор функций, прошитых изготовителем. Android же в большей степени относится к категории платформенных операционных систем, поддерживающих разнообразные приложения, различающиеся на разных платформах. Кроме того, Android является вертикально интегрированной системой, включая модификации ядра Linux, специально сделанные для Android.



**Рис. 13.7.** Программная архитектура встроенной системы, построенной на основе *μClinux* и *μClibc*

Основное внимание в Android уделяется вертикальной интеграции ядра Linux и компонентов из пользовательского пространства Android. В конечном счете это вопрос семантики, а “официальное” определение Android как встроенной системы Linux, на которое можно было бы положиться, отсутствует.

## 13.4. TINYOS

В системе TinyOS представлен более рациональный подход к разработке встроенных операционных систем, чем основанный на коммерческих универсальных операционных системах (например, встроенной версии Linux). Следовательно, TinyOS и аналогичные ей системы в большей степени пригодны для разработки небольших встроенных систем с жесткими требованиями к памяти, времени обработки и реагирования, потребляемой мощности и т.д. Процесс рационализации заходит в TinyOS довольно далеко, а в результате получается самая маленькая операционная система для встроенных систем. Базовой операционной системе требуется всего 400 байт памяти для кода и данных, вместе взятых.

Система TinyOS представляет собой существенный отход от других встроенных операционных систем. Разительное отличие заключается, в частности, в том, что TinyOS не является операционной системой реального времени. Причиной тому служит предполагаемая рабочая нагрузка, особенно в контексте сети беспроводных датчиков, как поясняется в следующем подразделе. В силу жестких ограничений на потребляемую мощность эти устройства большую часть времени выключены. Приложения обычно просты, а их соперничество за вычислительные ресурсы процессора не вызывает особых осложнений.

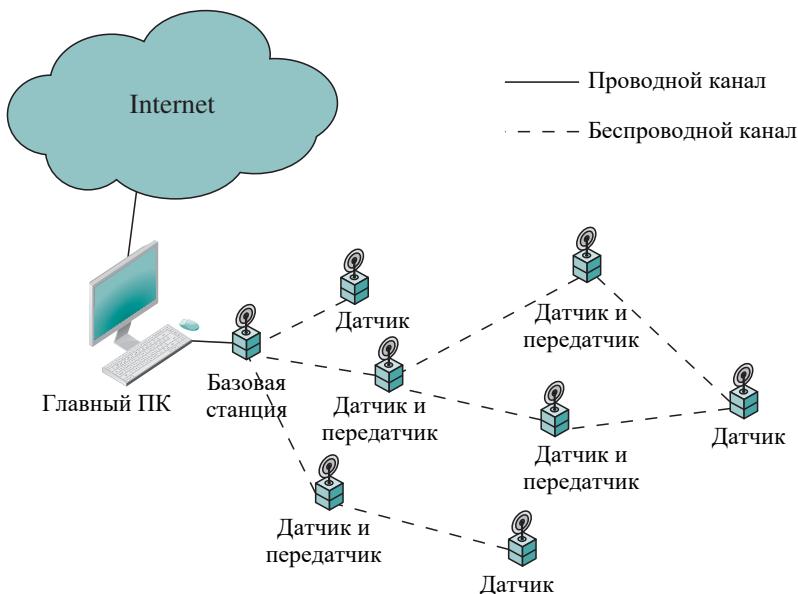
Кроме того, у системы TinyOS нет ядра, поскольку в ней не предусмотрена защита памяти, и она является операционной системой, основанной на компонентах. В этой операционной системе отсутствуют процессы и система распределения памяти, хотя в некоторых редко применяемых компонентах она все же внедрена. Обработка прерываний и исключений выполняется в зависимости от конкретных периферийных устройств и совершенно не блокируется, а следовательно, в этой системе имеется совсем немного явных примитивов синхронизации.

Система TinyOS нашла широкое применение в качестве подхода к реализации программного обеспечения сети беспроводных датчиков. В настоящее время более 500 организаций разрабатывают и принимают участие в разработке открытого стандарта на TinyOS.

## Беспроводные сети датчиков

Система TinyOS была разработана в основном для применения в сетях небольших беспроводных датчиков. Целый ряд возникших тенденций побудил к разработке весьма компактных маломощных датчиков. Хорошо известный закон Мура продолжает определять тенденции к сокращению габаритов электронных элементов оперативной памяти и логической обработки, а уменьшение габаритов приводит к сокращению потребляемой мощности. Тенденции к снижению потребляемой мощности и габаритов с очевидностью проявляются и в аппаратуре беспроводной связи, микроэлектромеханических датчиках (MEMS) и преобразователях. В итоге можно разработать датчик, вся логика работы которого умещается в одном кубическом миллиметре. Прикладное и системное программное обеспечение таких устройств должно быть достаточно компактным, чтобы функции опознавания, измерения, передачи данных и вычисления можно было внедрить в завершенную, но крошечную архитектуру.

Недорогие, малогабаритные, маломощные беспроводные датчики могут применяться во многих приложениях [209]. Типичная конфигурация сети подобных датчиков приведена на рис. 13.8. Базовая станция подключает сеть датчиков к главному ПК и передает данные от датчиков через сеть на главный ПК и/или через корпоративную сеть или Интернет на сервер, где полученные данные могут быть проанализированы. Отдельные датчики собирают данные и передают их на базовую станцию как непосредственно, так и через другие датчики, действующие в качестве передатчиков данных. Для передачи данных по сети датчиков на базовую станцию требуются функциональные возможности маршрутизации. Как отмечается в [32], во многих приложениях пользователю требуется возможность быстро развернуть большое количество недорогих устройств, не конфигурируя их и манипулируя ими. Это означает, что из таких устройств можно собрать специальную самоорганизующуюся сеть. Подвижность отдельных датчиков и наличие радиопомех означает, что их сеть должна быть способна перестраивать свою конфигурацию в считанные секунды.



**Рис. 13.8.** Типичная топология сети беспроводных датчиков

## Цели TinyOS

Принимая во внимание крошечный, распределенный характер применяемых датчиков, группа исследователей из Калифорнийского университета в Беркли [106] установила для TinyOS следующие цели.

- **Допущение высокой степени параллелизма.** В типичном приложении для сети беспроводных датчиков устройства интенсивно работают в параллельном режиме. Это означает, что по такой сети могут одновременно проходить несколько различных потоков данных. В то время как данные от датчиков вводятся устойчивым потоком, обработанные результаты должны передаваться столь же устойчивым потоком. Необходимо также обеспечить дистанционное управление от удаленных датчиков или базовых станций.
- **Работа с ограниченными ресурсами.** Целевая платформа для TinyOS будет обладать ограниченными ресурсами памяти и вычислений, а пытаться — от аккумуляторных или солнечных батарей. На одной платформе может быть выделено лишь несколько килобайтов для хранения программ в ПЗУ и сотни байтов для хранения данных в ОЗУ. Поэтому программное обеспечение должно эффективно использовать доступные ресурсы процессора и памяти, допуская в то же время маломощный режим передачи данных.
- **Адаптация к эволюции оборудования.** Большая часть оборудования постоянно развивается, и поэтому приложения и большинство системных служб должны благополучно переноситься из одного поколения оборудования в другое. Таким образом, обновление оборудования должно приводить к минимальным изменениям в программном обеспечении или вообще не требовать их, если его функциональные возможности остаются теми же самыми.

- **Поддержка обширного ряда приложений.** Приложения предъявляют обширный ряд требований к сроку действия, передаче данных, опознаванию, измерению и т.д. В данном случае требуется модульная, универсальная, встроенная операционная система, поэтому стандартизованный подход приводит к экономии масштабов при разработке приложений и поддерживающего программного обеспечения.
- **Поддержка разнообразного ряда платформ.** Как и в предыдущем пункте, здесь требуется универсальная встроенная операционная система.
- **Надежность.** Однажды разработанная сеть датчиков должна работать без обслуживания и сопровождения месяцами или даже годами. В идеальном случае должно быть резервирование как в одной системе, так и во всей сети датчиков. Но для обеспечения этих двух видов резервирования потребуются дополнительные ресурсы. К числу тех характеристик, которые способны повысить надежность, относится применение программных компонентов с высокой степенью модульности и стандартизации.

Здесь стоит уточнить требование к параллелизму. В типичном приложении могут присутствовать десятки, сотни или даже тысячи датчиков, связанных в сеть. Как правило, принимаемые от них данные требуется хотя бы незначительно буферизировать из-за возможных задержек. Так, если сбор данных выполняется через каждые 5 минут и требуется буферизировать четыре выборки данных перед их отправкой, среднее время задержки составит 10 минут. Таким образом, данные, как правило, фиксируются, обрабатываются и направляются непрерывным потоком по сети. Кроме того, если собирается значительный объем данных от датчиков, то ограниченное пространство доступной памяти накладывает свой предел на количество выборок, которые могут быть буферизованы. Тем не менее в некоторых приложениях каждый из потоков данных может включать в себя большое количество событий, возникающих на низком уровне и перемежающихся с обработкой данных на более высоком уровне. Часть обработки данных на высоком уровне распространяется и на многие события, наступающие в реальном времени. А поскольку датчики находятся в сети, где допускается маломощная передача данных, то они ограничены небольшой физической удаленностью. Таким образом, данные от удаленных датчиков должны передаваться одной или более базовым станциям через промежуточные узлы сети.

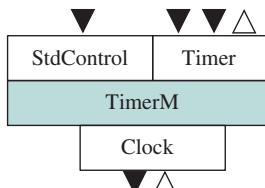
## Компоненты TinyOS

Встроенная программная система, построенная с помощью TinyOS, состоит из ряда небольших модулей, называемых *компонентами*. Каждый из них выполняет простую задачу или ряд задач и сопрягается как с другими компонентами, так и с оборудованием, ограниченным и вполне определенным числом способов. Единственным другим программным модулем, который при этом требуется, является рассматриваемый далее планировщик. На самом деле никакой операционной системы здесь нет, поскольку нет ядра. Но на это обстоятельство можно взглянуть иначе.

Прикладной областью интересов в данном случае является сеть беспроводных датчиков (*wireless sensor network* — WSN). Чтобы удовлетворить требованиям программного обеспечения для этой прикладной области, придется разработать жесткую, упрощенную программную архитектуру, состоящую из отдельных компонентов. Сообщество разработчиков TinyOS реализовало целый ряд компонентов с открытым исходным кодом и

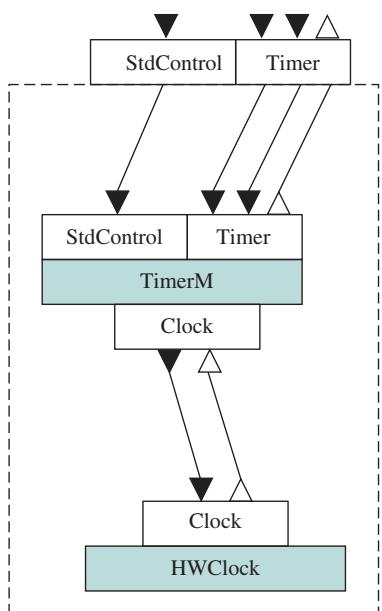
основными функциями, требующимися для приложения к сети беспроводных датчиков. К примерам таких стандартизованных компонентов относятся организация однопролетной сети, специальная маршрутизация, управление электропитанием и энергонезависимым запоминающим устройством. А для конкретных конфигураций и приложений пользователи могут построить дополнительные компоненты специального назначения, скомпоновать и загрузить все требующиеся компоненты. Таким образом, TinyOS состоит из комплекта стандартизованных компонентов. Некоторые, хотя и не все, из них применяются вместе с прикладными компонентами, специально разработанными пользователями для любой конкретной реализации. Операционная система для такой реализации состоит просто из ряда стандартизованных компонентов, выбираемых из комплекта TinyOS.

Все компоненты в конфигурации TinyOS имеют одинаковую структуру, пример которой схематически представлен на рис. 13.9, а, где затененные блоки обозначают компоненты, интерпретируемые как объекты, которые могут быть доступны только через определенные интерфейсы, показанные как прозрачные блоки. Компонент может быть аппаратным или программным.



(a) Компонент TimerM

```
module TimerM {
    provides {
        interface StdControl;
        interface Timer;
    }
    uses interface Clock as Clk;
} ...
```



(b) Конфигурация TimerC

```
configuration TimerC {
    provides {
        interface StdControl;
        interface Timer;
    }
}

implementation {
    components TimerM, HWClock;
    StdControl = TimerM.StdControl;
    Timer = TimerM.Timer;
    TimerM.Clk -> HWClock.Clock;
}
```

Рис. 13.9. Пример компонента и конфигурации

Программные компоненты реализуются на nesC — расширении языка C, которое обладает двумя отличительными свойствами: во-первых, моделью программирования, в которой компоненты взаимодействуют через интерфейсы, а во-вторых, моделью параллельной обработки на основе событий с обработчиками выполняемых до конца задач и прерываний, как поясняется далее.

Рассматриваемая здесь архитектура состоит из компонентов, расположенных на отдельных уровнях. Каждый компонент может быть связан лишь с двумя другими компонентами: с одним, расположенным по иерархии ниже, и с другим, расположенным выше. Такой компонент выдает команды своему компоненту более низкого уровня и получает от него сигналы о наступлении событий. Аналогично компонент принимает команды от своего компонента более высокого уровня и посыпает ему сигналы о наступлении событий. В нижней части этой иерархии находятся аппаратные компоненты, а на ее вершине — прикладные компоненты, которые могут и не входить в комплект стандартизованных компонентов TinyOS, но все же должны соответствовать их структуре.

Программный компонент реализует одно или более заданий. Каждое **задание** (task) в компоненте аналогично потоку выполнения в обычновенной операционной системе, хотя и с некоторыми ограничениями. А в самом компоненте задания атомарны: как только задание запущено, оно выполняется до своего завершения. Оно не может быть вытеснено другим заданием в том же самом компоненте, а следовательно, квантование времени отсутствует. Тем не менее задание может быть вытеснено событием, хотя и не может быть заблокировано или находиться в состоянии пережидания занятости. Такие ограничения заметно упрощают планирование заданий в компоненте и управление ими. Для каждого выполняющегося в настоящий момент задания имеется единственный стек. В заданиях можно выполнять вычисления, вызывать компоненты более низкого уровня (команды), сигнализировать о наступлении событий более высокого уровня и планировать другие задания.

**Команды** (command) представляют собой неблокируемые запросы. Это означает, что задание, выдавшее команду, не блокируется и не пережидает занятости до получения ответа от компонента более низкого уровня. Как правило, команда является запросом компонента более низкого уровня на выполнение некоторой службы, такой как начать чтение датчиков. Воздействие на компонент, принимающий команду, зависит от заданной команды и задания, которое требуется выполнить, чтобы удовлетворить эту команду. В общем случае, когда принимается команда, соответствующее задание планируется для последующего выполнения, поскольку команда не может вытеснить задание, выполняющееся в настоящий момент. Команда сразу же возвращает управление вызывающему компоненту, а некоторое время спустя наступит событие, сигнализирующее вызвавшему компоненту о завершении выполнения задания. Таким образом, команда не приводит ни к вытеснению в вызываемом компоненте, ни к блокировке в вызывающем компоненте.

**События** (event) в TinyOS могут быть прямо или косвенно привязаны к аппаратным событиям. Программные компоненты самого низкого уровня сопрягаются непосредственно с аппаратными прерываниями, которые могут быть внешними прерываниями, а также событиями от таймера или счетчика. Обработчик событий в компоненте самого низкого уровня может обработать само прерывание или же распространить сообщение о событии вверх по иерархии компонентов. Команда может отправить задание, которое впоследствии сигнализирует о наступлении события. В этом случае какая-либо привязка к аппаратному событию отсутствует.

Задание можно рассматривать как выполняющееся в три стадии. Сначала вызывающий код отправляет команду модулю. Затем модуль выполняет запрашиваемое задание. И наконец, модуль через механизм события уведомляет вызывающий код о том, что задача завершена.

Компонент TimerM, схематически представленный на рис. 13.9, *a*, является частью службы таймера в TinyOS. Этот компонент предоставляет интерфейсы StdControl и Timer и использует интерфейс Clock. Поставщики реализуют команды (т.е. логику работы данного компонента), а пользователи — события (т.е. внешние воздействия на компонент). Интерфейс StdControl применяется во многих компонентах TinyOS для выполнения их инициализации, запуска или остановки. Компонент TimerM предоставляет логику, преобразующую аппаратный таймер в его абстракцию, доступную в TinyOS. Абстракция таймера может использоваться для отсчета заданного промежутка времени. На рис. 13.9, *a* представлена также формальная спецификация интерфейсов из компонента TimerM.

Интерфейсы, связанные с компонентом TimerM, определяются следующим образом:

```
interface StdControl {
    command result_t init();
    command result_t start();
    command result_t stop();
}
interface Timer {
    command result_t start(char type, uint32_t interval);
    command result_t stop();
    event result_t fired();
}
interface Clock {
    command result_t setRate(char interval, char scale);
    event result_t fire();
}
```

Компоненты организуются в конфигурации путем их “связывания” вместе на уровне интерфейсов и отождествления интерфейсов конфигурации с некоторыми интерфейсами компонентов. Простой тому пример приведен на рис. 13.9, *б*, где компонент обозначается прописной буквой С, чтобы отличить интерфейс (например, Timer) от компонента, предоставляющего этот интерфейс (например, TimerC), а прописной буквой М — модуль. Такие условные обозначения применяются в том случае, если у одного логического компонента имеется конфигурация и модуля. В частности, компонент TimerC, представляющий интерфейс Timer, является конфигурацией, связывающей его реализацию (компонент TimerM) с поставщиками услуг системных часов и светодиодных индикаторов. В противном случае любому пользователю компонента TimerC придется явно привязать свои компоненты.

## Планировщик в TinyOS

Планировщик TinyOS работает во всех компонентах. Буквально все встроенные системы, в которых применяется TinyOS, будут однопроцессорными, а следовательно, среди всех заданий во всех компонентах одновременно может выполняться лишь одно задание. Планировщик является отдельным компонентом и той единственной составляющей TinyOS, которая непременно присутствовать в любой системе.

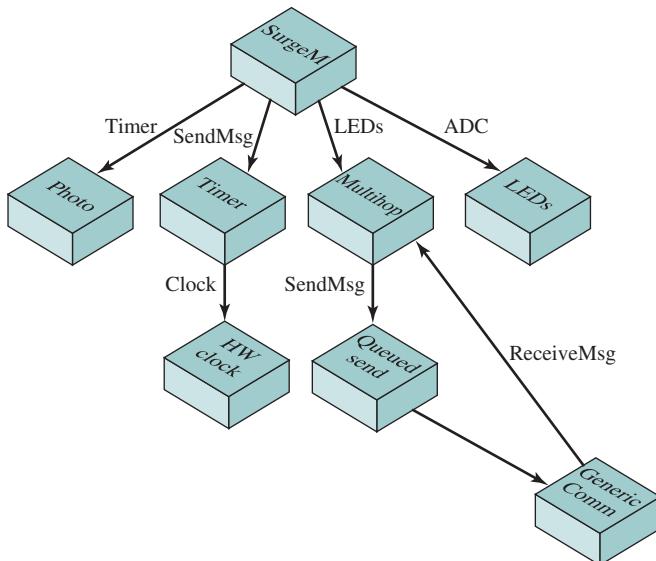
По умолчанию в TinyOS выбирается планировщик простой очереди, действующей по принципу “первым пришел — первым обслужен” (FIFO). Задание отправляется планировщику (т.е. помещается в очередь) в результате события, запускающего отправку задания, или же конкретного запроса при выполнении одного задания для планирования другого задания. Планировщик при работе учитывает потребляемую мощность. Это означает, что в отсутствие заданий в очереди планировщик переводит процессор в ждущий режим. А поскольку периферийные устройства продолжают работать, то одно из них может вывести систему из состояния ожидания с помощью аппаратного прерывания, о котором сигнализирует компонент самого низкого уровня. Как только очередь станет пустой, очередное задание может быть запланировано лишь в результате прямого аппаратного события. Такой режим работы обеспечивает эффективное использование заряда батареи питания.

Развитие планировщика прошло через два поколения. В версии TinyOS 1.x поддерживается очередь, общая для всех заданий, а компонент может неоднократно отправлять задание планировщику. Если очередь заданий заполнена, то операция отправки задания завершается неудачно. Как показывает опыт работы с сетевыми стеками, это обстоятельство может вызывать осложнения, поскольку задание может сигнализировать о завершении расщепляемой на фазы операции. Так, если отправка задания завершится неудачно, компонент верхнего уровня может оказаться навсегда заблокированным в ожидании, когда наступит событие завершения. В версии TinyOS 2.x для каждого задания резервируется место в очереди, а само задание может быть отправлено лишь один раз. Отправка задания завершается неудачно, только если задание уже было отправлено. Если же компоненту требуется отправить задание неоднократно, он может воспользоваться внутренней переменной состояния, чтобы при выполнении задание самостоятельно отправляло себя повторно. Это незначительное изменение в семантике значительно упрощает немалую долю исходного кода компонента. Вместо того чтобы перед отправкой задания проверять, не было ли оно уже отправлено, компонент может просто отправлять задание. В этой версии компонентам не нужно выполнять восстановление после неудачной отправки и пытаться ее повторить. Затраты на это решение составляют всего лишь один байт состояния для каждого задания.

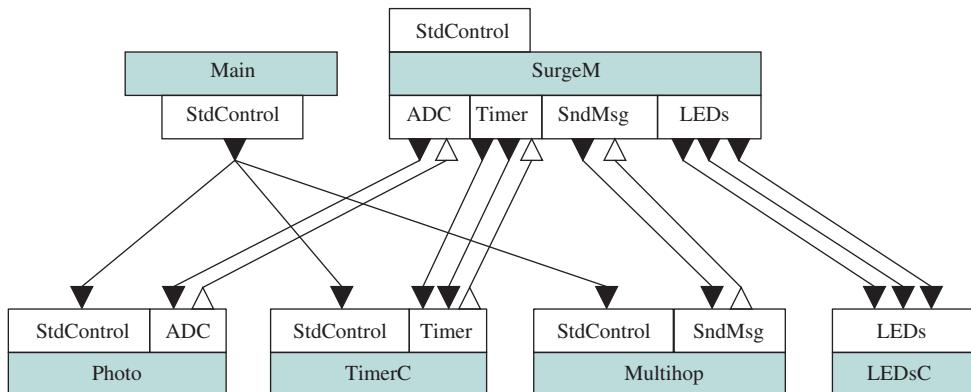
Пользователь может заменить выбираемый по умолчанию планировщик другим планировщиком, в котором применяется иная схема диспетчеризации (например, схема диспетчеризации по приоритетам или крайним срокам). Но при этом не следует пользоваться вытеснением и квантованием времени из-за издержек, возникающих в подобных системах. Что более важно, они нарушают модель параллельной обработки в TinyOS, где предполагается, что задания не вытесняют друг друга.

## Пример конфигурации

На рис. 13.10 приведен пример конфигурации, составленной из программных и аппаратных компонентов. Этот упрощенный пример приложения Surge описывается в [86] и демонстрирует периодический сбор данных от датчиков со специальной многоинтервальной маршрутизацией по беспроводной сети для доставки выборок данных на базовую станцию. В верхней части рис. 13.10 показаны компоненты Surge (обозначены блоками), а также интерфейсы (обозначены линиями со стрелками), через которые они связаны. Компонент SurgeM является компонентом прикладного уровня, направляющим действие конфигурации.



а) Упрощенное представление приложения Surge



б) Конфигурация Surge высокого уровня

LED — светодиод

ADC — аналого-цифровой преобразователь

Рис. 13.10. Пример приложения TinyOS

На рис. 13.10, б показана часть конфигурации для приложения Surge. Далее приведен упрощенный фрагмент, взятый из спецификации компонента SurgeM.

```
module SurgeM {
    provides interface StdControl;
    uses interface ADC;
    uses interface Timer;
    uses interface SendMsg;
    uses interface LEDs;
}
```

```

implementation {
    uint16_t sensorReading;
    command result_t StdControl.init() {
        return call Timer.start(TIMER_REPEAT, 1000);
    }
    event result_t Timer.fired() {
        call ADC.getData();
        return SUCCESS;
    }
    event result_t ADC.dataReady(uint16_t data) {
        sensorReading = data;
        ... отправка сообщения с данными ...
        return SUCCESS;
    }
    ...
}

```

Данный пример наглядно показывает сильные стороны подхода к разработке встроенных систем на основе TinyOS. Программное обеспечение организовано в виде взаимосвязанного ряда простых модулей, в каждом из которых определяется одно или несколько заданий. У одних компонентов имеется простой стандартизированный интерфейс с другими компонентами, будь то аппаратные или программные компоненты. Таким образом, компоненты могут быть легко заменены. Они могут быть как аппаратными, так и программными, а замена их границ невидима для прикладного программиста.

## Интерфейс ресурсов TinyOS

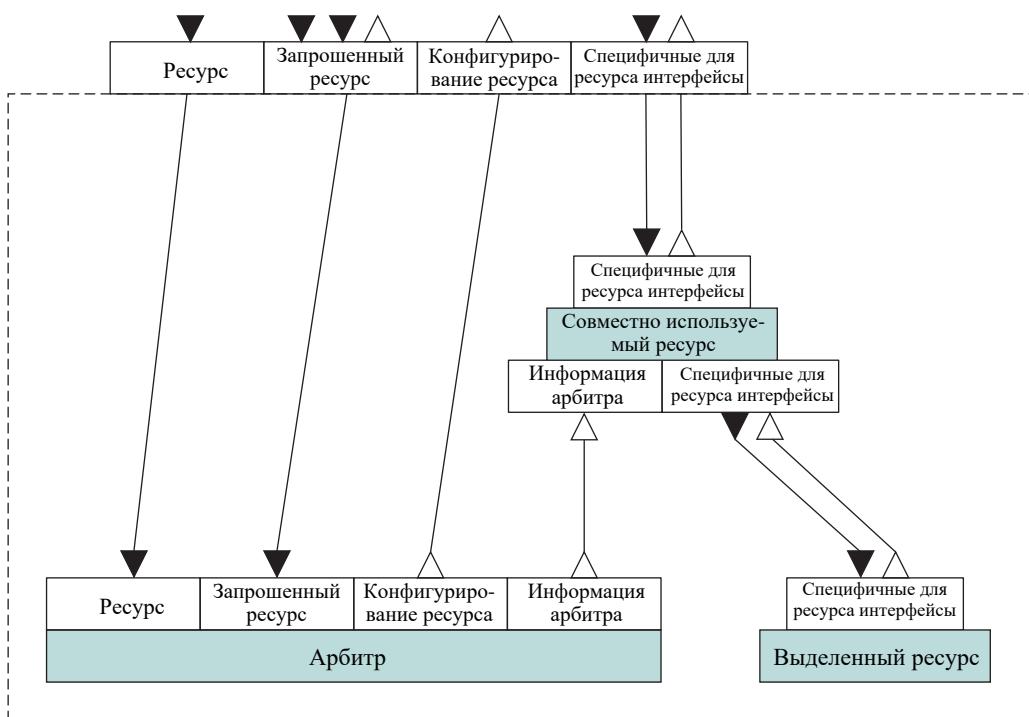
TinyOS предоставляет простое, но мощное множество условий для работы с ресурсами. Для ресурсов в TinyOS применяются следующие три абстракции.

- Выделенная.** Это ресурс, исключительный доступ к которому требуется подсистеме постоянно. В данной категории ресурсов стратегия совместного доступа не требуется, потому что пользоваться таким ресурсом требуется лишь одному компоненту. Примерами специально выделенных абстракций служат прерывания и счетчики.
- Виртуализованная.** Каждый клиент виртуализованного ресурса взаимодействует с ним, как будто он является специально выделенным ресурсом, причем все виртуализованные экземпляры используют один исходный ресурс. Виртуализованная абстракция может быть использована в том случае, если исходный ресурс не требуется защищать путем взаимоисключения. Характерным примером такой абстракции служит таймер или часы.
- Совместно используемая.** Такая абстракция ресурсов предоставляет доступ к специально выделенному ресурсу через арбитражный компонент. Этот арбитр соблюдает взаимоисключения, разрешая одновременный доступ к ресурсу лишь одному пользователю, называемому клиентом, а также обеспечивает блокировку данного ресурса.

В остальной части этого подраздела вкратце определяются возможности TinyOS для обращения с совместно используемыми ресурсами. Арбитр определяет того клиента, ко-

торому предоставляется доступ к ресурсу, а также время доступа. Пока клиент удерживает ресурс, он обладает полным и беспрепятственным контролем над ним. Арбитры полагают, что клиенты сотрудничают, захватывая ресурс лишь по мере необходимости и удерживая его не больше, чем требуется. Клиенты освобождают ресурсы явным образом, тогда как арбитр не имеет никакой возможности освободить его принудительно.

На рис. 13.11 показано упрощенное представление конфигурации совместно используемых ресурсов, обеспечивающей доступ к базовому ресурсу. С каждым ресурсом, предназначенным для совместного использования, связан арбитражный компонент. Арбитр обеспечивает выполнение стратегии, позволяющей клиенту заблокировать, использовать, а затем освободить ресурс. Конфигурация совместно используемых ресурсов предоставляет клиенту следующие интерфейсы.



**Рис. 13.11.** Конфигурация совместно используемых ресурсов

- **Ресурс.** Клиент выдает запрос через этот интерфейс, запрашивая доступ к ресурсу. Если в настоящий момент ресурс заблокирован, арбитр помещает запрос в очередь. Как только клиент окончит работать с ресурсом, он выдаст команду на освобождение данного ресурса.
- **Запрос ресурса.** Этот интерфейс подобен интерфейсу ресурса. Но в данном случае у клиента имеется возможность удерживать ресурс до тех пор, пока он не будет извещен, что данный ресурс требуется кому-то еще.
- **Конфигурирование ресурса.** Этот интерфейс позволяет автоматически сконфигурировать ресурс непосредственно перед предоставлением к нему доступа. В ком-

понентах, предоставляющих интерфейс конфигурирования ресурса, применяются интерфейсы, предоставляемые исходным специально выделенным ресурсом, чтобы сконфигурировать его, настроив для требуемого режима работы.

- **Специфичные для ресурса интерфейсы.** Как только клиент получает доступ к ресурсу, он пользуется специфичными для ресурса интерфейсами, чтобы обмениваться с ним данными и управляющей информацией.

В дополнение к выделенному ресурсу конфигурация совместно используемых ресурсов состоит из двух компонентов. В частности, арбитр принимает запросы на доступ и конфигурирование от клиента и выполняет блокировку базового ресурса. Компонент совместно используемого ресурса выступает посредником в обмене данными между клиентом и базовым ресурсом. Сведения об арбитре, передаваемые от арбитра к компоненту совместно используемого ресурса, управляют доступом клиента к исходному ресурсу.

## 13.5. КЛЮЧЕВЫЕ ТЕРМИНЫ, КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАЧИ

### Ключевые термины

eCos	Задание	Печатная плата
TinyOS	Интегральная схема	Прикладные процессоры
Встроенная операционная система	Команды	События
Встроенная система	Материнская плата	Специализированный процессор
Глубоко встроенная система	Микроконтроллер	
	Микросхема	

### Контрольные вопросы

- 13.1. Что такое встроенная система?
- 13.2. Каковы типичные требования к встроенным системам и накладываемые на них ограничения?
- 13.3. Что такое встроенная операционная система?
- 13.4. Каковы ключевые характеристики встроенной операционной системы?
- 13.5. Поясните относительные преимущества и недостатки встроенной операционной системы, основанной на существующей коммерческой операционной системе, по сравнению со специально разработанной встроенной операционной системой.
- 13.6. Каково целевое применение TinyOS?
- 13.7. Каковы цели разработки TinyOS?
- 13.8. Что собой представляет компонент TinyOS?
- 13.9. Какое программное обеспечение составляет TinyOS?
- 13.10. Какова стандартная стратегия планирования в TinyOS?

## Задачи

13.1. Планировщик в TinyOS обслуживает задачи в порядке FIFO (первым пришел — первым обслужен). Для TinyOS были предложены многие другие планировщики, но ни один из них так и не нашел широкого применения. Какие характеристики сети датчиков могут потребовать более сложного планирования?

13.2. а. Интерфейс ресурса TinyOS не позволяет компоненту, уже имеющему запрос ресурса в очереди, сделать второй запрос. Предложите пояснение такого поведения данного интерфейса.

б. Тем не менее интерфейс ресурса TinyOS позволяет компоненту, удерживающему блокировку ресурса, повторно запросить блокировку. Такой запрос ставится в очередь на удовлетворение в дальнейшем. Предложите пояснение такой стратегии. Указание: подумайте, что могло бы вызвать задержку между снятием блокировки одним компонентом и удовлетворением запроса на нее от следующего компонента?

**Примечание:** остальные задачи касаются системы eCos, рассматриваемой в приложении Р, “eCos”.

13.3. Что касается интерфейса драйвера устройства для ядра eCos, в драйверах устройств для запроса и снятия циклических блокировок рекомендуется применять варианты функции `_intsave()` вместо вариантов, отличающихся от `_intsave()`. Поясните причину такой рекомендации.

13.4. Рекомендуется также пореже пользоваться функцией `cug_drv_spinlock_spin()`, причем в тех случаях, когда не могут возникнуть взаимные и динамические блокировки. Поясните причину такой рекомендации.

13.5. Какие ограничения должны быть наложены на применение функции `cug_drv_spinlock_destroy()`?

13.6. Какие ограничения должны быть наложены на применение функции `cug_drv_mutex_destroy()`?

13.7. Почему в планировщике eCos не поддерживается квантование времени?

13.8. Реализация мьютексов в ядре eCos не поддерживает рекурсивные блокировки. Если поток заблокировал мьютекс, а затем пытается заблокировать его снова, это, как правило, приводит к рекурсивному вызову в усложненном графе вызовов, и в итоге либо проверкой утверждения будет выведено сообщение об ошибке, либо поток войдет в состояние взаимной блокировки. Поясните эту стратегию.

13.9. На рис. 13.12 приведен листинг исходного кода, предназначенного для применения в ядре eCos.

- Поясните работу кода, считая, что поток В начинает выполнение первым, а поток А начинает выполнение после некоторого события.
- Что произойдет, если при разблокировании мьютекса ожидающий в вызове функции `cug_cond_wait()` в строке 30 код будет неатомарным?
- Зачем требуется цикл `while` в строке 26 кода?

13.10. Описание циклических блокировок в eCos включает пример, наглядно показывающий, почему не следует пользоваться циклическими блокировками в однопроцессорной системе, если два потока с разными приоритетами могут соперничать за одну и ту же циклическую блокировку. Поясните, почему это затруднение останется, даже если одну и ту же циклическую блокировку запросят потоки с одинаковым приоритетом.

```
1 unsigned char buffer_empty = true;
2 cyg_mutex_t mut_cond_var;
3 cyg_cond_t cond_var;
4
5 void thread_a( cyg_addrword_t index )
6 {
7     while ( 1 ) // Бесконечное выполнение потока
8     {
9         // Захват данных в буфере...
10
11         // Теперь данные в буфере
12         buffer_empty = false;
13
14         cyg_mutex_lock( &mut_cond_var );
15
16         cyg_cond_signal( &cond_var );
17
18         cyg_mutex_unlock( &mut_cond_var );
19     }
20 }
21
22 void thread_b( cyg_addrword_t index )
23 {
24     while ( 1 ) // Бесконечное выполнение потока
25     {
26         cyg_mutex_lock( &mut_cond_var );
27
28         while ( buffer_empty == true )
29         {
30             cyg_cond_wait( &cond_var );
31         }
32
33
34         // Получение данных из буфера...
35
36         // Установка флага, что данные в буфере обработаны
37         buffer_empty = true;
38
39         cyg_mutex_unlock( &mut_cond_var );
40
41         // Обработка данных в буфере
42     }
43 }
```

Рис. 13.12. Пример кода, реализующего механизм условной переменной

# ВИРТУАЛЬНЫЕ МАШИНЫ

В ЭТОЙ ГЛАВЕ...

## 14.1. Концепции виртуальных машин

### 14.2. Гипервизоры

Назначение гипервизоров

Функции гипервизора

Гипервизор первого типа

Гипервизор второго типа

Паравиртуализация

Аппаратно поддерживаемая виртуализация

Виртуальное устройство

## 14.3. Контейнерная виртуализация

Группы управления ядром

Концепции контейнеров

Файловая система контейнера

Микрослужбы

Docker

## 14.4. Вопросы виртуализации на уровне процессоров

### 14.5. Управление памятью

### 14.6. Управление вводом-выводом

### 14.7. VMware ESXi

### 14.8. Варианты Hyper-V и Xen от корпорации Microsoft

### 14.9. Java VM

## 14.10. Архитектура виртуальной машины Linux VServer

Архитектура

Планирование процессов

## 14.11. Резюме

## 14.12. Ключевые термины, контрольные вопросы и задачи

Ключевые термины

Контрольные вопросы

Задачи

## УЧЕБНЫЕ ЦЕЛИ

- Обсудить два типа виртуализации.
- Пояснить метод контейнерной виртуализации и сравнить его с подходом гипервизора.
- Уяснить вопросы, связанные с реализацией виртуальной машины, на уровне процессоров.
- Уяснить вопросы, связанные с реализацией виртуальной машины, на уровне управления памятью.
- Уяснить вопросы, связанные с реализацией виртуальной машины, на уровне управления вводом-выводом.
- Сравнить и противопоставить виртуальные машины VMware ESXi, Hyper-V, Xen и Java VM.
- Пояснить принцип действия виртуальной машины Linux.

В этой главе основное внимание уделяется применению виртуализации при проектировании операционных систем. Виртуализация охватывает самые разные технологии для управления вычислительными ресурсами, предоставляя уровень преобразования программного обеспечения, известный под названием “уровень абстракции” и расположенный между программным обеспечением и физическим оборудованием. Виртуализация преобразует физические ресурсы в логические (или виртуальные). Она дает пользователям, приложениям и управляющим программам возможность действовать над уровнем абстракции, чтобы управлять ресурсами и пользоваться ими, даже не вникая в физические особенности расположенных ниже ресурсов.

В трех первых разделах этой главы описываются два основных подхода к виртуализации: виртуальные машины и контейнеры. А в остальной части этой главы рассматривается ряд конкретных систем, в которых эти подходы реализуются.

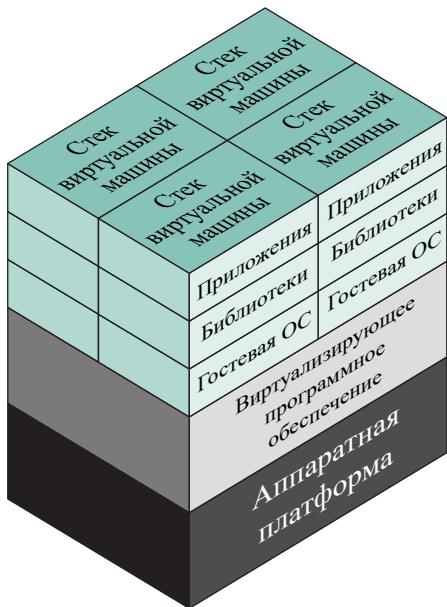
## 14.1. Концепции виртуальных машин

Традиционно приложения выполнялись непосредственно в операционной системе на персональном компьютере (ПК) или на сервере, причем на ПК или сервере одновременно могла действовать лишь одна операционная система. Следовательно, поставщику приложений приходилось переписывать отдельные их части для каждой платформы и операционной системы, в которой они должны выполняться и поддерживаться. Это приводит к задержке выпуска на рынок новых функциональных средств, увеличению вероятности появления дефектов, к дополнительным затратам труда на тестирование качества приложений, а в конечном счете — к повышению стоимости. Ради совместимости со многими операционными системами поставщикам приложений приходится создавать, управлять и поддерживать многие инфраструктуры оборудования и операционных систем, а это дорогостоящий и затратный в отношении ресурсов процесс.

Одной из эффективных стратегий, обеспечивающих выход из столь затруднительно-го положения, является так называемая **аппаратная виртуализация**. Такая технология виртуализации позволяет на единственном ПК или сервере одновременно работать не- скольким операционным системам (или сеансам работы одной операционной системы). Машина с программным обеспечением виртуализации позволяет разместить на одной платформе многочисленные приложения, в том числе те, которые выполняются в разных операционных системах. По сути, базовая операционная система (или хостовая опера- ционная система, операционная система-хозяин) может поддерживать целый ряд **виртуаль- ных машин** (virtual machine — VM), каждая из которых обладает характеристиками конкретной операционной системы, а в некоторых версиях виртуализации — характе-ristиками определенной аппаратной платформы. Виртуальные машины известны также как *системные виртуальные машины*, что подчеркивает то обстоятельство, что виртуа-лизации подлежат именно аппаратные средства системы.

Виртуализация — совсем не новая технология. Еще в 1970-е годы мейнфреймы IBM предоставали первые функциональные возможности, которые позволяли прикладным программам использовать лишь часть системных ресурсов. С тех пор подобные возмож-ности стали доступны в разных формах на разных платформах. Виртуализация стала мейнстримом в области вычислений в начале 2000-х годов, когда данная технология стала коммерчески доступной на серверах, построенных на процессорах x86. Органи-зации страдали избытком серверов из-за стратегии “одно приложение — один сервер”, принять которую побуждала операционная система Windows от корпорации Microsoft. А закон Мура стимулировал быстрые темпы развития аппаратных средств, превышав-шие возможности программных средств, и поэтому большинство серверов повсеместно оказывались недогруженными, и на каждом из них зачастую употреблялось менее 5% доступных ресурсов. Кроме того, избыток серверов, заполонивших центры обработки данных и потреблявших немало электроэнергии, расходующейся в том числе на охлаж-дение, внес немалое напряжение в способность корпораций управлять и поддерживать их инфраструктуру. Это напряжение помогла снять виртуализация.

Решением, обеспечивающим возможность виртуализации, является **монитор виртуаль- ных машин** (virtual machine monitor — VMM), широко известный ныне как *гипер-визор*. Это программное обеспечение располагается между аппаратными средствами и виртуальными машинами, действуя в качестве посредника в предоставлении ресурсов. Проще говоря, оно позволяет виртуальным машинам благополучно сосуществовать на одном физическом сервере узла и совместно пользоваться его ресурсами. Такого рода виртуализация наглядно представлена в общих чертах на рис. 14.1. Над аппаратной платформой располагается программное обеспечение виртуализации определенного рода, которое может состоять из хозяйской операционной системы и специализирован-ного виртуализирующего программного обеспечения или же просто программного па-кета, включающего в себя функции хозяйской операционной системы, а также функции виртуализации, как поясняется далее. Виртуализирующее программное обеспечение предоставляет абстракцию всех физических ресурсов, в том числе процессора, опера-тивной памяти, сети и запоминающих устройств, а следовательно, допускает наличие многих вычислительных стеков, именуемых виртуальными машинами, на одном физи-ческом узле.



**Рис. 14.1.** Концепция виртуальной машины

Каждая виртуальная машина включает в себя операционную систему, называемую **гостевой операционной системой**. Это может быть та же самая операционная система, что и хозяйская узла, или же другая операционная система. Например, гостевая операционная система Windows может выполняться в виртуальной машине в хозяйской операционной системе Linux. В свою очередь, гостевая операционная система поддерживает ряд стандартных библиотечных функций и различные бинарные файлы и приложения. С точки зрения приложений и пользователей, такой комплекс выглядит как реальная машина со своим оборудованием и операционной системой, что делает вполне уместным употребление термина *виртуальная машина*. Иными словами, это оборудование, которое виртуализируется.

Количество гостевых систем, которые могут существовать в одной хозяйской системе, измеряется **степенью консолидации** (consolidation ratio). Так, о системе, в которой поддерживаются 4 виртуальные машины, говорят, что она обладает степенью консолидации 4 к 1, записываемой также как 4:1 (см. рис. 14.1). Первые коммерчески доступные гипервизоры обеспечивали степень консолидации в пределах от 4:1 до 12:1, но даже если организации осуществляли виртуализацию на нижнем пределе (т.е. 4:1), им удавалось избавиться от 75% всех серверов, имевшихся в их центрах обработки данных. При этом им удавалось сократить затраты, которые нередко исчислялись миллионами или даже десятками миллионов долларов США ежегодно. Чем меньше было физических серверов, тем меньше требовалось электроэнергии на их работу и охлаждение. Кроме того, требовалось меньше кабелей, сетевых коммутаторов и площадей под оборудование. Повышение степени консолидации серверов стало (и продолжает оставаться) чрезвычайно ценным способом решения дорогостоящей и затратной задачи. Ныне во всем мире виртуальных серверов развернуто больше, чем физических, и процесс их развертывания продолжает ускоряться.

Основные причины, по которым организации пользуются виртуализацией, могут быть сведены к следующим.

- **Устаревшее оборудование.** Приложения, разработанные для устаревшего оборудования, можно по-прежнему выполнять, виртуализируя (или эмулируя) устаревшее оборудование, обеспечивая постепенный его вывод из эксплуатации.
- **Быстрое развертывание.** Как поясняется далее, новая виртуальная машина может быть развернута в считанные минуты, хотя на развертывание новых серверов в конкретной инфраструктуре могут уйти целые недели, а то и больше. Дело в том, что виртуальная машина состоит из файлов. Дублируя эти файлы, можно получить идеальную копию сервера в виртуальной среде.
- **Разносторонность.** Применение оборудования можно оптимизировать, увеличив до максимума количество видов приложений, которые могут быть выполнены на одном компьютере.
- **Консолидация.** Ресурсом большой емкости и быстродействия (например, сервером) можно пользоваться более эффективно, сделав его общим и одновременно доступным для многих приложений.
- **Агрегирование.** Виртуализация упрощает объединение многих ресурсов в единый виртуальный ресурс, как, например, при виртуализации запоминающих устройств.
- **Динамичность.** Применение виртуальных машин упрощает выделение аппаратных ресурсов в динамическом режиме. Благодаря этому улучшается распределение нагрузки и повышается отказоустойчивость.
- **Простота управления.** Виртуальные машины упрощают развертывание и тестирование программного обеспечения.
- **Повышенная доступность.** Узлы виртуальных машин группируются в кластеры, образуя пулы вычислительных ресурсов. На физических серверах каждого из этих узлов размещаются многие виртуальные машины, и, если выйдет из строя какой-нибудь физический сервер, виртуальные машины в отказавшем узле могут быть быстро перезапущены в автоматическом режиме в другом узле данного кластера. Если сравнить это с аналогичной доступностью на физическом сервере, то виртуальные среды могут обеспечить более высокую доступность при значительно меньших затратах и степени сложности.

Коммерческие варианты виртуальных машин, предлагаемые такими организациями, как VMware и Microsoft, широко применяются на серверах, и уже проданы миллионы их копий. Помимо возможности выполнять многие виртуальные машины на одном хосте, главная особенность виртуализации серверов состоит в том, что виртуальные машины могут рассматриваться как сетевые ресурсы. Виртуализация серверов маскирует от пользователей их ресурсы, в том числе количество и отличительные признаки физических серверов, процессоров и операционных систем. Это дает возможность разделить один узел на несколько независимых серверов и тем самым сберечь аппаратные ресурсы, а также быстро перенести сервер с одной машины на другую, чтобы равномерно распределить нагрузку или динамически подключить резерв при выходе машины из строя. Виртуализация серверов стала центральным элементом в разработке приложений для “больших данных” и реализации инфраструктур облачных вычислений.

Помимо серверных сред, подобные технологии применяются в настольных средах для одновременного выполнения нескольких операционных систем — как правило, Windows и Linux.

## 14.2. ГИПЕРВИЗОРЫ

Для различных подходов, принятых при разработке виртуальных машин, не существует какой-то определенной классификации. В этом разделе рассматривается понятие гипервизора, которое может служить наиболее общим основанием для классификации походов к разработке виртуальных машин.

### Назначение гипервизоров

Виртуализация является формой абстракции. Подобно тому, как операционная система абстрагирует от пользователя команды ввода-вывода на жесткие диски, применяя программные уровни и интерфейсы, виртуализация абстрагирует физическое оборудование от виртуальных машин, которые оно поддерживает. Монитор виртуальных машин, или гипервизор, — это программное обеспечение, предоставляющее такую абстракцию. Он служит своего рода брокером, или регулировщиком уличного движения, действуя в качестве посредника для гостевых виртуальных машин, когда они запрашивают или потребляют ресурсы физического узла.

Виртуальная машина является программной конструкцией, имитирующей характеристики физического сервера. Она конфигурируется с определенным количеством процессоров, объемом памяти, ресурсов запоминающих устройств и возможностью соединения через сетевые порты. Как только виртуальная машина создана, она может быть включена, способна загрузить операционную систему и программные комплексы и использована как физический сервер. Но, в отличие от физического сервера, виртуальному серверу доступны только те ресурсы, с которыми он был сконфигурирован, а не все ресурсы самой физической машины. Такая изоляция позволяет выполнять на машине-хозяине несколько гостевых виртуальных машин, на каждой из которых без особых затруднений выполняются одинаковые или разные копии операционных систем, используются общая память, а также вторичные запоминающие устройства и пропускная способность сети. Операционная система виртуальной машины получает доступ к ресурсам, предоставляемым ей гипервизором, который упрощает трансляцию и ввод-вывод как из виртуальной машины на физические устройства серверов, так и обратно на нужную виртуальную машину. Подобным образом гипервизор, действуя в качестве посредника для виртуальной машины, перехватывает и выполняет некоторые привилегированные команды, которые ее “родная” операционная система будет выполнять на оборудовании своего узла. Это создает предпосылки к некоторому снижению производительности в процессе виртуализации, хотя со временем подобные издержки будут сведены к минимуму путем усовершенствования аппаратных и программных средств.

Экземпляр виртуальной машины определяется как набор файлов. Типичная виртуальная машина может состоять всего лишь из нескольких файлов. К их числу относится файл конфигурации, описывающий свойства виртуальной машины. Он содержит определение сервера, количество виртуальных процессоров, выделяемых данной виртуальной машине, объем выделяемой памяти, устройства ввода-вывода, доступные данной виртуальной машине, количество сетевых адаптеров на виртуальном сервере и многое

другое. В этом файле описываются также запоминающие устройства, доступные данной виртуальной машине. Зачастую эти запоминающие устройства представлены в виде виртуальных дисков, существующих в виде файлов в физической файловой системе. Когда виртуальная машина включается или инстанцируется, создаются дополнительные файлы для протоколирования, страничной организации памяти и прочих функций. А поскольку виртуальная машина, по существу, состоит из файлов, то некоторые функции могут быть определены в виртуальной среде проще и быстрее, чем в физической среде. С самого начала развития вычислительной техники критически важной функцией считалось резервное копирование данных. А поскольку виртуальные машины уже определены в виде файлов, копирование последних, по существу, означает резервное копирование не только данных, но и всего сервера в целом, включая операционную систему, приложения и саму аппаратную конфигурацию.

Типичный способ быстрого развертывания новых виртуальных машин состоит в применении шаблонов. Шаблон предоставляет стандартизированную группу параметров настройки оборудования и программного обеспечения, с помощью которых можно создавать новые виртуальные машины, сконфигурированные с этими параметрами. Процедура создания новой виртуальной машины из шаблона состоит в том, чтобы предоставить уникальные идентификаторы для новой виртуальной машины, построить ее из шаблона в инициализирующем программном обеспечении и внести изменения в ее конфигурацию по ходу развертывания.

## Функции гипервизора

Ниже перечислены основные функции, выполняемые гипервизором.

- **Управление выполнением виртуальных машин.** Включает в себя планирование выполнения виртуальных машин, управление виртуальной памятью с целью обеспечить изоляцию одной виртуальной машины от остальных, переключение контекста между разными состояниями процессора. К этой функции относятся также изоляция виртуальных машин с целью предотвратить конфликты в использовании ресурсов и эмуляция таймера и механизмов прерываний.
- **Эмуляция устройств и управление доступом.** Эмуляция всех сетевых и запоминающих (блочных) устройств, предполагаемых в различных платформенно-ориентированных драйверах виртуальных машин, посредничество в доступе к физическим устройствам из разных виртуальных машин.
- **Выполнение гипервизором привилегированных операций для гостевых виртуальных машин.** Некоторые операции, вызываемые из гостевых операционных систем, в силу их привилегированного характера могут быть выполнены от имени гипервизора вместо непосредственного выполнения аппаратными средствами хозяина.
- **Управление виртуальными машинами, иначе называемое управлением жизненным циклом виртуальных машин.** Конфигурирование гостевых виртуальных машин и управление их состояниями (например, запуском, приостановкой или остановкой).
- **Администрирование платформы и программного обеспечения гипервизора.** Включает в себя установку параметров для взаимодействия пользователя с узлом, а также с программным обеспечением гипервизора.

## Гипервизор первого типа

Имеются два типа гипервизоров, различающихся наличием операционной системы между гипервизором и хостом. Гипервизор первого типа (рис. 14.2, а) загружается как программный уровень непосредственно над физическим уровнем — подобно тому, как загружается операционная система. Гипервизор первого типа может непосредственно управлять физическими ресурсами узла. Как только он будет установлен и сконфигурирован, сервер сможет поддерживать гостевые виртуальные машины. В серьезных, отработанных средах, в которых хосты виртуализации ради большей доступности и равномерного распределения нагрузки группируются в кластеры, гипервизор может быть размещен в новом узле. Затем новый узел присоединяется к существующему кластеру, а виртуальные машины перемещаются в новый узел без прерывания обслуживания. К некоторым примерам гипервизоров первого типа относятся VMware ESXi, Microsoft Hyper-V, а также различные варианты Xen.

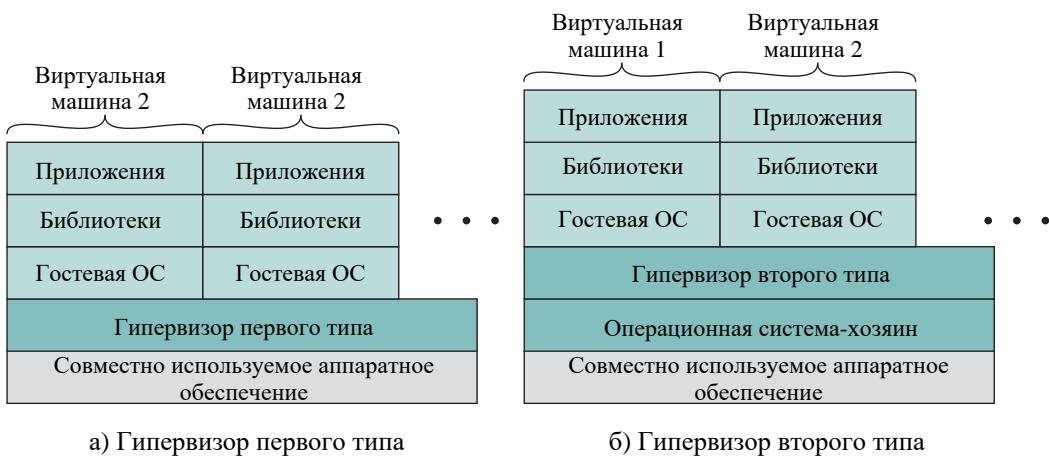


Рис. 14.2. Гипервизоры первого и второго типов

## Гипервизор второго типа

В гипервизоре второго типа используются ресурсы и функции операционной системы-хозяина, а действует он как программный модуль над операционной системой (см. рис. 14.2, б). Чтобы организовать все виды взаимодействия с оборудованием от имени самого гипервизора, он опирается на операционную систему-хозяина. К примерам гипервизоров второго типа относятся VMware Workstation и Oracle VM Virtual Box.

Ниже перечислены главные особенности гипервизоров обоих типов.

- Как правило, гипервизоры первого типа действуют лучше, чем гипервизоры второго типа. Поскольку гипервизор первого типа не соперничает за ресурсы с операционной системой, на сервере доступно больше ресурсов, так что с помощью гипервизоров первого типа на сервере виртуализации может быть размещено больше виртуальных машин.
- Кроме того, гипервизоры первого типа считаются более безопасными, чем гипервизоры второго типа. Виртуальные машины в гипервизоре первого типа делают запросы ресурсов, которые обрабатываются вне данной гостевой операционной

системы, и поэтому они не могут оказывать никакого влияния на другие виртуальные машины или гипервизор, который их поддерживает. Для виртуальных машин в гипервизоре второго типа это не обязательно, так что злонамеренный гость может потенциально оказывать влияние не только на самого себя.

- Гипервизоры второго типа дают пользователю возможность извлечь выгоду из виртуализации, обходясь без выделенного сервера. Те разработчики, которым по ходу производственного процесса требуется работать в разных средах, могут не только извлечь дополнительную выгоду из личного продуктивного рабочего пространства, предоставляемого операционной системой ПК, но и установить гипервизор второго типа в виде приложения в своей настольной системе под управлением Linux или Windows, добившись тем самым возможности работать с обеими операционными системами. Создаваемые и применяемые виртуальные машины могут быть перенесены или скопированы из среды одного гипервизора в среду другого гипервизора, сокращая время развертывания, повышая достоверность того, что развертывается, а также сокращая время выхода проекта на рынок.

## Паравиртуализация

По мере того как в коммерческих организациях стала преобладать виртуализация, поставщики оборудования и программного обеспечения начали искать пути еще большего повышения ее эффективности. Неудивительно, что эти пути привели к виртуализации, обеспечиваемой как с помощью программного обеспечения, так и с помощью оборудования. **Паравиртуализация** является методикой программной виртуализации, применяющей специализированные прикладные интерфейсы API для связывания виртуальных машин с гипервизором с целью оптимизировать их производительность. Операционная система виртуальной машины (Linux или Microsoft Windows) обеспечивает специализированную паравиртуализацию как составную часть ядра, а также в виде специальных драйверов паравиртуализации, что позволяет операционной системе и гипервизору работать вместе более эффективно, хотя и с определенными издержками на преобразования гипервизора. Такая методика программной виртуализации обеспечивает ее оптимизированную поддержку на серверах с процессорами, как предоставляющими расширения виртуализации, так и без таковых. Поддержка паравиртуализации стала предоставляться как составная часть многих из распространенных дистрибутивов Linux, начиная с 2008 года.

Несмотря на то что конкретные подробности реализации такого подхода в разных вариантах различаются, здесь приводится общее его описание (рис. 14.3). В отсутствие паравиртуализации гостевая операционная система может действовать без модификации, если гипервизор эмулирует оборудование. В этом случае вызовы оборудования из драйверов гостевой операционной системы перехватываются гипервизором, который выполняет все необходимые преобразования для используемого аппаратного обеспечения и переадресовывает вызов реальному драйверу. При наличии паравиртуализации исходный код операционной системы модифицируется таким образом, чтобы действовать как гостевая операционная система в среде определенной виртуальной машины. Вызовы оборудования заменяются вызовами гипервизора, способного принять их и без модификаций переадресовать настоящим драйверам. Такая организация вычислительных ресурсов обеспечивает большее быстродействие и меньшие издержки, чем конфигурация без паравиртуализации.

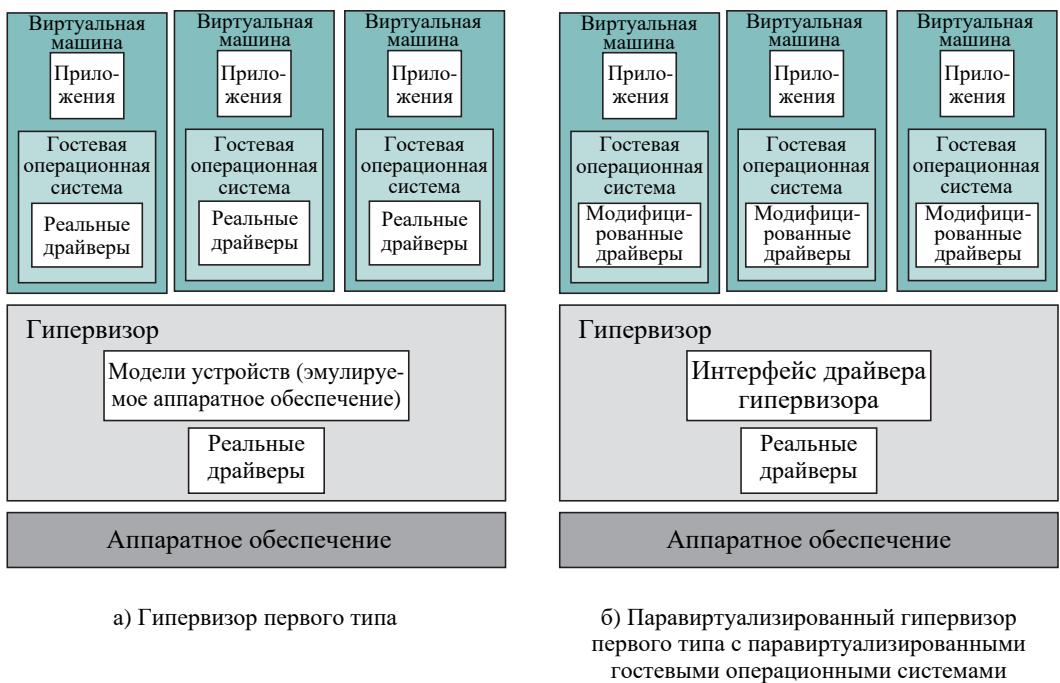


Рис. 14.3. Паравиртуализация

## Аппаратно поддерживаемая виртуализация

Аналогично производители процессоров AMD и Intel внедрили в свои процессоры дополнительные функциональные средства, чтобы повысить производительность с помощью гипервизоров. В частности, в обозначениях “AMD-V” и “Intel VT-x” указываются расширения аппаратной поддержки виртуализации, которыми гипервизоры могут воспользоваться во время работы. В процессорах Intel предоставляется дополнительный набор команд под названием **расширения виртуальных машин** (Virtual Machine Extensions — VMX). Наличие такого набора команд в процессоре избавляет гипервизоры от необходимости поддерживать соответствующие функции как часть своей кодовой базы. Сам код при этом становится более компактным и эффективным, а быстродействие поддерживаемых операций оказывается существенно выше, поскольку они полностью выполняются процессором. В отличие от паравиртуализации, такая обеспечиваемая аппаратно поддержка не требует модификации гостевой операционной системы.

## Виртуальное устройство

Виртуальное устройство (virtual appliance) представляет собой автономное программное обеспечение, которое может распространяться в виде образа виртуальной машины. Таким образом, оно состоит из упакованного множества приложений и гостевой операционной системы. Оно не зависит от архитектуры гипервизора или процессора и может работать с гипервизором либо первого, либо второго типа.

Развернуть предварительно настроенное и сконфигурированное виртуальное устройство намного проще, чем подготовить систему, установить приложение, а затем сконфигурировать и настроить его. Виртуальные устройства фактически становятся средствами распространения программного обеспечения и породили новый вид коммерческой деятельности в качестве поставщиков виртуальных устройств.

Помимо многих полезных примеров применения виртуальных устройств в прикладных областях, относительно недавно появилось очень важное направление в разработке виртуальных устройств защиты (security virtual appliance — SVA). Такое устройство является инструментальным средством защиты, выполняющим функцию технического контроля и защиты других (пользовательских) виртуальных машин и действующим за их пределами в виртуальной машине с особо усиленной защитой. Виртуальное устройство защиты получает доступ к состоянию виртуальной машины, включая состояние процессора, регистров, оперативной памяти и устройств ввода-вывода, а также к сетевому трафику как между самими виртуальными машинами, так и между гипервизором и виртуальными машинами через прикладной интерфейс API гипервизора, предназначенный для самоанализа виртуальных машин. Преимущества такого решения указываются в документе NIST SP 800-125 (*Security Recommendations for Hypervisor Deployment* — Рекомендации по обеспечению безопасности для разработки гипервизоров), выпущенном в октябре 2014 года. В частности, виртуальное устройство защиты должно быть:

- неуязвимым к дефектам гостевой операционной системы;
- независимым от конфигурации виртуальной сети и не должно перенастраиваться всякий раз, когда конфигурация виртуальной сети изменяется вследствие переноса виртуальных машин или изменений в подключаемости к сети тех виртуальных машин, которые резидентны для узла гипервизора.

## 14.3. КОНТЕЙНЕРНАЯ ВИРТУАЛИЗАЦИЯ

Относительно недавний подход к виртуализации известен под названием **контейнерная виртуализация**. При таком подходе программное обеспечение, называемое **контейнером виртуализации**, выполняется над ядром операционной системы-хозяина, представляя изолированную среду для выполнения приложений. В отличие от виртуальных машин на основе гипервизоров, контейнеры не предназначены для эмуляции физических серверов. Вместо этого все контейнерные приложения совместно используют общее ядро операционной системы. Благодаря этому исключаются ресурсы, необходимые для выполнения отдельной операционной системы для каждого приложения, что позволяет значительно сократить накладные расходы.

### Группы управления ядром

Большая часть применяемой в настоящее время технологии контейнеров была разработана для Linux, так что наиболее широко распространены контейнеры, основанные на Linux. Прежде чем перейти к обсуждению контейнеров, полезно ввести понятие группы управления ядром Linux. Как отмечалось в [169], прикладной интерфейс API для стандартного процесса Linux был расширен в 2007 году с целью внедрить контейнеризацию пользовательской среды, чтобы разрешить группирование нескольких процессов, а также обеспечить права доступа и управление системными ресурсами. Первоначально

употреблялся термин *контейнеры процессов* (process container), но в конце 2007 года на смену ему пришел термин *группы управления* (control group, cgroup) во избежание путаницы, обусловленной разным толкованием термина *контейнер* в контексте ядра Linux. Функциональные возможности групп управления были путем слияния присоединены к основной ветви разработки ядра Linux в версии 2.6.24, выпущенной в январе 2008 года.

Пространство имен процессов Linux имеет вид иерархии, в которой все процессы порождены от общего процесса, выполняемого на стадии начальной загрузки и называемого *init* (от англ. *initialization* — “инициализация”). При этом образуется единая иерархия процессов, а группа управления ядром допускает существование нескольких иерархий процессов в единой операционной системе. Во время конфигурирования каждая иерархия присоединяется к системным ресурсам.

Группа управления ядром обеспечивают следующее.

- **Ограничение ресурсов.** Группы могут быть настроены таким образом, чтобы не превышать сконфигурированный предел памяти.
- **Приоритизация.** Некоторые группы могут получать большую долю времени процессора или пропускной способности дискового ввода-вывода.
- **Учет используемых ресурсов.** Определяет меру употребления ресурса группой, которая может быть использована, например, для выставления счетов на оплату.
- **Управление.** Замораживание групп процессов, создание точек их контроля и перезапуск.

## Концепции контейнеров

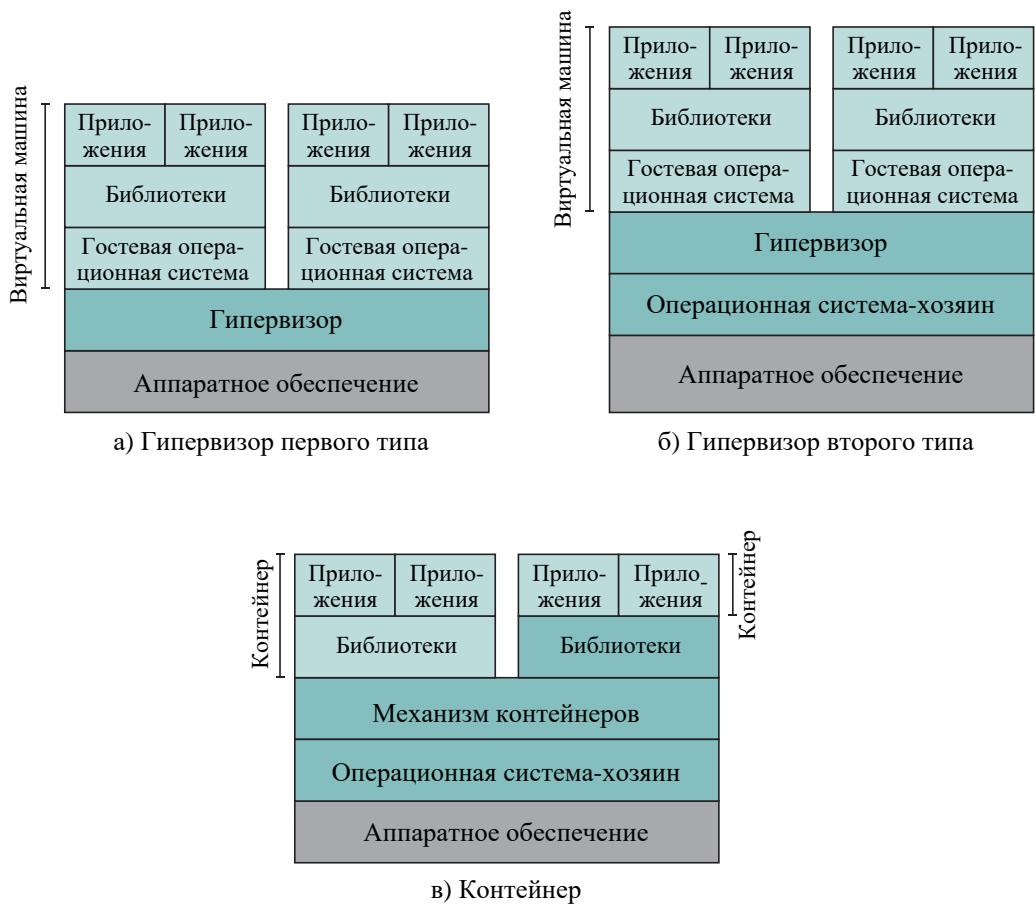
На рис. 14.4 приведено сравнение программных стеков контейнеров и гипервизоров. Для контейнеров требуется лишь небольшой механизм их поддержки. Механизм контейнеров настраивает каждый контейнер в качестве изолированного экземпляра, запрашивая для него ресурсы, выделенные операционной системой-хозяином, а затем эти ресурсы используются непосредственно в приложении каждого контейнера. И хотя подробности описываемой здесь процедуры различаются в зависимости от конкретного механизма контейнеров, ниже перечислены типичные задачи, выполняемые этим механизмом.

- Поддержание упрощенной среды выполнения и набора инструментальных средств для управления контейнерами, образами и сборками.
- Создание процесса для контейнера.
- Управление точками монтирования файловой системы.
- Запрашивание у ядра таких ресурсов, как, например, оперативная память, устройства ввода-вывода и IP-адреса.

Типичный жизненный цикл контейнеров, построенных на основе Linux, можно пояснить, описав стадии их существования.

- **Подготовка.** На стадии подготовки создается среда и запускаются контейнеры Linux. Типичным примером стадии подготовки служит активизация флагов или установка пакетов в ядре Linux для разрешения разделения пользовательского пространства. Подготовка включает в себя также установку набора инструментальных средств и утилит (например, `lxc`, `bridge`), чтобы инстанцировать среду контейнера и сетевую конфигурацию в операционной системе-хозяине.

- Конфигурирование.** Контейнеры конфигурируются для выполнения конкретных приложений или команд. Конфигурация контейнеров Linux включает в себя сетевые параметры (например, IP-адрес), корневые файловые системы, операции монтирования и т.е. устройства, доступ к которым разрешается через среду контейнеров. В общем случае контейнеры конфигурируются для того, чтобы разрешить выполнение приложений в контролируемых системных ресурсах (таких, как верхняя граница доступа приложения к оперативной памяти).
- Управление.** Как только контейнер будет подготовлен и сконфигурирован, им придется каким-то образом управлять, чтобы обеспечить его начальную загрузку и остановку. Как правило, к числу управляемых операций, предназначенных для выполнения в контейнерной среде, относятся запуск, остановка, замораживание и перенос. Кроме того, имеются метакоманды и наборы инструментов, позволяющие контролировать выделение контейнеров в одном узле для доступа со стороны конечных пользователей и управлять им.



**Рис. 14.4.** Сравнение виртуальных машин и контейнеров

Все контейнеры на одной машине выполняются в одном и том же ядре, а следовательно, совместное использование большей части базовой операционной системы и конфигурирование с помощью контейнеров оказывается меньшим бременем в сравнении с организацией виртуальной машины в виде гипервизоров и гостевых операционных систем. Соответственно, операционная система может иметь множество работающих поверх нее контейнеров, что особенно ярко отличается при сравнении с ограниченным количеством поддерживаемых гипервизоров и гостевых операционных систем.

Виртуальные контейнеры возможны благодаря управлению ресурсами и изоляции процессов с помощью таких методик, как группа управления ядром, как пояснялось выше. Такой подход позволяет разделять общие системные ресурсы среди многих экземпляров изолированных контейнеров. Группы управления предоставляют механизм для управления и мониторинга системными ресурсами. Производительность приложений оказывается близкой к производительности базовой системы благодаря совместному использованию общего ядра всеми экземплярами контейнеров в пространстве имен, а накладные расходы требуются лишь для предоставления механизма, изолирующего контейнеры через группы управления. Подсистемы Linux, разделяемые с помощью примитивов групп управления, включают в себя файловую систему, пространство имен процессов, сетевой стек, имя сетевого узла, взаимодействие процессов (IPC) и пользователей.

Чтобы сравнить виртуальные машины с контейнерами, рассмотрим операцию ввода-вывода прикладным процессом P, выполняемым в виртуализированной среде. В классической среде виртуализации (без аппаратной поддержки) процесс P будет выполняться в гостевой виртуальной машине. Операция ввода-вывода направляется через стек гостевой операционной системы для эмуляции гостевого устройства ввода-вывода. Вызов устройства ввода-вывода перехватывается далее гипервизором, который направляет его через стек гостевой операционной системы физическому устройству. Контейнер основан главным образом на косвенном механизме, предоставляемом расширениями каркаса контейнеров, внедренными в основное ядро. Здесь единственное ядро совместно используется многими контейнерами (в системных виртуальных машинах у каждой гостевой операционной системы имеется отдельное ядро). На рис. 14.5 схематически показано прохождение потока данных через виртуальные машины и контейнеры.

Ниже перечислены две примечательные характеристики контейнеров.

1. В среде контейнеров не требуется гостевая операционная система. Таким образом, контейнеры легковесны и требуют меньше накладных расходов по сравнению с виртуальными машинами.
2. Программное обеспечение управления контейнерами упрощает процедуру создания контейнеров и управления ими.

Благодаря своей легковесности контейнеры служат привлекательной альтернативой виртуальным машинам. Еще одно привлекательное свойство контейнеров состоит в том, что они обеспечивают переносимость приложений. Контейнеризованные приложения можно быстро переносить из одной системы в другую.



**Рис. 14.5.** Прохождение потока данных в операции ввода-вывода через гипервизор и контейнер

Эти преимущества контейнеров совсем не означают, что их следует всегда предпочитать виртуальным машинам, как следует из приведенных ниже соображений.

- Контейнерные приложения переносятся лишь среди тех систем, которые поддерживают одно и то же ядро операционной системы и функциональные средства виртуализации, а это обычно означает Linux. Следовательно, контейнерное приложение Windows будет выполняться только на машинах под управлением Windows.
- Для виртуальной машины может потребоваться особая настройка ядра, непригодная для других виртуальных машин хоста. Эта проблема решается путем применения гостевой операционной системы.
- Виртуализация виртуальной машины функционирует на границе оборудования и операционной системы. С помощью суженного интерфейса между виртуальными машинами и гипервизорами можно обеспечить прочную изоляцию режима работы и гарантировать надлежащую безопасность. Контейнеризация же, располагающаяся между операционной системой и приложениями, влечет за собой меньшие накладные расходы, но потенциально вносит большую степень уязвимости в безопасность системы.

Один из примеров применения рассматриваемого здесь подхода к виртуализации описан в [127] и относится к Kubernetes — технологии с открытым исходным кодом для манипулирования контейнерами, разработанной в компании Google, но теперь управляемой некоммерческой организацией CNCF (Cloud Native Computing Foundation — Фонд облачно-ориентированных вычислений). Сам этот фонд представляет собой проект Linux Foundation Collaborative. Например, если администратор выделяет канал связи на 500 Мбит/с для выполнения приложения в Kubernetes, то при планировании этого приложения может быть задействовано управление сетью, чтобы найти наилучшее место, гарантирующее такую пропускную способность. При взаимодействии с прикладным интерфейсом Kubernetes API на уровне управления сетью можно приступить к составлению таких правил брандмауэра, в которых принимаются во внимание контейнерные приложения.

## Файловая система контейнера

Каждый контейнер должен поддерживать свою обособленную файловую систему как часть изоляции контейнера. Отдельные свойства такой файловой системы варьируются от одного конкретного контейнерного продукта к другому, но основные принципы ее организации одинаковы.

Рассмотрим в качестве примера контейнерную файловую систему, применяемую в OpenVZ и приведенную на рис. 14.6. Процесс инициализации планировщика выполняется с целью запланировать пользовательские приложения, причем у каждого контейнера имеется свой процесс инициализации, который с точки зрения аппаратных узлов является всего лишь очередным выполняющимся процессом.

Множественные контейнеры хоста, вероятнее всего, выполняют одни и те же процессы, но у каждого из них отсутствует отдельная копия, несмотря на то что по команде `ls` выводится каталог `/bin` контейнера, заполненный программами. Вместо этого контейнеры совместно пользуются общим шаблоном — конструктивной особенностью, которая состоит в том, что все приложения, входящие в состав операционной системы, а также многие из наиболее распространенных приложений упакованы вместе в виде групп файлов, размещаемых в операционной системе платформы и символически связанных с каждым контейнером. К их числу относятся и файлы конфигурации, если только они не модифицируются самим контейнером; в последнем случае операционная система копирует файл шаблона в файловую систему контейнера (копирование при записи). Благодаря применению такой схемы совместного использования виртуальных файлов достигается немалая экономия свободного места, когда в файловой системе контейнера фактически находятся лишь локально созданные файлы.

На уровне дисков контейнер является файлом, который можно масштабировать в сторону увеличения или уменьшения. А с точки зрения проверки на вирусы файловая система контейнера монтируется под особой точкой монтирования в аппаратном узле, а потому системные инструментальные средства, доступные на уровне аппаратного узла, могут благополучно и безопасно проверить каждый файл по мере надобности.

## Микрослужбы

С контейнерами связано понятие микрослужбы. В документе NIST SP 800-180 (*NIST Definition of Microservices, Application Containers and System Virtual Machines* — Определение NIST микрослужб, контейнеров приложений и виртуальных машин системы),

вышедшем в феврале 2016 года, микрослужбы определяются как основной элемент, образующийся в результате архитектурного разложения компонентов приложения на слабо связанные шаблоны, состоящие из самостоятельных служб, сообщающихся вместе с помощью стандартного протокола 219 для передачи данных, и ряда вполне определенных прикладных интерфейсов API, независимо от конкретного поставщика, продукта и технологии.

Основной замысел, положенный в основу микрослужб, состоит в том, чтобы разделить каждую конкретную службу в цепочке доставки приложений на отдельные части вместо того, чтобы иметь в своем распоряжении монолитный стек приложений. Пользуясь контейнерами, люди вполне осознанно разделяют свою инфраструктуру на более понятные блоки. При этом в сетевых технологиях открывается возможность принимать от имени пользователей такие решения, которые раньше были просто невозможны в машино-ориентированном мире.

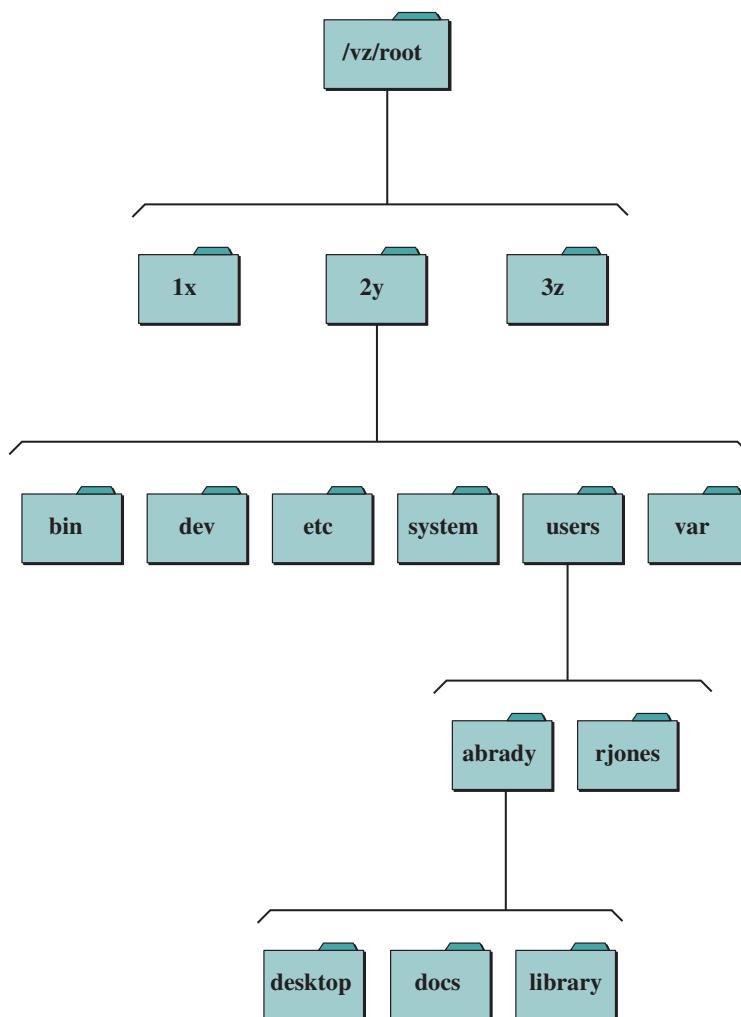


Рис. 14.6. Схематическое представление контейнерной файловой системы в OpenVZ

Ниже перечислены главные преимущества микрослужб.

- Микрослужбы реализуют намного более мелкие развертываемые блоки, что затем дает пользователю возможность быстрее выпускать обновления или внедрять новые функциональные средства и возможности. Это согласуется с нормами практики непрерывной поставки программных продуктов, в соответствии с которыми цель состоит в том, чтобы выпускать их мелкими блоками, не создавая монолитную систему.
- Микрослужбы поддерживают также точную масштабируемость. А поскольку микрослужба является частью намного более крупного приложения, то ее можно без особого труда реплицировать для создания множества экземпляров, и распределить нагрузку лишь на одну мелкую часть приложения, а не на все приложение в целом.

## Docker

Первоначально контейнеры появились как удобное и гибкое средство выполнения приложений. Контейнеры Linux позволяли выполнять легковесные приложения непосредственно в операционной системе Linux. Не требуя гипервизора и виртуальных машин, приложения можно было выполнять обособленно в одной и той же операционной системе. Контейнеры Linux стали применяться в центрах обработки данных компании Google, начиная с 2006 года. Но контейнерный подход к виртуализации нашел более широкое применение с появлением контейнеров Docker в 2013 году. Инструментальное средство Docker предоставляет более простой и стандартизированный способ выполнения контейнеров в сравнении с более ранними их версиями. Контейнер Docker также работает в Linux, хотя Docker и не единственное средство для работы контейнеров. Еще одним таким средством является Linux Containers (LXC). Оба эти средства — LXC и Docker — берут свое начало в Linux. Более широкая распространенность контейнеров Docker в сравнении с другими соперничающими с ними контейнерами (в том числе LXC) объясняется их способностью просто и быстро загружать свой образ в операционную систему узла. Контейнеры Docker хранятся в виде образов в облаке, из которого пользователи могут очень просто вызывать их на выполнение по мере необходимости.

Docker состоит из следующих основных компонентов.

- **Образ Docker.** Образы Docker являются доступными только для чтения шаблонами, из которых получаются экземпляры контейнеров Docker.
- **Клиент Docker.** Такой клиент выдает запрос на создание нового контейнера из применяемого образа. Клиент Docker может находиться на той же самой платформе, где находятся хост и машина Docker.
- **Узел Docker.** Платформа с собственной операционной системой хоста, на которой выполняются контейнерные приложения.
- **Механизм Docker.** Легковесный пакет времени выполнения, предназначенный для построения и выполнения контейнеров Docker в системе-хозяине.
- **Машина Docker.** Такая машина может выполняться в отдельной системе из узлов Docker, используя установленные механизмы Docker. Машина Docker инсталлирует механизм Docker в узел и конфигурирует клиент Docker для взаимодействия с механизмом Docker. Кроме того, машина Docker может быть использована локально, чтобы установить образ Docker в том же самом узле в качестве машины Docker.

- **Реестр Docker.** В реестре Docker хранятся образы Docker. Как только образ Docker будет сформирован, его можно ввести в открытый реестр (например, в концентраторе Docker) или же в закрытый реестр, защищенный брандмауэром. В таком реестре можно также искать существующие образы и затем извлекать их оттуда в узел.
- **Концентратор Docker.** Это платформа для сотрудничества и хранилище образов контейнеров Docker. Одни пользователи могут воспользоваться образами, сохраненными в этом концентраторе другими пользователями, а также ввести в него собственные образы.

## 14.4. ВОПРОСЫ ВИРТУАЛИЗАЦИИ НА УРОВНЕ ПРОЦЕССОРОВ

В виртуальной среде применяются две основные стратегии для предоставления ресурсов процессора. Первая из них состоит в программной эмуляции микросхемы и предоставлении доступа к данному ресурсу. Характерными примерами такой стратегии служат эмуляторы QEMU и Android Emulator из комплекта инструментальных средств разработки Android SDK. Эти эмуляторы выгодно отличаются простотой переноса, поскольку не зависят от конкретной платформы, хотя неэффективны с точки зрения производительности из-за того, что в процессе эмуляции интенсивно потребляются ресурсы. Вторая стратегия состоит в том, чтобы вообще не виртуализовать процессоры, а предоставлять кванты времени обработки на физических процессорах в хосте виртуализации отдельным виртуальным процессорам тех виртуальных машин, которые размещаются на физическом сервере. Именно таким образом большинство гипервизоров виртуализации предоставляют ресурсы процессора своим гостям. Когда операционная система в виртуальной машине передает команды процессору, гипервизор перехватывает запрос. Затем он планирует время обработки на физических процессорах узла, отправляет запрос на выполнение и возвращает результаты операционной системе виртуальной машины. Подобным образом обеспечивается наиболее эффективное использование доступных ресурсов процессора на физическом сервере. Дело, однако, усложняется тем, что гипервизору приходится действовать как регулировщику уличного движения, когда за доступ к процессору соперничают несколько виртуальных машин. В этом случае гипервизор планирует время процессора по запросу от каждой виртуальной машины, а также направляет запросы и данные как на виртуальные машины, так и от них.

Помимо памяти, одним из самых важных количественных показателей при масштабировании сервера служит количество процессоров на сервере. Это особенно справедливо и в какой-то степени более важно в виртуальной, чем физической среде. На физическом сервере, как правило, приложение имеет исключительное право пользоваться всеми вычислительными ресурсами, сконфигурированными в системе. Так, на сервере с четырьмя четырехъядерными процессорами в общем может быть использовано шестнадцать ядер процессоров. Но, как правило, требования приложения к ресурсам процессора намного скромнее. Дело в том, что физический сервер масштабирован с учетом возможного в будущем состояния данного приложения, включая его расширение в ближайшие три-пять лет, а также всплески производительности до некоторой высокой степени. Но в действительности вычислительная мощность процессоров на большинстве серверов в

значительной степени недоиспользуется, что служит сильной побудительной причиной для консолидации через виртуализацию, как пояснялось ранее в этой главе.

Когда приложения переносятся в виртуальные среды, возникает один из следующих вопросов, заслуживающих отдельного обсуждения: сколько виртуальных процессоров должно быть выделено для их виртуальных машин? Так, если на физическом сервере, который эти приложения высвобождают, доступно шестнадцать ядер процессоров, то разработчикам приложений обычно ставится задача продублировать эти ресурсы в виртуальной среде независимо от того, как они будут фактически использоваться. Кроме игнорирования степени использования физического сервера, из виду зачастую упускают усовершенствованные возможности процессоров на новом сервере виртуализации. Так, если приложение было перенесено в самом начале срока службы или аренды сервера, этот срок составит от трех до пяти лет. Но даже в течение трех лет быстродействие процессоров возрастет по закону Мура в четыре раза в сравнении с их быстродействием на исходном физическом сервере. Чтобы помочь в выборе “правильной” конфигурации виртуальных машин, имеются инструментальные средства, способные сначала проконтролировать фактическое использование ресурсов (процессора, памяти, сети и ввода-вывода на запоминающие устройства) на физических серверах, а затем выдать рекомендации относительно оптимальной масштабируемости виртуальных машин. Если же такое инструментальное средство для оценивания возможности консолидации отсутствует или не может быть применено, то вместо него можно воспользоваться целым рядом эмпирических правил. Одно из основных правил, соблюдаемых при создании виртуальной машины, состоит в том, чтобы начать с одного виртуального процессора и произвести текущий контроль производительности приложения. Добавить дополнительные виртуальные процессоры в виртуальную машину нетрудно, поскольку для этого достаточно всего лишь скорректировать параметры настройки виртуальной машины. Современные операционные системы не требуется даже перезагружать, чтобы они могли распознать и использовать дополнительный виртуальный процессор. Еще одна хорошая практика состоит в том, чтобы не переусердствовать, распределяя количество виртуальных процессоров виртуальным машинам. В частности, для определенного количества виртуальных процессоров виртуальной машины должно выполняться планирование соответствующего количества физических процессоров. Так, если виртуальной машине выделено четыре виртуальных процессора, то гипервизор от имени виртуальной машины должен одновременно планировать работу четырех физических процессоров. На очень занятых машинах наличие слишком большого количества виртуальных процессоров может отрицательно сказаться на производительности приложения виртуальной машины, поскольку планирование одного физического процессора выполняется значительно быстрее. Но это совсем не означает, что нет таких приложений, которым требуется несколько виртуальных процессоров. Такие приложения, безусловно, имеются и должны быть соответственным образом сконфигурированы, хотя это и не относится к большинству приложений.

“Родные” операционные системы управляют оборудованием, действуя в качестве посредника между запросами из прикладного кода и оборудованием. По мере выдачи запросов данных или их обработки операционная система передает их подходящим драйверам устройств через физические контроллеры для обращения к запоминающим устройствам или устройствам ввода-вывода и в обратном направлении. Операционная система является центральным маршрутизатором информации и управляет доступом ко всем физическим ресурсам оборудования. Одной из ключевых функций операционной

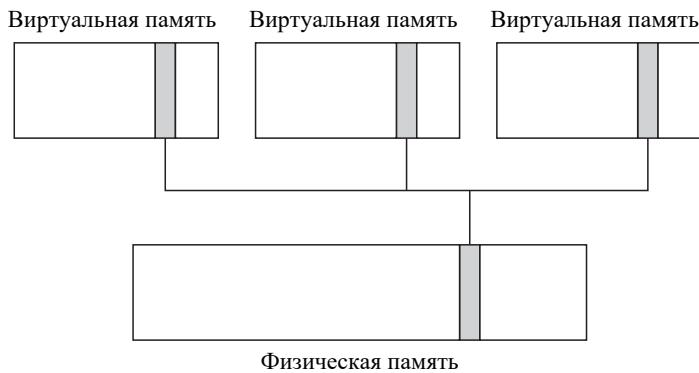
системы является оказание помощи в предотвращении нарушения нормальной работы приложений и самой операционной системы вследствие злонамеренных или случайных системных вызовов. Кольца защиты описывают уровни доступа или привилегии в компьютерной системе, и такая модель защиты выгодно используется во многих операционных системах и архитектурах процессоров. Наиболее доверительный уровень нередко называется нулевым кольцом защиты (Ring 0), где фактически действует ядро и допускается непосредственное взаимодействие с оборудованием. В пределах первого (Ring 1) и второго (Ring 2) колец защиты действуют драйверы устройств, тогда как пользовательские приложения выполняются в наименее доверительной области третьего кольца защиты (Ring 3). На практике первое и второе кольца защиты применяются нечасто, упрощая тем самым модель защиты до уровня доверительных и недоверительных пространств выполнения. Прикладной код не может непосредственно взаимодействовать с оборудованием, поскольку он выполняется в третьем кольце защиты, и поэтому операционная система должна выполнять прикладной код от его имени в нулевом кольце защиты. Это разделение предотвращает выполнение непrivилегированного кода, приводящее к таким действиям, как остановка системы или неразрешенный доступ к данным на диске или через сетевое соединение.

Гипервизоры выполняются в нулевом кольце защиты, контролируя доступ к оборудованию для тех виртуальных машин, которыми они управляют. Операционные системы в этих виртуальных машинах считают, что они выполняются в нулевом кольце защиты, и в какой-то степени так оно и есть, но лишь на виртуальном оборудовании, созданном как часть виртуальной машины. Так, если потребуется остановить систему, гостевая операционная система запросит команду остановки в нулевом кольце защиты. Гипервизор перехватит такой запрос, иначе физический сервер остановится, нарушив нормальную работу не только гипервизора, но и любых других виртуальных машин. Вместо этого гипервизор ответит гостевой операционной системе, что запрос остановки будет обработан, как требуется, что позволит гостевой операционной системе нормально завершить все необходимые для окончания работы программного обеспечения процессы.

## 14.5. УПРАВЛЕНИЕ ПАМЯТЬЮ

Подобно количеству виртуальных процессоров, одним из важных проектных решений является выбор объема оперативной памяти, выделяемой виртуальной машине. Фактически именно ресурсы памяти обычно оказываются первым узким местом, которого по мере своего роста достигают виртуальные инфраструктуры. Подобно виртуализации процессоров, использование оперативной памяти в виртуальных средах в большей степени связано с управлением физическим ресурсом, чем с созданием виртуального объекта. Как и в случае физического сервера, для эффективного функционирования виртуальную машину следует сконфигурировать с достаточным объемом оперативной памяти, выделив свободное место для операционной системы и приложений. И в этом случае виртуальная машина конфигурируется с меньшими ресурсами, чем обладает физический узел. Простым тому примером может служить физический сервер с объемом памяти 8 Гбайт. Так, если выделить виртуальной машине 1 Гбайт оперативной памяти, то ей будет доступен лишь этот объем, несмотря на то, что физический сервер, на котором она размещается, обладает большим объемом оперативной памяти. Когда виртуальная машина пользуется ресурсами памяти, гипервизор управляет запросами этой памяти че-

результатом преобразований, и поэтому гостевая операционная система (виртуальная машина) обращается к пространству памяти по предполагаемым адресам. И хотя это неплохой первый шаг, трудности все же остаются. Как и в случае процессора, владельцы приложений запрашивают выделение памяти, зеркально отображающее физические инфраструктуры, из которых они перенесены, независимо от того, гарантируется ли выделение запрашиваемого объема памяти. Это приводит к избыточному резервированию свободного места под виртуальные машины и напрасному расходованию ресурсов памяти. Так, если вернуться к примеру сервера с объемом памяти 8 Гбайт, то на нем можно разместить лишь семь виртуальных машин с объемом памяти 1 Гбайт, поскольку оставшийся 1 Гбайт требуется для самого гипервизора. Помимо “правильной” масштабируемости виртуальных машин, определяемой на основании их фактических показателей производительности, в гипервизоры встроены другие средства, помогающие оптимизировать использование оперативной памяти. Одним из таких средств является **совместное использование страниц** (page sharing) (рис. 14.7). Подобно устраниению дубликатов данных, методика разделения страниц служит для сокращения количества применяемых блоков хранения данных. Когда инстанцируется экземпляр виртуальной машины, страницы операционной системы и приложений загружаются в оперативную память. Если же несколько виртуальных машин загружают одну и ту же версию операционной системы или выполняют одни и те же приложения, то многие блоки памяти дублируются. Гипервизор уже управляет преобразованиями виртуальной памяти в физическую и поэтому в состоянии определить, загружена ли страница в оперативную память. Вместо того чтобы загружать дубликат страницы в физическую память, гипервизор предоставляет ссылку на совместно используемую страницу в таблице преобразования из виртуальной машины. В тех узлах, где гости выполняют одну и ту же операционную систему и одни и те же приложения, можно освободить от 10 до 40% физической памяти. При уровне освобождения в 25% физической памяти на сервере с объемом памяти 8 Гбайт на нем можно дополнительно разместить две виртуальные машины с объемом памяти по 1 Гбайт.



**Рис. 14.7.** Совместное использование страниц

Совместным использованием страниц управляет гипервизор, и поэтому операционным системам виртуальных машин ничего неизвестно о том, что происходит в физической системе. Еще одна стратегия эффективного использования оперативной памяти

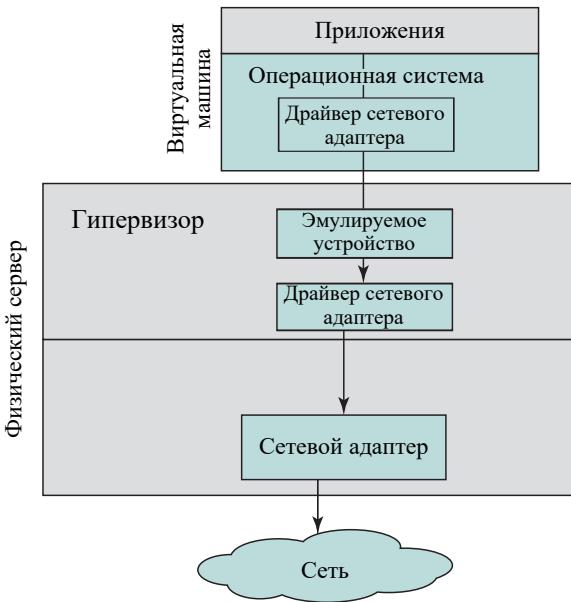
сродни “тонкому” резервированию в управлении памятью. Это дает администратору возможность выделять пользователю больше памяти, чем фактически присутствует в системе. Побудительной причиной для этого служит стремление довести предоставляемый объем памяти до наивысшего предела, который зачастую вообще не достигается. То же самое можно сделать и с памятью виртуальной машины. Для виртуальной машины обычно выделяется 1 Гбайт памяти, но ведь именно такой объем памяти доступен операционной системе виртуальной машины. Часть этой выделенной памяти гипервизор может использовать для другой виртуальной машины, освободив страницы, которые больше не используются. Процесс освобождения памяти выполняется через ее **накачку** (*ballooning*). Гипервизор активизирует драйвер накачки, который (буквально) надувается, вынуждая гостевую операционную систему сбросить страницы памяти на диск. Как только страницы сброшены, драйвер накачки сдувается, а гипервизор может воспользоваться освободившейся физической памятью для других виртуальных машин. Этот процесс происходит во время конфликтов в памяти. Так, если бы виртуальные машины использовали в среднем половину выделенной им памяти объемом 1 Гбайт, то для размещения девяти виртуальных машин на физическом сервере потребовалось бы 4,5 Гбайт памяти, а остальную ее часть можно было бы употребить на общий пул, управляемый гипервизором, а также на некоторые его накладные расходы. И хотя на физическом сервере удается разместить еще три виртуальные машины с объемом памяти 1 Гбайт, при этом все еще остается общий резерв памяти. Такая возможность выделить больше оперативной памяти, чем существует физически, называется **перегрузкой памяти** (*memory overcommit*). Она нередко используется в виртуализованных средах, чтобы увеличить в 1,2–1,5 раза объем выделяемой памяти, а в крайних случаях — и гораздо больше.

Имеются и другие методики управления памятью, позволяющие лучше пользоваться имеющимися ресурсами. Но в любом случае операционным системам в виртуальных машинах доступен именно тот объем памяти, который для них выделен. Гипервизор управляет доступом к физической памяти, чтобы обеспечить, а не гарантировать своевременное обслуживание всех запросов, не оказывая никакого влияния на виртуальные машины. В тех же случаях, когда требуется больше оперативной памяти, чем доступно, гипервизор вынужден прибегать к сбросу страниц памяти на диск. В кластерных средах, состоящих из многих узлов, виртуальные машины могут быть автоматически перенесены в другие узлы, как только станет не хватать определенных ресурсов.

## 14.6. УПРАВЛЕНИЕ ВВОДОМ-ВЫВОДОМ

Производительность приложений зачастую связана непосредственно с пропускной способностью, выделяемой сервером. Будь то затрудненный доступ к запоминающему устройству или ограниченный сетевой трафик, приложение в любом случае воспринимается как работающее недостаточно эффективно. Таким образом, виртуализация ввода-вывода во время виртуализации рабочих нагрузок приобретает решающее значение. Архитектура управления вводом-выводом в виртуальной среде довольно проста, как показано на рис. 14.8.

Сначала операционная система в виртуальной машине вызывает драйвер устройства, как если бы это было на физическом сервере. Затем драйвер устройства подключается к самому устройству, хотя на виртуальном сервере это эмулируемое устройство, размещаемое на нем и управляемое гипервизором.



**Рис. 14.8.** Ввод-вывод в виртуальной среде

Обычно эмулируются конкретные устройства, например плата сетевого интерфейса Intel e1000 или простые типовые контроллеры периферийных устройств SVGA или IDE. Это виртуальное устройство подключается в гипервизоре к стеку ввода-вывода, сообщающемуся с драйвером устройства, который назначается для физического устройства на сервере узла, преобразуя гостевые адреса ввода-вывода в аналогичные адреса хозяина. Гипервизор осуществляет управление и текущий контроль запросов от драйвера устройства виртуальной машины через стек ввода-вывода к физическому устройству и в обратном направлении, направляя вызовы ввода-вывода нужным устройствам на соответствующих виртуальных машинах. Несмотря на некоторые различия в архитектуре у разных поставщиков, основная модель управления вводом-выводом в виртуальной среде одна и та же.

Виртуализация путем ввода-вывода под рабочей нагрузкой дает немало преимуществ. Она обеспечивает аппаратную независимость, абстрагируя драйверы отдельных поставщиков до более обобщенных версий, выполняемых в гипервизоре. Так, виртуальную машину, работающую на сервере IBM в качестве хоста, можно динамически перенести на блейд-сервер HP, не особенно беспокоясь об аппаратной несовместимости или несогласованности версий. Такая абстракция допускает динамический перенос, являющийся одной из самых сильных сторон виртуализации. Совместное использование совокупных ресурсов (например, сетевых путей) также является следствием подобной абстракции. В более зрелых решениях имеется возможность точнее контролировать типы сетевого трафика и пропускную способность, доступную отдельным виртуальным машинам или их группам, чтобы гарантировать надлежащую производительность в общей среде, а следовательно, и выбранный уровень качества обслуживания. Помимо этих преимуществ, имеются и другие свойства, повышающие безопасность и доступность. А компромисс состоит в том, что гипервизор управляет всем сетевым трафиком, для чего он, собствен-

но, и предназначен, хотя это и требует затрат вычислительных ресурсов процессора. На ранней стадии развития виртуализации это был трудноразрешимый вопрос, который мог стать ограничивающим фактором, но по мере повышения быстродействия процессоров и усовершенствования гипервизоров данный вопрос был практически снят.

Более быстродействующий процессор позволяет гипервизору быстрее выполнять свои функции управления вводом-выводом, а также ускоряет обработку гостевым процессором. Явные аппаратные изменения в поддержке виртуализации также способствуют повышению производительности. Так, компания Intel предлагает технологию I/OAT (I/O Acceleration Technology — Технология ускорения ввода-вывода) как физическую подсистему, осуществляющую копирование памяти через механизм прямого доступа к памяти (direct memory access — DMA) от основного процессора в данную специализированную часть материнской платы. И хотя удаленный DMA предназначен для повышения производительности сети, он позволяет также увеличить скорость динамического переноса. Перенос части нагрузки из процессора на интеллектуальные устройства открывает еще один путь к повышению производительности. Интеллектуальные платы сетевого интерфейса поддерживают целый ряд технологий в данной области. В частности, механизм TOE (TCP Offload Engine — Механизм разгрузки по сетевому протоколу TCP) полностью переносит обработку по сетевому протоколу TCP/IP из процессора сервера на сетевой адаптер. К другим вариантам в данной области относятся технология LRO (Large Receive Offload — Укрупненная разгрузка приема данных), объединяющая входящие пакеты в комплекты для более эффективной их обработки, а также противоположная ей технология LSO (Large Segment Offload — Укрупненная разгрузка сегментов данных), позволяющая гипервизору объединять несколько исходящих по сетевому протоколу TCP/IP пакетов и сегментировать их по отдельным пакетам на аппаратном уровне сетевого адаптера.

Помимо описанной выше модели, некоторым приложениям или пользователям потребуется специально выделенный путь. В таком случае имеют две возможности: обойти стек ввода-вывода и надзор гипервизора, а также подключить драйвер устройства из виртуальной машины непосредственно к физическому устройству в узле виртуализации. Это дает преимущество в том, что в вашем распоряжении оказывается специально выделенный ресурс без каких-либо накладных расходов, обеспечивающий максимальную пропускную способность. Помимо большей пропускной способности, благодаря минимальному участию гипервизора снижается нагрузка на процессор сервера хоста. А недостаток непосредственного подключения к устройству ввода-вывода заключается в том, что виртуальная машина привязывается к тому физическому серверу, на котором она работает. В отсутствие абстракции устройств ввода-вывода нелегко осуществить динамический перенос, что может потенциально сделать некоторые функциональные возможности менее доступными. В частности, такие функциональные возможности гипервизора, как перегрузка памяти или управление вводом-выводом, оказываются вообще недоступными, а следовательно, не задействованные полностью ресурсы могут расходоваться напрасно, ставя под сомнение потребность в виртуализации. И хотя модель специально выделяемых устройств ввода-вывода обеспечивает большую производительность, в настоящее время она применяется редко, поскольку центры обработки данных отдают предпочтение гибкости, которую обеспечивает виртуализованный ввод-вывод.

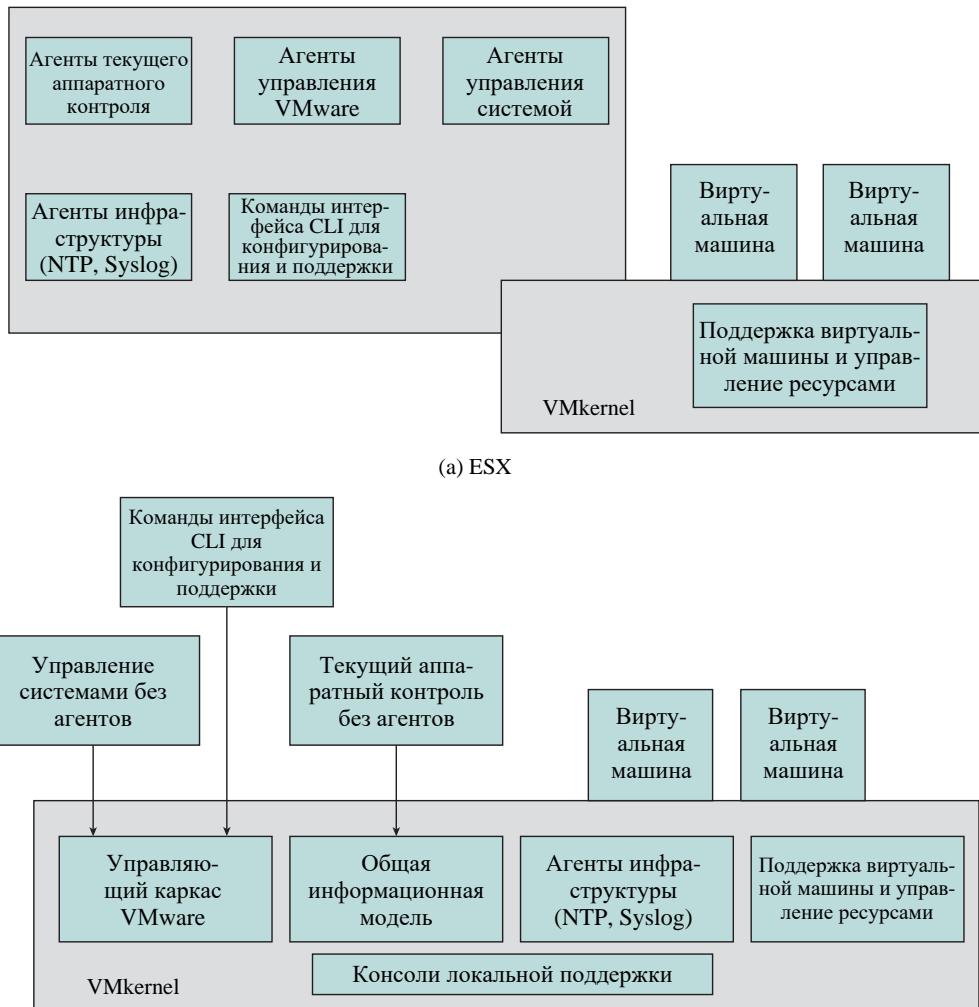
## 14.7. VMWARE ESXi

ESXi — коммерчески доступный программный продукт компании VMware, представляющий пользователям гипервизор первого типа, действующий на аппаратном уровне для размещения виртуальных машин на серверах данной компании. Свои первые решения на основе процессоров x86 компания VMware разработала в конце 1990-х годов и была среди первых, выпустивших на рынок соответствующий коммерческий продукт. Благодаря этому первому по срокам продукту на рынке и постоянным нововведениям компания VMware прочно завоевала немалую долю рынка, но важнее то, что она заняла ведущее место с точки зрения широты предоставляемых функциональных средств и зрелости решений. Не вдаваясь в подробное описание причин роста рынка виртуализации и изменений в решениях, предлагаемых компанией VMware, необходимо все же подчеркнуть некоторые основополагающие отличия архитектуры ESXi от других имеющихся решений.

Ядро виртуализации (VMkernel) образует сердцевину гипервизора и выполняет все функции виртуализации. В первых выпусках ESX (рис. 14.9, а) гипервизор развертывался вместе с установкой Linux, которая служила в качестве уровня управления. Некоторые функции управления наподобие протоколирования, служб именования, а зачастую и сторонних агентов для резервного копирования или текущего аппаратного контроля устанавливались на этой служебной консоли. Эта консоль стала также отличным местом для выполнения администраторами других сценариев и программ. Но ей были присущи два недостатка. Первый из них состоял в том, что она была намного крупнее гипервизора. В частности, при типичной установке требовалось около 32 Мбайт для гипервизора и около 900 Мбайт для служебной консоли. Второй недостаток заключался в том, что служебная консоль на основе Linux была вполне понятным интерфейсом и системой, а следовательно, была уязвима к атакам со стороны зловредных программ и злоумышленников. И тогда компания VMware переделала архитектуру ESX, чтобы устанавливать этот программный продукт и управлять им без служебной консоли.

Эта новая архитектура под названием “ESXi”, где “i” обозначает интегрированность, обладает всеми управляющими службами, входящими в ядро VMkernel (см. рис. 14.9, б). Благодаря этому весь пакет становится более компактным и безопасным, чем прежде. Текущие его версии занимают порядка 100 Мбайт. Столь малый размер пакета позволяет поставщикам серверов поставлять оборудование с версией ESXi, уже доступной во флеш-памяти на сервере. Все функции управления конфигурацией, текущего контроля и поддержки сценариев теперь доступны через соответствующие утилиты интерфейса командной строки. Сторонние агенты выполняются в ядре VMkernel после того, как пройдут надлежащие процедуры сертификации и цифрового подписания. Это, например, дает поставщику сервера, предоставляющему средства аппаратного текущего контроля, возможность включить в ядро VMkernel агент, который может бесперебойно возвращать количественные показатели работы оборудования, в том числе внутреннюю температуру или состояния компонентов, средствам управления VMware или другим инструментальным средствам.

Виртуальные машины размещаются через инфраструктурные службы в ядре VMkernel. А когда виртуальные машины запрашивают ресурсы, гипервизор выполняет их запросы, подбирая подходящие драйверы устройств. Как пояснялось ранее, гипервизор согласовывает все транзакции между многими виртуальными машинами и аппаратными ресурсами на физическом уровне.



**Рис. 14.9. Архитектуры ESX и ESXi**

Несмотря на всю упрощенность рассмотренных до сих пор примеров, VMware ESXi предлагает передовые и логически развитые функциональные возможности для достижения необходимого уровня доступности, масштабируемости, безопасности, управляемости и производительности. В каждом выпуске ESXi внедряются дополнительные возможности, а также совершенствуются уже имеющиеся возможности данной платформы. Ниже приведены некоторые тому примеры.

- **Функция сохранения VMotion.** Разрешает перемещение файлов данных, составляющих виртуальную машину, “на лету”, в процессе работы последней.
- **Отказоустойчивость.** Создает синхронную копию виртуальной машины в другом узле. При сбое исходного хоста соединения виртуальной машины переносятся в ее копию, не прерывая работу пользователей или применяемых ими приложений.

Отказоустойчивость отличается от высокой доступности (High Availability), которая потребовала бы в данном случае перезапуска виртуальной машины на другом сервере.

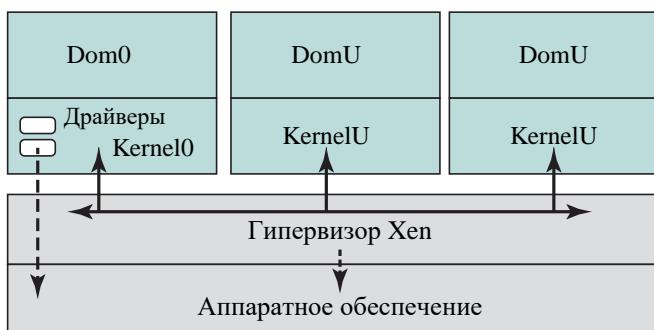
- **Диспетчер восстановления узла.** Применяет различные технологии тиражирования для копирования выбранных виртуальных машин на вторичный узел в случае аварии в центре обработки данных. Вторичный узел может быть введен в действие в считанные минуты, а виртуальные машины — автоматически включены в избранном и многоуровневом режиме, чтобы гарантировать плавный и аккуратный переход.
- **Управление сохранением данных и сетевым вводом-выводом.** Дает администратору возможность очень точно выделять пропускную способность в виртуальной сети. Эти правила активизируются при возникновении конфликта в сети и способны гарантировать требующийся приоритет и пропускную способность для нормального функционирования отдельных виртуальных машин, их групп, составляющих конкретное приложение, классов данных или трафика сохранения данных.
- **Планировщик распределенных ресурсов (Distributed Resource Scheduler — DRS).** Интеллектуально распределяет виртуальные машины по узлам и способен автоматически выравнивать рабочие нагрузки через VMotion, исходя из правил предметной области и рационального использования ресурсов. Распределенное управление электропитанием (Distributed Power Management — DPM) как часть функций этого планировщика позволяет включать и отключать физические узлы по мере надобности в них. Планировщик распределенных запоминающих устройств способен переносить файлы виртуальных машин, исходя из емкости запоминающего устройства и задержки ввода-вывода. В этом случае он действует исходя из правил предметной области и использования имеющихся ресурсов.

Имеется совсем немного функциональных возможностей для расширения предоставляемого компанией VMware решения ESXi за пределы самого гипервизора, способного поддерживать виртуальные машины, до платформы для нового центра обработки данных и основания для облачных вычислений.

## 14.8. Варианты Hyper-V и Xen от корпорации Microsoft

В начале 2000-х годов работы, проведенные на базе Кембриджского университета, привели к разработке гипервизора Xen с открытым исходным кодом. Со временем по мере роста потребности в виртуализации появилось немало вариантов в виде ответвлений от основного гипервизора Xen. Ныне, помимо гипервизора с открытым исходным кодом, имеется целый ряд коммерчески доступных гипервизоров, разработанных на основе Xen и предлагаемых Citrix, Oracle и другими компаниями. Для нормальной работы с гипервизором Xen, обладающим совсем иной архитектурой, чем у VMware, требуется специально выделенная операционная система или домен, подобный служебной консоли в VMware (рис. 14.10). Этот исходный домен называется нулевым (Dom0), запускает стек инструментальных средств Xen и как привилегированная область имеет прямой доступ к оборудованию. В состав многих версий Linux входит гипервизор Xen, спо-

собный создать виртуальную среду. К числу таких версий относятся CentOS, Debian, Fedora, Ubuntu, OracleVM, Red Hat (RHEL), SUSE и XenServer. Компании пользуются решениями виртуализации, основанными на Xen, из-за малой их стоимости (а то и полного ее отсутствия) или же исходя из собственного опыта работы с Linux.



**Рис. 14.10.** Архитектура Xen

Гости в Xen являются непривилегированными, или пользовательскими доменами (DomU). Нулевой домен предоставляет гостям доступ к ресурсам сети и памяти через серверные драйверы, сообщающиеся с клиентскими драйверами в пользовательском домене. Если только не сконфигурированы сквозные драйверы (обычно USB), весь сетевой ввод-вывод и ввод-вывод на запоминающие устройства выполняется через нулевой домен. А поскольку сам нулевой домен является экземпляром Linux, то если с ним происходит нечто неожиданное, это оказывает влияние на все виртуальные машины. Стандартное сопровождение операционной системы с использованием “заплат” может также оказывать потенциальное влияние на общую доступность.

Как и большинство других предлагаемых программных продуктов с открытым исходным кодом, Xen не обладает расширенными возможностями, предоставляемыми в VMware ESXi, хотя в каждом выпуске Xen появляются дополнительные функциональные возможности и совершенствуются уже существующие.

В распоряжении корпорации Microsoft имелся целый ряд технологий виртуализации, в том числе Virtual Server — гипервизор второго типа, приобретенный в 2005 году и по-прежнему доступный бесплатно. А гипервизор первого типа Hyper-V был впервые выпущен в 2008 году как часть выпуска операционной системы Windows Server 2008 корпорации Microsoft. Как и в архитектуре Xen, в Hyper-V имеется родительский раздел, который служит в качестве административного дополнения к гипервизору первого типа (рис. 14.11), а гостевые виртуальные машины представлены как дочерние разделы. Помимо таких основных функций, как управление гипервизором, гостевыми разделами и драйверами устройств, в родительском разделе выполняется операционная система Windows Server. И подобно драйверам в Xen, в родительском разделе из Hyper-V применяется поставщик служб виртуализации (Virtualization Service Provider — VSP), предоставляющий порожденным разделам службы устройств. Порожденные разделы сообщаются с поставщиками служб виртуализации, используя для своих потребностей клиент службы виртуализации (Virtualization Service Client — VSC).

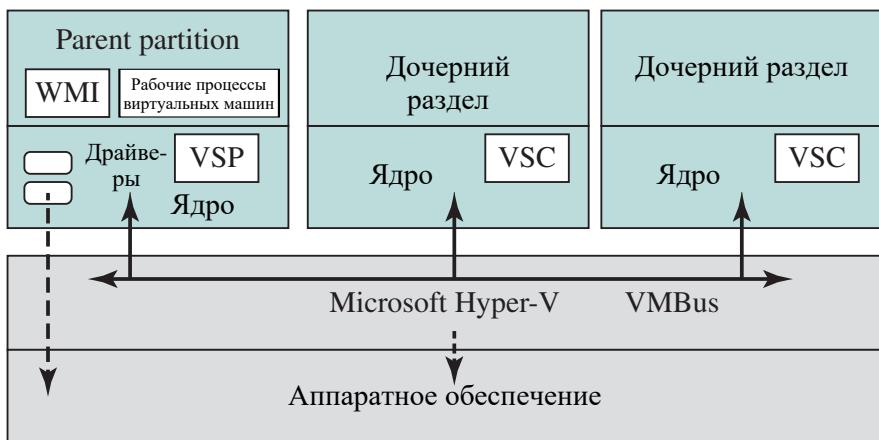


Рис. 14.11. Архитектура Hyper-V

Те же самые трудности, связанные с доступностью, что и у Xen, присущи и Microsoft Hyper-V вследствие потребностей операционной системы в родительском разделе, дополнительной копии Windows на сервере — из-за конфликта ресурсов, а также единого канала ввода-вывода. С точки зрения функциональных возможностей гипервизор Hyper-V весьма надежен, хотя применяется не так широко, как ESXi, поскольку все еще является относительно новым программным продуктом на рынке. Но со временем, когда у него появятся новые функциональные возможности, темпы внедрения Hyper-V, вероятно, возрастут.

## 14.9. JAVA VM

Несмотря на то что в названии программного продукта Java Virtual Machine (JVM) имеется термин *виртуальная машина*, его реализация и применение отличаются от рассмотренных ранее моделей виртуализации. Гипервизоры поддерживают одну или несколько виртуальных машин в узле. Данные же виртуальные машины являются самостоятельными приложениями, каждое из которых поддерживает операционную систему и приложения и имеет доступ к ряду аппаратных устройств, предоставляющих ресурсы для вычисления, сохранения и ввода-вывода. Цель Java Virtual Machine — предоставить динамическое пространство для выполнения множества кода Java в любой операционной системе, размещаемой на любой аппаратной платформе, не внося изменения в код, чтобы приспособить его к разным операционным системам или оборудованию. Обе модели предназначены в качестве независимых платформ, образуемых через определенную степень абстракции.

JVM описывается как абстрактная вычислительная машина, состоящая из **набора команд**, **регистра счетчика команд PC**, **стека** для хранения переменных и результатов, **кучи** для хранения данных и сборки мусора в динамическом режиме, а также области **методов** для хранения кода и констант. JVM может поддерживать многие потоки выполнения, и каждый из них со своими областями регистров и стека, хотя области кучи и методов совместно используются всеми потоками выполнения. Когда инстанцируется экземпляр JVM, сначала запускается среда выполнения, выделяются структуры памяти,

которые заполняются выбранным методом (кодом) и переменными, а затем начинается выполнение программы. Код, выполняемый в JVM, интерпретируется в реальном времени из языка Java в соответствующий бинарный код. Если этот код корректен и соответствует предполагаемым стандартам, то начинается его обработка. Если код некорректен, процесс завершается неудачно, генерируется исключение и состояние ошибки возвращается как самой виртуальной машине JVM, так и пользователю.

Язык Java и виртуальные машины JVM применяются в самых разных областях, включая веб-приложения, мобильные устройства, а также интеллектуальные устройства — от телевизионных и игровых приставок до проигрывателей формата Blu-ray и прочей аппаратуры, в которой используются интеллектуальные платы. Обещанный в Java принцип “написано однажды — работает везде” обеспечивает гибкую и простую модель развертывания, позволяющую развертывать приложения независимо от платформы, на которой они выполняются.

## 14.10. АРХИТЕКТУРА ВИРТУАЛЬНОЙ МАШИНЫ LINUX VSERVER

Linux VServer представляет собой быстродействующий, виртуализованный, контейнерный подход с открытым исходным кодом к реализации виртуальных машин на сервере Linux [156, 240]. Для этой цели задействована единственная копия ядра Linux. VServer состоит из относительно скромной модификации ядра, а также небольшого набора инструментальных средств ОС пользовательского пространства.<sup>1</sup> Ядро Linux в VServer поддерживает целый ряд отдельных *виртуальных серверов* и управляет всеми системными ресурсами и задачами, включая планирование процессов, выделение оперативной памяти, дискового пространства и времени процессора.

### Архитектура

Каждый виртуальный сервер изолируется от остальных серверов с помощью функциональных возможностей ядра Linux. Тем самым обеспечивается безопасность и упрощается установка многих виртуальных машин на одной платформе. Такая изоляция включает в себя четыре элемента: команду `chroot`, утилиты `chcontext` и `chbind`, а также функциональные возможности.

Команда `chroot` относится к набору команд UNIX или Linux и служит для замены выбиравшего по умолчанию назначения корневого каталога (/) каким-нибудь другим назначением в течение срока действия текущего процесса. Эта команда доступна для выполнения только привилегированными пользователями и применяется для того, чтобы предоставить процессу (обычно — сетевому серверу, подобному FTP- или HTTP-сервера) доступ к ограниченной части файловой системы. Тем самым она обеспечивает изоляцию файловой системы.

---

<sup>1</sup> Термин *пользовательское пространство* (userLand) означает все прикладное программное обеспечение, которое выполняется в пространстве пользователя, но не ядра, а термин *пользовательское пространство операционной системы* — различные программы и библиотеки, применяемые в операционной системе для взаимодействия с ядром, в том числе программное обеспечение, выполняющее ввод-вывод, манипулирующее объектами файловой системы и т.д.

Все команды, выполняемые виртуальным сервером, могут воздействовать на файлы, путь к которым начинается с корневого каталога, определенного для данного сервера.

Утилита `chcontext` из Linux служит для выделения нового контекста безопасности и выполнения команд в этом контексте. Обычный или *размещаемый* на сервере контекст безопасности является нулевым. Данный контекст обладает теми же самыми привилегиями, что и привилегированный пользователь с идентификатором UID 0. Так, в данном контексте можно выявлять и удалять задачи из других контекстов. Контекст с номером 1 служит для просмотра других контекстов, хотя и не может воздействовать на них, а все остальные контексты обеспечивают полную изоляцию. В частности, процессы из одного контекста могут выявлять процессы из другого контекста, но не взаимодействовать с ними. Тем самым обеспечивается возможность выполнять сходные контексты на одном и том же компьютере, но безо всякого взаимодействия на прикладном уровне. Таким образом, у каждого виртуального сервера имеется свой контекст выполнения, обеспечивающий изоляцию процессов.

Утилита `chbind` служит для выполнения команды и блокировки результирующего процесса и порожденных им процессов по конкретному IP-адресу. После вызова утилиты `chbind` всем пакетам, посыпаемым данным виртуальным сервером через сетевой интерфейс системы, присваивается IP-адрес отправки, указанный в качестве аргумента утилиты `chbind`. Именно этот системный вызов и обеспечивает *изоляцию сети*. На каждом виртуальном сервере используется отдельный и вполне определенный IP-адрес. Входящий трафик, предназначенный для одного виртуального сервера, не может быть доступным для других виртуальных серверов.

И наконец, каждому виртуальному серверу назначается ряд **возможностей** (*capabilities*). Концепция возможностей применяется в Linux с целью обозначить разделение привилегий, доступных привилегированному пользователю, например прав на чтение файлов или отслеживание процессов, принадлежащих другому пользователю. Таким образом, каждому виртуальному серверу может быть присвоено ограниченное подмножество привилегий, доступных привилегированному пользователю. И тем самым обеспечивается *изоляция корневых привилегий*. VServer позволяет также наложить ограничения на ресурсы, в том числе на объем виртуальной памяти, которой может воспользоваться процесс.

Общая архитектура Linux VServer приведена на рис. 14.12. В VServer предоставляется образ общей виртуализованной операционной системы, состоящий из корневой файловой системы и ряда совместно используемых системных библиотек и служб ядра. Каждую виртуальную машину можно загрузить, остановить и перегрузить независимо от других виртуальных машин.

На рис. 14.12 показаны три группы программного обеспечения, выполняющегося в компьютерной системе. Так, **платформа размещения**, иначе называемая **хостинг-платформой**, включает в себя совместно используемый образ операционной системы и виртуальную машину привилегированного узла, функция которой состоит в текущем контроле и управлении другими виртуальными машинами. На **виртуальной платформе** создаются виртуальные машины и формируется представление системы, доступное **приложениям**, выполняющимся на отдельных виртуальных машинах.

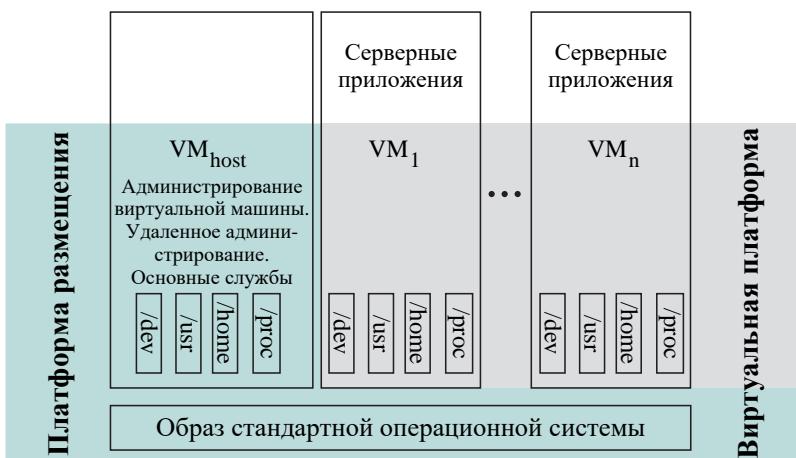


Рис. 14.12. Архитектура Linux VServer

## Планирование процессов

Средства виртуальной машины в Linux VServer предоставляют возможность управления использованием виртуальной машиной процессорного времени. В частности, VServer накладывает фильтр токенов (token bucket filter — TBF) на стандартный план работы Linux. Назначение фильтра TBF — определить, какое процессорное время (в однопроцессорной, многопроцессорной или многоядерной системе) выделено каждой виртуальной машине. Если для глобального планирования процессов среди всех виртуальных машин используется только базовый планировщик Linux, то ресурсоемкие процессы на одной виртуальной машине вытесняют процессы других виртуальных машин.

Принцип действия фильтра TBF наглядно показан на рис. 14.13.

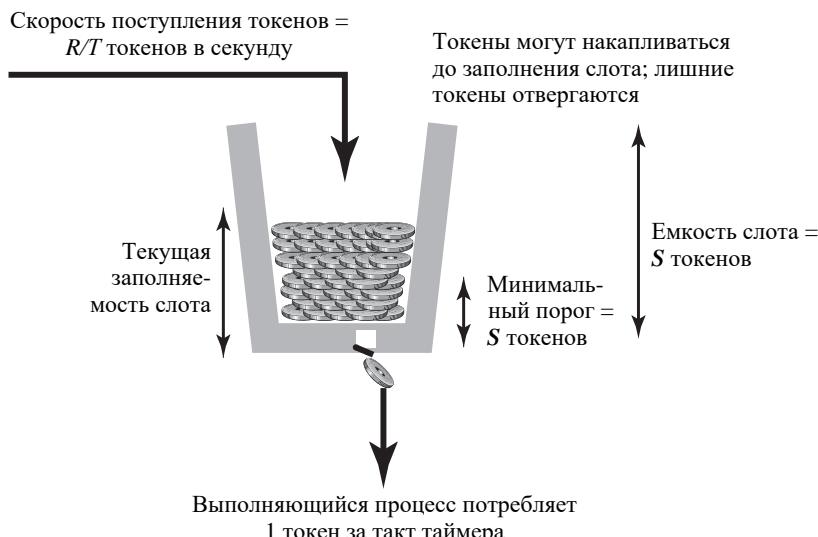


Рис. 14.13. Схематическое представление маркерного слота в Linux VServer

Для каждой виртуальной машины определяется слот емкостью  $S$  токенов. Эти токены вводятся в слот со скоростью  $R$  маркеров в течение промежутка времени длительностью  $T$ . Если слот заполнен, то дополнительные входящие токены просто отвергаются. Когда процесс выполняется на данной виртуальной машине, он потребляет по одному токену на каждый такт системного таймера. Если слот опорожняется, процесс приостанавливается и не может быть возобновлен до тех пор, пока слот не заполнится минимальным пороговым количеством маркеров  $M$ . В этот момент процесс планируется заново. Важное следствие из принципа действия фильтра TBF состоит в том, что виртуальная машина может накапливать токены в период бездействия, а в дальнейшем по мере необходимости использовать их пакетом.

Корректируя величины  $R$  и  $T$ , можно регулировать долю емкости маркерного слота в процентах, которая может быть затребована виртуальной машиной. Выделяемую емкость маркерного ведра для одного процессора можно определить следующим образом:

$$\frac{R}{T} = \text{Доля выделяемых ресурсов процессора}$$

В этом уравнении определяется доля участия одного процесса в системе. Так, если в системе применяется четырехъядерный процессор и в среднем каждой виртуальной машине требуется выделить одно ядро процессора, то в таком случае можно задать величины  $R = 1$  и  $T = 4$ . А общее ограничение на систему определяется следующим образом: если имеется  $N$  виртуальных машин, то

$$\sum_{i=1}^N \frac{R_i}{T_i} \leq 1$$

Параметры  $S$  и  $M$  задаются таким образом, чтобы “штрафовать” виртуальные машины по истечении определенного пакетного времени. Для виртуальной машины могут быть настроены или выделены следующие параметры: по истечении пакетного времени  $B$  виртуальная машина оштрафовывается на время ожидания  $H$ . С учетом этих параметров можно вычислить значения параметров  $S$  и  $M$  следующим образом:

$$M = W \times H \times \frac{R}{T}$$

$$S = W \times B \times \left(1 - \frac{R}{T}\right)$$

Здесь  $W$  — скорость, с которой принимаются решения. Рассмотрим в качестве примера виртуальную машину с ограничением в половину используемого времени процессора и допустим, что, как только процессор будет использован в течение 30 секунд, наступает ожидание в течение 5 секунд, а планировщик работает с частотой 1000 Гц. Такое требование удовлетворяется при следующих значениях:  $M = 1000 \times 5 \times 0,5 = 2500$  токенов;  $S = 1000 \times 30 \times (1 - 0,5) = 15\,000$  токенов.

## 14.11. Резюме

Технология виртуализации позволяет выполнять многие операционные системы или сеансы работы в одной операционной системе на единственном ПК или сервере. По существу, в исходной операционной системе может поддерживаться целый ряд виртуальных машин, каждая из которых обладает характеристиками конкретной операционной системы, а в некоторых версиях виртуализации — характеристиками конкретной аппаратной платформы.

В распространенной технологии виртуальных машин употребляется монитор виртуальных машин, или гипервизор, который действует на более низком уровне, чем виртуальная машина, и поддерживает многие виртуальные машины. По наличию или отсутствию другой операционной системы между гипервизором и хостом различаются два типа гипервизоров. В частности, гипервизор первого типа выполняется непосредственно на оборудовании машины, а гипервизор второго типа работает поверх исходной операционной системы узла.

Совсем другой, упрощенный, подход к реализации среды виртуальной машины принят в Java VM. Цель Java VM — предоставить динамическое пространство для выполнения множества кода Java в любой операционной системе, размещаемой на любой аппаратной платформе, без необходимости вносить изменения в сам код для его приспособления к разным операционным системам или оборудованию.

## 14.12. Ключевые термины, контрольные вопросы и задачи

### Ключевые термины

Docker	Гипервизор второго типа	Монитор виртуальной машины
Java Virtual Machine (JVM)	Гипервизор первого типа	Накачка памяти
Аппаратная виртуализация	Гостевая операционная система	Операционная система хоста
Аппаратно поддерживаемая виртуализация	Группа управления ядром	Паравиртуализация
Виртуализация	Контейнер	Перегрузка памяти
Виртуальная машина	Контейнерная виртуализация	Совместное использование страниц
Виртуальное устройство	Микрослужба	Степень консолидации

### Контрольные вопросы

- 14.1. Кратко опишите виртуализацию первого и второго типов.
- 14.2. Кратко опишите контейнерную виртуализацию контейнеров.
- 14.3. Поясните концепцию накачки памяти.
- 14.4. Дайте краткое описание Java VM.

## Задачи

- 14.1. Такие методики, как перегрузка памяти и разделение страниц, позволяют выделить виртуальным машинам больше ресурсов, чем физически доступно в одном узле виртуализации. Позволяет ли это собрать виртуальные машины вместе, чтобы выполнить больше реальной работы, чем допускает физическая нагрузка, на одном и том же оборудовании?
- 14.2. Гипервизоры первого типа действуют непосредственно на физическом оборудовании без промежуточной операционной системы, а гипервизоры второго типа — в качестве приложения, установленного в существующей операционной системе. Гипервизоры первого типа проявляют себя намного лучше, чем гипервизоры второго типа, во-первых, потому что отсутствует промежуточный уровень для согласования действий как с самими собой, так и с системным оборудованием, а во-вторых, потому что не нужно соперничать за ресурсы с другим управляющим уровнем программного обеспечения. Почему в таком случае гипервизоры второго типа находят широкое применение? Каковы другие примеры их применения?
- 14.3. Когда виртуализация впервые появилась на рынке вычислительных систем на основе процессоров x86, многие поставщики серверов скептически отнеслись к данной технологии и были озабочены тем, что консолидация вычислительных ресурсов повлияет на объем продаж серверов. Вместо этого поставщики серверов обнаружили, что продали больше дорогих серверов. Почему это произошло?
- 14.4. Для предоставления дополнительной пропускной способности серверам виртуализации первоначально требовались дополнительные сетевые адAPTERы, чтобы обеспечить больше сетевых соединений. Но по мере того как становилась доступной постоянно увеличивающаяся (от 10 до 40 и 100 Гбит/с) пропускная способность сетевых магистралей, сетевых адAPTERов требовалось меньше. Какие затруднения могли бы вызвать эти сложные сетевые соединения и как их можно было бы разрешить?
- 14.5. Запоминающие устройства доступны на виртуальных машинах таким же образом, как и на физических машинах через соединения по протоколу TCP/IP, Fibre-Channel или iSCSI. Основные функциональные возможности виртуализации позволяют оптимизировать использование оперативной памяти и процессоров, а дополнительные функциональные возможности обеспечивают более эффективное применение ресурсов ввода-вывода. Как вы считаете, можно ли улучшить использование ресурсов запоминающих устройств в виртуализированной среде?

# БЕЗОПАСНОСТЬ ОПЕРАЦИОННЫХ СИСТЕМ

В ЭТОЙ ГЛАВЕ...

## 15.1. Злоумышленники и зловредные программы

- Угрозы системного доступа
  - Злоумышленники
  - Зловредные программы
- Контрмеры
  - Обнаружение вторжений
  - Аутентификация
  - Управление доступом
  - Брандмауэры

## 15.2. Переполнение буфера

- Атаки типа переполнения буфера
- Защита времени компиляции
  - Выбор языка программирования
  - Методики безопасного программирования
  - Языковые расширения и применение безопасных библиотек
  - Механизмы защиты стека
- Защита времени выполнения
  - Защита адресного пространства выполняемых модулей
  - Рандомизация адресного пространства
  - Защитные страницы

## 15.3. Управление доступом

- Управление доступом к файловой системе
- Стратегии управления доступом
  - DAC
  - RBAC

## 15.4. Управление доступом в UNIX

- Традиционное управление доступом к файлам в UNIX
- Списки управления доступом в UNIX

### 15.5. Усиление защиты операционных систем

- Установка операционной системы и применение обновлений
- Удаление ненужных служб, приложений и протоколов
- Конфигурирование пользователей, групп и аутентификации
- Конфигурирование средств управления ресурсами
- Установка дополнительных средств управления защитой
- Тестирование защиты системы

### 15.6. Поддержание безопасности

- Протоколирование
- Резервное копирование и архивирование данных

### 15.7. Безопасность Windows

- Схема управления доступом
- Токен доступа
- Дескрипторы безопасности

### 15.8. Резюме

### 15.9. Ключевые термины, контрольные вопросы и задачи

- Ключевые термины
- Контрольные вопросы
- Задачи

## УЧЕБНЫЕ ЦЕЛИ

- Оценивать ключевые вопросы безопасности, связанные с операционными системами.
- Ориентироваться в вопросах проектирования, касающихся безопасности файловой системы.
- Различать модели поведения злоумышленников и понимать методики взлома, применяемые для нарушения защиты компьютеров.
- Сравнивать и противопоставлять два метода управления доступом.
- Понимать, как защититься от атак переполнения буфера

# 15.1. Злоумышленники и зловредные программы

Операционная система с каждым процессом связывает ряд привилегий. Эти привилегии определяют, какие именно ресурсы могут быть доступны процессу, — в том числе такие, как области оперативной памяти, файлы и привилегированные системные команды. Как правило, процесс, выполняющийся от имени пользователя, обладает привилегиями, которые операционная система признает за данным пользователем. Системный или служебный процесс может обладать привилегиями, назначаемыми во время конфигурирования.

В типичной системе наивысший уровень привилегии называется административным, супервизорным или корневым доступом. В частности, корневой доступ обеспечивает доступ ко всем функциям и службам операционной системы.<sup>1</sup> Имея корневой доступ, процесс полностью контролирует систему и может вводить или изменять программы и файлы, осуществлять текущий контроль других процессов, передавать и принимать сетевой трафик, а также изменять привилегии.

Главный вопрос обеспечения безопасности при проектировании любой ОС состоит в том, как предотвратить или по крайней мере обнаружить попытки злоумышленного пользователя или фрагмента зловредного программного обеспечения получить несанкционированные привилегии в системе и, в частности, корневой доступ. В этом разделе вкратце описываются угрозы и контрмеры, связанные с вопросом обеспечения безопасности. А в последующих разделах вопросы, поставленные в этом разделе, обсуждаются более подробно.

## Угрозы системного доступа

Угрозы системного доступа делятся на две общие категории: злоумышленников и злонамеренных программ.

<sup>1</sup> В системах UNIX учетная запись администратора (или супервизора) называется корневой (root), отсюда и термин корневой доступ.

## Злоумышленники

Одну из самых распространенных угроз безопасности представляет злоумышленник (а другую — вирусы), зачастую называемый хакером или взломщиком. В одном из первых и весьма важных исследований вторжения [5] Андерсон (Anderson) определяет следующие классы злоумышленников.

- **Притворщик.** Субъект, который не авторизован для пользования компьютером и который проникает сквозь элементы управления доступом к системе и злоупотребляет учетной записью законного пользователя.
- **Нарушитель.** Законный пользователь, получающий доступ к данным, программам или ресурсам, доступ к которым ему не разрешен, или пользователь, которому разрешен подобный доступ, но он злоупотребляет своими привилегиями.
- **Нелегальный пользователь.** Субъект, захватывающий супервизорный контроль над системой и пользующийся им с целью обойти проверку и миновать элементы управления доступом или подавить сбор данных для контрольной проверки.

Притворщиком, вероятнее всего, может быть постороннее лицо, нарушителем обычно является член организации, а скрытым пользователем — как постороннее лицо, так и член организации. Атаки злоумышленников разнятся в широких пределах: от безвредных до серьезных. К числу совершающих безвредные атаки относятся многие люди, просто стремящиеся исследовать Интернет и другие сети, чтобы выяснить, что же там есть интересного для них. А к числу совершающих серьезные атаки относятся субъекты, пытающиеся прочитать привилегированные данные, выполнить несанкционированные модификации данных или вообще нарушить нормальную работу системы.

Цель злоумышленника — получить доступ к системе или расширить привилегии, доступные для него в системе. В большинстве первоначальных атак используются уязвимости в системе или программном обеспечении, дающие злонамеренному пользователю возможность выполнить код, открывающий лазейку в системе. Злоумышленники могут получить доступ к системе, предпринимая атаки типа переполнения буфера в программе, которая выполняется с определенными привилегиями. Атаки на основе переполнения буфера будут рассмотрены в разделе 15.2.

В качестве альтернативы злоумышленник пытается заполучить информацию, которая должна быть защищена. В некоторых случаях эта информация представлена в форме пользовательского пароля. Зная пароль какого-либо пользователя, злоумышленник может войти в систему и реализовать все привилегии, предоставленные законному пользователю.

## Зловредные программы

Вероятно, самые изощренные типы угроз для компьютерных систем представляют программы, злоупотребляющие уязвимостями в компьютерных системах. Такие угрозы называют **зловредными программами** (malicious software или malware). В данном контексте мы рассматриваем угрозы как для прикладных, так и для служебных программ, например редакторов и компиляторов, а также программ уровня ядра.

Зловредные программы могут быть разделены на две категории: те, которым требуется главная программа, и являющиеся независимыми. Зловредные программы первой категории называются **паразитными** и, по существу, являются фрагментами программ, которые не могут существовать независимо от какой-то конкретной прикладной, слу-

жебной или системной программы. Характерными примерами служат вирусы, логические бомбы и лазейки. Зловредные программы второй категории являются самостоятельными программами, выполнение которых можно запланировать в операционной системе. Характерными примерами служат программы-черви и роботы.

Можно также различать программные угрозы, которые не размножаются и которые способны к размножению. К первой категории относятся программы или фрагменты программ, активизируемые специальным триггером. Характерными их примерами служат логические бомбы, лазейки и программы-роботы. Ко второй категории относятся независимые программы или их фрагменты, которые при выполнении могут создавать одну или несколько своих копий, чтобы впоследствии активировать их в той же самой или другой системе. Характерными примерами служат вирусы и программы-черви.

Зловредные программы могут быть относительно вредоносными или же выполнять одно или ряд вредоносных действий, в том числе повреждение файлов и данных в основной памяти, обход элементов управления доступом для получения привилегированного доступа, а также предоставление злоумышленникам средств для обхода элементов управления доступом.

## Контрмеры

### Обнаружение вторжений

В документе RFC 4949 (*Internet Security Glossary* — Словарь специальных терминов по безопасности Интернета) обнаружение вторжения определяется следующим образом: служба безопасности, осуществляющая мониторинг и анализ системных событий с целью обнаружить и предупредить в реальном или близком к нему времени попытки несанкционированного доступа к системным ресурсам.

Системы обнаружения вторжений (Intrusion detection systems — IDS) могут быть классифицированы следующим образом.

- **IDS хоста.** Осуществляет текущий контроль характеристик одного хоста и наступающих в нем событий на предмет подозрительных действий.
- **Сетевая IDS.** Осуществляет мониторинг сетевого трафика конкретных сетевых сегментов или устройств и анализирует сетевые, транспортные и прикладные протоколы с целью выявить подозрительные действия.

IDS состоит из следующих логических компонентов.

- **Датчики.** Отвечают за сбор данных. В качестве входных данных для датчика может служить любая часть системы, которая может содержать свидетельство вторжения. К типам входных данных для датчика относятся сетевые пакеты, файлы журналов и трассировки системных вызовов. Датчики собирают данные и направляют их анализатору.
- **Анализаторы.** Принимают входные данные от одного или нескольких датчиков или иных анализаторов и отвечают за выявление произошедшего вторжения. К выходным данным может относиться свидетельство, подкрепляющее вывод, что произошло вторжение.
- **Пользовательский интерфейс.** Такой интерфейс для IDS дает пользователю возможность просматривать выходные данные системы или управлять ее поведением. В некоторых системах пользовательский интерфейс можно приоронять к диспетчерскому, управляющему или консольному компоненту.

Системы обнаружения вторжений, как правило, предназначаются для выявления вредоносного поведения как злоумышленников, так и зловредных программ.

## Аутентификация

В большинстве контекстов компьютерной безопасности аутентификация пользователей является основополагающим стандартным блоком и первой линией защиты. Аутентификация пользователей служит основанием для большинства типов управления доступом и учета пользователей. В документе RFC 4949 аутентификация пользователей определяется как процесс верификации идентификации, требующийся системному объекту или затребованный им.

Процесс аутентификации состоит из следующих стадий.

1. **Идентификация.** Предоставление идентификатора системе защиты. Следует присваивать идентификаторы внимательно, поскольку аутентифицированные личности пользователей служат основанием для функционирования других служб, в том числе службы управления доступом.
2. **Верификация.** Предоставление или формирование аутентификационной информации, подтверждающей связь между объектом и идентификатором.

Например, у пользователя Алисы Токлас может быть идентификатор ABTOKLAS. Эту информацию необходимо хранить на всяком сервере или в компьютерной системе, которой Алиса желает воспользоваться, и она может быть известна как администраторам системы, как и другим ее пользователям. Типичным элементом аутентификационной информации, связанным с идентификатором данного пользователя, является пароль, который хранится в секрете (т.е. он известен только Алисе и самой системе). Если никому не удается получить или разгадать пароль Алисы, то комбинация пользовательского идентификатора и пароля Алисы позволяет администраторам установить для нее права доступа и осуществлять контрольную проверку ее действий в системе. А поскольку идентификатор Алисы не является секретным, другие пользователи системы могут посыпать ей сообщения по электронной почте, но никто из них не сможет притвориться Алисой, поскольку ее пароль держится в секрете.

По существу, идентификация служит средством, с помощью которого пользователь предъявляет затребованную системой идентификационную информацию. Аутентификация пользователя служит средством, с помощью которого устанавливается достоверность предъявляемой идентификационной информации.

Для аутентификации личности пользователя имеются четыре общих средства, которыми можно пользоваться как по отдельности, так и в определенном сочетании, чтобы выяснить следующее:

1. **Нечто, известное субъекту.** Примерами служат пароль, персональный идентификатор пользователя (PIN) и ответы на ряд заранее подготовленных вопросов.
2. **Нечто, принадлежащее субъекту.** Примерами служат электронные ключевые карточки, интеллектуальные карточки и физические ключи. Аутентификатор такого типа называется *токеном*.
3. **Нечто, представляющее субъекта (статическая биометрия).** Примерами служит опознавание личности по отпечаткам пальцев, сетчатой оболочке глаза и лицу.
4. **Нечто, совершающее субъектом (динамическая биометрия).** Примерами служат опознавание личности по тембрю голоса, почерку и скорости набора текста на клавиатуре.

Если реализовать и употребить все эти методы надлежащим образом, то они способны обеспечить безопасную аутентификацию пользователя. Тем не менее каждому из этих методов присущи свои недостатки, ведь злоумышленник может разгадать или украдь пароль. Аналогично он может подделать или украсть токен аутентификации. В то же время пользователь может забыть пароль или потерять свой токен аутентификации. Кроме того, имеются значительные административные накладные расходы на управление информацией о паролях и токенах, а также на защиту этой информации в системах. В отношении биометрических аутентификаторов имеется немало затруднений, включая обработку ложно-положительных и ложно-отрицательных срабатываний, приемлемость системы для пользователя, затраты и удобство.

## Управление доступом

Реализует стратегию обеспечения безопасности, определяющую, кто и что (например, процесс) может иметь доступ к каждому системному ресурсу в отдельности и какой тип доступа разрешается в каждом случае.

Механизм управления доступом служит посредником между пользователем (или процессом, действующим от имени пользователя) и такими системными ресурсами, как приложения, операционные системы, брандмауэры, маршрутизаторы, файлы и базы данных. Сначала система может аутентифицировать пользователя, ищущего возможность доступа. Как правило, функция аутентификации выясняет, разрешен ли пользователю вообще доступ к системе, а затем определяет, разрешен ли конкретный доступ, запрошенный пользователем. Администратор безопасности ведет базу данных аутентификации, в которой указывается, какого типа доступ разрешается данному пользователю к конкретным ресурсам. Функция управления доступом обращается к этой базе данных с целью выяснить, следует ли предоставить доступ данному пользователю. А функция контрольной проверки осуществляет текущий контроль и ведет учет попыток пользователя получить доступ к системным ресурсам.

## Брандмауэры

Брандмауэры могут быть эффективными средствами защиты локальной системы или даже целой сети систем от сетевых угроз безопасности, предоставляя в то же время доступ к внешнему миру через глобальные сети и Интернет. Традиционно в качестве брандмауэра служит специально выделенный компьютер, соединяющийся с внешними компьютерами через сеть и предпринимающий специально встроенные в него меры предосторожности, чтобы защитить файлы конфиденциальных данных, хранящиеся на компьютерах, подключенных к сети. Он применяется для обслуживания внешней сети, в особенности подключений к Интернету и коммутируемых линий связи. Распространены и личные брандмауэры, которые реализуются как аппаратно, так и программно и связанны с одной рабочей станцией или ПК.

Проектирование брандмауэров преследует следующие цели.

1. Весь трафик изнутри наружу и обратно должен быть передан через брандмауэр. Это достигается физической блокировкой всего доступа к локальной сети, кроме доступа через брандмауэр.
2. Через брандмауэр может проходить только сетевой график, разрешенный локальными правилами защиты. В различных применяемых на практике брандмауэрах реализуются разные правила защиты.

3. Сам брандмауэр стоек к проникновению. Это подразумевает применение защищенной системы с безопасной операционной системой. Для размещения брандмауэра подходят надежные вычислительные системы, которые нередко находят применение в государственных учреждениях.

## 15.2. ПЕРЕПОЛНЕНИЕ БУФЕРА

Основная и виртуальная память представляют собой системные ресурсы, подверженные угрозам нарушения безопасности, и поэтому для их защиты необходимо принять соответствующие контрмеры. Наиболее очевидным требованием безопасности является предотвращение несанкционированного доступа к содержимому процессов в оперативной памяти. Так, если часть памяти, выделенной процессу, не объявлена совместно используемой, то ее содержимое должно быть недоступно для других процессов. А если часть памяти, выделенной процессу, объявлена как совместно используемая с другими назначеными процессами, то служба безопасности операционной системы должна гарантировать доступ к этой части памяти только уполномоченным на то процессам. Угрозы безопасности и контрмеры, рассматривавшиеся в предыдущем разделе, имеют отношение и к данному типу защиты оперативной памяти. А в этом разделе кратко излагается другая угроза, которая влечет за собой потребность в защите оперативной памяти.

### Атаки типа переполнения буфера

**Переполнение буфера**, иначе называемое **выходом за границы буфера**, определяется в документе *Glossary of Key Information Security Terms* (Словарь основных терминов по информационной безопасности) организации *NIST* (National Institute of Standards and Technology — Национальный институт стандартов и технологий США) следующим образом.

**Переполнение буфера** — состояние интерфейса, когда в буфере размещается больше входных данных, чем предусмотрено его размером, и поэтому они перезаписываются вместо других данных. Атакующие злоумышленники злоупотребляют таким состоянием, чтобы вывести систему из строя или же внедрить специально подготовленный код, позволяющий им взять систему под свой контроль.

Переполнение буфера может произойти в результате ошибки программирования, когда процесс пытается сохранить данные за пределами буфера фиксированного размера, а следовательно, перезаписывает соседние ячейки памяти. Эти ячейки могут содержать переменные или параметры другой программы или данные из потока управления программой, в том числе возвращаемые адреса и указатели на предыдущие фреймы стека. Буфер может быть выделен в стеке, в куче или разделе данных процесса. К последствиям такой ошибки относится повреждение данных, используемых в программе, непредусмотренная передача управления в программе, возможные нарушения доступа к памяти, а также наиболее вероятное в конечном счете завершение программы. Если переполнение буфера совершается намеренно как часть атаки на систему, то управление может быть передано любому коду, выбиравшему атакующим злоумышленником, в результате чего у него появится возможность выполнить произвольный код с привилегиями атакуемого

процесса. Атаки типа переполнения буфера относятся к одному из самых преобладающих и опасных видов атак, нарушающих безопасность.

Чтобы продемонстрировать действие основного типа переполнения буфера, иначе называемое **переполнением стека** (stack overflow), рассмотрим функцию `main()`, написанную на языке С и приведенную на рис. 15.1, а. В теле этой функции определяются три переменные (`valid`, `str1` и `str2`),<sup>2</sup> значения которых, как правило, хранятся в соседних ячейках памяти. Порядок их следования и расположения зависит от типа переменной (локальной или глобальной), применяемого языка программирования и компилятора, а также от архитектуры целевой машины. Для данного примера допустим, что значения этих переменных сохраняются в следующих подряд ячейках памяти: от старших к младшим, как показано на рис. 15.2.<sup>3</sup> Это типичный случай для локальных переменных функции, написанной на языке С, в таких распространенных архитектурах процессоров, как, например, семейство Intel Pentium.

```
int main(int argc, char *argv[]) {
    int valid = FALSE;
    char str1[8];
    char str2[8];
    next_tag(str1);
    gets(str2);
    if (strncmp(str1, str2, 8) == 0)
        valid = TRUE;
    printf("buffer1: str1(%s), str2(%s), valid(%d) \n", str1, str2, valid);
}
```

### а) Код на языке программирования С с переполнением буфера

```
$ cc -g -o buffer1 buffer1.c
$ ./buffer1
START
buffer1: str1(START), str2(START), valid(1)
$ ./buffer1
EVILINPUTVALUE
buffer1: str1(TVALUE), str2(EVILINPUTVALUE), valid(0)
$ ./buffer1
BADINPUTBADINPUT
buffer1: str1(BADINPUT), str2(BADINPUTBADINPUT), valid(1)
```

### б) Пример выполнения кода с переполнением буфера

**Рис. 15.1.** Простой пример переполнения буфера

<sup>2</sup> В данном примере значение переменной `valid` сохраняется как целочисленное, а не логическое. Это сделано потому, что так принято в классическом стиле программирования на С, а также с целью избежать трудностей выравнивания слов там, где данное значение сохраняется. Буфера намеренно определяются небольшими, чтобы заострить внимание на демонстрируемом здесь затруднении, возникающем в связи с их переполнением.

<sup>3</sup> Значения адресов и данных указаны на этом и связанных с ним рисунках в шестнадцатеричном виде. Там, где это уместно, значения данных указаны также в коде ASCII.

Назначение рассматриваемого здесь фрагмента кода — вызов функции `next_tag(str1)`, чтобы скопировать в переменную `str1` предполагаемое значение дескриптора. Допустим, что это будет символьная строка "START". Затем из стандартного потока ввода в программу читается следующая строка текста с помощью функции `gets()` из библиотеки С, а затем эта строка сравнивается с предполагаемым признаком. Если следующая строка текста действительно содержит лишь символьную строку "START", сравнение завершится удачно, а в переменной `valid` будет установлено значение TRUE.<sup>4</sup> Этот случай показан в первом из трех примеров выполнения данной программы, приведенных на рис. 15.1, б. Если будет введен любой другой дескриптор, значение переменной `valid` останется равным FALSE. Подобный фрагмент кода может быть использован для анализа какого-нибудь взаимодействия с использованием сетевого протокола или содержимого отформатированного текстового файла.

Адрес памяти	До вызова функции <code>gets (str2)</code>	После вызова функции <code>gets (str2)</code>	Содержит значение
...	...	...	
bfffffbf4	34fcffbf 4 . . .	34fcffbf 3 . . .	argv
bfffffbf0	01000000	01000000	argc
bfffffbec	...	...	
bfffffbe8	c6bd0340 . . . @	c6bd0340 . . . @	return addr
bfffffbe4	08fcffbf . . . .	08fcffbf . . . .	old base ptr
bfffffbe0	00000000	01000000	valid
bfffffbe0	80640140 . d . @	00640140 . d . @	
bffffbdcc	54001540 T . . @	4e505554 N P U T	str1[4-7]
bffffbd8	53544152 S T A R	42414449 B A D I	str1[0-3]
bffffbd4	00850408 . . . .	4e505554 N P U T	str2[4-7]
bffffbd0	30561540 O V . @	42414449 B A D I	str2[0-3]
...	...	...	

Рис. 15.2. Значения в стеке для простого примера переполнения буфера

<sup>4</sup> В языке С логические значения FALSE и TRUE на самом деле представлены целочисленными значениями 0 и 1 (а на самом деле любым ненулевым значением) соответственно. Такие символьские имена нередко служат псевдонимами базовых значений, как это сделано в данной программе.

Недостаток рассматриваемого здесь кода объясняется тем, что в функции `gets()` из традиционной библиотеки С отсутствует всякая проверка объема копируемых данных<sup>5</sup>. Эта функция читает очередную строку текста из стандартного потока ввода в программу до появления символа новой строки<sup>6</sup>, а затем копирует ее в предоставляемый буфер, добавляя нулевой символ, применяемый в С для завершения символьных строк.<sup>7</sup> Если во введенной строке текста присутствует больше семи символов, то для их записи (вместе с завершающим нулевым символом) потребуется больше места, чем имеется в буфере `str2`. Следовательно, лишние символы, не умещающиеся в буфере, перезапишут значение соседней переменной (в данном случае — `str1`). Так, если бы входная строка содержала текст "EVILINPUTVALUE", то значение переменной `str1` было бы перезаписано символами `TVALUE`, а в буфере `str2` были бы использованы не только восемь выделенных для него символов, но и еще семь символов из переменной `str1`. Это наглядно показано во втором примере из рис. 15.1, б. В результате переполнения повредилось содержимое переменной, которая не используется непосредственно для сохранения входных данных. Поскольку сравниваемые строки не равны, переменная `valid` также сохраняет свое значение `FALSE`. Если бы было введено 16 или большее количество символов, то для их хранения пришлось бы перезаписать дополнительные ячейки памяти.

Рассмотренный выше пример наглядно демонстрирует базовое поведение при переполнении буфера. В простейшем случае любое непроверяемое копирование данных в буфер может привести к повреждению соседних ячеек памяти, в которых могут храниться значения других переменных, а возможно, управляющие программой адреса и данные. Но даже этот простой пример можно развить. Зная структуру кода обработки данных, атакующий злоумышленник мог бы устроить так, чтобы значение, перезаписываемое в переменной `str1`, стало равным значению, сохраняемому в переменной `str2`, а в итоге последующее сравнение завершилось бы удачно. Так, если бы была введена символьная строка "BADINPUTBADINPUT", это привело бы к удачному исходу сравнения, что демонстрируют третий пример выполнения рассматриваемой здесь программы на рис. 15.1, б, а также значения локальных переменных на рис. 15.2 до и после вызова функции `gets()`.

Следует также иметь в виду, что завершающий нулевой символ входной символьной строки записан в ячейку памяти после переменной `str1`. Это означает, что поток управления программой продолжится так, как будто предполагаемый дескриптор был обнаружен, в то время как на самом деле был прочитан совершенно другой дескриптор. И это, скорее всего, приведет к непредусмотренному поведению программы. В частности, одна из опасностей могла бы возникнуть в том случае, если бы вместо дескриптора в обоих буферах сохранялся пароль, требующийся для доступа к привилегированным функциям.

<sup>5</sup> Именно поэтому данная функция изъята из современных стандартов языков программирования С и C++. — Примеч. ред.

<sup>6</sup> Символ новой строки (*NL*), или перевода строки (*LF*), служит стандартным средством окончания строк в системах UNIX, а следовательно, и в С, и является символом кода ASCII со значением 0x0a.

<sup>7</sup> Символьные строки в языке С хранятся в массиве символов и оканчиваются нулевым символом, значение которого в коде ASCII равно 0x00. Любые оставшиеся символы в массиве не определены и, как правило, содержат произвольные значения, которые были ранее сохранены в данной области памяти. Это можно ясно наблюдать по значению переменной `str2` в столбце "До вызова функции `gets(str2)`" на рис. 15.2.

нальным возможностям. В таком случае переполнение буфера предоставляло бы атакующему злоумышленнику средства доступа к этим функциональным возможностям даже без знания правильного пароля.

Чтобы воспользоваться переполнением буфера любого типа (например, одним из продемонстрированных выше), атакующему злоумышленнику необходимо следующее.

1. Выявить уязвимость переполнения буфера в некоторой программе, которую можно запустить, используя данные, поставляемые извне под контролем атакующего злоумышленника.
2. Выяснить, каким образом буфер хранится в памяти, выделяемой процессам, и найти возможность так повредить соседние ячейки памяти, чтобы изменить ход выполнения программы.

Выявить уязвимые программы можно, проверив их исходный код и трассировку выполнения программ по мере обработки чрезмерного объема входных данных или воспользовавшись такими методиками, как *автоматическое тестирование безопасности* (так называемый *фаззинг*, от англ. *fuzzing* — “распыление”), которое подразумевает применение произвольно формируемых входных данных, чтобы автоматически выявлять потенциально уязвимые программы. А что атакующему злоумышленнику делать с результатами повреждения оперативной памяти — зависит от того, какие именно значения перезаписываются.

## Защита времени компиляции

Обнаружить переполнение буфера в стеке и воспользоваться им совсем не трудно. Это наглядно показывает огромное количество взломов систем за прошедшие два десятка лет. Следовательно, имеется насущная потребность в защите систем от подобных атак путем их предотвращения или хотя бы обнаружения. Меры против переполнения стека можно разделить на две категории.

1. Защита времени компиляции, нацеленная на повышение стойкости программ к атакам.
2. Защита времени выполнения, нацеленная на обнаружение и прекращение атак в выполняющихся программах.

Несмотря на то что за последние два десятка лет появились подходящие средства защиты, существующая в настоящее время весьма крупная база уязвимых программ и систем препятствует их развертыванию. Отсюда возникает интерес к средствам защиты времени выполнения, которые могут быть развернуты в операционных системах и в их обновлениях и обеспечивать защиту существующих уязвимых систем.

В этом разделе рассматриваются сначала средства защиты времени компиляции, а затем — средства защиты времени выполнения. Средства защиты времени компиляции предназначаются для предотвращения или обнаружения попыток переполнения буфера, обеспечивая программы этими возможностями во время их компиляции. Такие возможности простираются от выбора языка высокого уровня, не допускающего переполнение буфера, до стимулирования норм практики безопасного программирования с помощью безопасных стандартных библиотек или внедрения дополнительного кода для обнаружения повреждений кадров стека.

## Выбор языка программирования

Одна из возможностей состоит в том, чтобы написать программу на современном языке программирования высокого уровня, в котором строго определены типы переменных и допустимые операции над ними. Такие языки неувязимы к атакам типа переполнения буфера, поскольку их компиляторы дополняют код автоматическими проверками выхода за допустимые границы, а следовательно, программисту не нужно писать этот код вручную. Но за гибкость и безопасность, которую обеспечивают эти языки программирования, приходится расплачиваться используемыми ресурсами как во время компиляции, так и во время выполнения, когда требуется непременно выполнить дополнительный код различных проверок, в том числе связанный с ограничениями, накладываемыми размерами буферов. Впрочем, сегодня эти недостатки не столь существенны, как прежде, благодаря быстрому росту производительности процессоров. Поэтому постоянно растет число программ, написанных на этих языках программирования, а следовательно, и стойкость к переполнению буфера. Хотя эти программы могут по-прежнему оставаться уязвимыми, если они пользуются системными библиотеками или средами выполнения, написанными на менее безопасных языках. Отдаленность от базового машинного языка и архитектуры означает также утрату доступа к некоторым командам и аппаратным ресурсам. По всем этим причинам вполне вероятно, что какая-то часть кода окажется написанной на менее безопасных языках наподобие С.

## Методики безопасного программирования

Если применяются такие языки программирования, как С, программисты должны принимать во внимание их способность работать с адресами и указателями и обеспечивать непосредственный доступ к оперативной памяти, а за это приходится платить. Язык С предназначен для системного программирования, а написанные на нем программы — для выполнения в намного более мелких и ограниченных системах, чем используемые сегодня. Разработчики С уделили намного больше внимания вопросам эффективности используемого пространства и производительности, чем безопасности. Они предполагали, что программисты будут проявлять должную осмотрительность при написании кода на подобных языках и возьмут на себя ответственность за обеспечение безопасного использования всех структур данных и переменных.

К сожалению, как показал опыт нескольких десятилетий, это оказалось не совсем так. Например, в унаследованной кодовой базе операционных систем UNIX и Linux и в приложениях наблюдалось немало потенциально опасного кода, причем существенная часть уязвимостей была связана с переполнением буфера.

Чтобы защитить такие системы, программисту придется исследовать исходный код и переписать все небезопасные программные конструкции безопасным образом. В некоторых случаях этот процесс уже начался, принимая во внимание быстрые темпы роста попыток использования уязвимостей систем, связанных с переполнением буфера. Характерным примером служит проект OpenBSD по разработке свободно доступной, многоплатформенной UNIX-подобной операционной системы на основе свободно распространяемой версии 4.4BSD. Среди прочих технологических изменений программисты предприняли обширную ревизию существующей кодовой базы, включая саму операционную систему, стандартные библиотеки и распространенные утилиты. В итоге получилась операционная система, которая повсеместно признана одной из самых надежных среди широко используемых операционных систем. Как было заявлено в проекте OpenBSD,

на середину 2006 года, более чем за восемь лет использования, в стандартной установке операционной системы была выявлена лишь одна брешь. Это завидный показатель безопасности. Корпорация Microsoft также предприняла меры к пересмотру своей кодовой базы отчасти в ответ на постоянное негативное освещение в СМИ целого ряда уязвимостей, включая многие осложнения, связанные с переполнением буфера, которые были выявлены в исходном коде операционных систем и приложений этой корпорации.

## Языковые расширения и применение безопасных библиотек

Принимая во внимание трудности, которые могут возникнуть в С из-за небезопасных применений массивов и указателей, был внесен целый ряд предложений с целью усовершенствовать компиляторы таким образом, чтобы они автоматически вставляли проверку выхода за границы в таких случаях. Но если это совсем не трудно сделать для статически выделяемых массивов, то для динамически выделяемой памяти это намного труднее, поскольку сведения об ее объеме во время компиляции отсутствуют. Чтобы справиться с этой задачей, необходимо расширить семантику указателей таким образом, чтобы включить в них сведения о границах доступной памяти, а также использовать библиотечные функции, чтобы обеспечить правильность установки граничных значений. Некоторые из таких подходов перечислены в [155], но их применение, как правило, приводит к снижению производительности, что может оказаться неприемлемым. Кроме того, подобные методы предусматривают перекомпиляцию с помощью модифицированного компилятора всех программ и библиотек, в которых требуется принять меры по обеспечению безопасности. Хотя это вполне осуществимо в новом выпуске операционной системы и связанных с ней утилит, со сторонними приложениями все равно останутся определенные трудности.

Общее опасение при программировании на С вызывает применение небезопасных стандартных библиотечных функций, особенно предназначенных для работы с символьными строками. Один из подходов к повышению безопасности систем состоял в том, чтобы заменить эти функции более безопасными их вариантами. Например, семейство систем BSD, включая OpenBSD, можно снабдить новыми функциями наподобие `strlcpy()`. Но чтобы использовать такие функции, придется переписать исходный код в соответствии с новой более безопасной семантикой. С другой стороны, можно заменить стандартную библиотеку функций для работы с символьными строками более безопасной версией. Хорошо известным примером служит библиотека Libsafe, реализующая стандартную семантику, но включающая дополнительные проверки, которые гарантируют, что операции копирования не выйдут за пределы пространства локальных переменных в кадре стека. Хотя библиотека Libsafe и не позволяет предотвратить повреждение соседних локальных переменных, она все же предупреждает любую модификацию прежнего кадра стека и значения адреса возврата. Таким образом, она предотвращает упоминавшиеся ранее классические атаки типа переполнения буфера в стеке. Эта библиотека реализована как динамическая и загружается до стандартных библиотек, а следовательно, способна обеспечить защиту существующих программ, не требуя их перекомпиляции, — при условии, что они получают доступ к стандартным библиотечным функциям в динамическом режиме, как это обычно делается в большинстве программ. Модифицированный подобным образом библиотечный код, как правило, оказывается по крайней мере таким же эффективным, как и в стандартных библиотеках, а следовательно, с его помощью нетрудно защитить существующие программы от некоторых разновидностей атак на основе переполнения буфера.

## Механизмы защиты стека

Эффективный метод защиты программ от классических атак типа переполнения буфера состоит в том, чтобы реализовать код входа и выхода из функции для подготовки и последующей проверки кадра стека для обнаружения любого свидетельства его повреждения. Если при этом будет обнаружена любая модификация кадра, выполнение программы будет сразу же прервано, чтобы не допустить продолжение атаки. Имеется несколько рассматриваемых далее подходов, обеспечивающих подобную защиту.

Захист стека является одним из хорошо известных механизмов защиты. Это расширение компилятора GCC, внедряющее дополнительный код на входе и выходе из функции. Код, добавляемый на входе функции, записывает значение **канарейки** (canary)<sup>8</sup> ниже адреса указателя на прежний фрейм стека, прежде чем выделять пространство для локальных переменных. Код, добавляемый на выходе из функции, проверяет, не было ли изменено значение “канарейки”, прежде чем продолжить операции обычного выхода из функции, восстанавливающие указатель прежнего фрейма стека и передающие управление обратно по адресу возврата. Любая попытка классического переполнения буфера в стеке будет состоять в том, чтобы изменить данное значение, а значит, и указатели на прежний кадр стека и адрес возврата. Таким образом, попытка такого рода будет сразу же обнаружена, что приведет к немедленному прерыванию программы. Для того чтобы такой механизм защиты действовал успешно, крайне важно сделать значение канарейки необнаруживаемым, и оно должно быть различным в разных системах. Если бы это было не так, то атакующий злоумышленник просто позаботился бы о том, чтобы внедряемый им код запуска командной оболочки разместил подходящее значение канарейки в нужном месте. Как правило, при создании процесса в качестве значения канарейки выбирается случайное значение, которое сохраняется как часть состояния данного процесса. Это значение затем используется кодом, добавленным ко входу и выходу из функции.

Применение данного подхода вызывает определенные трудности. Во-первых, он предусматривает, что все программы, требующие защиты, должны быть перекомпилированы. Во-вторых, структура кадра стека изменяется, а это может вызвать осложнения в таких программах, как отладчики, которые анализируют кадры стека. Тем не менее методика канарейки была использована для перекомпиляции всего дистрибутива Linux, чтобы повысить его стойкость к атакам, основанным на переполнении стека. Аналогичные функциональные возможности доступны и для программ Windows, которые достаточно скомпилировать, используя опцию /GS командной строки компилятора Visual C++ корпорации Microsoft.

## Защита времени выполнения

Как отмечалось ранее, большая часть методов защиты времени компиляции требуют перекомпиляции существующих систем. Отсюда возникает интерес к средствам защиты времени выполнения, которые могут быть развернуты в виде обновлений операционных систем, чтобы обеспечить защиту существующих уязвимых программ. Такие средства защиты предусматривают изменения в управлении областью памяти, выделяемой под пространство виртуальных адресов процессов. Эти изменения служат как для измене-

<sup>8</sup> Название этой переменной происходит от шахтерской канарейки, использовавшейся для выявления отравленного воздуха в шахтах, чтобы предупредить шахтеров и вовремя покинуть забой (токсичный газ убивает канарейку гораздо быстрее, чем человека).

ния свойств областей оперативной памяти, так и для того, чтобы в достаточной степени затруднить прогнозирование расположения буферов, на которые нацелены многие виды атак, а следовательно, предотвратить их.

### **Защита адресного пространства выполняемых модулей**

Многие атаки типа переполнения буфера включают в себя копирование машинного кода в целевой буфер, а затем передачу ему управления. Для защиты от подобных атак можно, например, заблокировать выполнение кода в стеке, исходя из того, что выполняемый код должен находиться в каком-нибудь другом месте адресного пространства процесса.

Для эффективной поддержки такой возможности требуется, чтобы страницы виртуальной памяти помечались в блоке управления памятью (memory management unit — MMU) процессора как невыполнимые. Некогда такая поддержка имелась в некоторых процессорах (например, в процессорах SPARC, используемых операционной системой Solaris). Чтобы воспользоваться ею в Solaris, достаточно изменить простой параметр ядра. А в других процессорах (например, семейства x86) такая поддержка до сих пор отсутствовала, и лишь недавно в их блоках управления памятью была добавлена возможность устанавливать бит **запрета выполнения** (no-execute). Эта функциональная возможность позволила внедрить соответствующие расширения в Linux, BSD и другие UNIX-подобные системы. Некоторые из них действительно способны защищать не только кучу (динамическую область памяти), но и стек, который также является целью подобных атак. Поддержка защиты, запрещающей несанкционированное выполнение кода, была включена и в последние версии систем Windows.

Благодаря тому что стек и куча становятся невыполнимыми областями оперативной памяти, обеспечивается высокая степень защиты существующих программ от многих видов атак, основанных на переполнении буфера, и эта практика стала стандартной в целом ряде недавних выпусков операционных систем. Тем не менее остается одно затруднение, связанное с поддержкой тех программ, которым требуется размещать выполняемый код в стеке. Такое может, в частности, произойти в динамических компиляторах (например, применяемых в исполнительной системе Java). Код, выполняемый в стеке, применяется и для реализации вложенных функций в C (расширение GCC), а также обработчиков сигналов в Linux. Для удовлетворения подобных требований необходимо принять специальные меры. Тем не менее такой метод защиты существующих программ и повышения надежности систем считается одним из самых лучших.

### **Рандомизация адресного пространства**

Еще один метод защиты времени выполнения от рассматриваемых здесь видов атак подразумевает работу с расположением основных структур данных в адресном пространстве процесса. Напомним, что для реализации классической атаки типа переполнения буфера в стеке атакующему злоумышленнику необходимо суметь спрогнозировать приблизительное местоположение целевого буфера. Атакующий злоумышленник использует спрогнозированный в своей атаке адрес этого местоположения с целью определить подходящий адрес возврата, чтобы использовать его для передачи управления коду запуска командной оболочки. Один из методов, позволяющих существенно затруднить такое прогнозирование, состоит в том, чтобы произвольно изменять адрес, по которому размещается стек для каждого процесса. Диапазон адресов, доступный в современных процессорах, весьма широк (32 бита), и большинству программ требуется лишь

малая его часть. Следовательно, перемещение области памяти, выделяемой под стек, в пределах мегабайтного или около того адресного пространства оказывается на работе большинства программ в минимальной степени, но в то же время делает практически невозможным прогнозирование адреса расположения целевого буфера в оперативной памяти.

Еще одной целью подобных атак является местоположение стандартных библиотечных функций. Пытаясь обойти защиту (например, невыполнимые стеки), некоторые разновидности атак на основе переполнения буфера эксплуатируют уязвимость исходного кода, существующего в стандартных библиотеках, которые обычно в одной и той же программе загружаются по одному и тому же адресу. В качестве меры борьбы с такой формой атаки можно воспользоваться расширением безопасности, определяющим порядок загрузки стандартных библиотек в программе случайным образом, как и адреса их расположения в виртуальной памяти. Это позволяет существенно затруднить прогнозирование адреса расположения любой отдельной функции, а следовательно, вероятность правильно предсказать его в данной атаке. В состав системы OpenBSD версии таких расширений входят в качестве технологического обеспечения безопасности системы.

### **Защитные страницы**

Последний метод, который может быть использован для защиты от рассматриваемых здесь видов атак, подразумевает размещение **защитных страниц** (guard pages) между критическими областями памяти в адресном пространстве процессов. В этом случае используется то обстоятельство, что у процесса имеется намного больше доступной виртуальной памяти, чем обычно требуется. Между диапазонами адресов, используемых для каждой составляющей части адресного пространства, размещаются промежутки. Такие промежутки (защитные страницы) помечаются в блоке управления памятью как недопустимые адреса, и любая попытка получить к ним доступ сразу же приводит к прерыванию процесса. Благодаря этому предотвращаются атаки переполнения буфера (обычно на глобальные данные), которые пытаются перезаписать смежные области в адресном пространстве процессов.

Дальнейшее расширение позволяет расположить защищенные страницы между кадрами стека или разными выделяемыми участками памяти в куче. Благодаря этому можно обеспечить дополнительную защиту от атак переполнения стека и кучи, хотя и за счет времени выполнения, требующегося для поддержки большого количества операций отображения страниц.

## **15.3. УПРАВЛЕНИЕ ДОСТУПОМ**

Управление доступом — это функция, выполняемая на уровне операционной системы, файловой системы или на обоих уровнях, на которых, как правило, применяются одинаковые принципы такого управления. Сначала в этом разделе управление доступом рассматривается на уровне файловой системы, а затем обсуждение обобщается до правил, применяемых к самым разным системным ресурсам.

### **Управление доступом к файловой системе**

После успешного входа в систему пользователю предоставляется доступ к одному или ряду хостов и приложений. Но этого обычно оказывается недостаточно для работы

в системе с конфиденциальной информацией, хранящейся в ее базе данных. Через процедуру управления доступом пользователя можно установить его личность для системы. С каждым пользователем может быть связан соответствующий профиль, в котором определяются разрешенные операции и виды доступа к файлам, так что операционная система сможет обеспечить выполнение правил, основывающихся на профиле пользователя. Но система управления базой данных должна управлять доступом к отдельным записям или даже к их частям. Например, любому представителю администрации может быть разрешено получение списка сотрудников компании, но лишь избранные лица могут иметь доступ к сведениям о зарплате. И это не просто вопрос уровня детализации. Если операционная система может разрешить пользователю доступ к файлу или приложению, после чего не последует больше никаких проверок безопасности, то система управления базой данных должна принимать соответствующее решение по каждой попытке доступа в отдельности. И такое решение будет зависеть не только от личности самого пользователя, но и от отдельных частей данных, к которым осуществляется доступ, и даже от информации, уже предоставленной пользователю.

Общей моделью управления доступом, применяемой в файловой системе или же в системе управления базой данных, служит **матрица доступа** (access matrix, рис. 15.3, а). Ниже перечислены основные элементы такой модели.

- **Субъект.** Это лицо, способное получить доступ к объектам. Как правило, понятие субъекта приравнивается к понятию процесса. Любой пользователь или приложение фактически получает доступ к объекту с помощью процесса, представляющего данного пользователя или приложение.
- **Объект.** Это все, что подлежит управлению доступом. Примерами служат файлы и отдельные их части, программы, участки оперативной памяти и программные объекты (например, объекты Java).
- **Права доступа.** Это способ, посредством которого субъект получает доступ к объекту. Примерами служат операции чтения, записи, выполнения и функции в программных объектах.

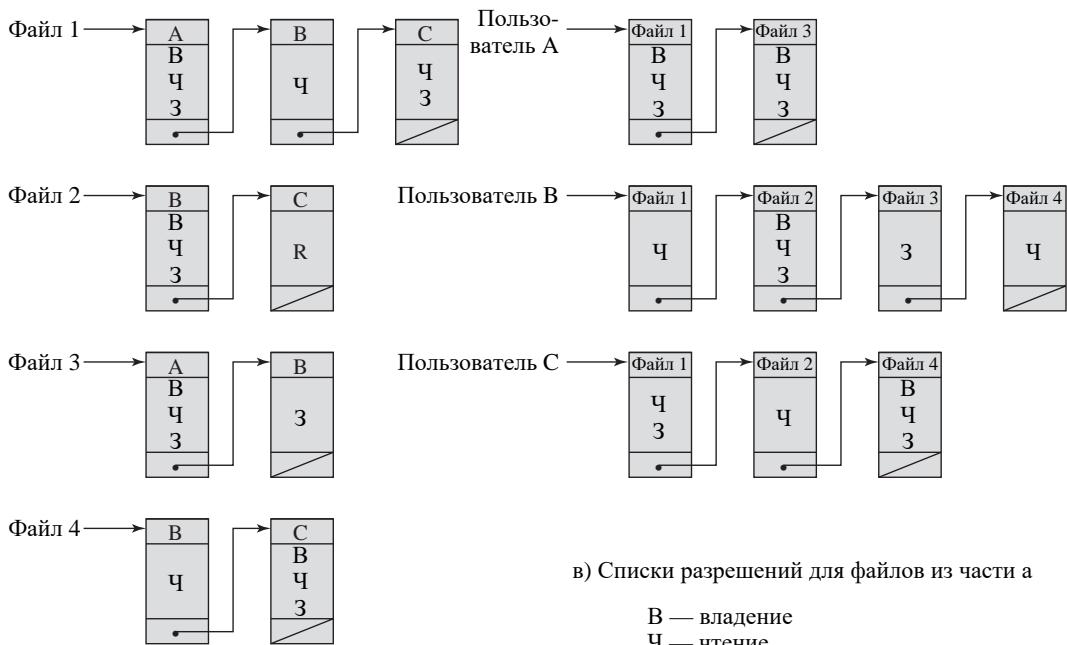
Одно измерение матрицы прав доступа состоит из идентифицированных субъектов, которые могут пытаться получить доступ к данным. Как правило, такой список состоит из отдельных пользователей или их групп, хотя управление доступом может распространяться и на терминалы, хосты или приложения, а не только на пользователей или вместо них. Второе измерение матрицы прав доступа состоит из объектов, которые могут быть доступны. На самом высоком уровне детализации объектами могут служить отдельные поля данных. Объектами в данной матрице могут быть и более укрупненные группы (например, записи, файлы или даже целая база данных). Каждый элемент матрицы обозначает права доступа данного субъекта по отношению к данному объекту.

На практике матрица прав доступа обычно разреженная и реализуется одним из двух способов декомпозиции. В частности, матрица прав доступа может быть разложена на столбцы, чтобы получить в итоге **справки доступа** (access control list, рис. 15.3, б). Таким образом, для каждого объекта имеется список доступа, в котором перечислены пользователи и предоставленные им права доступа. Список доступа может содержать устанавливаемый по умолчанию (или общедоступный) элемент. Это дает пользователям, не перечисленным в данном списке как явно наделенными особыми правами, возможность иметь права доступа, устанавливаемые по умолчанию. Элементы данного списка могут включать в себя как отдельных пользователей, так и их группы.

### Объекты

	Файл 1	Файл 2	Файл 3	Файл 4
Субъекты	Владение Чтение Запись		Владение Чтение Запись	
Пользователь А				
Пользователь В	Чтение	Владение Чтение Запись	Запись	Чтение
Пользователь С	Чтение Запись	Чтение		Владение Чтение Запись

а) Матрица доступа



б) Списки управления доступом к файлам из части а

**Рис. 15.3.** Пример структур управления доступом

Декомпозиция матрицы доступа на строки в итоге дает **списки разрешений** (capability tickets, рис. 15.3, в). Разрешение определяет объекты и операции, разрешенные пользователю. У каждого пользователя имеется ряд разрешений, и ему может быть предоставлено право запрашивать их у других пользователей или предоставлять их другим пользователям. А поскольку разрешения могут быть рассеяны по всей системе, они представляют большую трудность для обеспечения безопасности, чем списки управления доступом. В частности, разрешение не должно поддаваться подделке. Чтобы добить-

ся этого, можно, например, сохранять все разрешения в операционной системе от имени пользователей. Такие разрешения должны обязательно храниться в области оперативной памяти, недоступной для пользователей.

Вопросы управления доступом к данным в сети должны рассматриваться параллельно с управлением доступом пользователей. Так, если некоторым пользователям разрешен доступ к определенным элементам данных, то для целей защиты эти элементы данных, возможно, придется шифровать, чтобы защитить их при передаче авторизованным пользователям. Как правило, управление доступом к данным децентрализовано, т.е. осуществляется системами управления базами данных хостов. Если же в сети имеется сервер базы данных, то управление доступом к данным становится функцией сети.

## Стратегии управления доступом

Стратегии управления доступом предписывают, какие именно виды доступа, при каких условиях и кому разрешены. Такие стратегии обычно разделяются на следующие категории.

- **Дискреционное управление доступом (discretionary access control — DAC).** Осуществляется на основании личности запрашивающего лица и правил доступа (авторизаций), устанавливающих, что именно разрешено (или не разрешено) делать запрашивающему доступ субъекту. Такая стратегия доступа называется дискреционной (избирательной), потому что у отдельного лица могут быть права доступа, разрешающие ему по своей воле предоставлять доступ к тому же ресурсу другому лицу.
- **Принудительное управление доступом (mandatory access control — MAC).** Осуществляется на основании сравнения метки (грифа) секретности, обозначающей степень секретности или важности системных ресурсов, с допуском к секретам, обозначающим, какие именно системные объекты имеют право доступа к определенным ресурсам. Такая стратегия доступа не позволяет лицу с полномочиями доступа к ресурсу по своей воле разрешать другому лицу доступ к этому ресурсу.
- **Ролевое управление доступом (role-based access control — RBAC).** Осуществляется на основании ролей, которые имеются у пользователей в системе, а также правил, устанавливающих, какие именно виды доступа разрешены пользователям с заданными ролями.
- **Атрибутное управление доступом (attribute-based access control — ABAC).** Осуществляется на основании атрибутов пользователя, ресурсов, к которым требуется доступ, а также текущих внешних условий.

DAC является традиционным методом реализации управления доступом. Этот метод уже был представлен при рассмотрении управления доступом к файлам; в этом разделе мы рассмотрим его более подробно. Концепция MAC возникла из требований к защите информации военного характера, а потому ее рассмотрение выходит за рамки данной книги. Все более и более популярными становятся стратегии RBAC и ABAC. Далее будут рассмотрены стратегии DAC и RBAC.

Упомянутые выше четыре стратегии управления доступом не являются взаимоисключающими. В механизме управления доступом могут применяться две или даже три такие стратегии, охватывающие различные классы системных ресурсов.

## DAC

В этом разделе представлена общая модель для DAC, разработанная Лэмпсоном (Lampson), Грэхемом (Graham) и Деннигом (Denning) [61, 93, 142]. В этой модели предполагается наличие множества субъектов, множества объектов и множества правил, регулирующих доступ субъектов к объектам. Определим состояние защиты системы как множество сведений, которые в данный момент времени задают правила доступа каждого субъекта к каждому объекту. Мы можем идентифицировать три требования: представления состояния защиты, обеспечения прав доступа и разрешения субъектам изменять определенным образом состояние защиты. Данная модель удовлетворяет всем трем требованиям, предоставляя общее логическое описание системы DAC. Чтобы представить состояние защиты, расширим универсум объектов в матрице доступа таким образом, чтобы включить в нее следующее.

- Процессы.** Права доступа должны включать в себя возможность удалить, остановить (заблокировать) и возобновить (активировать) процесс.
- Устройства.** Права доступа должны включать в себя возможность чтения/записи устройства, управления им (например, выполнять поиск данных на диске), а также блокировать и разблокировать устройство.
- Ячейки и области памяти.** Права доступа должны включать в себя возможность чтения/записи определенных ячеек и областей оперативной памяти, которые защищены, а следовательно, доступ к ним по умолчанию не разрешен.
- Субъекты.** Права доступа для субъектов должны включать в себя возможность передавать права доступа данного субъекта другим субъектам или лишать их этих прав, как поясняется далее.

Пример применения рассматриваемой здесь модели управления доступом приведен на рис. 15.4 (сравните его с рис. 15.3, а). Каждый элемент  $A[S, X]$  матрицы доступа  $A$  содержит строки, именуемые атрибутами доступа, которые определяют права доступа субъекта  $S$  к объекту  $X$ . Например, как показано на рис. 15.4, субъект  $S_1$  может прочитать содержимое файла  $F_2$ , поскольку в элементе  $A[S_1, F_1]$  присутствует право **чтения**.

С логической или функциональной точки зрения с каждым типом объекта связан отдельный модуль управления доступом (рис. 15.4). Такой модуль оценивает каждый запрос субъекта на доступ к объекту, чтобы выяснить, имеются ли у него соответствующие права доступа.

		Объекты								
		Субъекты		Файлы		Процессы		Дисковые накопители		
Субъекты	$S_1$	Управление доступом	Владелец	Владелец управляет доступом	Чтение *	Чтение владельца	Активация	Активация	Поиск	Владелец
	$S_2$	Управление доступом			Запись *	Выполнение			Владелец	Поиск *
	$S_3$		Управление доступом			Запись	Остановка			

\* — Установлен флаг копирования

**Рис. 15.4.** Расширенная матрица прав доступа

При попытке доступа запускается процесс, состоящий из следующих стадий.

1. Субъект  $S_0$  выдает запрос на доступ типа  $\alpha$  к объекту  $X$ .
2. Этот запрос приводит к тому, что система (операционная система или некоторый модуль интерфейса управления доступом) генерирует сообщение вида  $(S_0, \alpha, X)$ , направляемое контроллеру объекта  $X$ .
3. Контроллер опрашивает матрицу доступа  $A$ , чтобы выяснить, находится ли доступ типа  $\alpha$  в элементе матрицы  $A[S_0, X]$ . Если это так, то доступ разрешен, иначе он запрещен, и в связи с этим возникает нарушение защиты, которое должно инициировать предупреждение и соответствующее действие.

На рис. 15.5, где схематически показано функционирование механизма управления доступом, предполагается, что при каждом доступе контроллер объекта выступает посредником между субъектом и данным объектом и что решение принимается в контроллере, исходя из текущего содержимого матрицы доступа. Кроме того, некоторые субъекты обладают полномочиями вносить определенные изменения в матрицу прав доступа. В частности, запрос на видоизменение матрицы прав доступа интерпретируется как доступ к этой матрице, а отдельные ее элементы — как объекты. Посредником при таких видах доступа выступает контроллер матрицы прав доступа, управляющий обновлениями в данной матрице.

В рассматриваемую здесь модель входит также ряд правил, регулирующих порядок модификации матриц прав доступа, как показано в табл. 15.1. С этой целью вводятся понятия *владельца* прав доступа и *управления* доступом, а также поясняемое далее понятие *признака копирования*.

**Таблица 15.1. Команды системы управления доступом**

Правило	Команда (от субъекта $S_0$ )	Авторизация	Операция
R1	<b>Передать</b> $\left\{ \begin{array}{l} \alpha^* \\ \alpha \end{array} \right\} \text{в } S, X$	" $\alpha^*$ " в $A[S_0, X]$	Сохранить $\left\{ \begin{array}{l} \alpha^* \\ \alpha \end{array} \right\}$ в $A[S, X]$
R2	<b>Предоставить</b> $\left\{ \begin{array}{l} \alpha^* \\ \alpha \end{array} \right\}$ <b>для</b> $S, X$	"Владелец" в $A[S_0, X]$	Сохранить $\left\{ \begin{array}{l} \alpha^* \\ \alpha \end{array} \right\}$ в $A[S, X]$
R3	<b>Удалить</b> $\alpha$ <b>из</b> $S, X$	"Управление" $A[S_0, S]$ или "владелец" в $A[S_0, X]$	Удалить $\alpha$ из $A[S, X]$
R4	$w \leftarrow$ <b>чтение</b> $S, X$	"Управление" $A[S_0, S]$ или "владелец" в $A[S_0, X]$	Копировать $A[S, X]$ в $w$
R5	<b>Создать объект</b> $X$	Отсутствует	Добавить столбец для $X$ в $A$ ; сохранить "владелец" в $A[S_0, X]$
R6	<b>Уничтожить объект</b> $X$	"Владелец" в $A[S_0, X]$	Удалить столбец для $X$ из $A$
R7	<b>Создать субъект</b> $S$	Отсутствует	Добавить строку для $S$ в $A$ ; выполнить команду <b>создать объект</b> $S$ ; сохранить "управление" в $A[S, S]$
R8	<b>Уничтожить субъект</b> $S$	"Владелец" в $A[S_0, S]$	Удалить из матрицы $A$ строку для $S$ из $A$ ; выполнить команду <b>уничтожить объект</b> $S$

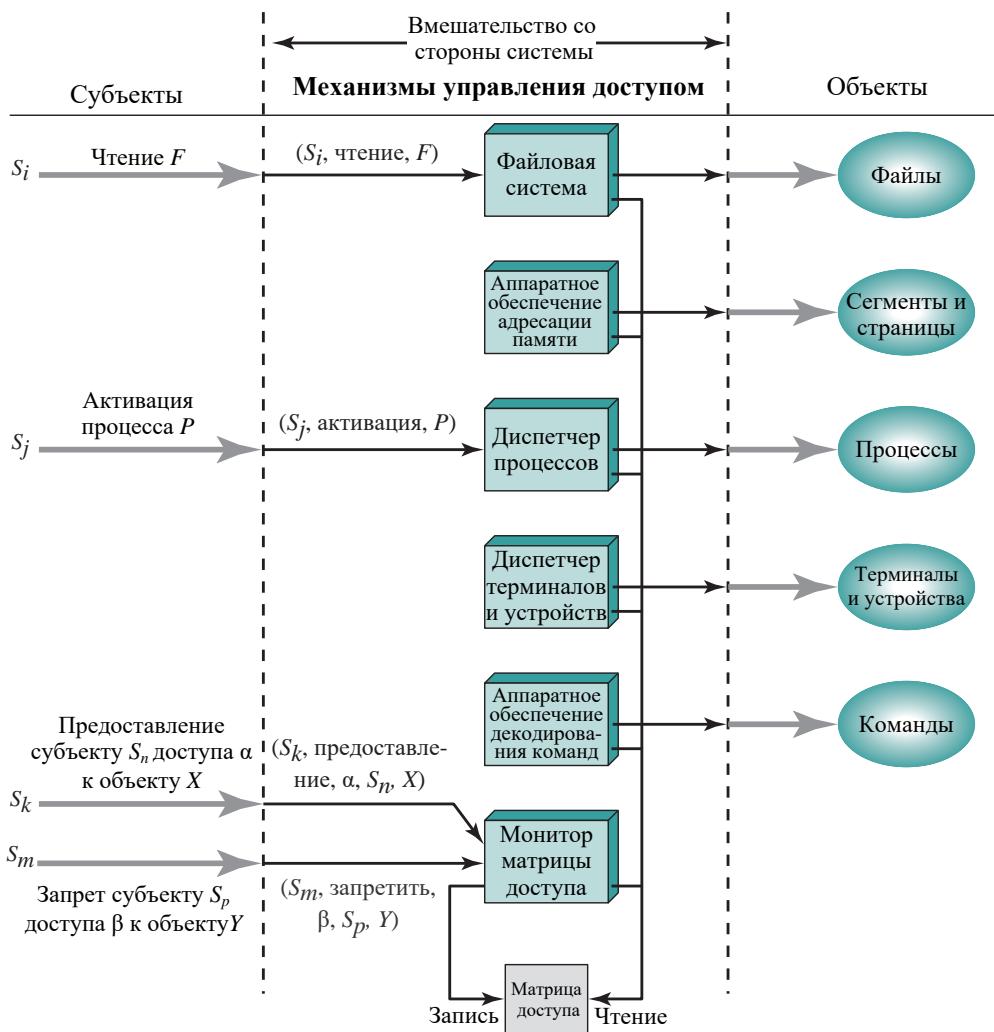


Рис. 15.5. Организация функционирования управления доступом

Первые три правила регулируют порядок передачи, предоставления и удаления прав доступа. Допустим, что в элементе матрицы  $A[S_0, X]$  содержится величина  $\alpha^*$ . Это означает, что у субъекта  $S_0$  имеется право доступа  $\alpha$  к объекту  $X$ , которое может быть предоставлено другому субъекту вместе с признаком копирования, обозначенным знаком \*. Такую возможность выражает правило R1. Субъект передаст право доступа без признака копирования, если возникнет подозрение, что новый субъект злонамеренно передаст это право другому субъекту, который не должен им обладать. Например, субъект  $S_1$  может разместить право доступа *чтение* или *чтение\** в любом элементе, находящемся в столбце  $F_1$  матрицы доступа. Правило R2 устанавливает, что если субъект  $S_0$  назначен в качестве владельца объекта  $X$ , то он может предоставить право доступа к данному объекту любому другому субъекту. Правило R2 также устанавливает, что субъект  $S_0$  может ввести любое право доступа в элемент матрицы  $A[S, X]$  для любого субъекта  $S$ , если он имеет доступ владельца к объекту  $X$ . Правило R3 разрешает субъекту  $S_0$  удалить

любое право доступа из любого элемента в той строке матрицы прав доступа, для которой субъект  $S_0$  управляет другим субъектом, а также для любого элемента матрицы в том столбце матрицы прав доступа, в котором субъект  $S_0$  владеет объектом. Правило R4 разрешает субъекту прочитать ту часть матрицы прав доступа, которой он владеет или управляет.

Остальные правила в табл. 15.1 регулируют порядок создания и удаления субъектов и объектов. Правило R5 устанавливает, что любой субъект может сначала создать новый объект, которым он будет владеть, а затем предоставить или запретить к нему доступ. Согласно правилу R6 владелец объекта может его уничтожить, что приведет к удалению соответствующего столбца из матрицы прав доступа. Правило R7 разрешает любому субъекту создать новый субъект и владеть им; новый субъект получает управление доступом к самому себе. И наконец, правило R8 разрешает владельцу субъекта удалить строку и столбец из матрицы доступа, если в ней имеются столбцы, отмеченные данным субъектом.

Правила, приведенные в табл. 15.1, служат примером определения ряда правил для системы управления доступом. Далее приведены примеры дополнительных и альтернативных правил, которые можно включить в эту таблицу. В частности, можно определить правило только для передачи прав доступа, в результате выполнения которого передаваемое право добавляется к целевому субъекту и удаляется из передающего субъекта. Число владельцев объекта или субъекта может быть ограничено одним, не позволяя флагу копирования сопутствовать праву владения.

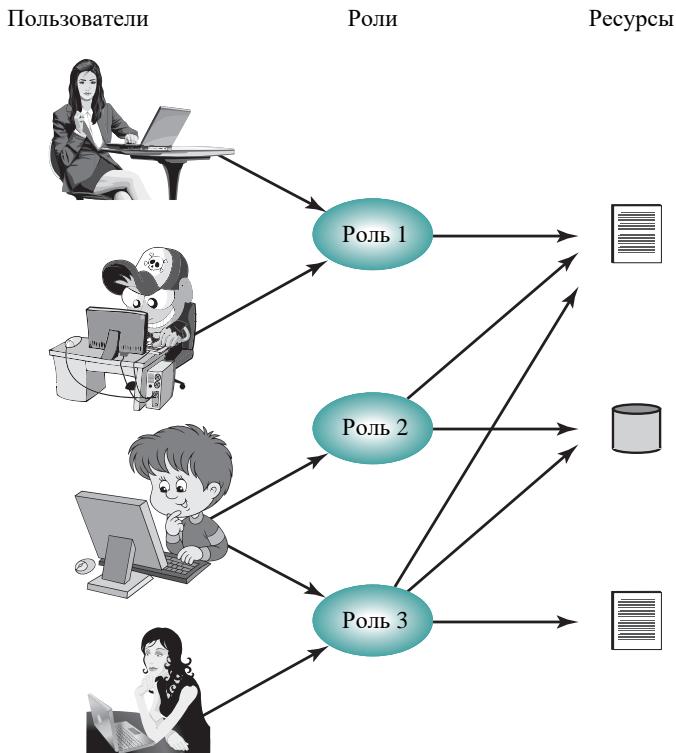
Способностью одного субъекта создавать другой субъект с правом доступа *владельца* к этому субъекту можно воспользоваться для определения иерархии субъектов. Так, на рис. 15.4 показано, что субъект  $S_1$  владеет субъектами  $S_2$  и  $S_3$ , поэтому они ему подчинены. По правилам из табл. 15.1 субъект  $S_1$  может как предоставить субъекту  $S_2$  уже имеющиеся у него права доступа, так и отменить их. Таким образом, один субъект может создать другой субъект с подмножеством своих прав доступа. Это может быть удобно, например, в том случае, если субъект вызывает приложение, не заслуживающее полного доверия, и поэтому нежелательно, чтобы приложение было в состоянии передавать права доступа другим субъектам.

## RBAC

В традиционных системах DAC определяются права доступа отдельных пользователей и их групп. А в системах RBAC управление доступом основывается на ролях, допустимых для пользователей в системе, а не на их личности. Как правило, роль в моделях RBAC определяется как должностная функция в организации. Системы RBAC присваивают права доступа ролям, а не отдельным пользователям. В свою очередь, пользователям присваиваются (как статически, так и динамически) разные роли в соответствии с их ответственностью.

RBAC в настоящее время широко распространен в коммерческой сфере, хотя и остается областью активных исследований. Национальный институт стандартов и технологий США (NIST) выпустил стандарт *Security Requirements for Cryptographic Modules* (Требования к безопасности криптографических модулей; документ FIPS PUB 140-2, от 25 мая 2001 года), в котором изложены требования к поддержке управления доступом и администрированию посредством ролей.

Отношение пользователей к ролям представляет собой отношение “многие ко многим”, как и отношение ролей к ресурсам или системным объектам (рис. 15.6). В некоторых средах состав пользователей часто меняется, а роли (одна или несколько) могут присваиваться пользователю динамически. Состав ролей в системе, вероятнее всего, остается устойчивым в большинстве сред и дополняется или сокращается лишь изредка. Каждая роль наделяется конкретными правами доступа к одному или более ресурсам. А состав ресурсов и конкретные права доступа к ним, связанные с отдельной ролью, вряд ли будут изменяться часто.



**Рис. 15.6.** Пользователи, роли и ресурсы

Чтобы объяснить основные элементы системы RBAC попроще, можно воспользоваться представлением матрицы прав доступа, как показано на рис. 15.7. Верхняя матрица на этом рисунке обозначает отношение пользователей к ролям. Как правило, пользователей намного больше, чем ролей. Каждый элемент этой матрицы оказывается пустым или помеченным “крестиком”, обозначающим, что данному пользователю присвоена конкретная роль. Следует, однако, иметь в виду, что одному пользователю может быть присвоено несколько ролей, на что указывает наличие нескольких отметок в столбце матрицы. Нижняя матрица на рис. 15.7 имеет такую же структуру, как и матрица DAC. Как правило, в системе имеется не очень много ролей и много объектов или ресурсов. Элементы матрицы представляют конкретные права доступа, которыми обладают роли. Следует также иметь в виду, что роль может трактоваться как объект, что позволяет определить иерархии ролей.

RBAC поддается эффективной реализации принципа наименьшей привилегии. Это означает, что каждая роль должна содержать минимальный набор правил доступа, необходимых данной роли. Пользователю присваивается роль, позволяющая ему выполнять только то, что требуется для данной роли. Многие пользователи, которым присвоена одна и та же роль, обладают одним и тем же минимальным набором прав доступа.

	R <sub>1</sub>	R <sub>2</sub>	• • •	R <sub>n</sub>
U <sub>1</sub>	✗			
U <sub>2</sub>	✗			
U <sub>3</sub>		✗		✗
U <sub>4</sub>				✗
U <sub>5</sub>				✗
U <sub>6</sub>				✗
•				
•				
•				
U <sub>m</sub>	✗			

		Объекты								
		R <sub>1</sub>	R <sub>2</sub>	R <sub>n</sub>	F <sub>1</sub>	F <sub>1</sub>	P <sub>1</sub>	P <sub>2</sub>	D <sub>1</sub>	D <sub>2</sub>
Роли	R <sub>1</sub>	Управление доступом	Владелец	Владелец управляет доступом	Чтение *	Чтение владельца	Активация	Активация	Поиск	Владелец
	R <sub>2</sub>		Управление доступом		Запись *	Выполнение			Владелец	Поиск *
	•									
	•									
	R <sub>n</sub>			Управление доступом		Запись	Остановка			

Рис. 15.7. Представление RBAC в виде матрицы прав доступа

## 15.4. УПРАВЛЕНИЕ ДОСТУПОМ В UNIX

В этом разделе описываются особенности реализации управления доступом в UNIX.

### Традиционное управление доступом к файлам в UNIX

Большинство систем UNIX зависят от схемы управления доступа к файлам, внедренной в первых версиях UNIX, или по крайней мере основываются на ней. Каждому пользователю UNIX присваивается уникальный идентификационный номер пользователя (идентификатор пользователя). Этот пользователь является также членом первичной группы, а возможно, и других групп, каждая из которых определена с помощью идентификатора группы. Когда создается файл, он обозначается как принадлежащий конкретному пользователю и помечается идентификатором данного пользователя. Он также принадлежит конкретной группе, которая первоначально является первичной группой его создателя или группой его родительского каталога, если для этого каталога установлены полномочия SetGID. С каждым файлом связано множество из 12 бит защиты. Идентификаторы пользователя и группы, а также биты защиты являются частью индексного узла (inode) файла.

Девять бит защиты обозначают полномочия для чтения, записи и выполнения для владельца файла, других членов группы, которым принадлежит данный файл, а также для всех остальных пользователей. Они образуют иерархию из владельца файла, группы и всех остальных пользователей с рядом соответствующих полномочий. В примере на рис. 15.8, *a* владелец файла обладает правом доступа для чтения и записи, все остальные члены группы, которой принадлежит данный файл, — правом доступа для чтение, а пользователи вне этой группы не имеют никаких прав доступа к данному файлу. Применительно к каталогу биты полномочий чтения и записи предоставляют право на вывод списка и создание, переименование и удаление файлов в данном каталоге.<sup>9</sup> А бит полномочий для выполнения предоставляет право на поиск в каталоге по компоненту имени файла.

Остальные три бита определяют специальное дополнительное поведение файлов и каталогов. Два из этих битов обозначают полномочия установки идентификатора пользователя во время выполнения (SetUID) и установки идентификатора группы (SetGID). Если они установлены для выполнимого файла, то, когда пользователь (с правами выполнения данного файла) выполняет запуск этого файла, система временно наделяет данного пользователя правами, которыми обладает создатель файла (или, соответственно, группа, владеющая данным файлом). Эти так называемые “действующий идентификатор пользователя” и “действующий идентификатор группы” используются в дополнение к “реальному идентификатору пользователя” и “реальному идентификатору группы”, когда программа принимает решение о предоставлении пользователю тех или иных прав. Эта функциональная возможность действует только во время выполнения программы и позволяет создавать и пользоваться привилегированными программами, в которых мо-

---

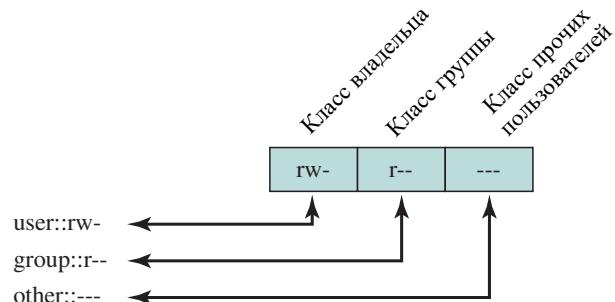
<sup>9</sup> Полномочия, распространяющиеся на каталог, отличаются от полномочий, распространяющихся на любой находящийся в нем файл или каталог. Если пользователь имеет право на запись в каталоге, это еще не дает ему право на запись в файл, находящийся в данном каталоге. Такое право регулируется полномочиями для конкретного файла. Тем не менее у пользователя может быть право на переименование файла.

гут применяться файлы, обычно недоступные другим пользователям. Применительно к каталогу, полномочия SetGID обозначают, что вновь созданные файлы унаследуют группу от данного каталога, тогда как полномочия SetUID игнорируются.

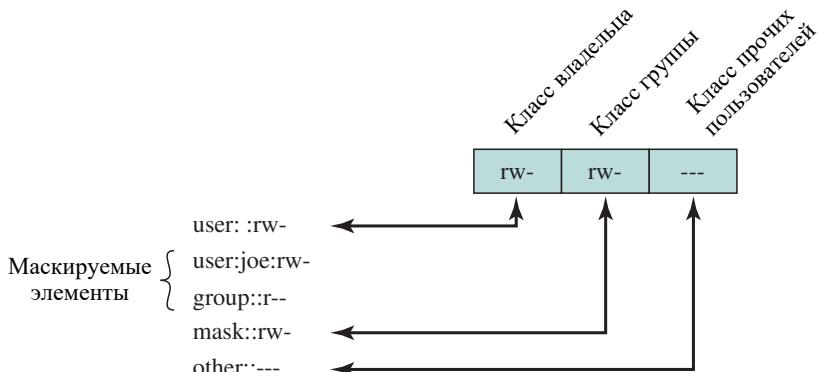
Последним специальным битом является бит “закрепления” (“липкий”, sticky). Будучи установленным для файла, этот бит изначально означал, что система должна сохранить содержимое файла в оперативной памяти после его выполнения. И хотя этот бит больше не используется, применительно к каталогу он обозначает, что только владелец любого файла в каталоге может переименовать, переместить или удалить данный файл. Это удобно для работы с файлами в общих временных каталогах.

Один особый идентификатор пользователя обозначает его как *привилегированного пользователя* (*суперпользователя*). Такой пользователь освобождается от обычных ограничений на управление доступом к файлам и имеет полный доступ ко всей системе. Любая программа, которой владеет привилегированный пользователь, с установленными полномочиями SetUID, потенциально предоставляет неограниченный доступ к системе любому пользователю, выполняющему эту программу. Поэтому при написании подобных программ следует проявлять особую осторожность.

Такая схема доступа оказывается пригодной в том случае, если требования доступа к файлам согласовываются с потребностями пользователей и небольшого количества их групп.



а) Традиционный подход UNIX (минимальный список управления доступом)



б) Расширенный список управления доступом

**Рис. 15.8.** Управление доступом к файлам в UNIX

Допустим, что какому-нибудь пользователю требуется наделить пользователей А и В правами чтения файла Х, а пользователей В и С — правами чтения файла У. Для этого потребуются как минимум две группы пользователей, причем пользователь В должен принадлежать обеим группам, чтобы получить доступ к обоим файлам. Но если имеется большое число различных групп пользователей, которым требуется целый ряд прав доступа к разным файлам, то для предоставления таких прав потребуется большое количество групп. Такая ситуация быстро станет трудно управляемой и выйдет из-под контроля, если вообще возможна.<sup>10</sup> В качестве выхода из столь затруднительного положения можно, например, воспользоваться списками управления доступом, предоставляемыми в большинстве современных систем UNIX. Наконец, следует заметить, что в традиционной для UNIX схеме управления доступом к файлам реализуется простая структура доменов защиты. Домен защиты связан с пользователем, а смена такого домена соответствует временному изменению идентификатора пользователя.

## Списки управления доступом в UNIX

Во многих современных UNIX и UNIX-подобных системах, в том числе в FreeBSD, OpenBSD, Linux и Solaris, поддерживаются списки управления доступом. В этом разделе описывается принятый в FreeBSD подход к поддержке подобных списков, хотя другие ее реализации обладают, по существу, такими же свойствами и интерфейсом. Рассматриваемое здесь функциональное средство называется расширенным списком управления доступом, тогда как в традиционном для UNIX подходе оно называется минимальным списком управления доступом.

Система FreeBSD разрешает администратору назначить файлу список идентификаторов пользователей и групп с помощью команды `setfacl`. С файлом может быть связано любое количество пользователей и групп, причем для каждой из этих категорий три бита защиты (чтения, записи, выполнения) предоставляют удобный механизм назначения прав доступа. Файл не обязан иметь список управления доступом (ACL) и может быть защищен только традиционным для UNIX механизмом доступа к файлам. Файлы в системе FreeBSD содержат дополнительный бит защиты, обозначающий, имеется ли у данного файла расширенный список ACL.

В FreeBSD и большинстве других реализаций UNIX, поддерживающих расширенные списки управления доступом, применяется следующая стратегия (рис. 15.8, б).

- Классы владельцев и другие классы в 9-разрядном поле полномочий, имеют то же назначение, что и в минимальном списке ACL.
- Класс групп определяет полномочия группы владельцев данного файла. Это максимальные полномочия, которые могут быть присвоены именованным пользователям или группам, отличным от владельца данного файла. В этой последней роли класс групп действует как маска.
- С файлом с помощью 3-разрядного поля полномочий могут быть связаны дополнительные именованные пользователи и группы. Полномочия, перечисляемые для именованного пользователя или группы, сравниваются с полем маски. Любые полномочия для именованного пользователя или группы, отсутствующие в поле маски, запрещены.

<sup>10</sup> В большинстве систем UNIX накладывается ограничение на максимальное количество групп, к которым может принадлежать любой пользователь, а также на общее количество групп, которые могут действовать в системе.

Когда процесс запрашивает доступ к объекту файловой системы, выполняются два шага. На первом шаге выбирается элемент списка ACL, наиболее точно соответствующий запрашиваемому процессу. Элементы списка ACL рассматриваются в следующем порядке: владелец, именованные пользователи, владеющие или именованные группы и прочие пользователи. Доступ определяет лишь один такой элемент. На втором шаге проверяется, содержит ли совпадающий элемент достаточные полномочия. Процесс может быть членом более чем одной группы, поэтому совпасть может не один элемент. Если же любой из совпадающих элементов групп содержит запрашиваемые полномочия, то выбирается именно он (результат будет таким же, какой бы совпавший элемент ни был выбран). Если ни один из совпадающих элементов групп не содержит запрашиваемые полномочия, то доступ будет запрещен, какой бы из этих элементов ни был выбран.

## 15.5. УСИЛЕНИЕ ЗАЩИТЫ ОПЕРАЦИОННЫХ СИСТЕМ

Первый важный шаг в усилении защиты состоит в том, чтобы защитить базовую операционную систему, на которую полагаются все остальные приложения и службы. Чтобы заложить прочное основание под безопасность, требуется надлежащим образом установленная, обновленная с помощью последних исправлений и сконфигурированная операционная система. К сожалению, в стандартной конфигурации для многих операционных систем зачастую основной акцент делается на простоте применения и функционирования, а не на безопасности. А поскольку у каждой организации свои потребности в безопасности, то у нее и свой особый профиль, а значит, и конфигурация системы защиты. Конкретные требования к безопасности системы должны быть поставлены на стадии планирования, как только что обсуждалось.

Общий подход к усилению защиты операционных систем остается одинаковым, несмотря на различия в подробностях реализации. Для большинства самых распространенных операционных систем существуют руководства и перечни действий по конфигурированию, и поэтому рекомендуется обращаться к ним, но при этом следует всегда учитывать конкретные потребности каждой организации и ее систем. В некоторых случаях оказать дополнительную помощь в усилении защиты системной конфигурации могут автоматизированные инструментальные средства.

В [177] предлагаются следующие основные шаги, которые необходимо предпринять для усиления защиты операционной системы.

- Установить операционную систему и все имеющиеся для нее обновления.
- Усилить и сконфигурировать операционную систему для надлежащего удовлетворения потребностей в безопасности, выполнив перечисленные ниже действия.
  - Удалить ненужные службы, приложения и протоколы.
  - Сконфигурировать пользователей, группы и полномочия.
  - Сконфигурировать средства управления ресурсами.
  - Установить и сконфигурировать дополнительные средства управления защитой, брандмауэры и системы обнаружения вторжений.
  - Проверить защиту базовой операционной системы, чтобы убедиться, что шаги, предпринятые по усилению ее защиты, надлежащим образом удовлетворяют потребности в ее безопасности.

## Установка операционной системы и применение обновлений

Системная безопасность начинается с установки операционной системы. Как отмечалось ранее, подключенная к сети и не обновленная с использованием последних обновлений операционная система уязвима к внешним атакам как на этапе установки, так и на этапе обычной эксплуатации. Именно поэтому так важно, чтобы система не оставалась незащищенной, находясь в уязвимом состоянии. В идеальном случае новые системы должны настраиваться в защищенной сети. Это должна быть полностью изолированная сеть, в которую образ операционной системы и все имеющиеся обновления перенесены на сменных носителях, подобных DVD-дискам и USB-накопителям. Учитывая существование вредоносных программ, которые могут распространяться и на сменных носителях, следует принять меры предосторожности против заражения применяемых носителей. С другой стороны, может быть использована сеть со строго ограниченным доступом к Интернету. В идеальном случае входящий доступ в такой сети должен быть запрещен, а исходящий доступ разрешен только к тем сайтам, которые требуются в процессе установки системы и ее обновления. В любом случае процесс полной установки и усиления защиты должен происходить до ее развертывания в предполагаемом более доступном, а следовательно, и более уязвимом месте.

Первоначальная установка должна состоять из самого необходимого минимума, требующегося для работы системы, а дополнительные программные пакеты должны быть включены в нее только в том случае, если они требуются для нормального функционирования системы.

Весь процесс начальной загрузки также должен быть защищен. Для этого может потребоваться настройка соответствующих параметров или задание пароля, требующегося для изменения кода BIOS, с помощью которого выполняется начальная загрузка системы. Возможно, придется также ограничить число носителей, с которых обычно разрешается загружать систему. Эта мера необходима для того, чтобы не дать атакующему злоумышленнику возможности изменить процесс начальной загрузки, чтобы установить скрытый гипервизор или просто выполнить начальную загрузку избранной им системы с внешнего носителя с целью обойти обычные средства управления доступом к системе, хранящиеся на локальных запоминающих устройствах. Как отмечается далее, для предотвращения такой угрозы можно также воспользоваться зашифрованной файловой системой.

Кроме того, следует внимательно отнестись к выбору и установке кода драйверов любых дополнительных устройств, поскольку такой код выполняется с полными привилегиями на уровне ядра системы, но зачастую поставляется сторонними производителями. Целостность и источник такого кода драйверов следует тщательно проверять, учитывая высокую степень доверия к нему. Зловредный драйвер способен обойти многие средства управления защитой, чтобы установить вредоносную программу. Принимая во внимание постоянно выявляющиеся программные и прочие уязвимости в наиболее распространенных операционных системах и приложениях, крайне важно в как можно большей степени поддерживать систему в актуальном состоянии, устанавливая имеющие решающее значение для ее безопасности обновления. Почти во всех наиболее распространенных сегодня системах предоставляются утилиты для автоматической загрузки и установки обновлений для повышения безопасности. Эти инструментальные средства

должны быть правильно сконфигурированы и использованы, чтобы свести к минимуму время, в течение которого в системе остаются уязвимыми слабые места.

Следует, однако, иметь в виду, что в некоторых системах с управлениями изменениями не следует выполнять обновления в автоматическом режиме, поскольку обновления для повышения безопасности могут в редких, но важных случаях вносить неустойчивость в работу системы. В тех системах, в которых доступность и постоянная готовность к работе имеют первостепенное значение, все обновления, прежде чем развертывать их на месте эксплуатации, следует устанавливать и проверять в тестовых системах.

## Удаление ненужных служб, приложений и протоколов

Любое программное обеспечение, выполняющееся в системе, может содержать уязвимости, поэтому очевидно, что чем меньше установлено программных пакетов, тем меньше риск злоупотребления их уязвимостями. Очевидно, что существует определенное равновесие между удобством, обеспечиваемым установкой всего программного обеспечения, которое может в какой-то момент потребоваться, и безопасностью и стремлением ограничить количество установленного программного обеспечения. Разнообразие служб, приложений и протоколов, требующихся как в самих организациях, так и в системах, которые в них эксплуатируются, изменяется в широких пределах. Поэтому в процессе планирования системы следует выяснить, что фактически требуется для ее работы, с тем чтобы обеспечить функционирование на подходящем уровне безопасности и исключить из нее все, что не пригодится для усиления защиты операционной системы.

В стандартной конфигурации большинства распределенных систем основной акцент делается на простоту применения и функционирования, а не на безопасность. Поэтому первоначальную установку системы следует выполнять не по умолчанию, а в режиме специальной настройки, чтобы выбрать для установки только требующиеся пакеты. Если впоследствии потребуются дополнительные пакеты, их можно будет установить по мере надобности. В [177] и многих других руководствах по усилению защиты представляются перечни служб, приложений и протоколов, которые не стоит устанавливать, если они требуются.

Кроме того, в [177] заявляется о предпочтении решения не устанавливать ненужное программное обеспечение, вместо того чтобы сначала установить, а затем удалить его или отключить. Такое предпочтение объясняется тем, что многие сценарии удаления оказываются не в состоянии удалить полностью все компоненты установленного пакета. А отключение службы означает, что даже если она и не доступна как первоначальное место совершения атаки, атакующий злоумышленник, добившись успеха в получении доступа к системе, может активировать эту отключенную службу и воспользоваться ею, чтобы продолжить нарушение безопасности системы. Поэтому в целях безопасности лучше не устанавливать ненужное программное обеспечение, которым в таком случае злоумышленник никак не сможет воспользоваться.

## Конфигурирование пользователей, групп и аутентификации

Не все пользователи, которым доступна система, будут иметь одинаковый доступ ко всем данным и ресурсам, присутствующим в ней. Во всех современных операционных

системах реализованы средства управления доступом к данным и ресурсам. И почти во всех из них предоставляется некоторая форма средств избирательного управления доступом. В некоторых системах могут также предоставляться механизмы ролевого или полномочного управления доступом.

В процессе планирования системы следует рассмотреть категории пользователей в системе, их привилегии, типы доступной им информации, а также порядок и место их определения и аутентификации. Одни пользователи будут иметь повышенные привилегии для администрирования системы, а другие — полномочия обычных пользователей, совместно пользующихся общим доступом к файлам и другим данным по мере надобности. Возможны даже гостевые учетные записи с весьма ограниченным доступом. Третья из четырех главных стратегий снижения риска в системе управлении военной связью состоит в ограничении круга пользователей, которым требуются повышенные привилегии. Кроме того, крайне желательно, чтобы такие пользователи получали повышенные привилегии лишь на то время, когда они необходимы для выполнения определенных задач, а в остальном имели права доступа обычных пользователей. Это повышает безопасность системы, сужая возможности атакующего злоумышленника для злоупотреблений действиями от имени привилегированных пользователей. В некоторых операционных системах представляются специальные инструментальные средства или механизмы доступа, помогающие администраторам повышать уровень своих привилегий по мере надобности и надлежащим образом протоколировать такие действия.

Одно из ключевых принимаемых администраторами решений — как определять пользователей, группы, к которым они принадлежат, и методы их аутентификации: локально в системе или же с помощью централизованного сервера аутентификации. Какой бы из этих вариантов ни был выбран, соответствующие детали их реализации теперь конфигурируются в системе.

На данной стадии должны быть также защищены любые учетные записи, создаваемые по умолчанию при установке системы. Те из них, которые не требуются для работы, должны быть удалены или хотя бы отключены. Системные учетные записи, управляемые службами в системе, должны быть установлены таким образом, чтобы ими нельзя было пользоваться при регистрации в диалоговом режиме. Любые устанавливаемые по умолчанию пароли должны быть заменены новыми значениями с надлежащей защитой.

Любые стратегии, применяемые к аутентификации и особенно к защите паролей, также подлежат конфигурированию. Сюда относится как выбор методов аутентификации, пригодных для различных методов доступа по учетной записи, так и такие детали, как минимальная длина, сложность и допустимый срок действия паролей.

## Конфигурирование средств управления ресурсами

Как только пользователи и их группы будут определены, для доступа к данным и ресурсам могут быть установлены полномочия, согласующиеся с определенными стратегиями. Это может быть сделано с целью ограничить права пользователей на выполнение программ, в особенности тех, которые модифицируют состояние системы, или же права чтения или записи данных в некоторых деревьях каталогов. Во многих из наставлений по усилению защиты системы представляются списки рекомендуемых изменений в стандартной конфигурации доступа для совершенствования безопасности.

## Установка дополнительных средств управления защитой

В еще большей степени усовершенствовать защиту системы можно, установив и сконфигурировав дополнительные инструментальные средства обеспечения безопасности, в том числе антивирусное программное обеспечение, бандмауэры, программное обеспечение обнаружения или предотвращения вторжений или ведения списков разрешенных приложений. Одни из этих средств могут быть предоставлены при установке операционной системы, но по умолчанию не быть сконфигурированы и активизированы; другие являются сторонними программными продуктами, которые приобретаются отдельно.

Принимая во внимание распространенность зловредного программного обеспечения, подходящий антивирус, который обнаруживает обширный ряд вредоносных программ, является крайне важной составляющей защиты во многих системах. Антивирусные программы традиционно применялись в системах Windows (которые в силу своей распространенности стали предпочтительной мишенью для атак злоумышленников). Развитие других платформ, особенно смартфонов, привело к тому, что и для них стало разрабатываться все больше и больше вредоносных программ. Таким образом, антивирусные программы должны считаться неотъемлемой частью профиля защиты любой системы.

Бандмауэры, программное обеспечение систем обнаружения и предотвращения вторжений также способны усовершенствовать защиту, ограничивая удаленный сетевой доступ к службам в системе. Если же удаленный доступ к системе не требуется, а нужен только определенный локальный доступ, то такие ограничения помогают защитить систему от удаленного злоупотребления атакующим злоумышленником. По традиции бандмауэры настраиваются на ограничение доступа через порт или сетевой протокол из некоторых или всех внешних систем. Некоторые из них могут быть также настроены на разрешение доступа от отдельных программ или же к конкретным программам в системах, чтобы еще больше локализовать доступные для атак места и не позволить атакующему злоумышленнику устанавливать свои вредоносные программы и получать к ним доступ. В состав программного обеспечения систем обнаружения и предотвращения вторжений могут входить дополнительные механизмы текущего контроля сетевого трафика или проверки целостности файлов, чтобы выявлять некоторые виды атак и реагировать на них.

Еще одним средством управления защитой является ведение списков разрешенных приложений. Таким образом, круг программ, которые можно выполнять в системе, ограничивается только указанными в точном списке. Такое средство способно помешать атакующему злоумышленнику устанавливать и выполнять свои вредоносные программы, и поэтому оно является последней из четырех главных стратегий снижения риска в системе. Хотя данное средство и совершенствует защиту, лучше всего оно функционирует в среде с предсказуемым набором приложений, которые требуются пользователям. Любые изменения в использовании программного обеспечения потребуют изменений в конфигурации, что может привести к повышению требований к его информационно-технологической поддержке. Не все организации и системы в достаточной степени прогнозируемые, чтобы отвечать данному типу управления защитой.

## Тестирование защиты системы

На последней стадии процесса первоначальной защиты базовой операционной системы производится тестирование ее защищенности. Его цель — убедиться, что предыдущие стадии конфигурирования защиты реализованы правильно, а также выявить любые возможные уязвимости, которые должны быть устраниены.

Подходящие перечни действий можно найти во многих руководствах по усилению защиты систем. Имеются также программы, специально предназначенные для критического обзора системы с целью убедиться, что она удовлетворяет основным требованиям к защите, и проверить ее на наличие известных уязвимостей и неудачного конфигурирования. Это должно быть сделано непосредственно после первоначального усиления защиты системы и периодически повторяться как один из этапов процесса поддержания безопасности.

# 15.6. Поддержание безопасности

Как только система надлежащим образом построена, защищена и развернута, начинается процесс непрерывного поддержания ее безопасности. Это необходимо в силу постоянного изменения рабочей среды, выявления новых уязвимостей, а следовательно, и подверженности новым угрозам. В [177] предлагается, чтобы этот процесс поддержания безопасности включал в себя следующие дополнительные шаги.

- Текущий контроль и анализ протоколируемой информации.
- Регулярное выполнение резервного копирования.
- Устранение нарушений правил безопасности.
- Регулярное тестирование защиты системы.
- Выполнение надлежащих процессов для сопровождения программного обеспечения по обновлению всего ответственного программного обеспечения, а также для текущего контроля и пересмотра конфигурации по мере надобности.

Ранее уже отмечалась необходимость настройки автоматического обновления везде, где это возможно, или ручного тестирования и установки обновлений в системах, в которых автоматическое обновление нежелательно. Упоминалось также, что система должна регулярно проверяться с использованием соответствующего перечня проверок или с помощью автоматизированных инструментальных средств, где это возможно.

## Протоколирование

Как отмечается в [177], “протоколирование является краеугольным камнем безопасности”. Протоколирование является средством контроля, способным лишь извещать о неполадках, уже возникших в системе. Но эффективное протоколирование в случае взлома или отказа системы помогает ее администраторам быстрее и точнее выяснить, что же произошло, а следовательно, эффективно сосредоточить свои усилия на исправлении и устранении возникших неполадок. Самое главное — убедиться, что в журналах регистрации зафиксированы корректные данные, а затем надлежащим образом проанализировать эти данные, которые могут генерироваться системой, сетью и приложениями. Разнообразие данных, собираемых при протоколировании, должно быть определено

на стадии планирования системы, поскольку оно зависит от требований к защите и информационной закрытости сервера.

При протоколировании могут быть сгенерированы значительные объемы информации, поэтому очень важно выделить для их хранения достаточное место. Должна быть также настроена подходящая система автоматической ротации и архивирования журналов регистрации, что обеспечит упорядочение всего объема протоколируемой информации.

Для обнаружения неблагоприятных событий ручной анализ журналов регистрации трудоемок и ненадежен. Вместо этого лучше использовать какой-нибудь автоматизированный анализ, поскольку он с большей вероятностью позволит выявить подозрительные действия в системе.

## Резервное копирование и архивирование данных

Регулярное резервное копирование данных в системе является еще одним очень важным средством, помогающим поддерживать целостность системных и пользовательских данных. Имеется немало причин, по которым данные в системе могут быть утрачены, включая аппаратные или программные сбои, случайное или намеренное повреждение. Существуют также юридические законные или производственные требования к сохранению данных. **Резервное копирование** (backup) — это процесс создания копий данных через регулярные промежутки времени, позволяющий восстановить потерянные или поврежденные данные в течение относительно коротких периодов времени, от нескольких часов до нескольких недель. **Архивирование** (archive) — это процесс сохранения копий данных в течение продолжительных периодов времени, насчитывающих месяцы или годы, чтобы удовлетворить юридическим или производственным требованиям доступа к старым данным. Эти процессы нередко связаны и организованы вместе, хотя и удовлетворяют разные потребности.

Нужды и стратегии, касающиеся резервного копирования и архивирования, должны быть определены на стадии планирования системы. К ключевым решениям, принимаемым на этой стадии, относятся следующие: как хранить резервные копии — в сети или автономно, локально или в удаленном месте. К компромиссным решениям относятся выбор между простотой реализации и затратами по сравнению с большей безопасностью и надежной защитой от различных угроз.

Наглядным примером последовательности неудачно выбранных вариантов хранения данных стала атака на австралийского хостинг-провайдера в 2011 году. Атаковавшие злоумышленники уничтожили не только оригиналы тысяч клиентских сайтов, но и все доступные в сети их резервные копии. В итоге многие клиенты, не хранившие резервные копии своих сайтов, потеряли все их содержимое и данные, что повлекло серьезные последствия не только для них, но и для провайдера. Другими примерами могут служить многие организации, хранившие резервные копии только в одном месте и потерявшие все свои данные в результате пожара или затопления информационного центра. Подобные риски должны быть оценены надлежащим образом.

## 15.7. БЕЗОПАСНОСТЬ WINDOWS

Характерным примером воплощения рассмотренных ранее понятий является устройство управления доступом в Windows, где для предоставления эффективной и удобной возможности управления доступом применяются объектно-ориентированные кон-

цепции. В операционной системе Windows предоставляется единообразное устройство управления доступом применительно к процессам, потокам выполнения, файлам, семафорам, окнам и прочим объектам. Управление доступом регулируется двумя элементами: токеном доступа, связанным с каждым процессом, а также дескриптором безопасности, связанным с каждым объектом, к которому возможен доступ из разных процессов.

## Схема управления доступом

Когда пользователь входит в систему Windows, для его аутентификации в Windows используется схема “имя–пароль”. Если вход в систему принят, для пользователя создается процесс, а с процессом объекта связывается токен доступа. Токен доступа, более подробно описанный далее, включает в себя идентификатор безопасности (security ID — SID), который представляет пользователя системе для целей безопасности. Этот токен содержит также идентификаторы SID для тех групп безопасности, к которым принадлежит пользователь. Если первоначальный пользовательский процесс порождает новый процесс, то объект нового процесса наследует тот же самый токен доступа.

Токен доступа служит двум целям.

1. Хранит вместе всю необходимую информацию по безопасности, для ускорения проверки достоверности доступа. Когда любой процесс, связанный с пользователем, пытается получить доступ, подсистема защиты может использовать токен, связанный с данным процессом, чтобы определить пользовательские привилегии доступа.
2. Позволяет каждому процессу ограниченно модифицировать свои характеристики безопасности, не оказывая влияния на другие процессы, выполняющиеся от имени этого пользователя.

Особая важность второй цели связана с привилегиями, которые могут быть назначены пользователю. Токен доступа указывает привилегии, которыми может обладать пользователь. В общем случае при инициализации токена все привилегии оказываются в отключенном состоянии. Таким образом, если в одном из процессов пользователя требуется выполнить привилегированную операцию, этот процесс может включить подходящую привилегию и попытаться получить доступ. Было бы нежелательно совместно использовать один и тот же токен доступа всеми процессами пользователя — в этом случае включение привилегии для одного процесса означало бы ее включение и для всех остальных процессов.

С каждым объектом, к которому возможен доступ из разных процессов, связан дескриптор безопасности. Главной составляющей дескриптора безопасности является список управления доступом, в котором определяются права доступа к данному объекту для разных пользователей и их групп. Когда процесс пытается получить доступ к данному объекту, идентификаторы безопасности из токена доступа этого процесса сопоставляются со списком управления доступом к данному объекту, чтобы выяснить, разрешен или запрещен такой доступ.

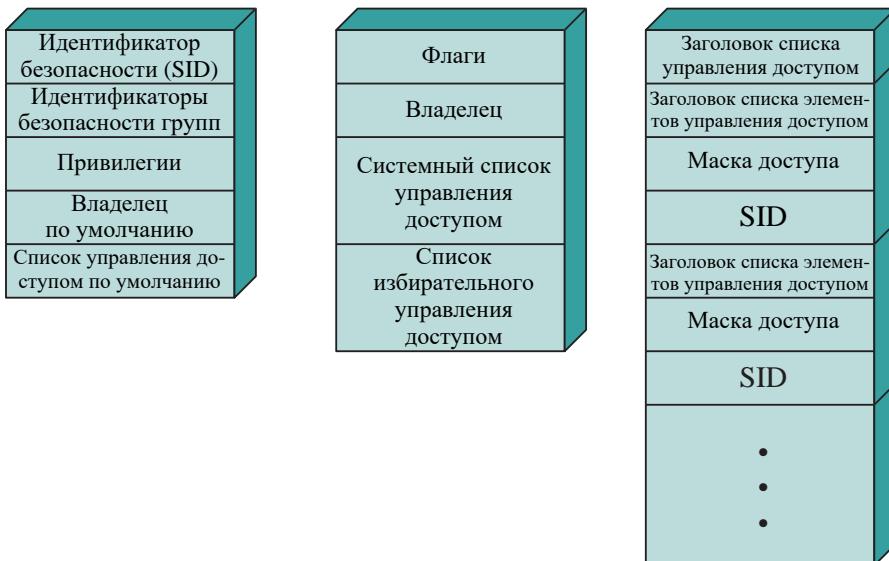
Когда приложение открывает ссылку на защищаемый объект, в Windows проверяется, предоставляет ли дескриптор безопасности объекта запрашиваемый доступ соответствующему процессу. Если такая проверка завершается удачно, Windows сохраняет в кеше предоставленные в итоге права доступа.

Важной особенностью безопасности Windows является понятие олицетворения (имперсонации), упрощающее обеспечение безопасности в среде “клиент/сервер”. Если клиент и сервер сообщаются через RPC-соединение (вызов удаленных процедур), то сервер может временно принять личность клиента, чтобы оценить запрос доступа по отношению к правам данного клиента. А после доступа сервер возвращается к собственной личности.

## Токен доступа

На рис. 15.9, а приведена общая структура токена доступа, которая включает в себя следующие параметры.

- **Идентификатор безопасности.** Уникальным образом идентифицирует пользователя на всех машинах сети и обычно соответствует регистрационному имени пользователя в системе. В версии Windows 7 были введены специальные идентификаторы безопасности для использования процессами и службами. Эти специальным образом управляемые идентификаторы предназначены для безопасного управления системой. В них не используются обычные стратегии паролей, как в учетных записях обычных пользователей.
- **Идентификаторы безопасности групп.** Это список групп, к которым принадлежит данный пользователь. Группа представляет собой множество идентификаторов пользователей, отождествляемых с данной группой для целей управления доступом. У каждой группы имеется свой уникальный идентификатор безопасности. Доступ к объекту может быть определен на основе идентификаторов безопасности групп, идентификаторов безопасности отдельных пользователей или определенного сочетания тех и других. Имеется также идентификатор безопасности, отражающий уровень целостности процессов (низкий, средний, высокий или системный).
- **Привилегии.** Это список чувствительных к безопасности системных служб, которые может вызывать данный пользователь (например, привилегия CreateToken для создания токена). Другим примером служит привилегия SetBackupPrivilege, обладая которой, пользователи могут выполнять с помощью специально предназначенного для этой цели инструментального средства резервное копирование файлов, которые им обычно не разрешено читать.
- **Владелец по умолчанию.** Если в одном процессе создается другой процесс, то в данном поле указывается владелец нового объекта. В общем случае владелец нового объекта оказывается тем же, что и у порождающего процесса. Тем не менее пользователь может указать, что владельцем по умолчанию любого процесса, запущенного данным, является идентификатор безопасности группы, к которой он принадлежит.
- **Список управления доступом по умолчанию.** Это изначальный список видов защиты, применяемых к объектам, создаваемым пользователем. Впоследствии пользователь может изменить список управления доступом к любому объекту, которым владеет он или его группы.



а) Токен доступа

б) Дескриптор безопасности

в) Список управления доступом

**Рис. 15.9.** Структуры безопасности Windows

## Дескрипторы безопасности

На рис. 15.9, б показана общая структура дескриптора безопасности, которая включает в себя следующие параметры.

- **Флаги.** Определяют тип и содержимое дескриптора безопасности. Они указывают наличие или отсутствие системного списка управления доступом (SACL) и списка избирательного управления доступом (DACL), размещаются ли они в объекте действующим по умолчанию механизмом, а также какая адресация применяется в указателях дескриптора — абсолютная или относительная. Относительные дескрипторы требуются для тех объектов, которые передаются по сети (например, для информации, передаваемой при удаленном вызове процедур).
- **Владелец.** Владелец объекта в общем случае может выполнить любое действие над дескриптором безопасности. Владельцем может быть отдельный пользователь или группа со своим идентификатором безопасности. Владелец обладает полномочиями изменить содержимое списка DACL.
- **Системный список управления доступом (SACL).** Указывает, какие именно виды операций над объектом должны формировать сообщения аудита. Чтобы читать или записывать список SACL для любого объекта, приложение должно обладать соответствующей привилегией в своем токене доступа. Благодаря этому несанкционированные приложения не могут читать списки SACL (чтобы узнать, чего нельзя делать, чтобы избежать генерации аудита) или записывать их (чтобы генерировать множество сообщений аудита и тем самым добиться того, чтобы выполнение запрещенных операций оказалось незамеченным). В списке SACL указывается также уровень целостности объекта. Процессы не могут изменить объект, если только

уровень целостности процесса не совпадает с соответствующим уровнем в данном объекте или не превышает его.

- **Список избирательного управления доступом (DACL).** Определяет, какие именно пользователи и группы могут иметь доступ к данному объекту и какие операции можно над ним выполнять. Он состоит из списка элементов управления доступом (ACE).

Когда создается объект, то создающий его процесс может присвоить ему свой идентификатор безопасности в качестве владельца или же идентификатор безопасности любой группы из своего токена доступа. Создающий процесс не может присвоить новому объекту владельца, отсутствующего в текущем токене доступа. Следовательно, любой процесс, которому предоставлено право изменения владельца объекта, может сделать это, но опять с тем же ограничением. Такое ограничение объясняется необходимостью помешать пользователю замести свои следы после попытки выполнить какое-нибудь несанкционированное действие.

Рассмотрим более подробно структуру списков управления доступом, поскольку они составляют ядро управления доступом в Windows (см. рис. 15.9, в). Каждый такой список состоит из общего заголовка и переменного числа элементов управления доступом, а каждый элемент определяет идентификатор безопасности отдельного пользователя или группы и маску доступа, указывающую права, предоставляемые данному идентификатору безопасности. Когда процесс пытается получить доступ к объекту, диспетчер объектов из модуля исполнительных служб Windows Executive читает идентификатор безопасности пользователя, а также идентификаторы безопасности групп из токена доступа наряду с идентификатором безопасности уровня целостности. Если запрашиваемый доступ включает модификацию объекта, уровень его целостности проверяется относительно уровня целостности объектов в списке SACL. Если эта проверка проходит успешно, диспетчер объектов просматривает список DACL данного объекта. И если обнаруживается совпадение, т.е. если найден элемент управления доступом с идентификатором безопасности, совпадающим с идентификаторами безопасности из токена доступа, то процесс может иметь права доступа, определяемые маской в данном элементе управления доступом. При этом маска может включать отказ в доступе, и тогда запрос доступа завершается неудачно. Результат проверки прав доступа определяется первым же совпадшим элементом управления доступом.

Содержимое маски доступа приведено на рис. 15.10. Младшие 16 бит в этой маске обозначают права доступа к определенному типу объекта. Например, нулевой бит для файлового объекта равен FILE\_READ\_DATA, а нулевой бит для объекта события — EVENT\_QUERY\_STATE.

Старшие 16 бит маски доступа обозначают права доступа ко всем типам объектов. Пять из них обозначают следующие типы стандартного доступа.

- **Synchronize.** Дает разрешение на синхронизацию выполнения с некоторым событием, связанным с данным объектом. В частности, данный объект может быть использован в функции ожидания.
- **Write\_owner.** Дает программе возможность сменить владельца объекта. Это удобно потому, что владелец объекта может всегда изменить его защиту. (Владельцу не может быть отказано в доступе Write\_DAC.)

- **Write\_DAC.** Предоставляет приложению возможность модифицировать список DACL, а следовательно, изменить защиту данного объекта.
- **Read\_control.** Предоставляет приложению возможность запросить поля владельца и списка DACL в дескрипторе безопасности данного объекта.
- **Delete.** Предоставляет приложению возможность удалить данный объект.

В старшей половине маски доступа содержатся также биты, обозначающие четыре типа обобщенного доступа. Эти биты представляют собой удобное средство установки конкретных типов доступа для большого количества разных типов объектов. Допустим, в приложении требуется создать несколько типов объектов и гарантировать пользователям доступ для чтения объектов, несмотря на то что для каждого типа объекта операция чтения имеет несколько иной смысл.

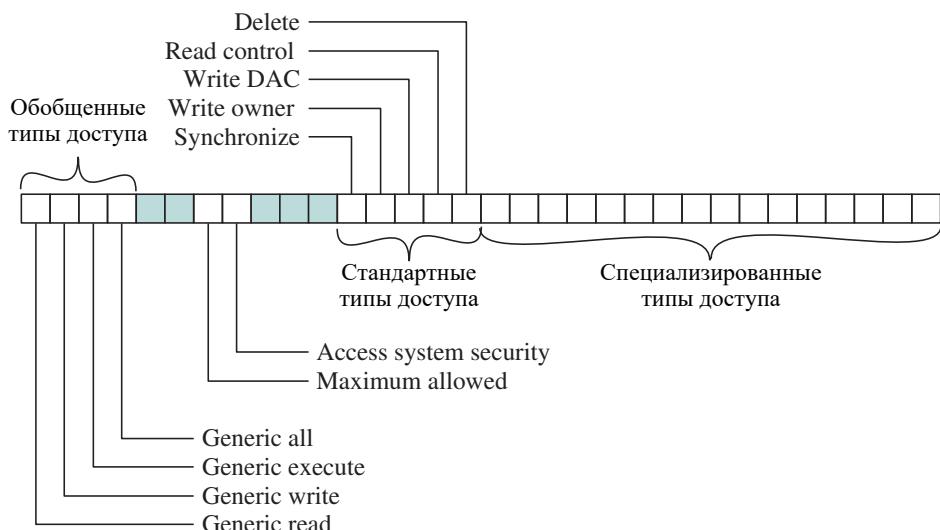


Рис. 15.10. Маски доступа

Чтобы защитить каждый объект каждого типа, без битов обобщенного доступа, в приложении пришлось бы создавать отдельный элемент управления доступом для каждого типа объекта и проявлять особую внимательность, передавая нужный элемент управления доступом при создании каждого объекта. Намного удобнее создать один элемент управления доступом, выражющий общий принцип “разрешения чтения” и применить этот элемент к каждому создаваемому объекту, чтобы все заработало как следует. Именно этой цели и служат следующие биты обобщенного доступа.

- **Generic\_all.** Разрешает все типы обобщенного доступа.
- **Generic\_execute.** Разрешает доступ для выполнения, если объект выполнимый.
- **Generic\_write.** Разрешает доступ для записи.
- **Generic\_read.** Разрешает доступ для чтения.

Биты универсального доступа оказывают также влияние на типы стандартного доступа. Например, для файлового объекта бит Generic\_Read соответствует битам Read\_Control и Synchronize стандартного доступа, а также битам File\_Read\_Data, File\_Read\_Attributes и File\_Read\_EA конкретного объекта. Если разместить элемент управления доступом для файлового объекта, предоставляющий права доступа Generic\_Read для некоторого идентификатора безопасности, то все эти пять прав доступа будут предоставлены так, как будто в маске доступа все их биты были установлены по отдельности.

Два оставшихся бита в маске доступа имеют специальное назначение. Бит Access\_System\_Security позволяет видоизменить аудит и управление предупреждениями для данного объекта. Однако недостаточно только установить этот бит в элементе управления доступом для конкретного идентификатора безопасности, — нужно также, чтобы токен доступа для процесса с этим идентификатором безопасности имел соответствующую привилегию.

И наконец, бит Maximum\_Allowed на самом деле является не битом доступа, а битом, который модифицирует алгоритм просмотра списка DACL для данного идентификатора безопасности. Как правило, Windows сканирует список DACL до тех пор, пока не будет найден элемент управления доступом, предоставляющий (бит установлен) или запрещающий (бит сброшен) запрашиваемый процессом доступ, или пока не будет достигнут конец списка DACL. В последнем случае доступ запрещен. Бит Maximum\_Allowed позволяет владельцу объекта определить набор прав доступа, которые представляют собой максимальный доступ, который может быть разрешен данному пользователю. С учетом этого предположим, что в приложении неизвестны все операции над объектом, выполнение которых может быть запрошено в течение сеанса работы приложения. В таком случае для запрашиваемого доступа имеются три следующие возможности.

1. Попытаться открыть объект для всех возможных видов доступа к нему. Недостаток такого подхода заключается в том, что в таком доступе может быть отказано, несмотря на то что у приложения могут быть все права доступа, фактически требующиеся для данного сеанса работы.
2. Открыть объект лишь в том случае, если требуется конкретный доступ, а для запроса любого другого типа доступа открыть новый дескриптор объекта. В общем случае это наиболее предпочтительный метод, поскольку он не запрещает доступ без всякой нужды и не разрешает больший доступ, чем нужно. Зачастую ссылаться на объект второй раз не требуется, а для создания копии дескриптора с более низким уровнем доступа можно воспользоваться функцией `DuplicateHandle()`.
3. Попытаться открыть объект для как можно большего доступа, который только объект может разрешить для данного идентификатора безопасности. Преимущество такого подхода заключается в том, что доступ клиентскому приложению не будет искусственно запрещен, но при этом оно может иметь больший доступ, чем нужно. Эта последняя ситуация может маскировать программные ошибки в приложении.

Важная особенность безопасности Windows состоит в том, что в приложениях можно использовать инфраструктуру безопасности Windows для определяемых пользователем объектов. Например, на сервере базы данных могут быть созданы свои дескрипторы, присоединяемые к отдельным частям базы данных. Помимо обычных ограничений доступа для чтения и записи, сервер может защитить специфичные для базы данных операции, такие как прокрутка в результирующем множестве или соединение таблиц. В этом случае ответственность за определение специальных прав доступа и их проверки возлагается на сервер. Но такие проверки должны производиться в стандартном контексте с использованием учетных записей пользователей и групп во всей системе, а также журналов аудита. Такая модель расширяемой безопасности должна быть полезной и для сторонних производителей файловых систем.

## 15.8. Резюме

Область безопасности операционных систем довольно обширна, поэтому в этой главе основное внимание было уделено самым важным вопросам из данной области. Первостепенное значение для безопасности операционных систем имеет противостояние угрозам от злоумышленников и зловредных программ. Если злоумышленники пытаются получить несанкционированный доступ к системным ресурсам, то зловредные программы предназначены для проникновения в целевые системы через средства их защиты, чтобы стать выполняемыми в этих системах. Контрмерами против обоих видов угроз служат внедрение систем обнаружения вторжений, протоколов аутентификации, механизмов управления доступом и брандмаузеров.

Одним из самых распространенных методов нарушения безопасности операционной системы является атака переполнения буфера. Это состояние интерфейса, при котором в буфере или области хранения данных может быть размещено больше входных данных, чем допускает выделенный объем, в результате чего происходит перезапись другой информации. Атакующие злоумышленники злоупотребляют этим состоянием, чтобы вывести систему из строя или внедрить специально подготовленный код, позволяющий им взять систему под свой контроль. Для противостояния такого рода атакам разработчики систем пользуются самыми разными средствами защиты как времени компиляции, так и времени выполнения.

Еще одной важной областью обеспечения безопасности является управление доступом. К мерам управления доступом относится безопасный доступ к файловой системе и пользовательскому интерфейсу операционной системы. Традиционные методы управления доступом относятся к избирательному управлению; более гибкий подход, нашедший значительную поддержку, представляет собой ролевое управление доступом, когда доступ зависит не только от личности пользователя, но и от конкретной роли, которую он берет на себя для решения одной или ряда задач.

## 15.9. Ключевые термины, контрольные вопросы и задачи

### Ключевые термины

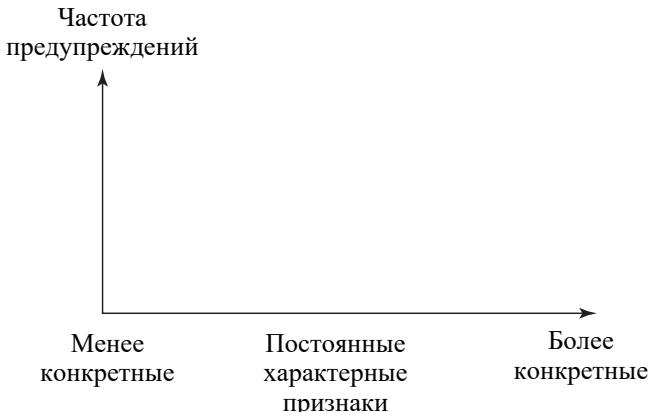
Аутентификация	Матрица доступа	Ролевое управление доступом (RBAC)
Брандмауэр	Обнаружение вторжений	Список управления доступом (ACL)
Вредоносные программы	Переполнение буфера	Стратегия управления доступом
Заделенная страница	Переполнение стека	Управление доступом
Зловредные программы	Протоколирование	Управление доступом к файловой системе
Злоумышленник	Рандомизация адресного пространства	
Избирательное управление доступом (DAC)		

### Контрольные вопросы

- Каковы типичные права доступа к конкретному файлу, которые могут быть предоставлены отдельному пользователю или которых он может быть лишен?
- Перечислите и кратко определите три категории злоумышленников.
- Каковы в общих чертах четыре средства аутентификации личности пользователя?
- Кратко опишите различия между DAC и RBAC.
- Какие типы языков программирования уязвимы к переполнению буфера?
- Каковы две обширные категории способов защиты от переполнения буфера?
- Перечислите и кратко опишите некоторые способы защиты от переполнения буфера, которыми можно воспользоваться во время компиляции новых программ.
- Перечислите и кратко опишите некоторые способы защиты от переполнения буфера, которыми можно воспользоваться во время выполнения существующих уязвимых программ.

### Задачи

- Укажите некоторые угрозы, возникающие в результате выполнения процесса с полномочиями администратора или привилегированного пользователя.
- В контексте систем обнаружения вторжений ложно-положительным является предупреждение, формируемое с целью предупредить о состоянии, которое фактически является безвредным. Ложно-отрицательный результат появляется в том случае, если системе обнаружения вторжений не удается сформировать предупреждение, когда возникает состояние, заслуживающее такого предупреждения. Нанесите на приведенную диаграмму две кривые, приблизительно отображающие ложно-положительные и ложно-отрицательные результаты соответственно.



- 15.3.** Перепишите функцию, приведенную на рис. 15.2, таким образом, чтобы она стала неуязвимой к переполнению буфера в стеке.
- 15.4.** Альтернативным представлением состояния защиты в модели избирательного управления доступом (DAC), описанной в разделе 15.3, является ориентированный граф. Каждый субъект и каждый объект в состоянии защиты представлен отдельным узлом графа, причем один узел служит для обозначения элемента, который является как субъектом, так и объектом. Направленная дуга, проведенная от субъекта к объекту, обозначает право доступа, которое помечается также меткой на ссылке.
- Постройте ориентированный граф, соответствующий матрице доступа, приведенной на рис. 15.3, а.
  - Постройте ориентированный граф, соответствующий матрице доступа, приведенной на рис. 15.4.
  - Имеется ли взаимно однозначное соответствие между представлениями прав доступа в виде ориентированного графа и в виде матрицы? Поясните свой ответ.
- 15.5.** Программы и сценарии с битом установки идентификатора пользователя (SetUID) или идентификатора группы (SetGID) служат эффективным механизмом, предоставляемым Unix для поддержки “контролируемого вызова” с целью управления доступом к уязвимым ресурсам. Но именно поэтому такой механизм открывает потенциальную брешь в защите, а программные ошибки привели ко многим фактам нарушения защиты в системах Unix. Опишите подробно команду, которой вы могли бы воспользоваться для обнаружения всех сценариев или программ с такими битами в Unix, и поясните, как воспользоваться полученными сведениями.
- 15.6.** Пользователь `ahmed` владеет каталогом `stuff`, содержащим текстовый файл `ourstuff.txt`, который совместно используется пользователями, принадлежащими группе `stuff`. Эти пользователи могут читать и изменять данный файл, но не удалять его. Они не могут также добавлять в данный каталог другие файлы. Другие пользователи не могут ни читать, ни записывать или выполнять что-нибудь в каталоге `stuff`. Как при этом будут выглядеть права владения и полномочия на доступ к каталогу `stuff` и файлу `ourstuff.txt`? (Напишите свои ответы в форме выводимого “длинного листинга”).

- 15.7.** Каталоги с файлами рассматриваются в UNIX как разновидность файлов. Это означает, что и те, и другие определяются с помощью одной и той же структуры данных, называемой индексным узлом (inode). Как и файлы, каталоги включают в себя 9-битовую строку защиты. Если быть невнимательным, это может вызвать трудности управления доступом. Допустим, что некоторый файл имеет режим защиты 644 и находится в каталоге с режимом защиты 730 (оба режима указаны в восьмеричной форме). Как в таком случае может быть нарушена безопасность этого файла?
- 15.8.** В традиционной для систем UNIX модели доступа к файлам предоставляется стандартная установка прав доступа ко вновь создаваемым файлам и каталогам, которую их владелец может впоследствии изменить. Как правило, владельцу предоставляется по умолчанию полный доступ к ним в сочетании с одним из следующих полномочий: запрет доступа для группы или другого пользователя; доступ для чтения и выполнения для группы и запрет доступа для других пользователей; доступ для чтения и выполнения для группы и для других пользователей. Опишите кратко преимущества и недостатки каждого из этих случаев, а также тип организации, в которой может оказаться пригодным каждый из этих механизмов управления доступом.
- 15.9.** Рассмотрим пользовательские учетные записи в системе с веб-сервером, настроенным для предоставления пользователям доступа к пользовательским областям. В общем случае такая схема использует стандартное имя каталога (например, `public_html`) в начальном каталоге пользователя. Однако чтобы разрешить веб-серверу доступ к страницам в данном каталоге, у него должен быть по крайней мере доступ для поиска (и выполнения) к начальному каталогу пользователя, доступ для чтения и выполнения к веб-каталогу, а также доступ для чтения к любым веб-страницам в нем. Рассмотрите взаимодействие данных требований с проблемами, рассматривавшимися в предыдущем задании. Какие последствия будет иметь выполнение данных требований? Имейте в виду, что веб-сервер, как правило, выполняется с полномочиями особого пользователя в группе, не используемой совместно с другими пользователями в системе. Возможны ли обстоятельства, когда подобное функционирование веб-сервера неприемлемо? Поясните свой ответ.
- 15.10.** Допустим, что имеется система с  $N$  должностями. На должности  $i$  работают отдельные пользователи в количестве  $U_i$ , а для выполнения их обязанностей требуется количество разрешений  $P_i$ .
- Какие взаимоотношения должны быть определены между пользователями и разрешениями в традиционной схеме избирательного управления доступом (DAC)?
  - Какие взаимоотношения должны быть определены между пользователями и разрешениями в схеме ролевого управления доступом (RBAC)?
- 15.11.** Почему так важно протоколирование? Каковы его ограничения в отношении управления защитой? Каковы преимущества и недостатки удаленного протоколирования?

- 15.12. Допустим, имеется некоторое автоматизированное инструментальное средство для анализа журналов проверки безопасности (например, swatch). Можете ли вы предложить какие-нибудь правила, которыми можно было бы воспользоваться, чтобы отличить подозрительные действия от нормального поведения пользователя в системе в некоторой организации?
- 15.13. Каковы преимущества и недостатки применения инструментального средства для проверки целостности файлов (например, tripwire)? Является ли это средство такой программой, которая регулярно уведомляет администратора о любых изменениях в файлах? Рассмотрите проблемы при выборе файлов, в которые требуется вносить изменения лишь изредка, чаще или часто. Выясните, как этот выбор может повлиять на конфигурацию данного инструментального средства, в особенности в отношении того, какие части файловой системы сканируются, и объема работы, который возлагается на администратора по текущему контролю над реакциями данного инструментального средства на состояние файловой системы.
- 15.14. Существует мнение, что в системах Unix/Linux повторно используется небольшое количество средств защиты во многих контекстах повсюду в системе, тогда как в системах Windows предоставляется намного больше средств, специально предназначенных для обеспечения безопасности в подходящих контекстах. Это можно рассматривать как своего рода компромисс между простотой и недостаточной гибкостью в подходе, принятом в системах Unix/Linux, по сравнению с более целенаправленным, но и более сложным подходом в Windows, затрудняющим правильное конфигурирование. Обсудите, каково влияние такого подхода на безопасность в соответствующих системах и какая нагрузка ложится на администраторов по обеспечению безопасности тех и других систем.

Tlgm: @it\_boooks

# ГЛАВА 16

---

# Облачные операционные системы и операционные системы Интернета вещей

В ЭТОЙ ГЛАВЕ...

## 16.1. Облачные вычисления

Элементы облачных вычислений

Модели предоставления услуг облачных вычислений

Программное обеспечение как услуга (SaaS)

Платформа как услуга (PaaS)

Инфраструктура как услуга (IaaS)

Модели развертывания облака

Открытое облако

Закрытое облако

Коллективное облако

Гибридное облако

Эталонная архитектура облачных вычислений

## 16.2. Облачные операционные системы

Инфраструктура как служба

Требования к облачной операционной системе

Общая архитектура облачной операционной системы

Виртуализация

Виртуальные вычисления

Виртуальное хранилище

Виртуальная сеть

Управление структурами данных

Управление и координирование

### OpenStack

- Вычисления (Nova)
- Образ (Glance)
- Сеть (Neutron)
- Хранилище объектов (Swift)
- Хранилище блоков (Cinder)
- Идентификация (Keystone)
- Информационная панель (Horizon)
- Мониторинг (Ceilometer)
- Координирование (Heat)
- Прочие дополнительные службы

## 16.3. Интернет вещей

- Вещи в Интернете вещей
- Эволюция
- Компоненты IoT-устройств
- Интернет вещей в контексте облака
- Границы
- Туманные вычисления
- Базовая сеть
- Облачная сеть

## 16.4. Операционные системы для Интернета вещей

- Устройства с ограниченными ресурсами
- Требования к операционным системам для Интернета вещей
- Архитектура операционной системы для Интернета вещей
- Операционная система RIOT
  - Ядро RIOT
  - Другие аппаратно-независимые модули
  - Уровень аппаратной абстракции

## 16.5. Ключевые термины, контрольные вопросы и задачи

- Ключевые термины
- Контрольные вопросы

## УЧЕБНЫЕ ЦЕЛИ

- Представить понятия облачных вычислений и сделать их общий обзор.
- Перечислить и определить основные облачные службы.
- Перечислить и определить модели развертывания облачных вычислений.
- Пояснить архитектуру облачных вычислений NIST.
- Описать основные функции облачной операционной системы.
- Представить общий обзор пакета OpenStack.
- Пояснить область действия Интернета вещей.
- Перечислить и обсудить пять основных компонентов IoT-устройств.
- Описать взаимосвязь между облачными вычислениями и Интернетом вещей.
- Определить устройства с ограниченными ресурсами.
- Описать основные функции облачной операционной системы.

За последние годы двумя наиболее значительными разработками в области вычислений стали облачные вычисления и Интернет вещей (*Internet of Things — IoT*). В обоих случаях операционные системы, удовлетворяющие конкретным требованиям этих сред, все еще находятся на стадии своего развития. Сначала в этой главе дается краткий обзор концепций облачных вычислений, а затем рассматриваются облачные операционные системы. Далее в этой главе исследуются понятия Интернета вещей, а завершается она рассмотрением операционных систем для IoT. Дополнительный материал по облачным вычислениям и Интернету вещей, рассматриваемым в разделах 16.1 и 16.3, можно найти в [243].

## 16.1. Облачные вычисления

В многих организациях все более и более заметной становится тенденция к переносу значительной части или даже всех информационно-технологических операций в подключенную к Интернету инфраструктуру, называемую корпоративными облачными вычислениями. В этом разделе дается общий обзор облачных вычислений.

### Элементы облачных вычислений

В документе NIST SP-800-145 (*The NIST Definition of Cloud Computing* — Определение облачных вычислений, данное в NIST) облачные вычисления определяются следующим образом.

**Облачные вычисления** — это модель, обеспечивающая повсеместный удобный сетевой доступ по требованию к общему пулу конфигурируемых вычислительных ресурсов (например, к сетям, серверам, хранилищам данных, приложениям и службам), которые могут быть быстро предоставлены и освобождены с минимальными усилиями, затрачиваемыми на управление или взаимодействие с провайдером услуг. Такая модель облачных вычислений способствует их доступности и обладает пятью важными характеристиками, а также состоит из трех моделей предоставления услуг и четырех моделей развертывания.

Приведенное выше определение различных моделей и характеристики, а также их взаимосвязи наглядно показаны на рис. 16.1. Ниже перечислены важные характеристики облачных вычислений.

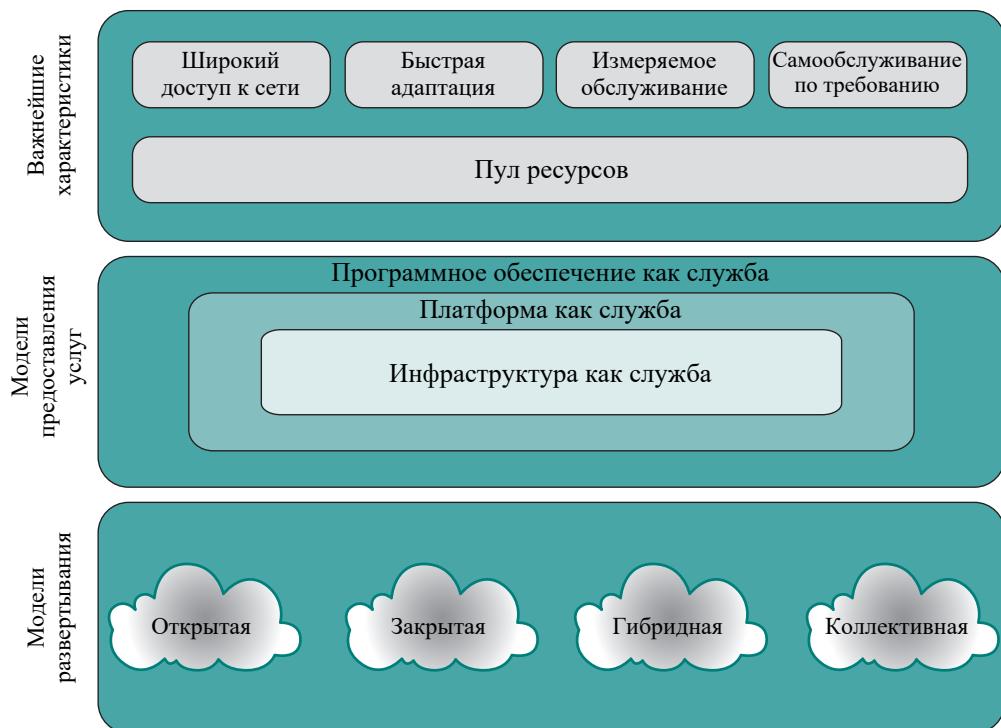


Рис. 16.1. Элементы облачных вычислений

- **Широкий доступ к сети.** Это возможности, доступные через сеть и стандартные механизмы, способствующие применению гетерогенных платформ “тонких” и “толстых” клиентов (например, мобильных телефонов, переносных и планшетных компьютеров), а также других традиционных или расположенных в облаке программных служб.
- **Быстрая адаптация.** Облачные вычисления дают возможность расширять или сокращать круг ресурсов, удовлетворяющих конкретным требованиям к обслуживанию. Например, во время выполнения конкретной задачи может потребоваться большее количество ресурсов сервера, которые по завершении задачи могут быть освобождены.
- **Измеряемое обслуживание.** Облачные системы автоматически управляют ресурсом и оптимизируют его применение, выгодно используя возможность измерений на некотором уровне абстракции, отвечающем типу услуг (например, хранения, обработки, предоставления пропускной способности и активных учетных записей пользователей). Применение ресурсов допускает текущий и общий контроль, отчетность и прозрачность как для провайдера, так и для потребителя используемых услуг.

- **Самообслуживание по требованию.** Потребитель услуг облачных вычислений (cloud service consumer — CSC) может автоматически и в одностороннем порядке обеспечить вычислительные возможности (например, время сервера или место для хранения в сети) по мере надобности, не требуя взаимодействия человека с провайдером услуг. Поскольку услуги предоставляются по требованию, эти ресурсы не являются постоянной частью информационно-технологической инфраструктуры их пользователя.
- **Пул ресурсов.** Вычислительные ресурсы провайдера объединяются в пул для обслуживания многих потребителей услуг облачных вычислений с помощью мультиарендаторной модели, в которой по требованию потребителя динамически назначаются и переназначаются разные физические и виртуальные ресурсы. Имеется определенная степень независимости от расположения в том плане, что потребитель услуг облачных вычислений не управляет предоставляемыми ему ресурсами и не знает о конкретном их расположении, хотя может указать их местоположение на более высоком уровне абстракции (например, страны, региона или центра обработки данных). Характерными примерами таких ресурсов служат услуги хранения, обработки, выделения оперативной памяти, пропускной способности сети и виртуальных машин. Даже в закрытых облаках наблюдается тенденция к объединению в пулы ресурсов среди разных подразделений организации.

## Модели предоставления услуг облачных вычислений

В Национальном институте стандартов США (NIST) определяются три **модели обслуживания** (service model), которые можно рассматривать как вложенные альтернативы обслуживания: программное обеспечение как услуга (software as a service — SaaS), платформа как услуга (platform as a service — PaaS), а также инфраструктура как услуга (infrastructure as a service — IaaS).

### Программное обеспечение как услуга (SaaS)

Представляет потребителям услуги в форме программного обеспечения и, в частности, прикладных программ, выполняемых и доступных в облаке, следуя известной модели веб-служб, которая в данном случае применяется к облачным ресурсам. Модель SaaS дает потребителю возможность воспользоваться приложениями, выполняемыми в облачной инфраструктуре провайдера услуг. Приложения доступны из различных клиентских устройств через простой интерфейс (например, веб-браузер). Вместо получения лицензий на программное обеспечение, применяемое в настольных системах и на серверах, ту же самую функциональность предприятие получает из облачной службы. Модель SaaS позволяет избежать сложностей установки, сопровождения, обновления и установки патчей в программное обеспечение. К примерам служб, действующих на данном уровне, относятся Google Gmail, Microsoft 365, Salesforce, Citrix GoToMeeting и Cisco WebEx.

Распространенными подписчиками на услуги SaaS являются те организации, сотрудникам которых требуется предоставить доступ к типичному программному обеспечению, повышающему их производительность труда в учреждении (например, обработку документов и электронной почты). Физические лица также пользуются моделью SaaS для приобретения облачных ресурсов. Как правило, подписчики пользуются конкретными приложениями по требованию; провайдер услуг облачных вычислений обычно

предоставляет такие средства манипулирования данными, как автоматическое резервирование и обмен данными между подписчиками.

### **Платформа как услуга (PaaS)**

Облачная модель PaaS предоставляет потребителям услугу в форме платформы, на которой они могут выполнять свои приложения. Она дает потребителям возможность развертывать в облачной инфраструктуре созданные или приобретенные ими приложения. Облачная модель PaaS предоставляет удобные программные строительные блоки, а также целый ряд инструментальных средств разработки (например, языковые средства программирования, среды выполнения, а также вспомогательные средства для развертывания новых приложений). По существу, PaaS — это операционная система в облаке. Она удобна для тех организаций, которым требуется разрабатывать новые или приспособливать уже существующие приложения, а платить за вычислительные ресурсы только по мере надобности и пользоваться ими столько, сколько потребуется. К примерам служб, действующих на уровне PaaS, относятся AppEngine, Engine Yard, Heroku, Microsoft Azure, Force.com и Apache Stratos.

### **Инфраструктура как услуга (IaaS)**

С помощью модели IaaS потребитель получает доступ к ресурсам базовой облачной инфраструктуры. Пользователь услуг облачных вычислений не управляет ресурсами базовой облачной инфраструктуры и не контролирует их, но контролирует операционные системы, развертываемые приложения, а возможно, имеет ограниченный контроль над избранными сетевыми компонентами (например, брандмауэрами узлов). Модель IaaS предоставляет виртуальные машины и прочее виртуализованное оборудование и операционные системы, а также услуги обработки, хранения, доступа к сетям и прочим основополагающим вычислительным ресурсам, чтобы потребитель мог развернуть и выполнить произвольное программное обеспечение, в том числе операционные системы и приложения. Модель IaaS дает потребителям возможность объединять основные вычислительные услуги (например, решение числовых задач и хранение данных) для построения в высшей степени адаптируемых компьютерных систем.

Как правило, потребители могут и сами предоставлять такую инфраструктуру, используя веб-ориентированный графический пользовательский интерфейс, который служит в качестве консоли для управления информационно-технологическими операциями для общей среды. В качестве дополнительной возможности может быть также предоставлен API для доступа к облачной инфраструктуре. К примерам служб, действующих на уровне IaaS, относятся Amazon Elastic Compute Cloud (Amazon EC2), Microsoft Windows Azure, Google Compute Engine (GCE) и Rackspace. Для сравнения на рис. 16.2 приведены функции, реализуемые провайдером услуг облачных вычислений по трем рассмотренным здесь моделям предоставления услуг.

## **Модели развертывания облака**

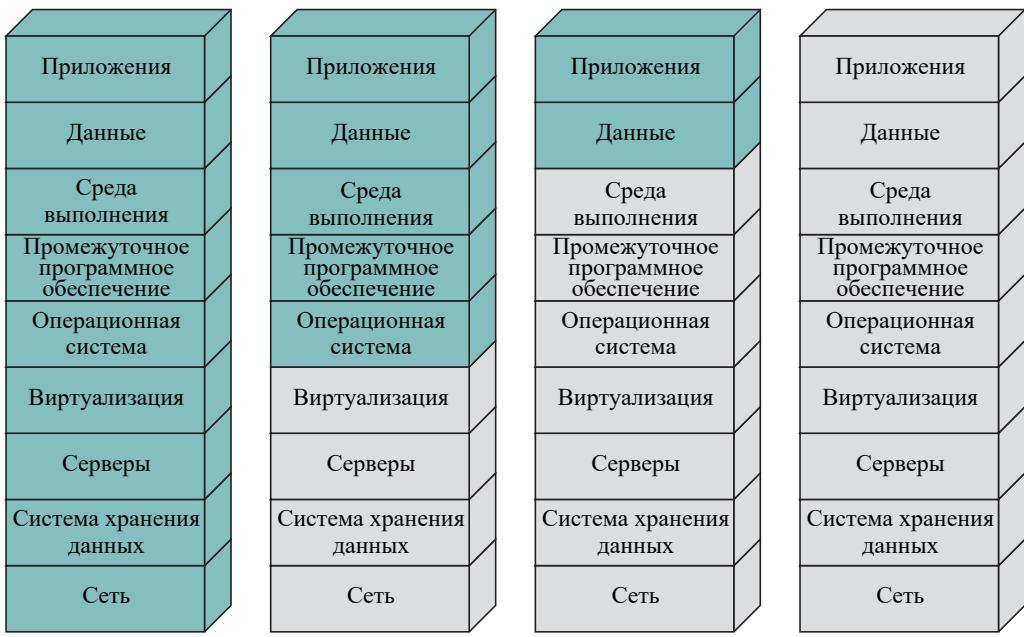
Как упоминалось ранее, во многих организациях все более заметно проявляется тенденция переносить значительную часть или даже все IT-операции в корпоративные облачные вычисления. При этом организациям приходится делать выбор среди целого ряда вариантов владения облачными ресурсами и управления ими. В этом разделе будут рассмотрены самые примечательные модели развертывания для облачных вычислений.

Традиционные  
IT-компоненты  
в локальной среде

IaaS

PaaS

SaaS



Управляется потребителем

Управляется провайдером облачных служб

**Рис. 16.2.** Разделение обязанностей в облачной операции

### Открытое облако

Открытая облачная инфраструктура (public cloud) предназначена для общедоступной или крупной промышленной группы. Ее владельцем может быть организация, предоставляющая платные услуги облачных вычислений.

Провайдер услуг облачных вычислений отвечает как за облачную инфраструктуру, так и за управление данными и выполнение операций в облаке. Открытым облаком может владеть, управлять и заведовать коммерческая, академическая, государственная организация или определенное их объединение. Облако располагается в локальной среде провайдера услуг облачных вычислений.

В модели открытого облака все основные компоненты находятся за пределами корпоративного брандмауэра, расположенного в мультиарендаторной инфраструктуре. Приложения и хранилища данных могут быть доступны через Интернет по защищенному IP-адресу как бесплатно, так и за определенную плату, взимаемую за их использование. Облако такого рода облако предоставляет простые в употреблении услуги потребительского типа, в том числе такие, как доступные по требованию веб-приложения или вычислительные мощности от компаний Amazon и Google, почта Yahoo и социальные сети Facebook или LinkedIn, в которых можно свободно размещать фотографии. Хотя

открытые облака недороги и масштабируются для удовлетворения насущных потребностей, они, как правило, вообще не предоставляют соглашения об уровне обслуживания (SLA) или предоставляют его в минимальной форме и могут вообще не гарантировать отсутствие потерь или повреждения данных, присущих закрытым или гибридным вариантам облачных инфраструктур. Открытое облако подходит для потребителей услуг облачных вычислений и субъектов, не требующих высокого уровня обслуживания. Кроме того, открытые облака типа SaaS совсем не обязательно накладывают ограничения и не обеспечивают соблюдение законов, защищающих частную жизнь граждан, возлагая ответственность за нарушения на конечного пользователя. Во многих открытых облаках основное внимание уделяется потребителям услуг облачных вычислений, а также представителям мелкого и среднего бизнеса, готовым платить за пользование услугами (зачастую — копейки за гигабайты). К характерным примерам услуг открытых облачных вычислений можно отнести обмен фотографиями, музыкальными записями и файлами, а также резервное копирование данных с ноутбуков.

Главным преимуществом открытого облака является стоимость. Подписавшаяся организация платит только за те услуги и ресурсы, которые ей требуются, и может корректировать свои потребности. Кроме того, издержки подписчика на управление значительно сокращены. А главным его недостатком является безопасность. Тем не менее имеется целый ряд провайдеров открытых облачных услуг, продемонстрировавших образцы строгого управления защитой; фактически такие провайдеры могут иметь больше опыта и ресурсов, чтобы уделить их обеспечению такого уровня безопасности, который обычно имеется в закрытом облаке.

### **Закрытое облако**

Закрытое облако (*private cloud*) реализуется во внутренней IT-среде организации, которая может выбрать управление облаком своими силами или по договору со сторонней организацией. Кроме того, облачные серверы и хранилища данных могут существовать как локально, так и удаленно.

Закрытые облака могут предоставлять услуги типа IaaS сотрудникам или структурным подразделениям внутри организации через корпоративную сеть или Интернет посредством виртуальной частной сети (*virtual private network* — VPN), а прикладное программное обеспечение или хранилища данных в виде услуг — филиалам организации. Но в обоих случаях закрытые облака позволяют выгодно воспользоваться существующей инфраструктурой и предоставлять и взимать плату за отдельные или полные услуги, соблюдая конфиденциальность в сети организации. К примерам услуг, предоставляемых через закрытое облако, относятся базы данных, электронная почта и хранилища данных, доступные по требованию.

Главной побудительной причиной для выбора закрытого облака является безопасность. Инфраструктура закрытого облака обеспечивает более строгий контроль над географическим местоположением хранилища данных и прочими аспектами безопасности. К другим преимуществам закрытого облака относятся простота совместного пользования ресурсами и быстрота развертывания в организационных единицах.

### **Коллективное облако**

Коллективное облако (*community cloud*) разделяет характеристики закрытых и открытых облаков. Доступ к коллективному облаку, как и к закрытому облаку, ограничен. А ресурсы коллективного облака, как и ресурсы открытого облака, совместно используют-

ся многими независимыми организациями. Такие организации предъявляют к коллективному облаку сходные требования и, как правило, испытывают потребность в обмене данными между собой. Одним из примеров отрасли, в которой применяется принцип коллективного облака, служит здравоохранение. Коллективное облако может быть реализовано в соответствии с постановлениями правительства о соблюдении конфиденциальности и прочими нормативными требованиями. Участники коллективного облака могут обмениваться данными в контролируемом режиме.

Инфраструктурой коллективного облака могут управлять участвующие в нем организации или же сторонняя организация, а существовать она может как локально, так и удаленно. В этой модели развертывания затраты распределяются среди меньшего числа пользователей, чем в открытом облаке, но среди большего числа пользователей, чем в закрытом облаке. Поэтому на коллективных облачных вычислениях можно сэкономить совсем немного.

### **Гибридное облако**

Инфраструктура гибридного облака (hybrid cloud) состоит из двух или более облаков (закрытого, коллективного или открытого), которые остаются особыми элементами, связанными вместе по стандартизированной или патентованной технологии, допускающей перенос данных и приложений (например, расширение облака для равномерного распределения нагрузки между облаками). Гибридный вариант позволяет размещать секретную информацию в закрытом облаке, а менее секретные данные — выгодно хранить в открытом облаке.

Гибридный вариант открытого и закрытого облака может быть особенно привлекательным для представителей малого бизнеса. Многие приложения, для которых вопросы безопасности не так важны, могут выгружаться со значительной экономией затрат, не вынуждая организации переносить в открытое облако данные и приложения, которым требуется большая защита.

Сильные и слабые стороны четырех рассмотренных здесь моделей развертывания перечислены в табл. 16.1.

**Таблица 16.1. Сравнение моделей развертывания облака**

	Закрытая	Коллективная	Открытая	Гибридная
<b>Масштабируемость</b>	Ограниченнaя	Ограниченнaя	Весьма высокая	Весьма высокая
<b>Безопасность</b>	Самая безопасная	Довольно безопасная	Умеренно безопасная	Довольно безопасная
<b>Производительность</b>	Очень хорошая	Очень хорошая	От низкой до средней	Хорошая
<b>Надежность</b>	Весьма высокая	Весьма высокая	Средняя	От средней до высокой
<b>Стоимость</b>	Высокая	Средняя	Низкая	Средняя

## Эталонная архитектура облачных вычислений

В документе NIST SP 500-292 (*NIST Cloud Computing Reference Architecture* — Определение эталонной архитектуры облачных вычислений, данное в NIST) эталонная архитектура облачных вычислений описывается следующим образом.

В эталонной архитектуре облачных вычислений, разработанной в NIST, основное внимание уделяется предоставляемым услугам облачных вычислений, а не выработке проектного решения и его реализации. Эталонная архитектура призвана облегчить понимание практических сложностей, возникающих в облачных вычислениях. Она не представляет архитектуру конкретной облачной вычислительной системы, а служит инструментальным средством для описания, обсуждения и разработки архитектуры конкретной системы, исходя из общей точки зрения.

Эталонная архитектура облачных вычислений была разработана в NIST со следующими целями.

- Наглядно показать и разъяснить различные доступные в облаке услуги в контексте общей концептуальной модели облачных вычислений.
- Предоставить техническое руководство для потребителей услуг облачных вычислений, чтобы они смогли уяснить, обсудить, разделить на категории и сравнить доступные в облаке услуги.
- Облегчить анализ подходящих стандартов на безопасность, взаимодействие, переносимость и эталонные реализации.

В эталонной архитектуре, показанной на рис. 16.3, определяются пять основных действующих лиц с точки зрения их ролей и обязанностей.

- **Потребитель услуг облачных вычислений (Cloud service consumer — CSC).** Физическое или юридическое лицо, поддерживающее деловые отношения с провайдером услуг облачных вычислений и пользующееся его услугами.
- **Провайдер услуг облачных вычислений (Cloud service provider — CSP).** Физическое или юридическое лицо, ответственное за предоставление услуг облачных вычислений заинтересованным сторонам.
- **Аудитор облака.** Сторона, которая может произвести независимую оценку услуг облачных вычислений, операций в информационной системе, производительности и безопасности конкретной реализации облака.
- **Облачный брокер.** Субъект, управляющий применением, производительностью и предоставлением услуг облачных вычислений и согласующий взаимоотношения между провайдерами услуг облачных вычислений и их потребителями.
- **Оператор облака.** Посредник, обеспечивающий связь и транспорт услуг облачных вычислений от их провайдера к потребителям.



**Рис. 16.3.** Эталонная архитектура облачных вычислений, разработанная в NIST

Роли потребителя и провайдера услуг облачных вычислений уже рассматривались. Подводя итог, можно сказать, что **провайдер услуг облачных вычислений** может предоставить одну или несколько подобных услуг, чтобы удовлетворить IT и прикладные потребности **потребителя услуг облачных вычислений**. В соответствии с каждой из трех моделей обслуживания (SaaS, PaaS, IaaS) провайдер услуг облачных вычислений предоставляет средства хранения и обработки данных, требующиеся для поддержки данной модели, а также облачный интерфейс для потребителей услуг облачных вычислений. Так, в модели SaaS провайдер услуг облачных вычислений развертывает, конфигурирует, сопровождает и обновляет прикладные программы, действующие в облачной инфраструктуре, чтобы услуги предоставлялись потребителям на предполагаемых уровнях обслуживания. Потребителями услуг в модели SaaS могут быть организации, предоставляющие своим членам доступ к прикладным программам, конечные пользователи, непосредственно применяющие прикладные программы, или же администраторы прикладных программ, настраивающие их для конечных пользователей.

В модели PaaS провайдер услуг облачных вычислений организует вычислительную инфраструктуру для платформы и выполняет облачное программное обеспечение, представляющее компоненты платформы, в том числе стеки выполнения программного обеспечения, базы данных и прочие компоненты промежуточного программного обеспечения. Потребители услуг в модели PaaS могут пользоваться инструментальными средствами и исполнительными ресурсами, предоставляемыми провайдером услуг облачных вычислений, для разработки и управления приложениями, размещаемыми в облачной среде.

В модели IaaS провайдер услуг облачных вычислений приобретает физические вычислительные ресурсы, на которых основывается обслуживание, включая серверы, сети, хранилища данных и инфраструктуру размещения в облаке. В свою очередь, потребитель услуг облачных вычислений пользуется этими вычислительными ресурсами (например, виртуальной машиной) для удовлетворения своих основных потребностей в вычислениях.

**Оператором облака** является объект организации сети, обеспечивающий связь и транспорт услуг облачных вычислений от их провайдера к потребителям. Как правило, провайдер услуг облачных вычислений заключает соглашения об уровне обслуживания (SLA) с оператором облака для предоставления потребителям услуг на уровне, соответствующем подобным соглашениям, и для этого он может потребовать от оператора облака обеспечить выделенные и защищенные соединения между потребителями и провайдерами услуг облачных вычислений.

**Облачный брокер** приносит пользу в том случае, если услуги облачных вычислений слишком сложны для потребителя, чтобы он мог легко справиться с ними. В таком случае облачный брокер может обеспечить поддержку в следующих трех областях.

- **Посредничество в предоставлении услуг.** Это дополнительные услуги, такие как управление пользователями, отчетность о производительности, а также усиление защиты.
- **Агрегирование услуг.** Облачный брокер объединяет многие услуги облачных вычислений, чтобы удовлетворить те потребности потребителей, которые не в состоянии удовлетворить единственный провайдер услуг облачных вычислений, и чтобы оптимизировать производительность или свести затраты к минимуму.
- **Арбитраж услуг.** Подобен агрегированию услуг, за исключением того что агрегируемые услуги не фиксируются. Арбитраж услуг означает, что брокер может свободно выбирать услуги многих агентств. Например, брокер может воспользоваться услугами оценки платежеспособности, чтобы определить и выбрать агентство с наилучшей оценкой.

**Аудитор облака** может оценивать услуги облачных вычислений, предоставляемые провайдером, с точки зрения средств управления защитой, влияния на конфиденциальность, производительность и т.д. Аудитором облака может быть независимый субъект, способный убедиться, что провайдер услуг облачных вычислений действительно удовлетворяет ряду стандартов.

Взаимодействия между действующими в облаке лицами показаны на рис. 16.4. Потребитель услуг облачных вычислений может запросить услуги непосредственно у провайдера или через облачного брокера. Аудитор облака проводит независимые контрольные проверки и может связываться с другими действующими в облаке лицами для сбора необходимой информации. Как показано на рис. 16.4, для решения вопросов сетевой организации облака привлекаются три отдельных типа сетей. В частности, для производителя облака архитектура сети состоит из типичного крупного центра обработки данных, в котором установлены стойки с высокопроизводительными серверами и запоминающими устройствами, соединенными через высокоскоростные коммутаторы сети Ethernet, расположенные в верхней части стоек.

В данном контексте интерес и основное внимание уделяются вопросам расположения и перемещения виртуальных машин, распределения нагрузки и доступности. Корпоративная сеть, вероятнее всего, будет иметь совсем другую архитектуру, как правило, включающую в себя целый ряд локальных сетей, серверов, рабочих станций, ПК и мобильных устройств с самыми разными требованиями, предъявляемыми к производительности сети, безопасности и управлению. Интерес производителя и потребителя к оператору облака, являющемуся общим для многих пользователей, состоит в том, чтобы создавать виртуальные сети с подходящими соглашениями об уровне обслуживания и гарантиями безопасности.



Рис. 16.4. Связи между действующими лицами в облачных вычислениях

## 16.2. ОБЛАЧНЫЕ ОПЕРАЦИОННЫЕ СИСТЕМЫ

Термин *облачная операционная система* означает распределенную операционную систему, предназначенную для выполнения в центре обработки данных провайдера услуг облачных вычислений и управления высокопроизводительными серверами, сетью, ресурсами для хранения данных и предоставления этих услуг их пользователям. По существу, облачная операционная система — это программное обеспечение, реализующее услуги по модели IaaS.

Здесь важно отметить отличие облачной операционной системы от платформы PaaS. Как обсуждалось в разделе 16.1, PaaS — это платформа для выполнения приложений потребителя, которая дает потребителю возможность развертывать приложения, созданные им в своей инфраструктуре или приобретенные в готовом виде. Она предоставляет удобные стандартные блоки для построения программ, а также целый ряд инструментальных средств, в том числе сред программирования, сред выполнения и прочих инструментальных средств, помогающих развертывать новые приложения. По сути, PaaS является операционной системой, доступной пользователям в облаке. А облачная операционная система, напротив, отличается от операционной системы, выполняемой пользователем услуг на виртуальных машинах в облаке. Поскольку провайдер предоставляет услуги IaaS, операционная система пользователя выполняется в облачной инфраструктуре. Облачная операционная система управляет этими услугами и может обеспечить пользователя инструментальными средствами, но является прозрачной для пользователя услуг облачных вычислений.

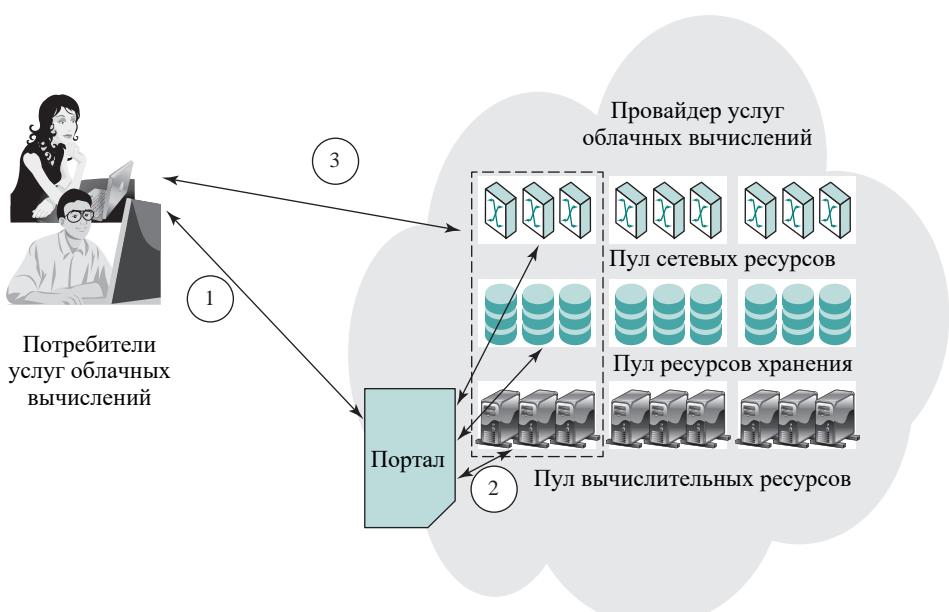
В начале этого раздела более подробно описывается модель IaaS, а затем исследуются характеристики облачной операционной системы, пригодной для реализации данной модели. Наконец в этом разделе рассматривается OpenStack — самая важная облачная операционная система с открытым исходным кодом.

## Инфраструктура как служба

Модель IaaS представляет уровень инфраструктуры, состоящий главным образом из виртуализованных сред, предоставляющих услуги вычисления, хранения и сетевые ресурсы. Гипервизоры выполняют совокупность виртуальных машин в реальных информационно-технологических ресурсах и предоставляют виртуализированные версии этих ресурсов пользователям услуг облачных вычислений. Пользователи могут свободно устанавливать любую ОС и прикладную среду, которая требуется в этих виртуализованных ресурсах. А провайдер отвечает за разрешение доступа к виртуализированным ресурсам, предоставляя их в нужном количестве и управляя ими. Потребитель услуг облачных вычислений не управляет и не контролирует базовую облачную инфраструктуру, но обладает контролем над операционными системами, хранилищами данных, развернутыми приложениями, а возможно, и ограниченным контролем над выбором сетевых компонентов (например, брандмаузеров).

Следует, однако, иметь в виду, что IaaS — это просто другое название виртуализованной среды. И хотя виртуализация является главной технологией, активизирующей облачные вычисления, виртуализованная среда способна удовлетворять самым важным характеристикам IaaS лишь в том случае, если основная среда расширяется для внедрения дополнительных средств управления (например, для перемещения виртуальных машин, текущего контроля и управления доступностью, восстановления, управления жизненным циклом, самообслуживания, взимания платы и т.д.).

На рис. 16.5 наглядно демонстрируются основные средства IaaS, доступные потребителю услуг облачных вычислений. Три взаимодействия, обозначенные нумерованными линиями с двумя стрелками на концах, являются наиболее важными элементами на рис. 16.5.



**Рис. 16.5.** Концептуальная модель IaaS

Первое взаимодействие между потребителем и порталом провайдера услуг облачных вычислений служит для предоставления потребителю доступа к облачным ресурсам с подходящими механизмами защиты. Оно охватывает следующие действия.

1. Потребитель услуг облачных вычислений получает доступ к услуге IaaS с подходящими механизмами защиты и запрашивает у провайдера услуг облачных вычислений список поддерживаемых функций, связанных с инфраструктурой (например, шаблоны инфраструктур).
2. Потребитель услуг облачных вычислений выбирает подходящий шаблон инфраструктуры из результатов запроса и далее запрашивает у провайдера услуг облачных вычислений создание инфраструктуры на основе сделанного выбора.
3. Потребитель услуг облачных вычислений осуществляет управление и текущий контроль созданной инфраструктуры в течение ее жизненного цикла. Сюда относятся следующие действия (но управление и контроль ими не ограничиваются).
  - **Назначение.** Запуск услуги типа IaaS путем выделения для нее доступных ресурсов, определенных в конфигурации (например, для создания, инициализации, запуска, активизации и подачи питания).
  - **Модификация.** Изменение количества используемого ресурса согласно потребностям (например, для обновления, добавления, активации и деактивации).
  - **Освобождение.** Завершение услуги типа IaaS путем высвобождения используемого в ней ресурса (например, для удаления, остановки, отмены и отключения питания).

Второе взаимодействие включает следующие действия.

1. Потребитель услуг облачных вычислений выбрал шаблон или сконфигурировал конкретную виртуальную машину и/или физический хост.
2. Потребитель услуг облачных вычислений выбрал ресурсы хранения (например, блок, файл и хранилище объектов), а затем присоединил их через вычислительные возможности или же воспользовался ими непосредственно.
3. Потребитель услуг облачных вычислений выбрал услуги подключения к сети (например, IP-адрес, виртуальную локальную сеть, брандмауэр и выравнивание нагрузки), а затем применил их к соответствующим возможностям вычисления и/или хранения.
4. Потребитель услуг облачных вычислений подтвердил соглашения об уровне обслуживания (SLA) и модель начислений за выбранные услуги вычисления, хранения и подключения к сети, предоставляемые провайдером услуг облачных вычислений.

Как только провайдер услуг облачных вычислений предоставит доступ к своим ресурсам и сконфигурирует их для потребителя этих услуг, третье взаимодействие продолжится следующим образом.

1. Провайдер услуг облачных вычислений осуществляет управление и мониторинг возможностей вычисления, хранения и организации сети с помощью произвольных приложений.

2. Потребитель услуг облачных вычислений конфигурирует, развертывает и поддерживает ресурсы гипервизора и хранения.
3. Провайдер услуг облачных вычислений устанавливает, конфигурирует и поддерживает связь по сети с потребителем этих услуг.
4. Провайдер услуг облачных вычислений предоставляет инфраструктуру безопасности потребителю этих услуг.

## Требования к облачной операционной системе

Облачная операционная система должна управлять потребителями услуг облачных вычислений и предоставлять им доступ к среде IaaS. Чтобы определить требования к облачной операционной системе, полезно рассмотреть функции, которые должны поддерживаться для услуг типа IaaS. В рекомендации Y.3513 (*Cloud computing — Functional requirements of Infrastructure as a Service — Облачные вычисления*. Функциональные требования к инфраструктуре как услуге, от августа 2014 года), выданной сектором по стандартизации телесвязи Международного союза электросвязи (ITU-T), перечислены функциональные требования, предъявляемые к провайдеру услуг облачных вычислений для предоставления услуг типа IaaS. В этих требованиях сведены конечные цели облачной операционной системы (табл. 16.2).

**Таблица 16.2. Функциональные требования к провайдеру услуг облачных вычислений в модели типа IaaS**

Область	Требования
<b>Общая</b>	<ul style="list-style-type: none"> <li>• Предоставление таких функций IaaS, как композиция ресурсов обработки, хранения и организации сети, с использованием логики обслуживания, конкретных соглашений об уровне обслуживания (SLA) и модели начислений за услуги</li> <li>• Предоставление сведений о состоянии инфраструктуры в ответ на запросы потребителя услуг облачных вычислений</li> <li>• Предоставление потребителю услуг облачных вычислений шаблона, связанного с инстанцированием инфраструктуры, что позволяет снабжать ресурсами обработки, хранения и организации сети, которые могли быть реализованы на основе конфигурации</li> </ul>
<b>Услуга вычисления</b>	<ul style="list-style-type: none"> <li>• Предоставление виртуальных машин на основе шаблона или конфигурации, указанной потребителем услуг облачных вычислений</li> <li>• Предоставление потребителю услуг облачных вычислений механизмов для выполнения таких операций, как создание, удаление, запуск, остановка, приостановка, возобновление, гибернация и выход из нее</li> <li>• Предоставление виртуальной машины со следующими функциями: перенос с одного хоста на другой; масштабирование, включая изменения конфигурации (например, увеличение или уменьшение количества процессоров, объема оперативной памяти, пропускной способности каналов связи), а также изменения в составе компонентов (например, добавление или удаление виртуальной машины); получение моментальных снимков; клонирование и резервное копирование</li> </ul>

Окончание табл. 16.2

Область	Требования
<b>Услуга хранения</b>	<ul style="list-style-type: none"> <li>Предоставление таких функций хранения, как хранение данных на уровне блоков, файлов и объектов</li> <li>Предоставление механизмов для выполнения таких операций, как создание, присоединение, отсоединение, запрос и удаление объема хранения данных на уровне блока или файловой системы, запись, чтение и удаление данных в заданном месте хранения</li> <li>Предоставление следующих функций: перенос хранилища данных, получение мгновенного снимка и резервное копирование</li> </ul>
<b>Услуга организации сети</b>	<ul style="list-style-type: none"> <li>Предоставление таких сетевых функций, как IP-адрес, виртуальная локальная сеть, виртуальный коммутатор, перераспределение нагрузки и брандмауэр</li> </ul>

## Общая архитектура облачной операционной системы

Главная особенность облачной операционной системы состоит в том, что для предоставления ресурсов вычисления, хранения и организации сети через среду IaaS в ней применяется технология виртуализации. На рис. 16.6 схематически показаны основные составляющие облачной операционной системы на концептуальном уровне. Каждый из них поочередно будет рассмотрен далее.

### Виртуализация

Как пояснялось в главе 14, “Виртуальные машины”, технология виртуальных машин допускает перенос специально выделенных серверов приложений, организации сети и хранения на коммерчески доступные серверы. В традиционных сетях серверов и запоминающих устройствах все подобные устройства развертываются на закрытых платформах. Каждому устройству требуется дополнительное аппаратное обеспечение для повышения мощности, но такое оборудование приставляет, если система работает не на полную мощность. С помощью виртуализации элементы вычисления, хранения и организациисетки становятся независимыми приложениями, удобно развертываемыми на единообразной платформе, состоящей из стандартных серверов, запоминающих устройств и коммутаторов. Подобным образом программные и аппаратные средства оказываются разделенными, а вычислительные мощности для каждого приложения наращиваются или сокращаются введением или удалением виртуальных ресурсов.

Аппаратными ресурсами в облачной среде являются стандартные серверы, подключаемые через сеть запоминающие устройства и коммутаторы (как правило, коммутаторы Ethernet). Гипервизоры, выполняющиеся на этих аппаратных устройствах, обеспечивают поддержку, необходимую для разработки виртуальных машин, предоставляющих необходимые ресурсы для виртуальных вычислений, хранения данных и организации сети.

Провайдер услуг облачных вычислений сохраняет полный контроль над физическим оборудованием и административный контроль над уровнем гипервизора. Потребитель может делать запросы к облаку на создание новых виртуальных машин и управление ими, хотя эти запросы принимаются во внимание лишь в том случае, если они соответствуют стратегиям, установленным провайдером для назначения ресурсов. Через

гипервизор провайдера, как правило, предоставляет интерфейсы к сетевым средствам (например, сетевым коммутаторам), которыми потребители могут воспользоваться для конфигурирования специальных виртуальных сетей в инфраструктуре данного провайдера. Потребитель обычно сохраняет полный контроль над гостевой операционной системой, действующей на каждой виртуальной машине, а также над всеми находящимися выше программными уровнями.

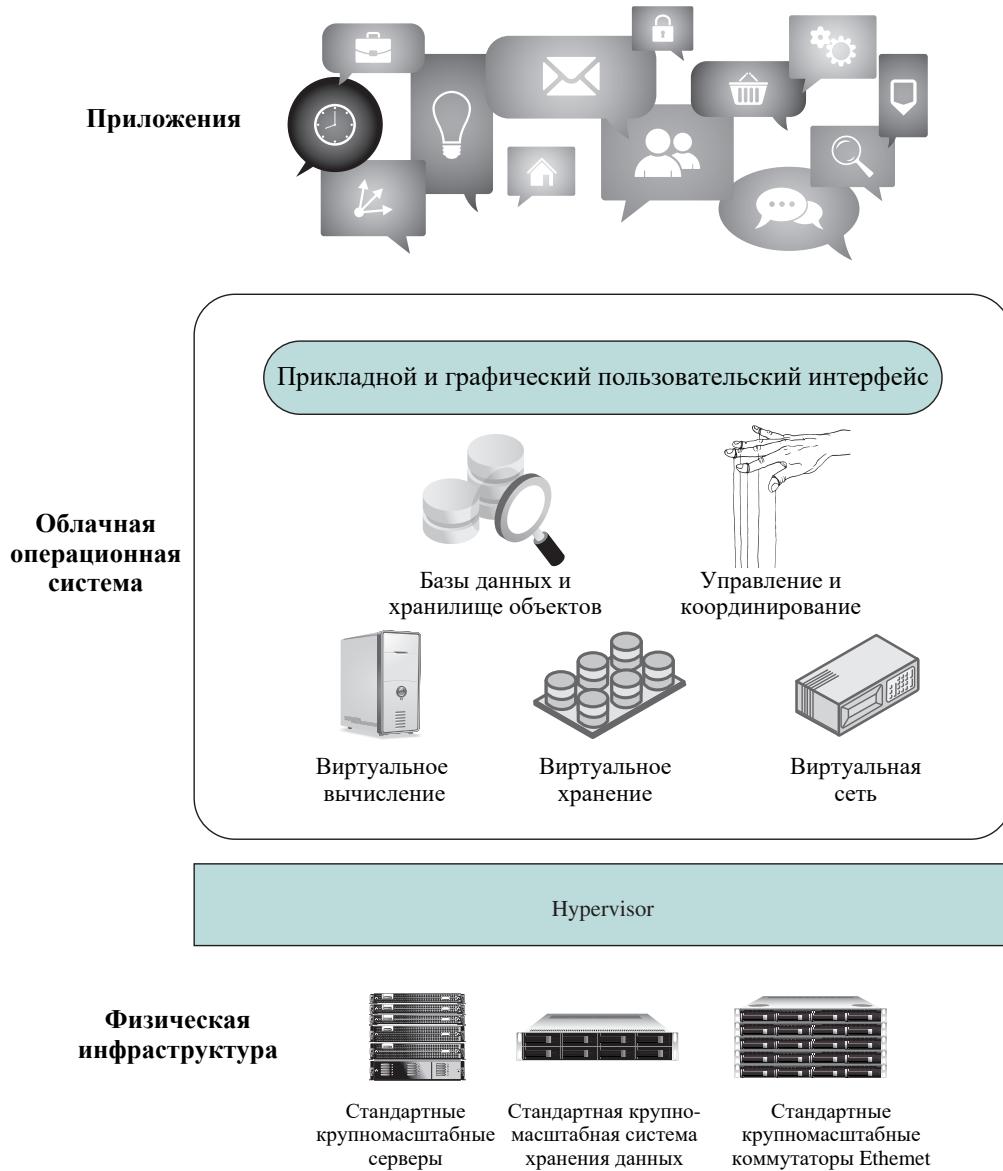


Рис. 16.6. Концептуальная схема облачной операционной системы

## Виртуальные вычисления

Компонент облачной операционной системы, отвечающий за виртуальные вычисления, управляет виртуальными машинами в среде IaaS облачных вычислений. Каждая виртуальная машина доступна для облачной операционной системы в виде экземпляра вычислений, основные элементы которого перечислены ниже.

- **Процессор/память.** Это процессор коммерчески доступного сервера с основной памятью для выполнения кода виртуальной машины.
- **Внутреннее хранилище.** Это энергонезависимое запоминающее устройство, расположенное в той же физической структуре, что и процессор (например, флеш-память).
- **Ускоритель.** К функциям ускорителя могут относиться защита, организация сети и обработка пакетов. Эти функции виртуального ускорителя соответствуют функциям аппаратного ускорителя, связанного с физическим сервером.
- **Внешнее хранилище со своим контроллером.** Обеспечивает доступ ко вторичным запоминающим устройствам. Эти запоминающие устройства присоединяются к физическому серверу, в отличие от запоминающих устройств, подключаемых к сети.

В состав компонента виртуальных вычислений входит также программное обеспечение взаимодействия с другими компонентами облачной операционной системы и с прикладными и графическими пользовательскими интерфейсами.

## Виртуальное хранилище

Компонент облачной операционной системы, отвечающий за виртуальное хранение данных, предоставляет услуги хранения данных в облачной инфраструктуре. К услугам этого компонента относятся следующие.

- Хранение информации управления облаком, включая определения виртуальных машин и сетей.
- Выделение рабочего пространства для различных приложений, выполняющихся в облачной среде.
- Предоставление механизмов, связанных с хранением данных, включая перенос приложений, автоматическое резервное копирование, интегрированный контроль версий и механизмы, оптимизированные для конкретных приложений.

Для потребителя услуг облачных вычислений этот компонент обеспечивает блочное хранение данных с дополнительными функциональными возможностями, реализованными в виде совокупности виртуальных накопителей на дисках в гипервизоре. Этот компонент должен изолировать сохраняемые данные из разных приложений потребителя услуг облачных вычислений.

Хранилище данных имеет следующие виды топологии.

- **Непосредственно подключаемое хранилище (Direct Attached Storage — DAS).** Несмотря на то что такое хранилище, как правило, связано с внутренними накопителями на дисках сервера, его лучше рассматривать как привязанное к тому серверу, к которому оно присоединено.

- **Сетевое хранилище (Storage Area Network — SAN).** Это специально выделенная сеть, предоставляющая доступ к различным типам запоминающих устройств, включая библиотеки на магнитных лентах, накопители на автоматически сменяемых оптических дисках и дисковые массивы. Для серверов и прочих устройств в сети запоминающие устройства SAN подобны локально присоединяемым устройствам. Благодаря технологии блочного хранения данных на диске система SAN является, вероятно, самой распространенной формой хранения в очень крупных центрах обработки данных и фактически стала основной для приложений, интенсивного пользующихся базами данных. Таким приложениям требуется совместно используемая память, большая пропускная способность и разнесение стоек в центре обработки данных на большие расстояния.
- **Подключаемое к сети хранилище (Network Attached Storage — NAS).** Такая система состоит из подключаемых к сети запоминающих устройств, содержащих один или более накопителей на дисках, которые могут совместно использоваться многими разнородными компьютерами. Ее особая роль в сети состоит в хранении и обслуживании файлов. Как правило, накопители на дисках, подключаемые к сети, поддерживают встроенные механизмы защиты данных, в том числе накопители большой емкости с избыточностью или массивы RAID. NAS позволяет отделять обязанности по обслуживанию файлов от других серверов в сети и, как правило, обеспечивает более быстрый доступ к данным, чем традиционные файловые серверы.

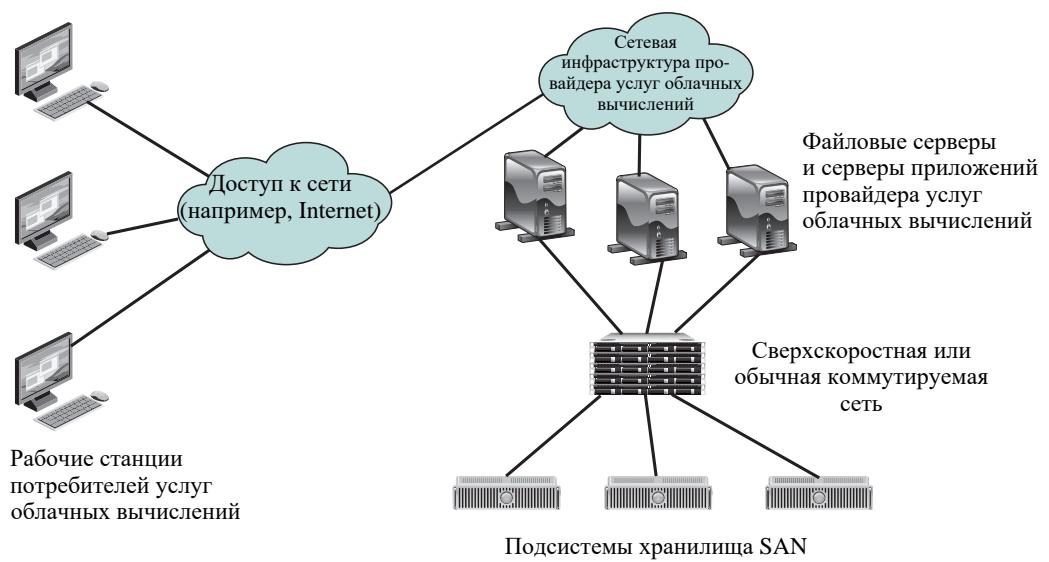
На рис. 16.7 показано различие хранилищ SAN и NAS. Как правило, потребитель услуг облачных вычислений реализует систему SAN в облачной инфраструктуре, но может воспользоваться и системой NAS. Облачная операционная система должна уметь приспособиться к обеим топологиям и предоставить прозрачный доступ потребителю услуг облачных вычислений, которому совсем не обязательно знать внутреннюю топологическую структуру хранения данных в облаке.

### Виртуальная сеть

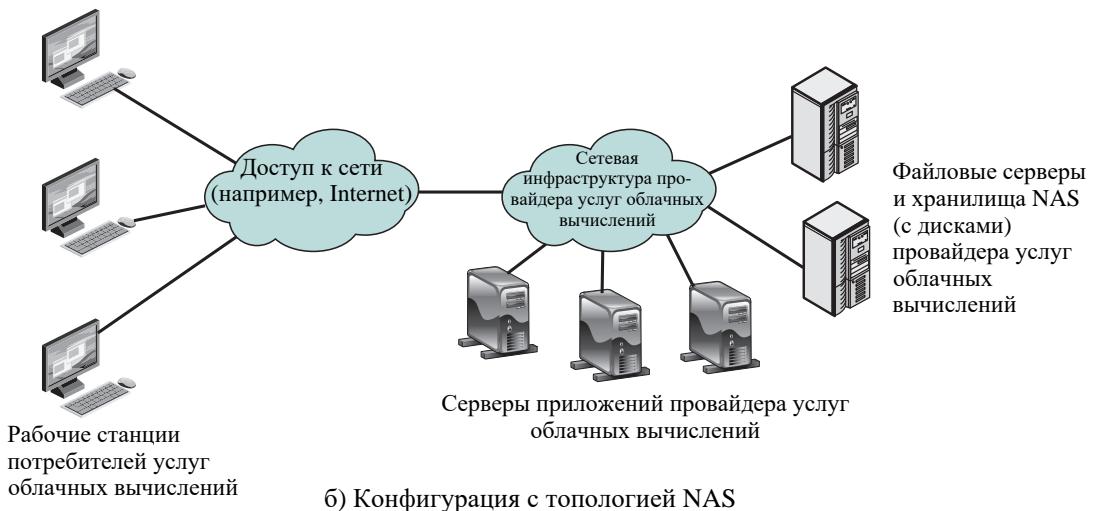
Компонент облачной операционной системы, отвечающий за виртуальную сеть, предоставляет услуги организации такой сети в облачной инфраструктуре. Он активизирует подключение к сети компьютеров, запоминающих устройств и прочих элементов данной инфраструктуры, а также более обширной среды за пределами облака. Этот компонент предоставляет также потребителям услуг облачных вычислений возможность создавать виртуальные сети среди виртуальных машин и сетевых устройств.

Помимо основных услуг подключения к сети, компонент виртуальной сети может предоставлять услуги и выполнять функции, перечисленные ниже.

- Поддержка схемы адресации инфраструктуры (таких схем может быть несколько) с распределением адресов и управлением ими.
- Процесс маршрутизации, способный соотносить адреса инфраструктуры с маршрутами через ее сетевую топологию.
- Процесс выделения пропускной способности, включая средства для установления приоритетов и обеспечения должного качества обслуживания (quality of service — QoS).
- Поддержка таких сетевых функций, как организация виртуальной локальной сети, распределение нагрузки и установка брандмаузеров.



а) Конфигурация с топологией SAN



б) Конфигурация с топологией NAS

Рис. 16.7. Хранилища SAN и NAS в облачной инфраструктуре

## Управление структурами данных

В облачной операционной системе предоставляется не только возможность хранить данные в исходном виде, но и услуги структурированного доступа к ним. Тремя распространенными структурами данных, поддерживаемыми в облачной операционной системе и модели IaaS, являются блочная, файловая и объектная.

При **блочном хранении** данные сохраняются на жестком диске в виде фиксированных блоков. Каждый блок является непрерывной последовательностью байтов. В системах SAN, как, впрочем, и в системах DAS, предоставляется доступ к хранилищу блоков данных. Блочное хранение пригодно для получения моментальных снимков и реализации отказоустойчивых схем вроде зеркального отображения. Как правило, в контроллерах SAN применяется механизм чтения при записи для сохранения локальной копии и синхронно отображаемого зеркального тома.

**Файловое хранение** данных, как правило, является синонимом NAS и состоит из массива хранилищ, контроллера некоторого типа, операционной системы и одного (или нескольких) протоколов сетевого хранения данных. В крупномасштабных средах виртуализации чаще всего применяется протокол NFS (Network File System). При этом данные хранятся на жестком диске в файлах, которые образуют структуру каталогов. У таких запоминающих устройств имеются свои процессоры и операционные системы, а доступ к ним осуществляется по стандартному протоколу через сеть TCP/IP. К типичным протоколам файлового хранения данных относятся следующие.

- NFS (Network File System — сетевая файловая система). Протокол NFS распространен в сетях на платформах Unix и Linux.
- SMB (Server Message Block — блок серверных сообщений). Протокол SMB (или CIFS) обычно применяется в сетях Windows.
- HTTP (Hypertext Transfer Protocol — протокол передачи гипертекста). Протокол HTTP чаще всего применяется при работе с веб-браузером.

Устройства NAS развертываются относительно просто, и клиенты довольно легко получают доступ к ним по распространенным протоколам. Все серверы и устройства NAS подключаются через общую сеть TCP/IP, а данные, хранящиеся на устройствах NAS, могут быть доступны буквально на любом сервере независимо от операционной системы сервера.

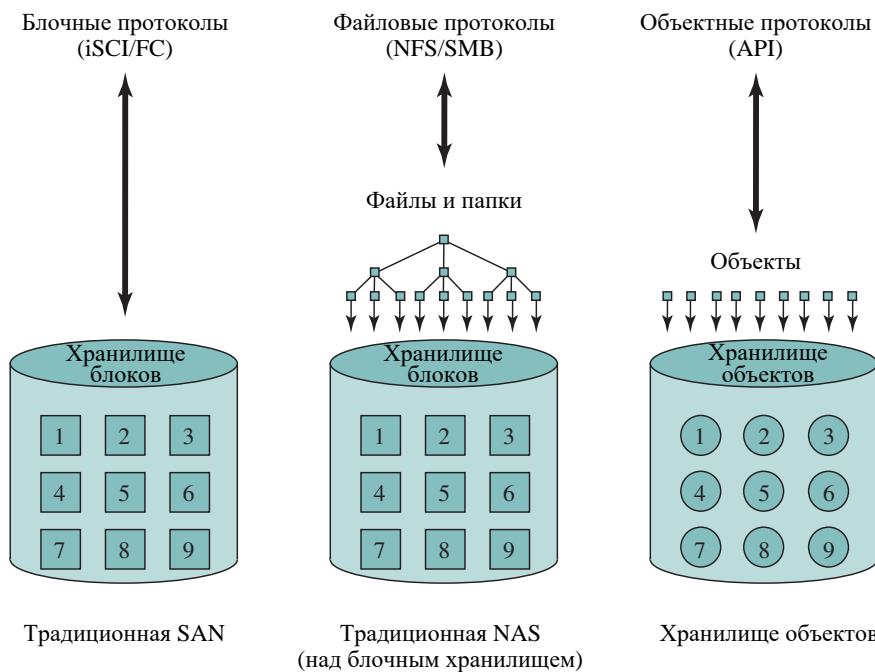
К преимуществам файлового хранения данных относится возможность рассматривать файл как блочное устройство или дисковый накопитель. Файлы можно легко добавлять, чтобы создавать более крупные виртуальные накопители, а также копировать в разные места. К недостаткам файлового хранилища данных относится трудность быстрого клонирования существующего образа виртуального диска в новый образ. Кроме того, хранилища данных на основе NAS, как правило, работают медленнее, чем аналогичные системы на основе DAS или SAN.

В отличие от файлового хранения данных, для **хранения объектов** применяется плоское адресное пространство вместо иерархической схемы, основанной на каталогах [170, 254]. Каждый объект состоит из контейнера, в котором хранятся как данные, так и метаданные, описывающие эти данные (например, дата, размер и формат). Каждому объекту присваивается уникальный идентификатор, по которому можно обращаться не-

посредственно к данному объекту. Идентификатор объекта хранится в базе данных или приложении и применяется для обращения к объектам в одном или нескольких контейнерах. Хранение объектов широко применяется в облачных системах.

Данные в при хранении в виде объектов, как правило, доступны по протоколу HTTP из веб-браузера или непосредственно через программный интерфейс API. Плоское адресное пространство в объектном хранилище обеспечивает простоту и большую степень масштабируемости, хотя данные в подобных системах нельзя модифицировать. Одним из главных преимуществ хранения объектов является возможность непосредственно связывать особые методы или реализации безопасности с конкретными данными, в отличие от необходимости получать такую возможность из соседней системы или службы.

На рис. 16.8 приведены для сравнения блочное, файловое и объектное хранение данных.



**Рис. 16.8.** Блочное, файловое и объектное хранение данных

### Управление и координация

На компонент облачной операционной системы, отвечающий за управление и координацию (management and orchestration — MANO), возложена функция управления средой IaaS. В документе NIST (*US Government Cloud Computing Technology Roadmap Volume I — Стратегический план правительства США по развитию технологий облачных вычислений, том I. SP 500-293, октябрь 2014 года*) эта функция определяется как композиция системных компонентов для поддержки действий провайдера услуг облачных вычислений по расположению, координации и управлению вычислительными ресурсами с целью предоставить услуги облачных вычислений их потребителям.

Компонент MANO охватывает следующие функции и услуги.

- **Координирование.** Отвечает за установку и конфигурирование новых сетевых служб, управление их жизненным циклом, глобальное управление ресурсами, проверку достоверности и разрешение запрашивать ресурсы.
- **Диспетчер виртуальных машин.** Осуществляет надзор над управлением жизненным циклом экземпляров виртуальных машин.
- **Диспетчер инфраструктуры.** Контролирует и управляет взаимодействием виртуальных машин с находящимися под его началом ресурсами вычисления, хранения и организации сети, а также их виртуализацией.

## OpenStack

OpenStack является программным проектом с открытым исходным кодом, который был основан фондом OpenStack (OpenStack Foundation) с целью выпустить свободно доступную облачную операционную систему [210, 218]. Главная цель такой операционной системы — обеспечить возможность создания больших групп виртуальных выделенных серверов и управления ими в среде облачных вычислений. OpenStack (в той или иной степени) встраивается в инфраструктуру центра обработки данных и программные продукты, предлагаемые для облачных вычислений компаниями Cisco, IBM, Hewlett-Packard и прочими провайдерами. Эта операционная система предоставляет мультиарендаторную модель IaaS и нацелена на удовлетворение потребностей в простой реализации масштабируемых открытых и закрытых облаков независимо от их размеров.

Операционная система OpenStack состоит из целого ряда независимых модулей, у каждого из которых имеется свое проектное и функциональное имя. Модульная структура легко поддается масштабированию и предоставляет наиболее употребительный набор базовых услуг. Как правило, компоненты конфигурируются вместе, чтобы предоставить исчерпывающие возможности IaaS. Тем не менее модульная структура такова, что ее компонентами можно, как правило, пользоваться независимо.

Чтобы особенности OpenStack стали понятнее, стоит различать три типа хранилищ данных, входящих в состав среды OpenStack и перечисленных ниже.

- **Сетевое блочное хранилище.** Этот тип хранилища обеспечивает перманентное хранение данных, монтируя одно или несколько сетевых блочных хранилищ. Оно представляет собой выделенное перманентное, доступное для чтения и записи блочное хранилище, которое может быть реализовано в виде корневого диска для экземпляра виртуальной машины или вторичного запоминающего устройства, присоединяемого к экземпляру виртуальной машины и/или отсоединяемого от него.
- **Хранилище объектов.** Это постоянное хранилище объектов в сети. С точки зрения самого хранения объекты рассматриваются как произвольные, неструктурированные данные. Сохраняемые объекты, как правило, записываются однократно, а читаются многократно. Это надежное средство хранения объектов с их резервными копиями. Списки управления доступом определяют порядок доступа к объектам для их владельцев и уполномоченных пользователей.
- **Хранилище образов виртуальных машин.** Образами виртуальных машин являются образы дисков, которые могут быть изначально загружены гипервизором в

виртуальную машину. Это может быть единственный образ, содержащий начальный загрузчик, ядро и операционную систему или же начальный загрузчик отдельно от ядра. Данный тип хранилищ данных допускает наличие специальных ядер и образов с изменяемыми размерами.

На рис. 16.9, взятом из [35], схематически представлена концептуальная архитектура операционной системы OpenStack вместе с взаимодействием ее основных программных компонентов. А в табл. 16.3 определяется их функциональное взаимодействие; в левом столбце указывается источник действия, в верхней строке — назначение этого действия. Компоненты OpenStack могут быть приблизительно разделены на пять перечисленных ниже функциональных групп.

- **Вычисления:** вычисления (Nova), образ (Glance).
- **Организация сети:** сеть (Neutron).
- **Хранение:** хранилище объектов (Swift), хранилище блоков (Cinder).
- **Общие службы:** безопасность (Keystone), информационная панель (Horizon), мониторинг (Ceilometer), координация (Heat).
- **Прочие дополнительные службы:** рассматриваются далее.

Рассмотрим сначала каждый из компонентов, перечисленных выше в первых четырех пунктах, а затем опишем вкратце другие компоненты.

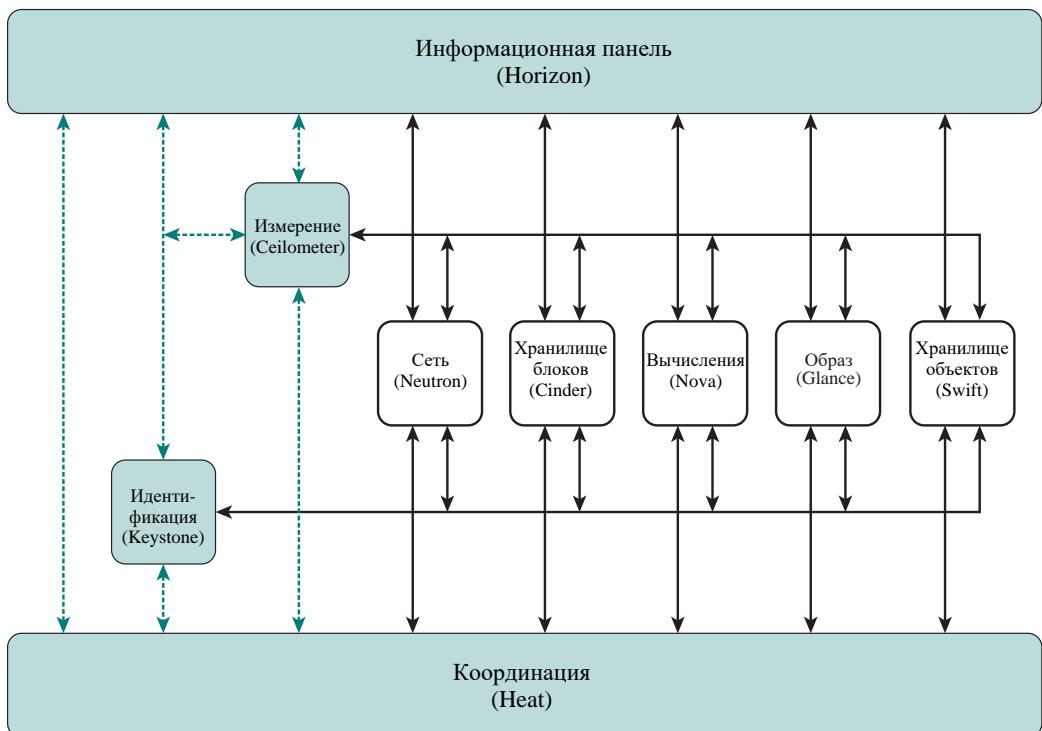


Рис. 16.9. Обобщенная архитектура ОС OpenStack

**Таблица 16.3. Функциональное взаимодействие компонентов OpenStack**

	<b>Glance (образ)</b>	<b>Horizon (информационная панель)</b>	<b>Nova (вычисления)</b>	<b>Swift (хранилище объектов)</b>	<b>Cinder (хранилище блоков)</b>	<b>Neutron (сеть)</b>
<b>Glance (образ)</b>			Отсылает сообщения	Сохраняет файлы на диске	Сохраняет блоки	
<b>Horizon (информационная панель)</b>	Предоставляет пользовательский интерфейс		Предоставляет пользовательский интерфейс	Предоставляет пользовательский интерфейс		Представляет пользовательский интерфейс
<b>Nova (вычисление)</b>		Получает сообщения			Сохраняет тома	
<b>Swift (хранилище объектов)</b>	Поставляет файлы на диске		Предоставляет тома			
<b>Cinder (хранилище блоков)</b>		Предоставляет тома				
<b>Neutron (сеть)</b>					Представляет экземпляр	
<b>Keystone (идентификация)</b>	Аутентифицирует с помощью	Аутентифицирует с помощью	Аутентифицирует с помощью	Аутентифицирует с помощью	Аутентифицирует с помощью	Аутентифицирует с помощью
<b>Heat (координация)</b>	Координирует	Координирует	Координирует	Координирует	Координирует	Координирует
<b>Trove (база данных)</b>						
<b>Ceilometer</b>	Осуществляет мониторинг	Осуществляет мониторинг	Осуществляет мониторинг	Осуществляет мониторинг	Осуществляет мониторинг	Осуществляет мониторинг
<b>Виртуальная машина</b>	Извлекает файлы образов					

**Таблица 16.3. (Окончание)**

	<b>Keystone (идентификация)</b>	<b>Heat (координация)</b>	<b>Trove (база данных)</b>	<b>Ceilometer</b>	<b>Виртуальная машина</b>
<b>Glance (образ)</b>	Аутентифицирует с помощью				Предоставляет файлы образа
<b>Horizon (информацион- ная панель)</b>	Аутентифицирует с помощью	Предоставляет пользовательский интерфейс	Предоставляет пользовательский интерфейс		Предоставляет пользовательский интерфейс
<b>Nova (вычисление)</b>	Аутентифицирует с помощью		Получает экземпляры		Запускает том
<b>Swift (хранилище объектов)</b>	Аутентифицирует с помощью				
<b>Cinder (хранилище блоков)</b>	Аутентифицирует с помощью				
<b>Neutron (сеть)</b>	Аутентифицирует с помощью				
<b>Keystone (идентифика- ция)</b>		Аутентифицирует с помощью	Аутентифицирует с помощью	Обеспечивает подключение к сети	
<b>Heat (координация)</b>	Аутентифицирует с помощью				
<b>Trove (база данных)</b>	Аутентифицирует с помощью				
<b>Ceilometer</b>	Аутентифицирует с помощью	Осуществляет мониторинг	Осуществляет мониторинг		
<b>Виртуальная машина</b>	Аутентифицирует с помощью				

## Вычисления (Nova)

Компонент Nova — это программа, управляющая виртуальными машинами на платформе облачных вычислений IaaS. Он управляет жизненным циклом экземпляров в среде OpenStack. В обязанности данного компонента входит порождение, планирование и вывод из эксплуатации машин по требованию. Таким образом, компонент Nova позволяет предприятиям и провайдерам услуг предоставлять вычислительные ресурсы по требованию, подготавливая крупные сети виртуальных машин к работе и управляя ими. По своим возможностям компонент Nova подобен веб-службе Amazon Elastic Compute Cloud (EC2). Он способен взаимодействовать с различными гипервизорами, как с открытым исходным кодом, так и с коммерческими. В состав компонента Nova не входит никакого программного обеспечения виртуализации, но вместо этого определяются драйверы, взаимодействующие с базовыми механизмами виртуализации, действующими в операционной системе хоста, а функциональные возможности предоставляются через API для веба. Таким образом, компонент Nova обеспечивает управление крупными сетями виртуальных машин и поддержку резервируемых и масштабируемых архитектур, включая управление экземплярами виртуальных машин на серверах и в сетях, а также управление доступом к ним. Для компонента Nova не требуется никакого предварительно устанавливаемого оборудования, и он полностью независим от гипервизора.

Компонент Nova состоит из пяти основных компонентов, перечисленных ниже и приведенных на рис. 16.10.

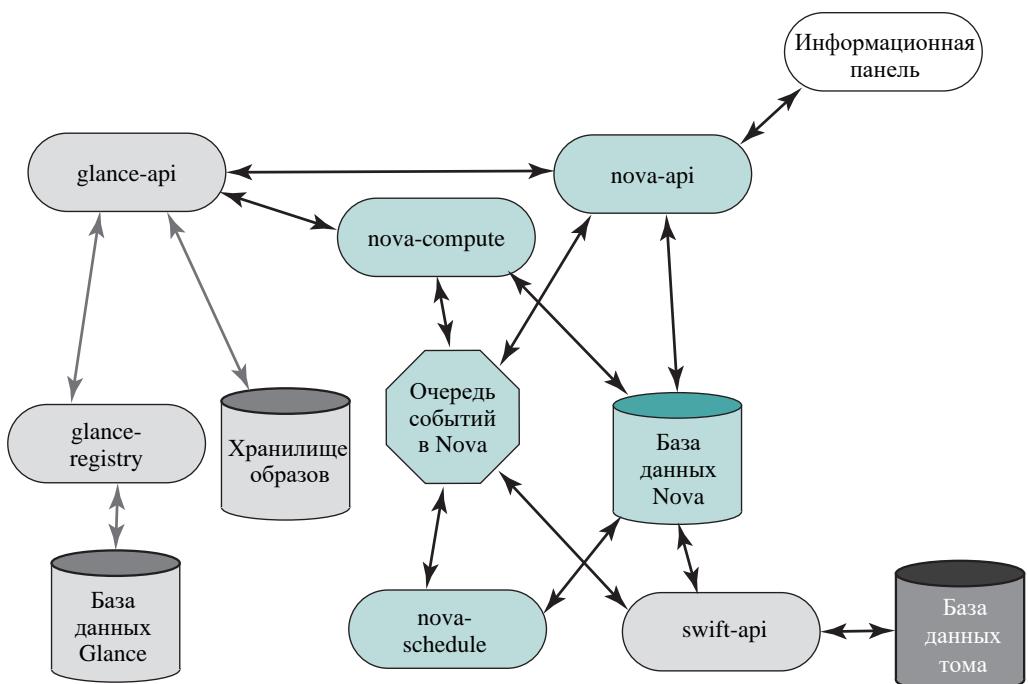


Рис. 16.10. Логическая архитектура компонента Nova

- **Сервер API.** Это внешний интерфейс информационной панели для пользователей и приложений.
- **Очередь сообщений.** Компоненты Nova обмениваются сведениями через очередь (действиями) и базу данных (информацией) для выполнения запросов API. Очередь сообщений реализует механизм для диспетчеризации обмениваемых команд с целью упростить такой обмен.
- **Контроллер вычислений.** Оперирует жизненным циклом экземпляров виртуальных машин; отвечает за создание и манипулирование виртуальными серверами и взаимодействует с компонентом Glance.
- **База данных.** Хранит большую часть состояния облачной инфраструктуры во время построения и выполнения, включая типы экземпляров, доступные для применения, применяемые экземпляры, доступные сети и проекты.
- **Планировщик.** Принимает запросы на экземпляры виртуальных машин и определяет, где именно они должны выполняться (т.е. на каком именно сервере вычислений).

Следует, однако, иметь в виду, что с компонентом Swift взаимодействует несколько компонентов. Этот компонент управляет созданием, присоединением и отключением томов от экземпляров вычислений.

### *Образ (Glance)*

Компонент Glance представляет собой систему поиска и извлечения образов дисков виртуальных машин, предоставляющую услуги выявления, регистрации и извлечения виртуальных образов через API. Компонент Glance предоставляет также интерфейс в стиле SQL для запроса информации в образах, размещаемых в различных хранилищах данных. Компонент вычисления операционной системы OpenStack пользуется такой возможностью во время предоставления экземпляра.

### *Сеть (Neutron)*

Компонент Neutron является проектом OpenStack, предназначенный для предоставления услуг подключения к сети интерфейсным устройствам, управляемым другими службами OpenStack (например, Nova). С этой целью сервер Neutron предоставляет веб-сервер, открывающий доступ к API компонента Neutron и передающий все вызовы веб-служб на обработку в модуль, подключаемый к Neutron. По существу, компонент Neutron предоставляет согласованный набор сетевых услуг для применения в других элементах (например, виртуальных машинах, модулях управления системами и иных сетях). Пользователи взаимодействуют с сетевыми функциями через графический пользовательский интерфейс информационной панели, а остальные системы управления и сети — с сетевыми службами через прикладной интерфейс API компонента Neutron.

В настоящее время компонент Neutron реализует виртуальные локальные сети второго уровня и маршрутизаторы третьего уровня на основе сетевого протокола IP. Имеются также расширения для поддержки брандмауэров, выравнивателей нагрузки и виртуальных частных сетей по стандарту IPSec.

Ниже перечислены главные преимущества от применения компонента Neutron [183].

- Применяя согласованный подход к организации сети для разнотипных виртуальных машин, компонент Neutron помогает провайдерам услуг эффективно работать в разнородных средах, что зачастую требуется в системах провайдеров услуг.
- Предоставляя согласованный набор прикладных интерфейсов API для подключения к разнообразным средам на нижнем физическом уровне, провайдеры услуг выигрывают от удобства изменения структуры своих базовых физических сетей, вообще не затрагивая логику предоставления услуг облачных вычислений.
- Провайдеры услуг координирования и управления системами, а также их собственный технический персонал могут воспользоваться прикладным интерфейсом API компонента Neutron, чтобы интегрировать управление сетью в облаке вместе с задачами управления услугами более высокого уровня. Это открывает немало возможностей, в том числе для текущего контроля соглашений об уровне обслуживания, а также для интеграции с такими платформами автоматизации, как каталоги и порталы, чтобы динамически управлять потребительскими облаками.

### Хранилище объектов (*Swift*)

Это распределенное хранилище объектов, создающее резервируемое и масштабируемое пространство для хранения многих петабайтов данных. Такое хранилище объектов представляет собой не традиционную файловую систему, а скорее распределенное хранилище статических данных вроде образов виртуальных машин, фотоархивов, архивов электронной почты, резервных копий и прочих архивов. Может использоваться компонентами Cinder для резервного копирования томов виртуальных машин.

### Хранилище блоков (*Cinder*)

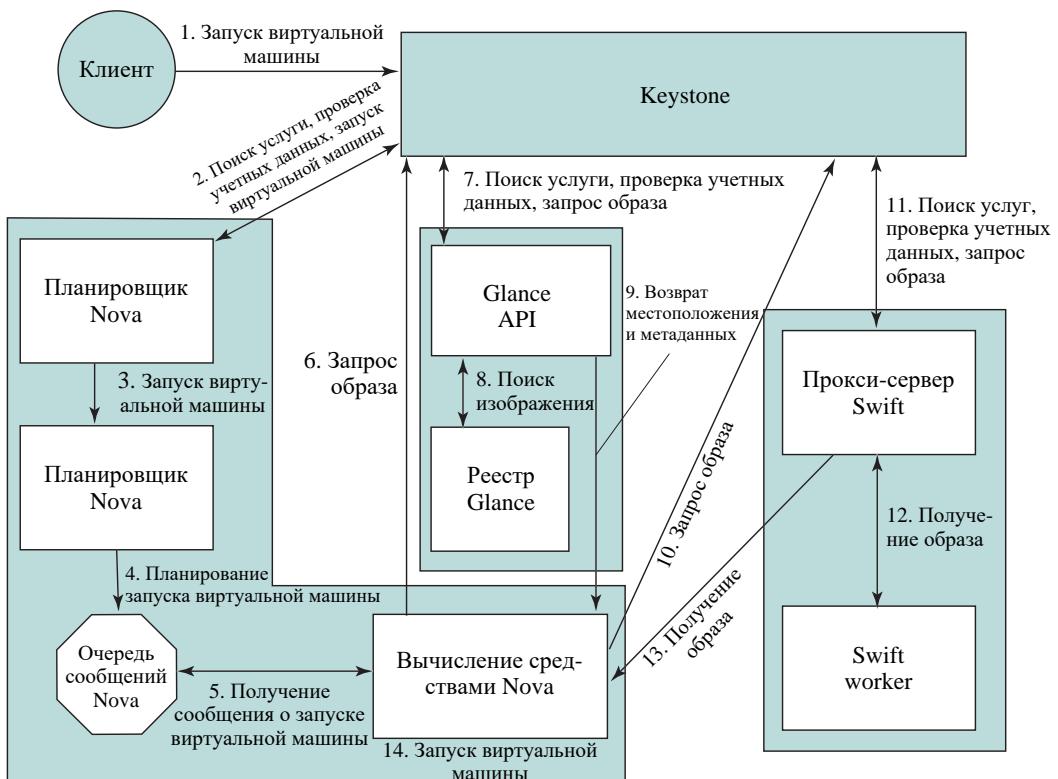
Компонент Cinder предоставляет перманентное хранилище блоков (или томов) для гостевых виртуальных машин. Для резервного копирования томов виртуальных машин в этом компоненте может использоваться компонент Swift. Кроме того, компонент Cinder взаимодействует с компонентом Nova, предоставляя тома для его экземпляров и допуская манипулирование томами, их типами и моментальными снимками через его прикладной интерфейс API.

### Идентификация (*Keystone*)

Компонент Keystone предоставляет общие услуги защиты, имеющие существенное значение для функционирования инфраструктуры облачных вычислений. Ниже перечислены основные услуги, предоставляемые этим компонентом.

- **Идентификация.** Это аутентификация пользовательской информации. Эта информация определяет роль и полномочия пользователя в данном проекте, а также служит основанием для механизма ролевого управления доступом (RBAC).
- **Токен.** После входа пользователя в систему по его имени и паролю ему присваивается токен, используемый в дальнейшем для управления доступом. Службы OpenStack хранят такие токены, используя их для запроса компонента Keystone во время выполнения операций.
- **Каталог услуг.** Конечные точки обслуживания в операционной системе OpenStack регистрируются компонентом Keystone для создания каталога услуг. Клиент, которому требуется обслуживание, обращается к компоненту Keystone и определяет конечную точку для вызова на основании возвращаемого каталога.
- **Стратегии.** Эта услуга приводит в действие разные уровни доступа пользователя.

На рис. 16.11 схематически показано, каким образом компонент Keystone взаимодействует с другими компонентами OpenStack для запуска новой виртуальной машины.



**Рис. 16.11.** Запуск виртуальной машины

### Информационная панель (Horizon)

Информационная панель является пользовательским веб-интерфейсом для управления облачной инфраструктурой. Она предоставляет администраторам и пользователям графический интерфейс для доступа, выделения и автоматизации ресурсов в облаке. Расширяемая структура данной панели упрощает подключение и открытие доступа к сторонним программным продуктам и таким услугам, как начисление платы за пользование ресурсами, текущий контроль и предоставление дополнительных инструментальных средств управления. Она взаимодействует с прикладными интерфейсами API всех остальных программных компонентов. Например, компонент Horizon дает пользователю или приложению возможность запустить экземпляр, назначить IP-адреса и сконфигурировать средства управления доступом.

### Мониторинг (Ceilometer)

Компонент Ceilometer предоставляет конфигурируемую совокупность функций для измерения таких показателей, как степень использования процессора и памяти, а также сетевого трафика. Это своеобразная справочная служба для начисления платы за пользование ресурсами, оценки производительности, масштабируемости и статистических целей.

## Координирование (Heat)

Компонент Heat координирует многие облачные приложения. Его цель — создать доступную человеку и машине услугу для управления всем жизненным циклом инфраструктуры и приложений в облаках OpenStack. Он реализует механизм координирования для запуска многих составных облачных приложений на основании шаблонов в форме текстовых файлов, которые могут трактоваться как исходный код. Компонент Heat совместим с инфраструктурой Amazon Cloudformation, которая становится фактическим стандартом.

## Прочие дополнительные службы

По мере развития проекта OpenStack разные его участники разрабатывают новые компоненты. На момент написания данной книги были доступны или находились на стадии разработки следующие компоненты.

- **База данных (Trove).** Это база данных как услуга, предназначенная для предоставления механизмов как реляционных, так и нереляционных баз данных. По умолчанию в компоненте Trove применяется система управления базой данных MySQL, позволяющая другим службам хранить конфигурации и управляющую информацию.
- **Служба обмена сообщениями (Zaqar).** Компонент Zaqar является мультиарендаторной облачной службой обмена сообщениями, предназначеннной для разработчиков как веб-приложений, так и мобильных приложений. Эта служба отличается наличием API, которым разработчики могут пользоваться для обмена сообщениями между различными компонентами их программ типа SaaS и мобильных приложений на основе разнообразных шаблонов взаимодействия. В основу этого API положен эффективный механизм обмена сообщениями, разработанный с учетом масштабирования и безопасности.
- **Управление ключами (Barbican).** Компонент Barbican предоставляет API для безопасного хранения данных, инициализации секретных значений (например, паролей, ключей шифрования и сертификатов по стандарту X.509) и управления ими.
- **Самоорганизация (Congress).** Компонент Congress предоставляет правило как услугу для любой совокупности услуг облачных вычислений, чтобы обеспечить самоорганизацию и согласованность динамических инфраструктур.
- **Эластичное уменьшение отображения (Sahara).** Компонент Sahara предназначен для снабжения пользователей простыми средствами, с помощью которых они могут настраивать кластеры Hadoop, устанавливая такие параметры, как версия Hadoop, кластерная топология и узловое оборудование. Как только пользователь заполнит поля всех параметров, компонент Sahara развернет кластер. Кроме того, компонент Sahara предоставляет средства для масштабирования уже настроенного кластера, добавляя и удаляя рабочие узлы по требованию.
- **Совместно используемые файловые системы (Manila).** Компонент Manila предоставляет координированный доступ к совместно используемым или распределенным файловым системам. И хотя основная доля потребляемых файловых ресурсов приходится на экземпляры вычислений в OpenStack, данная услуга предназначена быть доступной и как независимая возможность.

- **Контейнеры (Magnum).** Компонент Magnum предоставляет услуги API, чтобы сделать доступными в качестве ресурсов OpenStack такие механизмы коорденирования контейнеров, как Docker и Kubernetes.
- **Снабжение встраиваемыми средствами (Ironic).** Компонент Ironic предоставляет встраиваемые, а не виртуальные машины, ответвляемые от встраиваемого драйвера из компонента Nova. Этот компонент лучше всего рассматривать как интерфейс API для встроенного гипервизора и ряд подключаемых модулей, взаимодействующих с такими гипервизорами.
- **Служба DNS (Designate).** Компонент Designate предоставляет услуги DNS для пользователей OpenStack, в том числе API для управления доменами и записями.
- **Каталог приложений (Murano).** Компонент Murano внедряет каталог приложений в OpenStack, давая их разработчикам и администраторам облака возможность публиковать в просматриваемом и составленном по категориям каталоге различные готовые для применения в облаке приложения.

Перечисленные выше модульные компоненты легко настраиваются, позволяя провайдеру облачных услуг IaaS приспособливать рассмотренную здесь облачную операционную систему для решения конкретных задач.

## 16.3. ИНТЕРНЕТ ВЕЩЕЙ

Интернет вещей является последней разработкой в долгой и непрерывной революции в области вычислений и обмена данными. Его масштабы, повсеместность и влияние на повседневную жизнь, коммерческую и государственную деятельность затмевают любые прежние технические достижения. В этом разделе дается краткий обзор Интернета вещей, а более подробное описание особенностей его реализации приведено далее в данной книге.

### Вещи в Интернете вещей

Термин *Интернет вещей* (Internet of Things — IoT) означает расширение взаимосвязи интеллектуальных устройств: от бытовых приборов до крошечных датчиков. Преобладающим направлением в данной области является встраивание мобильных приемопередатчиков с малым радиусом действия в самые разные принадлежности и бытовые приборы, что дает возможность для проявления новых форм общения людей с вещами, а самим вещам — между собой. Ныне в Интернете поддерживается взаимосвязь миллиардов промышленных и частных объектов — как правило, через облачные системы. Объекты предоставляют информацию от датчиков, действуют в своей среде, а иногда видоизменяются, создавая условия для общего управления крупной системой вроде фабрики или города.

Интернет вещей приводится в действие главным образом глубоко встроеннымми устройствами. Такими устройствами являются низкоскоростные бытовые приборы с малой частотой сбора и использования данных, которые сообщаются между собой и представляют эти данные через пользовательский интерфейс. Встраиваемым устройствам (например, камерам видеонаблюдения с высоким разрешением, VoIP-телефонам и ряду других подобных устройств) требуются возможности потоковой передачи данных на высокой скорости. А подавляющему числу остальных устройств просто требуется время от времени доставлять пакеты данных.

## ЭВОЛЮЦИЯ

Применительно к поддерживаемым конечным системам эволюция Интернета прошла приблизительно через четыре поколения и достигла разработки IoT.

1. **Информационные технологии (IT).** Это персональные компьютеры, серверы, маршрутизаторы, брандмауэры и прочее оборудование, приобретаемое в качестве IT-устройств сотрудниками IT-отделов организаций и соединяемое главным образом через проводные каналы связи.
2. **Операционные технологии (OT).** Это машины и устройства со встроенными IT-разработанными не IT-организациями, такие как медицинская техника, системы оперативно-диспетчерского управления (supervisory control and data acquisition — SCADA), средства контроля технологических процессов, а также киоски, приобретенные как аппаратура сотрудниками OT-отделов организаций и соединяемые в основном через проводные каналы связи.
3. **Персональные технологии.** Это смартфоны, планшетные компьютеры и устройства для чтения электронных книг, приобретенные в качестве IT-устройств потребителями (сотрудниками организаций) и соединяемые исключительно через каналы беспроводной связи, которые зачастую доступны во многих формах.
4. **Устройства “датчик/привод”.** Это узкоспециализированные устройства, приобретаемые потребителями, сотрудниками IT- и OT-отделов организаций и соединяемые исключительно через каналы беспроводной связи (как правило, в одной форме) в качестве составляющих более крупных систем.

Именно четвертое поколение Интернета обычно считается Интернетом вещей, который заметно отличается применением миллиардов встроенных устройств.

## Компоненты IoT-устройств

Ниже перечислены главные компоненты IoT-устройства.

- **Датчик.** Измеряет некоторый параметр физического, химического или биологического объекта и выдает электронный сигнал, пропорциональный наблюдаемой характеристике, в аналоговой (в виде уровня напряжения) или цифровой (в виде последовательного ряда единиц и нулей) форме. Но в обоих случаях выход датчика, как правило, подается на вход микроконтроллера или другого управляющего элемента.
- **Привод.** Получает электронный сигнал от контроллера и соответственно реагирует на него, взаимодействуя со своим окружением, чтобы добиться определенного результата по некоторому параметру физического, химического или биологического объекта.
- **Микроконтроллер.** Это развитая логика работы интеллектуального устройства, обеспечиваемая глубоко встроенным микроконтроллером.
- **Приемопередатчик.** Состоит из электронных компонентов, требующихся для передачи и приема данных. Большинство устройств в Интернете вещей состоят из беспроводного приемопередатчика, способного обмениваться данными по WiFi, ZigBee или какой-нибудь другой схеме беспроводной связи.

- **Радиочастотная идентификация (Radio-Frequency Identification — RFID).** Это технология, в которой радиоволны применяются для идентификации объектов и которая все больше и больше становится основой для реализации Интернета вещей. Основными элементами системы RFID являются метки и считыватели. В частности, радиочастотные метки — это небольшие программируемые устройства, применяемые для слежения за объектами, животными и людьми, имеющие самые разные формы, размеры, функции и стоимость. А радиочастотные считыватели получают, а иногда и перезаписывают, информацию, хранящуюся в радиочастотных метках, находящихся в радиусе действия считывателей (от нескольких сантиметров до нескольких метров). Радиочастотные считыватели обычно подключаются к вычислительной системе, регистрирующей и форматирующей полученную информацию для последующего применения.

## Интернет вещей в контексте облака

Чтобы стали понятнее функции Интернета вещей в контексте облака, полезно сделать общий его обзор в контексте целой корпоративной сети, состоящей из сторонней сети и элементов облачных вычислений. Общее схематическое представление контекста Интернета вещей приведено на рис. 16.12.

## Границы

Границей типичной корпоративной сети служит сеть устройств, функционирующих через IoT и состоящих из датчиков, а возможно, и приводов. Такие устройства могут общаться одно с другим. Например, вся группа датчиков может передавать свои данные одному датчику, накапливающему данные, собираемые объектом более высокого уровня. На этом уровне может также иметься целый ряд **шлюзов**. В частности, шлюз устанавливает соединение между устройствами, функционирующими через IoT, с сетями передачи данных более высокого уровня. Он выполняет необходимый обмен данными между протоколами, применяемыми в сетях передачи данных, а также теми протоколами, которые используются устройствами. Он может выполнять и основную функцию накопления данных.

## Туманные вычисления

Во многих развернутых вариантах IoT массивные объемы данных могут быть сформированы распределенной сетью датчиков. Например, морские нефтепромысловые и нефтеперерабатывающие объекты могут ежедневно формировать терабайты данных. А в самолете ежесекундно могут формироваться многие терабайты данных. Вместо того чтобы хранить все эти данные постоянно (или хотя бы в течение длительного периода) в центральном хранилище, доступном для приложений IoT, зачастую желательно обработать как можно больше данных поближе к сенсорам. Таким образом, цель того, что иногда называется уровнем граничных вычислений, состоит в преобразовании потоков проходящих через сеть данных в информацию, пригодную для хранения и обработки на более высоком уровне. Обрабатывающим элементам на этом уровне, возможно, придется манипулировать большими объемами данных и выполнять операции их преобразования, чтобы в конечном итоге хранить намного меньшие объемы данных.

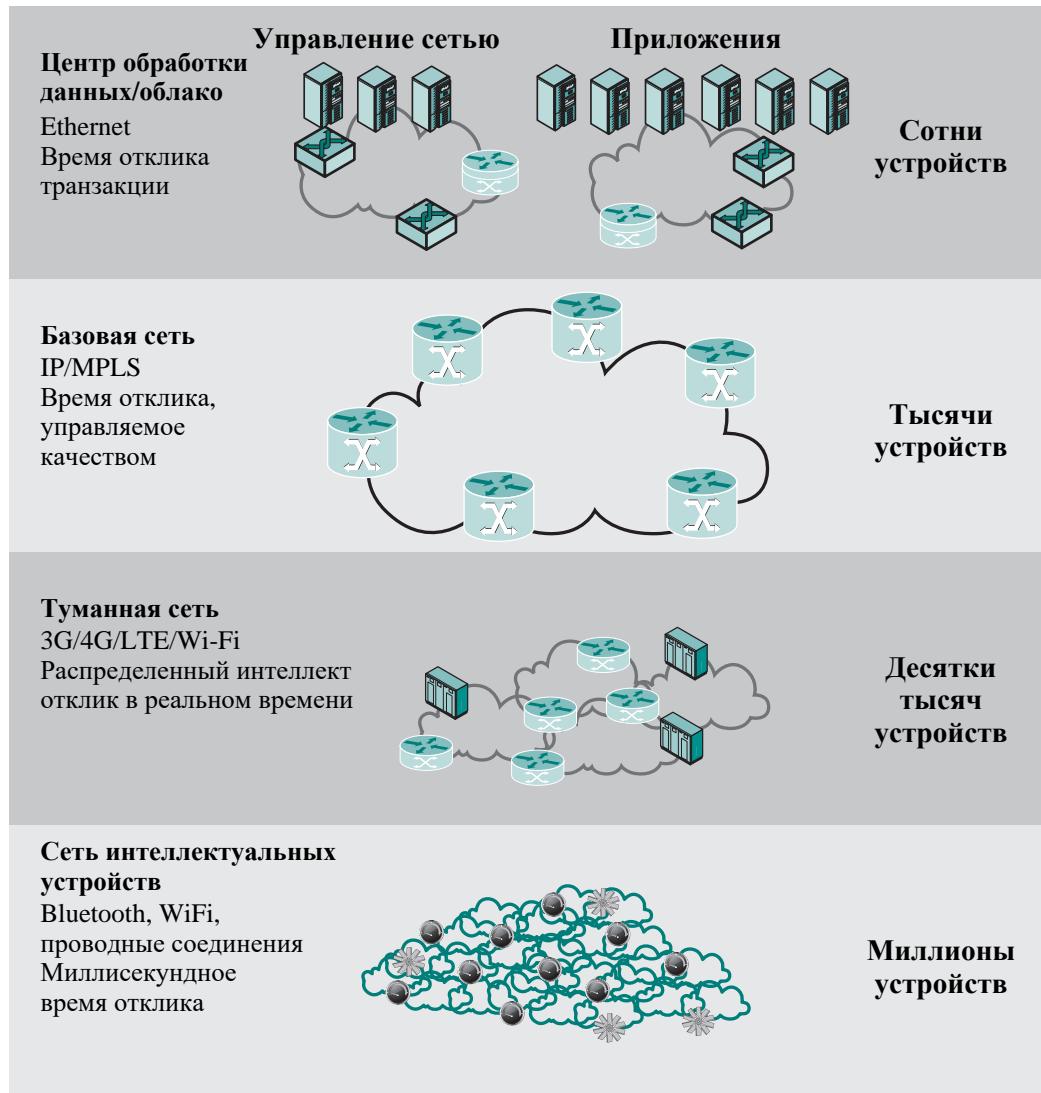


Рис. 16.12. Контекст Интернета вещей

Ниже приведены некоторые примеры операций туманных вычислений.

- **Оценивание.** Это оценивание данных по критериям, определяющим необходимость их обработки на более высоком уровне.
- **Форматирование.** Переформатирование данных для последовательной обработки на более высоком уровне.
- **Расширение/декодирование.** Обработка зашифрованных данных в дополнительном контексте (например, в исходном виде).

- **Очищение/сокращение.** Это сокращение и/или сведение данных для минимизации их влияния и сетевого трафика в сети и системах обработки на более высоком уровне.
- **Аналитическое оценивание.** Определение, являются ли данные пороговыми; может включать в себя переадресацию данных в другие места назначения.

Как правило, устройства для туманных вычислений физически развертываются у границы сети IoT, т.е. рядом с датчиками и другими устройствами, формирующими данные. Таким образом, часть основной обработки больших объемов формируемых данных снимается с прикладного программного обеспечения IoT, действующего в центре обработки данных, и передается далее.

Туманные вычисления и услуги обещают стать отличительной особенностью Интернета вещей. Они представляют собой тенденцию в организации современных сетей, противоположную облачным вычислениям. При облачных вычислениях массивные, централизованные ресурсы хранения и обработки данных доступны распределенным потребителям через средства организации сети в облаке и предоставляются относительно небольшому числу пользователей. А при туманных вычислениях массовое число отдельных интеллектуальных объектов соединено с помощью средств организации сети, предоставляющих ресурсы обработки и хранения данных вблизи конечных устройств на границе Интернета вещей. Туманные вычисления разрешают трудности, возникающие в результате деятельности тысяч или миллионов интеллектуальных устройств, в том числе вопросы безопасности, конфиденциальности, ограничений на пропускную способность сети и времени задержки. А сам термин *туманные вычисления* навеян тем обстоятельством, что туман обычно стелется по земле, тогда как облака плывут высоко в небе.

## Базовая сеть

Базовая сеть, иначе называемая **магистральной** (backbone network), соединяет географически распределенные туманные сети, а также предоставляет доступ к другим сетям, не являющимся частью корпоративной сети. Как правило, в базовой сети применяются высокопроизводительные маршрутизаторы для повышения степени резервирования и пропускной способности. Кроме того, базовая сеть может быть подключена к высокопроизводительным серверам большой емкости (например, к серверам крупных баз данных и объектам размещения закрытых облаков). Некоторые из маршрутизаторов базовой сети могут быть исключительно внутренними, обеспечивая резервирование и дополнительную пропускную способность, но не выполняя функции граничных маршрутизаторов.

## Облачная сеть

В облачной сети предоставляются возможности хранения и обработки массивных объемов данных, накопленных из устройств, функционирующих на границе IoT. На облачных серверах размещаются также приложения, взаимодействующие с подобными устройствами, управляющими ими и анализирующими данные, сформированные в IoT.

Сравнительные характеристики облачных и туманных вычислений приведены в табл. 16.4.

**ТАБЛИЦА 16.4. СРАВНЕНИЕ ХАРАКТЕРИСТИК ОБЛАЧНЫХ И ТУМАННЫХ ВЫЧИСЛЕНИЙ**

	Облачные вычисления	Туманные вычисления
<b>Местоположение ресурсов обработки и хранения</b>	В центре	На границе
<b>Время задержки</b>	Большое	Малое
<b>Доступ</b>	Фиксированный или беспроводный	В основном беспроводный
<b>Поддержка мобильности</b>	Отсутствует	Присутствует
<b>Управление</b>	Централизованное/иерархическое (полное)	Распределенное/иерархическое (частичное)
<b>Доступ к услугам</b>	Через базовую сеть	На оконечных/ручных устройствах
<b>Доступность</b>	99,99%	Непостоянная/избыточная
<b>Количество пользователей и устройств</b>	Десятки и сотни миллионов	Десятки миллиардов
<b>Основной генератор содержимого</b>	Человек	Устройства и датчики
<b>Формирование содержимого</b>	В центральном месте	Где угодно
<b>Потребление содержимого</b>	На конечном устройстве	Где угодно
<b>Программная виртуальная инфраструктура</b>	Центральные корпоративные серверы	Пользовательские устройства

## 16.4. ОПЕРАЦИОННЫЕ СИСТЕМЫ для Интернета вещей

Устройства для Интернета вещей являются встроенными, а следовательно, у них имеется встроенная операционная система. Тем не менее большинство подобных устройств обладают весьма ограниченными ресурсами, включая ограниченный объем памяти, ограниченную производительность процессора, а также учитывая жесткие требования к потребляемой мощности. Следовательно, многие встроенные операционные системы попросту слишком велики и требуют слишком много ресурсов, чтобы их можно было применять в устройствах из IoT, хотя некоторые операционные системы (например, TinyOS) все же пригодны для этой цели. В этом разделе сначала определяются типы устройств, которые обычно считаются целевыми для операционных систем, применяемых в IoT, затем исследуются характеристики встроенной операционной системы, подходящей для таких устройств, а в конце рассматривается RIOT — распространенная в IoT операционная система.

## Устройства с ограниченными ресурсами

Термин *устройство с ограниченными ресурсами* (*constrained device*) все чаще применяется для обозначения обширной разновидности устройств для Интернета вещей. Устройством с ограниченными ресурсами для IoT называется такое устройство, которое обладает ограниченным объемом энергозависимой и энергонезависимой памяти, ограниченной вычислительной мощностью процессора, а также низкоскоростным приемопередатчиком данных. Многие устройства для IoT (особенно небольшие и многочисленные устройства) обладают ограниченными ресурсами. Как отмечается в [216], благодаря постоянным технологическим усовершенствованиям согласно закону Мура встроенные устройства становятся все более и более дешевыми, компактными и энергоэффективными, хотя и не обязательно более мощными. Типичные встроенные устройства для IoT оснащены 8- или 16-разрядными микроконтроллерами, располагающими очень малым объемом памяти, а следовательно, довольно скромными возможностями хранения данных. Устройства с ограниченными ресурсами зачастую оснащены радиоканалом стандарта IEEE 802.15.4 для передачи данных по беспроводным персональным сетям фреймами размером до 127 октетов на небольшой скорости, порядка 20–250 Кбит/с, при малом потреблении электроэнергии.

В документе RFC 7228 (Terminology for Constrained-Node Networks — Терминология для сетей с ограниченным числом узлов) [27] определяются три класса устройств с ограниченными ресурсами (табл. 16.5).

**Таблица 16.5. Классы устройств с ограниченными ресурсами**

Класс	Объем данных, Кбайт	Размер кода, Кбайт
0	<< 10	<< 100
1	~ 10	~ 100
2	~ 50	~ 250

**Класс 0.** Это устройства с весьма ограниченными ресурсами, как правило датчиками, называемыми *пылинками* или *интеллектуальной пылью*. Такие “пылинки” могут быть внедрены или распределены по участку местности для сбора данных и обмена с другими пылинками или центральной точкой сбора данных. Например, фермер, виноградарь или эколог может оснастить пылинки датчиками для измерения температуры, влажности и прочего, превратив каждую пылинку в миниатюрную метеостанцию. С помощью таких пылинок, рассеянных по полю, винограднику или лесу, можно следить за микроклиматом на отдельном участке местности. Устройства класса 0 обычно нельзя полностью защитить или управлять ими в традиционном смысле. Вероятнее всего, они предварительно настраиваются (и редко перенастраиваются, если это вообще возможно) очень малым набором данных.

- **Класс 1.** Это устройства с настолько ограниченным местом для хранения кода и возможностями обработки данных, что им нелегко обмениваться информацией с другими узлами Интернета, пользующимися полным стеком сетевых протоколов. Тем не менее они способны в достаточной степени пользоваться стеком протоколов, специально предназначенным для сетевых узлов с ограниченными ресурсами (например, протоколом CoAP — Constrained Application Protocol), а также принимать участие в полноценном диалоге без помощи шлюзового узла.

- **Класс 2.** Это устройства с менее ограниченными ресурсами, по существу, способные поддерживать большинство тех же самых стеков протоколов, что и у переносных компьютеров или серверов. Тем не менее их ресурсы весьма ограничены по сравнению с устройствами более высокого класса, применяемыми в IoT. Поэтому им требуются упрощенные и энергоэффективные протоколы и сетевой трафик на низкой скорости передачи.

Устройства класса 0 обладают настолько ограниченными ресурсами, что для них не годится обыкновенная операционная система. Такие устройства выполняют весьма ограниченную, специализированную функцию или ряд функций, которые могут быть запрограммированы непосредственно на оборудовании. Устройства класса 1 и 2, как правило, менее специализированы. Операционная система вместе с функциями ее ядра и поддерживаемыми библиотеками дают разработчикам программного обеспечения возможность проектировать приложения, в которых применяются функции операционной системы и которые могут быть выполнены на самых разных устройствах. Тем не менее многие встроенные операционные системы (например,  $\mu$ Clinix) потребляют слишком много ресурсов и энергии, чтобы их можно было применять в устройствах с ограниченными ресурсами. Такие операционные системы, как правило, называются операционными системами для Интернета вещей.

## Требования к операционным системам для Интернета вещей

В [99] перечислены следующие характеристики, которыми должна обладать операционная система для IoT.

- **Небольшой объем занимаемой памяти.** В табл. 16.5 указаны ограничения на объем памяти для устройств с ограниченными ресурсами. Такой объем памяти на много порядков меньше, чем у смартфонов, планшетных компьютеров и разнообразных более крупных встроенных устройств. В результате таких требований нужны библиотеки, оптимизированные как по размеру, так и по производительности, а также структуры данных, эффективно использующие отведенное им пространство.
- **Поддержка разнородного оборудования.** В самых крупных системах (например, серверах, персональных и переносных компьютерах) преобладает архитектура процессоров семейства Intel x86. В менее крупных системах (например, смартфонах и целом ряде классов устройств для IoT) господствует архитектура ARM. Но устройства с ограниченными ресурсами основываются на различных архитектурах микроконтроллеров и семействах процессоров, особенно 8- и 16-разрядных. На устройствах с ограниченными ресурсами развертываются также самые разные технологии связи.
- **Подключение к сети.** Это требование имеет существенное значение для сбора данных, разработки распределенных приложений для Интернета вещей и обслуживания удаленных систем. В устройствах с низким потреблением энергии и минимальными ресурсами применяются самые разные методы и протоколы передачи данных, в том числе следующие.
  - IEEE 802.15.4 (низкоскоростная беспроводная персональная сеть)
  - Bluetooth Low Energy (BLE)

- 6LoWPAN (протокол IPv6 для маломощных беспроводных персональных сетей)
- CoAP (Constrained Application Protocol — Прикладной протокол для устройств с ограниченными ресурсами)
- RPL (Routing Protocol for Low and Lossy Networks — Протокол маршрутизации для маломощных сетей с потерями)
- **Энергоэффективность.** Для любого встроенного устройства, и особенно с ограниченными ресурсами энергоэффективность имеет первостепенное значение. В ряде случаев устройства должны работать в Интернете вещей годами с одним зарядом батареи питания [173]. Производители микросхем удовлетворяют это требование, достигая как можно большей энергоэффективности процессоров (см., например, [223]). Кроме того, был разработан целый ряд схем беспроводной передачи данных, предназначенных для сведения к минимуму потребляемой мощности [84]. Не менее важна роль, которую должна играть операционная система. Так, в [99] предполагается, что основные требования к операционной системе для IoT должны быть следующими: 1) предоставление возможности для энергосбережения на верхнем уровне и 2) как можно большее использование специализированных методов, например таких как циклический режим передачи данных по радиоканалу или сведение к минимуму количества периодически выполняемых задач.
- **Возможности работы в реальном времени.** Обширному ряду устройств для Интернета вещей требуется поддержка работы в реальном времени [244]. К их числу относятся следующие возможности.
  - Передача потоков данных от датчиков в реальном времени. Так, большинство приложений для сетей датчиков (например, в наружном наблюдении) обычно имеют чувствительный ко времени характер, когда пакеты должны быть пересланы своевременно. Поэтому гарантия работы в реальном времени является необходимым требованием, предъявляемым к подобным приложениям [67].
  - Двунаправленный контроль в широких пределах. Например, обмениваясь информацией, автомашины (или самолеты) контролируют друг друга во избежание столкновений; люди при встрече автоматически обмениваются информацией, что может оказывать влияние на их последующие действия; когда физиологические данные пациентов передаются врачам, те реагируют на них в реальном времени.
  - Реакция в реальном времени на события, связанные с безопасностью.Таким образом, операционная система для IoT должна быть в состоянии удовлетворить требованиям своевременного выполнения операций и гарантировать требуемое время выполнения или задержки прерывания даже в наихудшем случае.
- **Безопасность.** Устройства для Интернета вещей многочисленны и зачастую развертываются в небезопасных местах, а также имеют слишком ограниченные ресурсы для обработки и хранения данных в памяти, чтобы поддерживать протоколы безопасности и механизмы повышенной сложности. Обычно такие устройства связываются в беспроводном режиме, что повышает их уязвимость. Поэтому безопасность в IoT имеет высокий приоритет, но в то же время она оказывается труднодостижимой [243]. В рекомендации МСЭ-Т Y-2060 (*Overview of the Internet of things* — Обзор Интернета вещей, от 2 июня 2012 года) перечислены следующие возможности для обеспечения безопасности, желательные в устройствах IoT.

- **На прикладном уровне:** авторизация, аутентификация, защита конфиденциальности и целостности прикладных данных, защита частной информации, аудит безопасности и защита от вирусов.
- **На сетевом уровне:** авторизация, аутентификация, соблюдение конфиденциальности данных, сигнализация о нарушении как конфиденциальности, так и целостности данных.
- **На уровне устройства:** авторизация, аутентификация, проверка целостности устройства, управление доступом, конфиденциальность и защита целостности данных.

Таким образом, в операционных системах для IoT должны быть предоставлены механизмы, необходимые для обеспечения безопасности в устройствах с ограниченными ресурсами, а также механизмы обновления программного обеспечения устройств, уже развернутых в IoT.

## Архитектура операционной системы для Интернета вещей

Имеется целый ряд встроенных операционных систем, которые можно считать пригодными для устройств с ограниченными ресурсами, применяемых в IoT. Полезный обзор таких операционных систем приведен в [99]. Два других обзора, сделанных в [67] и [213], сосредоточены на беспроводных сетях датчиков. И хотя эти системы во многом различны, в общей их структуре, приведенной на рис. 16.13, продемонстрированы главные компоненты типичной операционной системы для Интернета вещей. Эти компоненты рассматриваются ниже.

- **Системные библиотеки и библиотеки поддержки.** Упрощенный набор библиотек, включая командную оболочку, протоколирование и функции шифрования.
- **Драйверы устройств и логическая файловая система.** Модульный набор упрощенных драйверов устройств и поддержка файловой системы, которым требуется минимальная настройка на конкретное устройство и приложения.
- **Стек протоколов для маломощных сетей.** К сетевой связности различных устройств с ограниченными ресурсами для IoT предъявляются разные требования. Так, во многих сетях датчиков устройствам для IoT требуются лишь ограниченные возможности связи, позволяющие датчикам обмениваться данными между собой или со шлюзом, через который данные будут передаваться дальше. А в других случаях устройства (даже с ограниченными ресурсами) для IoT должны плавно интегрироваться с Интернетом и поддерживать связь с другими машинами в Интернете. Таким образом, операционная система для IoT должна предоставлять возможность конфигурировать сетевой стек для поддержки протоколов, специально предназначенных для маломощных сетей, хотя сюда должна быть включена возможность поддерживать и уровень межсетевого протокола IP [189].
- **Ядро.** Как правило, ядро предоставляет планировщик, модель для заданий, механизм взаимных исключений и другие формы синхронизации, а также таймеры.

**Уровень аппаратной абстракции (HAL).** На этом уровне находится программное обеспечение, предоставляющее согласованный прикладной интерфейс API для верхних уровней и отображающее операции верхнего уровня на конкретную аппаратную платформу. Ниже перечислены наиболее употребительные интерфейсы, которые могут поддерживаться на данном уровне.

- **Интерфейс ввода-вывода общего назначения (General Purpose Input/Output — GPIO).** Предоставляет обобщенный контакт, который может быть назначен пользователем как для ввода, так и для вывода данных во время выполнения. Оказывается удобным, когда число контактов ограничено.
- **Универсальный асинхронный приемопередатчик (Universal Asynchronous Receiver/Transmitter — UART).** Асинхронный канал последовательного обмена цифровыми данными.
- **Последовательный периферийный интерфейс (Serial Peripheral Interface — SPI).** Синхронный канал последовательного обмена цифровыми данными.
- **Шина для соединения интегральных схем (Inter-Integrated Circuit — I2C).** Последовательная компьютерная шина, применяемая для присоединения низкоскоростных периферийных интегральных схем к процессорам и микроконтроллерам для обмена данными на коротком расстоянии в пределах печатной платы.



Рис. 16.13. Типичная структура операционной системы для IoT

## Операционная система RIOT

Как упоминалось ранее, далеко не все встроенные операционные системы пригодны для устройств с ограниченными ресурсами, применяемых в IoT. Например, из двух таких операционных систем, рассмотренных в главе 13, “Встроенные операционные системы”, операционной системе  $\mu$ Clinix требуется слишком много памяти, тогда как операционная система TinyOS вполне пригодна для IoT. В этом разделе рассматривается RIOT — операционная система с открытым исходным кодом, специально предназначенная для устройств с ограниченными ресурсами, применяемых в IoT [13]. В табл. 16.6 приведены сравнительные характеристики операционных систем  $\mu$ Clinix, TinyOS и RIOT, а на рис. 16.14 — общая структура операционной системы RIOT.

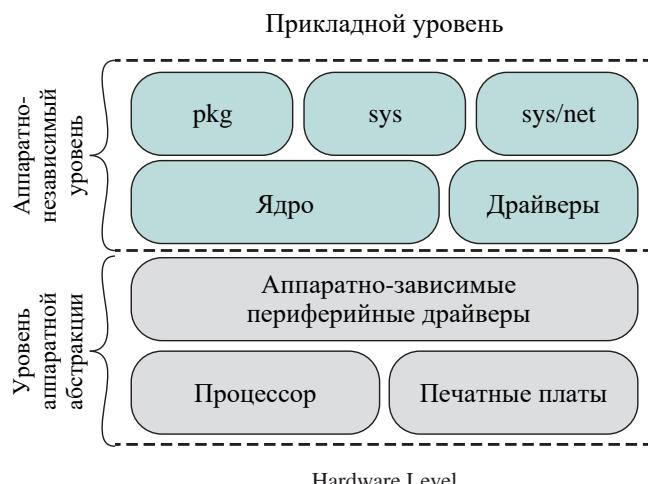
**Таблица 16.6. СРАВНИТЕЛЬНЫЕ ХАРАКТЕРИСТИКИ ОС  $\mu$ Clinix, TinyOS И RIOT**

	$\mu$ Clinix, Мбайт	TinyOS, Кбайт	RIOT, Кбайт
Минимальный объем оперативной памяти	< 32	< 1	~ 1,5
Минимальный объем постоянной памяти	< 2	< 4	~ 5
Поддержка языка C	✓	✗	✓
Поддержка языка C++	✓	✗	✓
Многопоточность	✓	○	✓
Микроконтроллеры без (MMU)	✓	✓	✓
Модульность	○	✗	✓
Режим реального времени	○	✗	✓

✓ — полная поддержка

○ — частичная поддержка

✗ — поддержка отсутствует



**Рис. 16.14. Общая структура операционной системы RIOT**

## Ядро RIOT

В системе RIOT применяется архитектура микроядра, а это означает, что ядро, называемое в RIOT основным (core) модулем, содержит только самые необходимые функции, в том числе для планирования, взаимодействия процессов (IPC), синхронизации и обработки запросов на прерывание (IRQ). Все остальные функции операционной системы, включая драйверы устройств и системные библиотеки, выполняются, как потоки. Благодаря такому применению потоков выполнения приложения и прочие части системы выполняются в собственном контексте, причем многие из таких контекстов могут действовать одновременно, а взаимодействие процессов обеспечит безопасный и синхронизированный обмен данными между ними по определенным приоритетам.

Преимущество архитектуры микроядра заключается в том, что систему нетрудно сконфигурировать только из того минимального состава программного обеспечения, который требуется для приложений на конкретном устройстве в IoT.

Ниже перечислены модули, входящие в состав ядра операционной системы RIOT.

- **Обработка запросов на прерывание.** Предоставляет API для управления обработкой прерываний.
- **Утилиты ядра.** Утилиты и структуры данных, применяемые в ядре.
- **Почтовые ящики.** Реализует ящики для хранения электронной почты.
- **Обмен сообщениями/взаимодействие процессов.** Предоставляет API для обмена сообщениями и взаимодействия между процессами.
- **Управление электропитанием.** Предоставляет интерфейс для управления электропитанием на уровне ядра.
- **Планировщик.** Планировщик заданий в RIOT.
- **Запуск и конфигурирование.** Конфигурационные данные и код запуска ядра.
- **Синхронизация.** Мьютексы для синхронизации потоков.
- **Поточная обработка.** Поддержка многопоточности.

В отличие от многих других операционных систем, одним из примечательных свойств RIOT является применение энергосберегающего планировщика. В отсутствие ожидающих выполнения задач операционная система RIOT переключается на бездействующий поток выполнения. Этот поток служит для определения режима самого глубокого сна в зависимости от конкретных применяемых периферийных устройств. В итоге планировщик продлевает до максимума время нахождения системы в спящем режиме, сводя к минимуму потребление энергии операционной системой. И только прерывания (внешние или генерируемые ядром) выводят систему из состояния бездействия. Кроме того, все функции ядра также сводятся к минимуму, что дает возможность выполнять ядро даже в системах с очень низкой тактовой частотой. Планировщик служит также для сведения к минимуму переключений потоков выполнения с целью сократить накладные расходы. Такая стратегия подходит для тех устройств в IoT, которым не требуется взаимодействовать с пользователем.

## Другие аппаратно-независимые модули

В состав библиотеки `sys` входят структуры данных (например, `bloom` и `color`), библиотеки шифрования (например, хеши и AES), высоконадежные API (например, реализации стандарта Posix), средства управления памятью (например, `malloc`), командная оболочка RIOT и другие наиболее употребительные модули системных библиотек.

Все программное обеспечение, связанное с организацией сети, находится в подкаталоге `sys/net`. Сюда относятся стек сетевых протоколов, API для работы в сети, а также программное обеспечение для конкретных типов сетей.

Библиотека `pkg` обеспечивает поддержку целого ряда внешних библиотек (например, OpenWSN и microsoap). В состав RIOT входит специальный файл `Makefile` для каждой поддерживаемой библиотеки, который загружает библиотеку (и возможно, вносит в нее ряд исправлений, чтобы она могла нормально работать с RIOT).

## Уровень аппаратной абстракции

Этот уровень RIOT состоит из трех наборов программного обеспечения. Для каждого поддерживаемого процессора в каталоге `CPU` имеется подкаталог с именем этого процессора. В этих подкаталогах содержатся все конфигурации, специфичные для конкретных процессоров, такие как реализации управления электропитанием, обработки прерываний, код запуска, инициализации системных часов и работы с потоками (например, переключения контекста).

Платформо-зависимый код разделяется на два логических элемента: процессоры и печатные платы. Отдельная плата имеет ровно один процессор, хотя он может быть частью нескольких плат. Процессорная часть содержит весь общий для конкретного процесса код.

Часть, относящаяся к платам, хранит конкретную конфигурацию процессора, который они содержат. Эта конфигурация состоит в основном из конфигурации периферийных устройств и схемы расположения контактов, конфигурации внутриплатных устройств, а также конфигурации системных часов. Помимо исходных и заголовочных файлов, требующихся для каждой платы, в ее каталоге могут дополнительно содержаться некоторые файлы сценариев и конфигурации, требующиеся для сопряжения с ней.

В каталоге драйверов конкретного периферийного оборудования предоставляется API для программного обеспечения драйверов логических устройств. Этот каталог сконфигурирован для конкретных периферийных устройств исходной системы. Главная цель разделения драйверов на каталоги `drivers` и `drivers/peripherals` состоит в том, чтобы дать возможность писать переносимый код для доступа к оборудованию. В этом состоит одна из главных особенностей операционной системы RIOT. Каталог `drivers` содержит код драйверов конкретного оборудования (например, датчиков или радиоаппаратуры). Каталог `drivers/peripherals` содержит заголовки и некоторый общий код аппаратной абстракции в RIOT, что обеспечивает единообразный API для абстракции таких интерфейсов ввода-вывода микроконтроллеров, как, например, UART, I<sup>2</sup>C и SPI. Замысел состоит в том, чтобы писать драйверы (или приложения) один раз для API, предоставляемого в каталоге `drivers/peripherals`, а затем выполнять их без изменения на всех микроконтроллерах, предоставляющих реализацию требующегося интерфейса.

## 16.5. Ключевые термины и контрольные вопросы

### Ключевые термины

Аудитор облака	Модель обслуживания	Приемопередатчик
Блочное хранилище данных	Непосредственно подключаемое хранилище (DAS)	Привод
Гибридное облако	Облако	Приемопередатчик
Датчик	Облачные вычисления	Провайдер услуг облачных вычислений (CSP)
Закрытое облако	Облачный брокер	Программное обеспечение как услуга (SaaS)
Интернет вещей (IoT)	Оператор облака	Радиочастотная идентификация (RFID)
Инфраструктура как услуга (IaaS)	Открытое облако	Сетевое хранилище данных (SAN)
Коллективное облако	Платформа как услуга (PaaS)	Устройство с ограниченными ресурсами
Магистральная сеть	Подключаемое к сети хранилище (NAS)	Файловое хранилище
Микроконтроллер	Потребитель услуг облачных вычислений (CSC)	Хранилище объектов

### Контрольные вопросы

- 16.1. Дайте определение облачным вычислениям.
- 16.2. Перечислите и кратко опишите три модели обслуживания в облаке.
- 16.3. Что собой представляет эталонная архитектура облачных вычислений?
- 16.4. Перечислите и кратко опишите основные компоненты облачной операционной системы.
- 16.5. В чем состоит взаимосвязь облачной операционной системы с моделью IaaS?
- 16.6. Что такое OpenStack?
- 16.7. Дайте определение Интернета вещей.
- 16.8. Перечислите и кратко опишите основные компоненты устройства, функционирующего через IoT.
- 16.9. Каким требованиям должна удовлетворять операционная система для IoT?
- 16.10. Что такое RIOT?



# ГЛАВА 17

## СЕТЕВЫЕ ПРОТОКОЛЫ

В ЭТОЙ ГЛАВЕ...

### 17.1. Потребность в архитектуре протоколов

### 17.2. Архитектура протоколов TCP/IP

Уровни протоколов TCP/IP

Протоколы TCP и UDP

Протоколы IP и IPv6

Принцип действия протоколов TCP/IP

Приложения протокола TCP/IP

### 17.3. Сокеты

Сокет

Вызовы интерфейса Socket

Настройка сокета

Соединение сокетов

Обмен данными между сокетами

### 17.4. Организация сетей в Linux

Передача данных

Прием данных

### 17.5. Резюме

### 17.6. Ключевые термины, контрольные вопросы и задачи

Ключевые термины

Контрольные вопросы

Задачи

### Приложение 17.А. Простой протокол передачи файлов

Введение в протокол TFTP

Пакеты TFTP

Краткий обзор передачи данных

Ошибки и задержки

Синтаксис, семантика и синхронизация

## УЧЕБНЫЕ ЦЕЛИ

- Пояснить побудительные причины для организации функций передачи данных в многоуровневую архитектуру протоколов.
- Описать архитектуру протоколов TCP/IP.
- Объяснить назначение сокетов и порядок их применения.
- Описать доступные в Linux средства для организации сетей.
- Объяснить принцип действия протокола TFTP.

По мере того как все более и более доступными становились недорогие, хотя и довольно мощные, персональные компьютеры и серверы, стала постепенно проявляться тенденция к распределенной обработке данных (*distributed data processing — DDP*), при которой процессоры, данные и другие компоненты системы обработки данных могут быть рассредоточены в пределах организации. Система DDP подразумевает разделение вычислительной функции, а возможно, и распределенную организацию баз данных, управления устройствами и их взаимодействием (через сеть).

Во многих организациях делается сильный акцент на персональные компьютеры, связанные с серверами. Персональные компьютеры служат для выполнения самых разных пользовательских приложений, в том числе редакторов текста, электронных таблиц и средств графического представления информации. А на серверах размещается корпоративная база данных вместе с логически развитой системой управления ею, а также программное обеспечение информационных систем. Поэтому между самими персональными компьютерами, с одной стороны, и каждым персональным компьютером и сервером, с другой стороны, должны быть установлены определенные связи. С этой целью обычно применяются самые разные подходы: от наделения персонального компьютера функциями простого терминала до реализации высокой степени интеграции приложений на персональных компьютерах с базой данных на сервере.

Такие тенденции к применению нашли поддержку в ходе эволюции распределенных возможностей операционных систем и соответствующих утилит. Весь спектр исследованных распределенных возможностей включает следующие.

- **Архитектура коммуникаций.** Это программное обеспечение, поддерживающее группу подключенных к сети компьютеров. Оно поддерживает такие распределенные приложения, как электронная почта, передача файлов и доступ с удаленных терминалов. Тем не менее каждый компьютер сохраняет свою отчетливую индивидуальность для пользователей и приложений, которым требуется связываться с другими компьютерами по прямой ссылке. На каждом компьютере установлена своя отдельная операционная система, и поэтому вполне возможно разнородное сочетание компьютеров и операционных систем, при условии, что на всех машинах поддерживается одна и та же архитектура коммуникаций. Наиболее широко употребляемой архитектурой является набор (или стек) сетевых протоколов TCP/IP, рассматриваемый далее в этой главе.

- **Сетевая операционная система.** Это конфигурация, в которой имеется сеть, состоящая из прикладных машин (как правило, однопользовательских рабочих станций) и одной или более серверных машин. При этом серверные машины предоставляют по всей сети свои услуги и приложения (например, для хранения файлов и управления печатающими устройствами). У каждого компьютера имеется собственная операционная система. Сетевая операционная система служит дополнением к локальной операционной системе, позволяя прикладным машинам взаимодействовать с серверными. Пользователю известно, что в сети имеется много независимых компьютеров, и он должен обращаться к ним явно. Как правило, для таких сетевых приложений применяется распространенная архитектура коммуникаций.
- **Распределенная операционная система.** Это общая для сети компьютеров операционная система. Для своих пользователей она выглядит, как обыкновенная операционная система, но предоставляет им прозрачный доступ к ресурсам целого ряда машин. Распределенная операционная система может полагаться на архитектуру коммуникаций для выполнения основных функций передачи данных, но чаще всего ради большей эффективности урезанный набор таких функций встраивается в саму операционную систему.

Технология архитектуры коммуникаций хорошо развита и поддерживается всеми поставщиками. Сетевые операционные системы — более недавнее явление, хотя они уже существуют в виде ряда коммерческих продуктов. На переднем крае прикладных исследований распределенных систем находится область распределенных операционных систем. И хотя некоторые коммерческие образцы подобных систем уже были представлены, полностью функционирующие распределенные операционные системы все еще находятся на стадии экспериментов.

В этой и последующей главах дается обзор возможностей распределенной обработки. Основное внимание в этой главе уделяется программному обеспечению, положенному в основу сетевых протоколов.

## 17.1. ПОТРЕБНОСТЬ В АРХИТЕКТУРЕ ПРОТОКОЛОВ

Когда компьютеры, терминалы и/или другие устройства обработки данных обмениваются ими, задействованные в этом процессе процедуры могут быть довольно сложными. Рассмотрим в качестве примера передачу файла из одного компьютера на другой. Между двумя компьютерами должен быть установлен канал передачи данных как напрямую, так и через сеть передачи данных, но этого мало. Ниже перечислены типичные задачи, которые при этом выполняются.

1. Исходная система должна активизировать прямой канал передачи данных или же известить сеть передачи данных о подлинности требующейся целевой системы.
2. Исходная система должна убедиться, что целевая система готова к приему данных.
3. Приложение, передающее файл в исходной системе, должно убедиться, что программа управления файлами в целевой системе подготовлена к приему и сохранению файла для данного конкретного пользователя.
4. Если форматы файлов или иные способы представления данных, применяемые в обеих системах, несовместимы, в той или иной системе должна быть выполнена функция преобразования формата.

Обмен информацией между компьютерами для выполнения совместного действия обычно называется *компьютерной связью* (computer communication). А если два или более компьютеров соединяются через сеть передачи данных, то это множество компьютерных станций называется *компьютерной сетью* (computer network). Поскольку между терминалом и компьютером требуется аналогичный уровень взаимосвязи, эти термины нередко употребляются и в тех случаях, когда связывающиеся объекты представляют собой терминалы.

При обсуждении компьютерной связи и компьютерных сетей первостепенное значение приобретают следующие понятия.

- Протоколы
- Архитектура компьютерной связи, или архитектура протокола

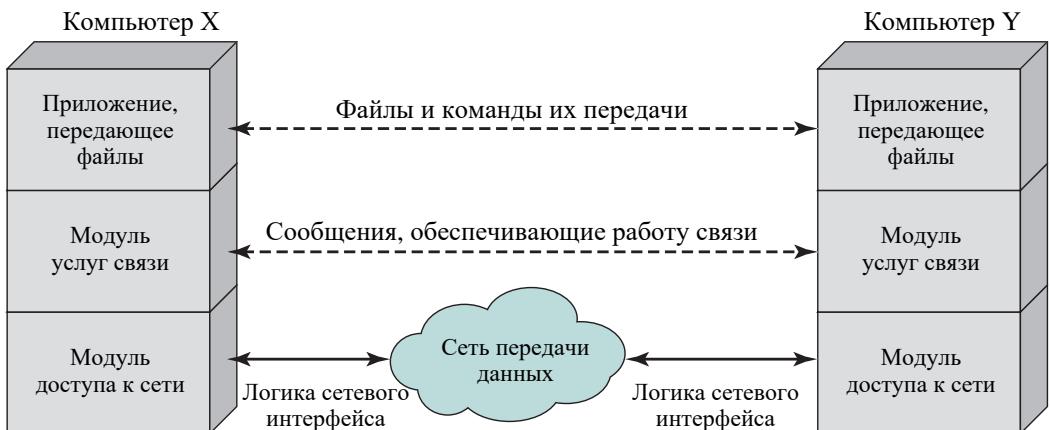
Протокол используется для обмена данными между объектами в разных системах. Термины *объект* и *система* употребляются в самом общем смысле. Примерами объектов служат прикладные программы пользователей, пакеты для передачи данных, системы управления базами данных, средства электронной почты и терминалы. А примерами систем служат компьютеры, терминалы и удаленные датчики. Следует, однако, иметь в виду, что в некоторых случаях объект и система, в которой он находится, совпадают (примером тому служат терминалы). В общем случае объект — это все, что способно передавать или принимать информацию, а система — физически отличный объект, содержащий один или несколько объектов. Для успешного обмена данными между двумя объектами они должны “говорить на одном и том же языке”. Чем именно, как и когда они обмениваются — все это должно отвечать соглашениям, взаимно согласованным объектами, участвующими в таком обмене. Такие условия называются протоколом, который может быть определен как ряд правил, регулирующих обмен данными между двумя объектами. Ниже перечислены основные элементы протокола.

- **Синтаксис.** Включает в себя такие понятия, как формат данных и уровни сигналов.
- **Семантика.** Включает в себя управляющую информацию для координации и обработки данных.
- **Синхронизация.** Включает в себя согласование скорости и установление последовательности.

В приложении к этой главе представлен конкретный пример стандартного протокола TFTP (Trivial File Transfer Protocol — простой протокол передачи файлов).

Теперь, когда вы знакомы с концепцией протокола, можно поговорить об **архитектуре протокола**. Очевидно, что между двумя компьютерными системами должна существовать кооперация высокой степени. Вместо реализации для этой цели логики в виде отдельного модуля, задача разделяется на подзадачи, каждая из которых реализуется отдельно. В качестве примера на рис. 17.1 наглядно показано, каким образом может быть реализовано средство передачи файлов. В данном примере для этой цели применяются три модуля. Задачи, перечисленные выше, в пп. 3 и 4, могут быть выполнены модулем передачи файлов. В обеих взаимодействующих системах имеются два модуля, которые обмениваются файлами и командами. Но вместо того, чтобы требовать от модулей передачи файлов самим заниматься передачей данных и команд, можно поручить каждому из них обращаться к специальному модулю услуг связи. Такой модуль отвечает за на-

дежный обмен командами и данными при передаче файлов между системами. Порядок функционирования модуля услуги связи будет рассмотрен далее, а пока что следует отметить, что данный модуль выполняет задачу, перечисленную выше, в п. 2. И наконец, сам характер обмена данными между двумя модулями услуг связи не зависит от особенностей сети, через которую они соединяются. Следовательно, вместо того чтобы встраивать функции сетевого интерфейса в модуль услуг связи, стоит иметь в своем распоряжении третий модуль для доступа к сети, выполняющий задачу, указанную выше, в п. 1, путем взаимодействия непосредственно с сетью.



**Рис. 17.1.** Упрощенная архитектура для передачи файлов

Итак, модуль передачи файлов содержит всю логику, специфичную для приложения передачи файлов, в том числе пароли, файловые команды и файловые записи. Передаваться такие файлы и команды должны надежно. Однако в плане надежности одни и те же требования предъявляются к самым разным приложениям (например, к приложениям электронной почты или передачи документов). Таким образом, эти требования удовлетворяются в отдельном модуле услуг связи, который может использоваться самыми разными приложениями. Такой модуль обязан убедиться, что обе компьютерные системы активны и готовы к передаче данных, а также отслеживать данные, которыми они обмениваются, чтобы гарантировать их доставку. Однако эти задачи не зависят от типа применяемой сети, так что логика работы с сетью выносится в отдельный модуль доступа к сети. Если изменится используемая сеть, это скажется только на модуле доступа к сети.

Таким образом, вместо одного модуля для передачи данных требуется структурированный набор модулей, реализующих функцию передачи данных. Такая структура называется архитектурой протокола. В данный момент может оказаться весьма полезной следующая аналогия. Допустим, что ответственному служащему в учреждении X требуется отправить документ ответственному служащему в учреждении Y. С этой целью ответственный служащий в учреждении X подготавливает документ, а возможно, и прикрепляет к нему пояснительную записку. Это соответствует действиям приложения для передачи файлов, приведенного на рис. 17.1. Затем ответственный служащий в учреждении X передает документ секретарю или помощнику по административным вопросам в данном учреждении. Этот помощник вкладывает документ в конверт и пишет на нем

адрес учреждения Y и обратный адрес учреждения X, а возможно, еще и помечает его грифом “совершенно секретно, перед прочтением сжечь”. Таким действиям помощника по административным вопросам соответствуют функции модуля услуг связи, показанного на рис. 17.1. Далее помощник по административным вопросам передает пакет с документом в отдел доставки. Служащий в отделе доставки выбирает способ отправки пакета: почтой, через специальную службу срочной доставки или фельдъегерскую курьерскую службу. Отдел доставки добавляет к пакету соответствующие почтовые или сопроводительные документы и отправляет его. Таким действиям отдела доставки соответствуют функции модуля доступа к сети, приведенного на рис. 17.1. Когда пакет прибывает в учреждение Y, происходит ряд аналогичных действий. В частности, отдел доставки в учреждении Y получает пакет и передает его соответствующему помощнику по административным вопросам или секретарю, исходя из указанного на пакете имени. Помощник по административным вопросам открывает пакет и передает вложенный в него документ тому ответственному служащему, которому он адресован.

## 17.2. АРХИТЕКТУРА ПРОТОКОЛОВ TCP/IP

Архитектура протоколов TCP/IP является результатом исследований и разработки протоколов, проведенных в экспериментальной сети с коммутацией пакетов ARPANET, учрежденной Управлением перспективного планирования оборонных научно-исследовательских работ в США (Defense Advanced Research Projects Agency — DARPA), и обычно именуется набором (или стеком) протоколов TCP/IP. Этот набор состоит из большой совокупности протоколов, выпущенных в виде стандартов Комитетом по надзору за сетью Интернет (Internet Activities Board — IAB).

### УРОВНИ ПРОТОКОЛОВ TCP/IP

В общих чертах компьютерную связь можно описать как включающую в себя следующие три составляющие: приложения, компьютеры и сети. К примерам приложений относятся передача файлов и электронная почта. Особый интерес представляют распределенные приложения, работа которых включает обмен данными между двумя компьютерными системами. Эти и другие приложения действуют на тех компьютерах, на которых нередко поддерживается одновременное выполнение нескольких приложений. Компьютеры подключаются к сетям, а данные, которыми они обмениваются, передаются по сети от одного компьютера к другому. Таким образом, передача данных от одного приложения к другому требует сначала перенести данные на тот компьютер, на котором находится целевое приложение, а затем доставить данные этому приложению.

Официальной модели для протоколов TCP/IP не существует. Однако, исходя из разработанных стандартов сетевых протоколов, задачу передачи данных по протоколам TCP/IP можно разделить на пять относительно независимых уровней, перечисленных ниже по порядку от нижнего к верхнему.

- Физический уровень
- Уровень доступа к сети
- Межсетевой уровень (уровень Интернета)
- Транспортный или межузловый уровень
- Прикладной уровень

**Физический уровень** охватывает физический интерфейс между устройством передачи данных (например, рабочей станцией или персональным компьютером) и передающей средой или сетью. На этом уровне задаются характеристики передающей среды, основные свойства сигналов, скорость передачи и все, что с этим связано.

На уровне **доступа к сети** осуществляется обмен данными между конечной системой (сервером, рабочей станцией и т.д.) и той сетью, к которой она подключена. Передающий компьютер должен снабдить сеть адресом принимающего компьютера, чтобы она могла направить данные к соответствующему месту назначения. Кроме того, передающий компьютер может вызвать определенные службы, которые могут предоставляться сетью. Конкретное программное обеспечение, используемое на данном уровне, зависит от типа сети. Для коммутации каналов и коммутации пакетов (например, ретрансляции фреймов), локальных сетей (например, Ethernet) и прочего были разработаны разные стандарты. Таким образом, функции, связанные с доступом к сети, имеет смысл вынести на отдельный уровень. Благодаря этому остальное программное обеспечение, находящееся выше, над уровнем доступа к сети, избавляется от необходимости учитывать особенности конкретной сети. В итоге программное обеспечение, находящееся на том же самом верхнем уровне, должно функционировать надлежащим образом независимо от конкретной сети, к которой подключен компьютер.

За доступ и маршрутизацию данных через одну и ту же сеть, к которой подключены две конечные системы, отвечает уровень доступа к сети. В тех же случаях, когда два устройства подключены к разным сетям, требуются особые процедуры, позволяющие переносить данные через многие соединенные между собой сети. И в этом состоит функция межсетевого уровня. На этом уровне применяется **межсетевой протокол IP** (Internet Protocol), обеспечивающий возможность маршрутизации данных через несколько сетей. Этот протокол реализован не только в конечных системах, но и в маршрутизаторах. **Маршрутизатор (router)** — это процессор, соединяющий две сети, а его основная функция состоит в том, чтобы пересыпать данные из одной сети в другую по маршруту, проложенному от исходной к целевой конечной системе.

Независимо от характера приложений, обменивающихся данными, к ним обычно предъявляется требование осуществлять такой обмен надежно. Это означает, что было бы желательно гарантировать поступление всех данных в целевое приложение, причем в том же самом порядке, в каком они были отправлены. Как будет показано далее, механизмы, обеспечивающие надежность обмена данными, по существу, не зависят от общего для всех приложений уровня, иначе называемого межузловым или **транспортным уровнем**. Для предоставления таких функциональных возможностей чаще всего служит протокол TCP (Transmission Control Protocol — протокол управления передачей).

И наконец, на **прикладном уровне** содержится логика, требуемая для поддержки различных приложений пользователей. Для каждого типа приложений (например, для передачи данных) требуется отдельный модуль, свойственный именно этому типу приложений.

## Протоколы TCP и UDP

Для большинства приложений, выполняющихся как часть архитектуры протоколов TCP/IP, на транспортном уровне предоставляется протокол TCP. Этот протокол обеспечивает надежное соединение для обмена данными между приложениями. Такое соединение попросту является временной логической связью между объектами в разных

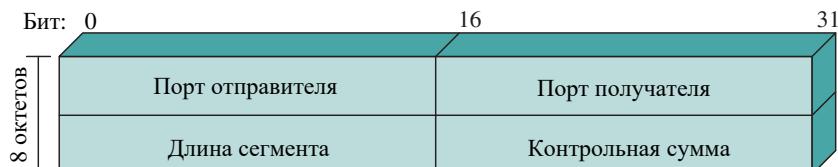
системах. В течение этого соединения каждый объект отслеживает сегменты данных, поступающие из другого объекта и отправляемые ему, чтобы регулировать поток сегментов данных и восстанавливать потерянные или поврежденные их сегменты.

На рис. 17.2, а схематически показан формат заголовка для протокола TCP, состоящего как минимум из 20 октетов, или 160 битов. Так, поля портов отправителя и получателя идентифицируют приложения в исходной и целевой системах, пользующиеся данным соединением, а в полях порядкового номера, номера подтверждения и окна предоставлена информация для управления потоком и ошибками. В поле контрольной суммы хранится 16-битовое контрольное значение, вычисляемое на основании содержимого сегмента данных и используемое для обнаружения ошибок в TCP-сегменте.

Помимо протокола TCP, на транспортном уровне имеется еще один протокол — UDP (User Datagram Protocol — протокол пользовательских дейтаграмм), широко употребляемый как часть набора протоколов TCP/IP. Протокол UDP не гарантирует доставку, сохранение последовательности данных или защиту от дублирования. Этот протокол дает одному процессу возможность отсылать сообщения другим процессам, применяя минимальный протокольный механизм. Протокол UDP применяется в некоторых транзакционных приложениях; одним из примеров тому служит стандартный протокол SNMP (Simple Network Management Protocol — простой протокол сетевого управления), предназначенный для управления сетями TCP/IP. А поскольку протокол UDP не требует устанавливать соединение, он мало на что способен. По существу, он добавляет к протоколу IP возможность указания портов. Лучше всего это проявляется в ходе анализа заголовка пакета UDP, показанного на рис. 17.2, б.



а) Заголовок пакета TCP



б) Заголовок пакета UDP

**Рис. 17.2.** Заголовки протоколов TCP и UDP

## Протоколы IP и IPv6

На протяжении десятилетий краеугольным камнем архитектуры протоколов TCP/IP был протокол IP. На рис. 17.3, *a* наглядно показан формат заголовка IP с минимальной длиной 20 октетов или 160 битов. Этот заголовок вместе с сегментом данных из транспортного уровня образует блок IP-уровня, называемый IP-дейтаграммой или IP-пакетом. В этот заголовок входят 32-битовые адреса отправителя и получателя. Поле контрольной суммы заголовка служит для выявления ошибок в заголовке во избежание ошибки при доставке пакетов. В поле протокола указывается, применяется ли протокол IP в TCP, UDP или каком-нибудь другом протоколе верхнего уровня. Поля идентификации, флагов и смещения фрагмента служат для обозначения процесса сборки при фрагментации, в ходе которой одна IP-дейтаграмма при передаче разделяется на несколько более мелких IP-дейтаграмм, а затем снова собирается воедино в месте получения.

В 1995 году группа IETF (Internet Engineering Task Force — группа инженерной поддержки Internet), разрабатывающая стандарты протоколов Интернета, выпустила спецификацию протокола IP следующего поколения, позже известного как IPng. В 1996 году эта спецификация превратилась в стандарт под названием “IPv6”, обеспечивший целый ряд функциональных усовершенствований существующего протокола IP с целью приспособить его к повышенным скоростям передачи данных в современных сетях, а также сочетать разные потоки данных, включая графические и видеоданные, которые сегодня становятся преобладающими. Однако побудительной причиной для разработки нового протокола стала потребность в дополнительных адресах. Так, в протоколе IP применяется 32-разрядный адрес для обозначения отправителя и получателя. Вследствие бурного развития Интернета и частных сетей, подключаемых к Интернету, такой длины адреса оказалось недостаточно, чтобы удовлетворить потребности в адресах всех имеющихся систем. Как показано на рис. 17.3, *b*, в заголовке протокола IPv6 имеются поля 128-битовых адресов.

В конечном счете ожидается, что все разновидности установки протоколов TCP/IP перейдут с текущего протокола IP на более перспективный протокол IPv6, хотя этот процесс может продолжаться многие годы, если не десятки лет.

## Принцип действия протоколов TCP/IP

На рис. 17.4 показано, каким образом протоколы конфигурируются для передачи данных. Так, для подключения компьютера к сети применяется некоторый протокол доступа к сети (например, протокол, реализующий логику Ethernet). Этот протокол позволяет одному узлу посылать данные через сеть другому узлу или же маршрутизатору, если целевой узел находится в другой сети. Протокол IP реализуется во всех конечных системах и маршрутизаторах. Он действует как ретранслятор, перенося блок данных из одного узла через один или несколько маршрутизаторов другому узлу. Протокол TCP реализуется только в конечных системах, чтобы отслеживать передаваемые блоки данных и тем самым гарантировать, что все они будут надежно доставлены соответствующему приложению.



а) Заголовок пакета IPv4



б) Заголовок пакета IPv6

IHL — длина заголовка (Internet header length)

DS — поле дифференцированных услуг  
(Differentiated services)

ECN — Поле явного уведомления о перегрузке  
(Explicit congestion notification)

*Примечание.* 8-битовые поля DS/ECN первоначально назывались Type of Service (Тип услуги) в заголовке IPv4 и Traffic Class (Класс трафика) — в заголовке IPv6

Рис. 17.3. Заголовки разновидностей протоколов IP

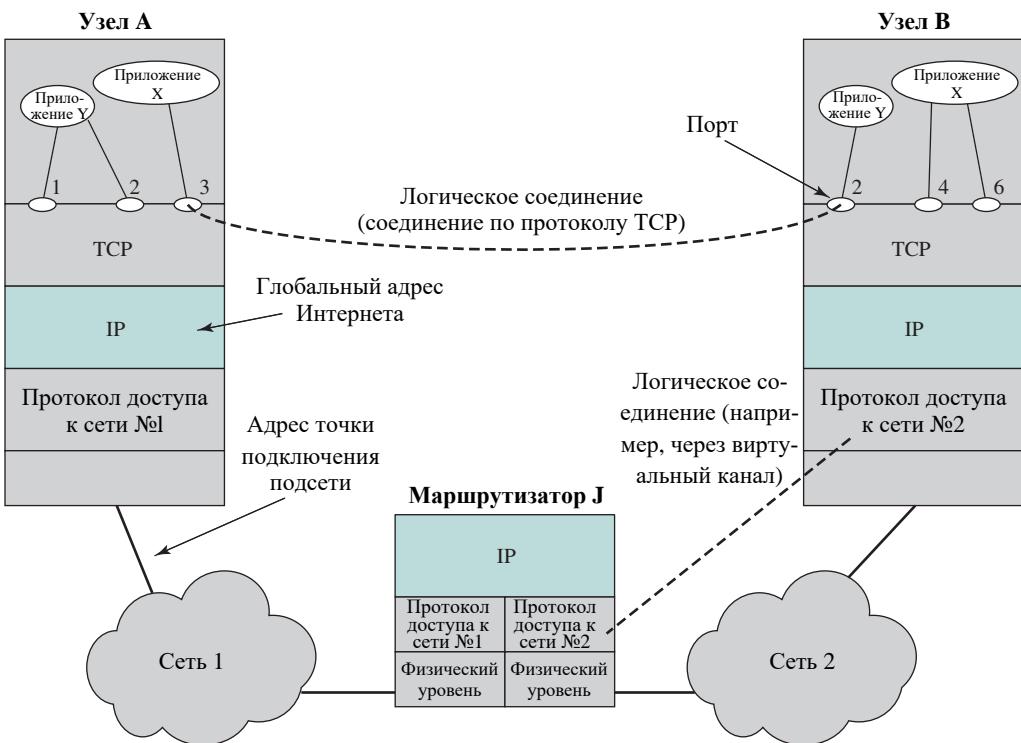


Рис. 17.4. Принципы действия протоколов TCP/IP

Для успешного обмена данными у каждого объекта в общей системе должен быть уникальный адрес. На самом деле для этой цели требуются два уровня адресации. У каждого узла в сети должен быть уникальный глобальный адрес Интернета, что дает возможность доставлять данные в нужный узел. Этот адрес используется в протоколе IP для маршрутизации и доставки данных. У каждого приложения в узле должен быть адрес, уникальный в рамках узла, что дает протоколу соединения узлов (TCP) возможность доставлять данные нужному процессу. В последнем случае адреса называются **портами**.

Проанализируем в качестве примера простую операцию. Допустим, что процессу, связанному с портом 3 в узле А, требуется отправить сообщение другому процессу, связанному с портом 2 в узле В. Процесс в узле А передает сообщение на нижний уровень протоколу TCP с инструкциями по его отправке узлу В, порт 2. Протокол TCP передает сообщение на нижний уровень протоколу IP вместе с инструкциями по его отправке узлу В. Следует, однако, иметь в виду, что протоколу IP совсем не обязательно сообщать целевой порт. Достаточно лишь указать, что данные предназначены для узла В. Далее протокол IP передает сообщение на нижний уровень доступа к сети (например, протоколу логики Ethernet) вместе с инструкциями по его отправке маршрутизатору J, находящемуся на первом транзитном участке по пути к узлу В.

Чтобы управлять данной операцией, необходимо передать управляющую информацию, а также пользовательские данные, как показано на рис. 17.5.

Допустим, что отправляющий процесс формирует блок данных и передает его протоколу TCP. А протокол TCP может разделить этот блок на более мелкие части, чтобы упростить его обработку. К каждой из этих частей протокол TCP присоединяет управляющую информацию, известную как заголовок TCP (см. рис. 17.2, а), формируя **TCP-сегмент**. Эта управляющая информация предназначена для использования объектом протокола TCP того же ранга в узле В.



**Рис. 17.5.** Протокольные блоки данных (PDU) в архитектуре TCP/IP

Далее протокол TCP передает каждый сегмент протоколу IP вместе с инструкциями по его передаче узлу В. Такие сегменты должны передаваться через одну или несколько сетей и пересыпаться через один или несколько промежуточных маршрутизаторов. Для этой операции будет использоваться управляющая информация. Протокол IP присоединяет заголовок с управляющей информацией (см. рис. 17.3) к каждому сегменту, формируя **IP-дейтаграммы**. Примером элемента, хранящегося в заголовке IP, служит адрес узла получателя (в данном примере — узла В).

Наконец, каждая IP-дейтаграмма передается на уровень доступа к сети для передачи через первую сеть на пути к получателю. На уровне доступа к сети к ней присоединяется его собственный заголовок, и в итоге создается пакет, или фрейм, который передается по сети маршрутизатору J. Заголовок пакета содержит информацию, требующуюся для передачи данных по сети. Ниже приведены примеры элементов, которые могут входить в этот заголовок.

- **Сетевой адрес получателя.** В сети должно быть известно, какому именно подключенному к ней устройству доставляется пакет (в данном случае это маршрутизатор J).
- **Запросы сетевых средств.** Протокол доступа к сети может запросить использование определенных сетевых возможностей (например, приоритетов).

В маршрутизаторе J заголовок пакета обрезается и анализируется IP-заголовок. На основании сведений об адресе получателя, находящемся в IP-заголовке, IP-модуль маршрутизатора направляет дейтаграмму через вторую сеть узлу В.

Как только данные будут приняты в узле В, произойдет обратный процесс. На каждом уровне удаляется соответствующий заголовок, а оставшаяся часть пакета передается на следующий по очереди верхний уровень до тех пор, пока исходные пользовательские данные не будут доставлены целевому процессу.

## Приложения протокола TCP/IP

Был стандартизирован целый ряд приложений, работающих поверх протокола TCP. Здесь упоминаются три наиболее распространенные из них.

**Протокол простого обмена электронной почтой** (Simple Mail Transfer Protocol — SMTP) предоставляет основные возможности электронной почты, а также механизм для обмена сообщениями между сетевыми узлами. К средствам протокола SMTP относятся списки рассылки, уведомления о доставке почты и ее пересылка. В протоколе SMTP не определяется, каким образом создаются сообщения, поэтому для их составления требуется некоторое средство редактирования или создания электронной почты. Как только сообщение будет создано, оно принимается SMTP и отправляется с использованием протокола TCP модулю SMTP в другом сетевом узле. А в целевом модуле SMTP будет использован локальный пакет электронной почты для сохранения входящего сообщения в почтовом ящике пользователя.

**Протокол передачи файлов** (File Transfer Protocol — FTP) служит для отправки файлов из одной системы в другую по команде пользователя. Этот протокол в состоянии работать как с текстовыми, так и с бинарными файлами, и предоставляет средства управления пользовательским доступом. Когда пользователь приступает к передаче файла, протокол FTP устанавливает TCP-соединение с целевой системой для обмена управляющими сообщениями. Это соединение позволяет пользователю передать свой идентификатор и пароль, а также указать файл и действия, которые требуется над ним выполнить. Как только передача файла одобрена, устанавливается второе TCP-соединение по протоколу TCP, но без дополнительных издержек на заголовки и управляющую информацию прикладного уровня. По завершении передачи для сигнализации об ее окончании и приема команд на передачу других файлов используется управляющее соединение.

**Протокол безопасной оболочки** (Secure Shell — SSH) обеспечивает безопасный вход в удаленную систему, что дает пользователю возможность войти с терминала или персонального компьютера в систему на удаленном компьютере и работать на нем непосредственно как на собственном компьютере. Кроме того, в протоколе SSH поддерживается обмен файлами между локальным узлом и удаленным сервером. Протокол SSH дает пользователю и удаленному серверу возможность аутентифицировать друг друга. Он также шифрует весь сетевой трафик в обоих направлениях. Сетевой трафик по протоколу SSH передается через TCP-соединение.

## 17.3. Сокеты<sup>1</sup>

Концепция сокетов и принципы их программирования были разработаны в 1980-е годы в среде UNIX и в конечном счете оформились в интерфейс сокетов Беркли (Berkeley Sockets Interface). По существу, сокет обеспечивает связь между клиентским и серверным процессами, которая может быть как ориентированной на установление соединения, так и без его установления. Сокет можно рассматривать как конечную точку в передаче данных. В клиентском сокете на одном компьютере используется адрес для вызова серверного сокета на другом компьютере. Оба компьютера могут обмениваться данными, как только будут задействованы соответствующие сокеты.

<sup>1</sup> В этом разделе дается общий обзор сокетов, а более подробное их изложение приведено в приложении М. “Введение в программирование сокетов”.

Как правило, на компьютерах с серверными сокетами поддерживается открытый TCP- или UDP-порт, готовый к приему незапланированных входящих вызовов. Клиент обычно указывает сокет требующегося сервера, обнаруживая его адрес в базе данных DNS (Domain Name System — система доменных имен). Как только соединение установлено, сервер переключает диалоговый режим на другой номер порта, чтобы освободить основной порт для приема других входящих вызовов.

Интернет-приложения наподобие TELNET или rlogin (удаленный вход в систему) используют сокеты, хотя подробности их применения скрыты от пользователя. Тем не менее сокеты могут быть созданы в прикладной программе, написанной на таком языке, как C или Java, что дает программисту возможность без труда поддерживать сетевые функции и приложения. Механизм программирования сокетов включает в себя достаточно семантики, чтобы разрешить обмен данными между несвязанными вместе процессами на разных узлах.

Интерфейс сокетов Беркли фактически является стандартным **интерфейсом прикладного программирования** (API) для разработки сетевых приложений, охватывающих обширный ряд операционных систем. Так, на основе спецификации Беркли построен прикладной интерфейс Windows Sockets (WinSock). API сокетов обеспечивает универсальный доступ к услугам обмена данными между процессами. Таким образом, возможности сокетов идеально подходят учащимся для изучения принципов действия сетевых протоколов и распределенных приложений в ходе практической разработки программ.

## Сокет

Напомним, что в состав TCP- и UDP-заголовков входят поля портов отправителя и получателя (см. рис. 17.2). Номера этих портов обозначают соответствующих пользователей (приложения) в двух TCP-объектах. Кроме того, в состав каждого заголовка IPv4 и IPv6 входят поля адресов отправителя и получателя (см. рис. 17.3). Эти **IP-адреса** идентифицируют узлы, а сочетание номера порта и IP-адреса образует **сокет**, уникальный для всего Интернета. Таким образом, на рис. 17.4 сочетание IP-адреса узла B и номера порта приложения X уникальным образом определяет местоположение сокета приложения X в узле B. Как показано на упомянутом рисунке, у приложения может быть несколько адресов сокетов — по одному на каждый порт в приложении.

Сокет служит для определения API, предназначенного для написания программ, в которых применяется протокол TCP или UDP. При применении API на практике сокет идентифицируется тремя составляющими: протоколом, локальным адресом и локальным процессом, где локальный адрес — это IP-адрес, а локальный процесс — номер порта. Поскольку номера портов в системе уникальны, номер порта подразумевает протокол (TCP или UDP). Но ради ясности и простоты реализации определение сокетов включает протокол, а также IP-адрес и номер порта.

В соответствии с двумя упомянутыми выше протоколами в Socket API распознаются два типа сокетов: потоковые и дейтаграммы. В **потоковых сокетах** применяется протокол TCP, обеспечивающий ориентированную на установление соединения надежную передачу данных. Таким образом, все блоки данных, которыми сетевые узлы обмениваются через пару потоковых сокетов, гарантированно доставляются и поступают по месту назначения в том порядке, в каком они были отправлены. А в **сокетах дейтаграмм** приме-

няется протокол UDP, не обладающий свойствами протокола TCP, ориентированными на установление соединения. Поэтому в сокетах дейтаграмм не гарантируется ни доставка блоков данных, ни сохранение порядка их следования. В Berkeley Sockets API поддерживается также третий тип сокетов, называемых “необработанными” (raw). Такие сокеты разрешают непосредственный доступ к протоколам нижнего уровня (таким, как протокол IP).

## Вызовы интерфейса Socket

В этом разделе кратко описываются ключевые системные вызовы.

### Настройка сокета

Первый шаг в применении прикладного интерфейса Sockets API состоит в создании нового сокета с помощью функции `socket()`. Эта функция принимает три параметра, и первый из них — семейство протоколов — для TCP/IP всегда равен `PF_INET`. Второй параметр обозначает *тип* сокета: потоковый или дейтаграмм, а третий параметр — сетевой *протокол*: TCP или UDP. Причина указания *типа* и *протокола* сокета — позволить в будущих реализациях включить в них протоколы транспортного уровня. Таким образом, может существовать несколько дейтаграммных, или несколько ориентированных на установление соединения транспортных протоколов. Функция `socket()` возвращает целочисленное значение, идентифицирующее созданный сокет аналогично дескриптору файла в UNIX. Точная структура данных сокета зависит от конкретной реализации и включает в себя номер порта и IP-адрес отправителя, номер порта и IP-адрес получателя, если соединение установлено или ожидается, а также различные режимы и параметры, связанные с соединением.

После создания сокет должен иметь адрес для прослушивания. Функция `bind()` привязывает сокет к адресу. Ниже приведена структура адреса сокета.

```
struct sockaddr_in {
    short int sin_family;           // Семейство адресов (TCP/IP)
    unsigned short int sin_port;    // Номер порта
    struct in_addr sin_addr;        // Адрес в Интернете
    unsigned char sin_zero[8];      // Такой же размер, как и
                                    // у структуры sockaddr
};
```

### Соединение сокетов

Как только потоковый сокет создан, должно быть установлено соединение с удаленным сокетом. Одна сторона такого соединения функционирует как клиентская, запрашивая соединение с другой стороной, которая действует как серверная.

Установление соединения на серверной стороне должно происходить в две стадии. Сначала серверное приложение вызывает функцию `listen()`, указывающую на готовность данного сокета принимать запросы на входящее соединение. В качестве параметра указывается максимально допустимое количество входящих соединений в очереди. Запрос на каждое входящее соединение содержится в очереди до тех пор, пока на серверной стороне не будет вызвана соответствующая функция `accept()`. Вызов `accept()` удаляет из очереди один запрос. Если очередь пуста, функция `accept()` блокирует

процесс до тех пор, пока не поступит запрос на соединение. Если имеется вызов, ожидающий обработки, функция `accept()` возвращает новый файловый дескриптор соединения. При этом создается новый сокет с IP-адресом и номером порта удаленной стороны соединения, IP-адресом данной системы и новым номером порта. Новый номер порта присваивается новому сокету для того, чтобы локальное приложение могло продолжить прием других запросов. В итоге приложение может оперировать несколькими активными соединениями в любой момент времени, причем для каждого из них выделяется другой номер локального порта. Этот новый номер порта возвращается через TCP-соединение запрашивающей системе.

Клиентское приложение вызывает функцию `connect()`, указывая как локальный сокет, так и адрес удаленного сокета. Если попытка установить соединение завершится неудачно, функция `connect()` возвратит значение `-1`. Если же попытка завершится удачно, функция `connect()` возвратит нулевое значение и заполнит параметр дескриптора файла, включив в него IP-адрес и номер порта локального и удаленного сокета. Напомним, что номер удаленного порта может отличаться от номера, указанного в параметре функции, поскольку номер порта в удаленном узле изменяется.

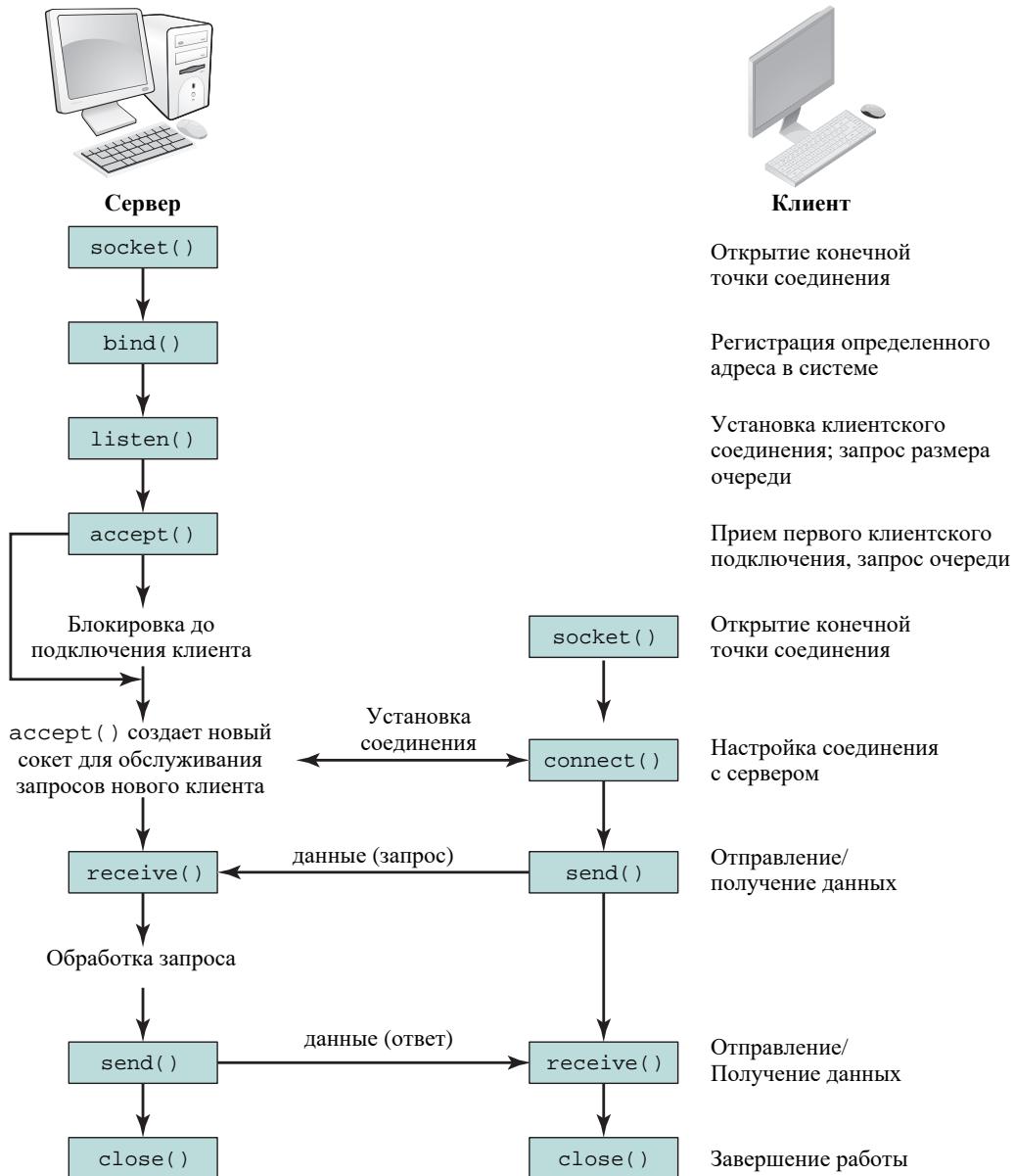
Как только соединение будет установлено, с помощью функции `getpeername()` можно выяснить, кто именно находится на другом конце соединения с потоковым сокетом. Эта функция возвращает значение в параметре `sockfd`.

### Обмен данными между сокетами

Для потоковой коммуникации (*stream communication*) служат функции `send()` и `recv()`, передающие и принимающие данные через соединение, идентифицированное параметром `sockfd`. При вызове функции `send()` параметр `msg` указывает на передаваемый блок данных, параметр `len` обозначает количество передаваемый байтов, а параметр `flags` содержит флаги, которые обычно сброшены в нуль. В результате вызова функции `send()` возвращается количество переданных байтов, которое может быть меньше, чем количество, заданное в параметре `len`. При вызове функции `recv()` параметр `buf` указывает на буфер для хранения входящих данных, причем верхний предел для количества сохраняемых байтов задается параметром `len`.

В любой момент времени любая сторона соединения может разорвать соединение, вызвав функцию `close()`, которая предотвращает дальнейший обмен данными. Вызов функции `shutdown()` позволяет вызывающей стороне прервать передачу данных, их прием или то и другое вместе. Взаимодействие клиентской и серверной сторон для установления, использования и завершения соединения наглядно показано на рис. 17.6.

Для дейтаграммной коммуникации (*datagram communication*) служат функции `sendto()` и `recvfrom()`. При вызове функции `sendto()` указываются все те же параметры, что и при вызове функции `send()`, а также адрес получателя (IP-адрес и номер порта). Аналогично при вызове функции `recvfrom()` указывается параметр адреса, который заполняется при приеме данных.



**Рис. 17.6.** Системные вызовы сокетов для ориентированного на соединения протокола

## 17.4. ОРГАНИЗАЦИЯ СЕТЕЙ В LINUX

В операционной системе Linux поддерживаются самые разные сетевые архитектуры и, в частности, протоколы TCP/IP через Berkeley Sockets API. Общая структура поддержки протоколов TCP/IP в Linux приведена на рис. 17.7. Процессы на уровне пользователя взаимодействуют с сетевыми устройствами посредством системных вызовов Berkeley Sockets API. В свою очередь, модуль Berkeley Sockets API взаимодействует с программным пакетом в ядре Linux, работающим с транспортным уровнем (TCP и UDP) и с протоколом IP. Этот программный пакет обменивается данными с драйвером устройства сетевого адаптера.

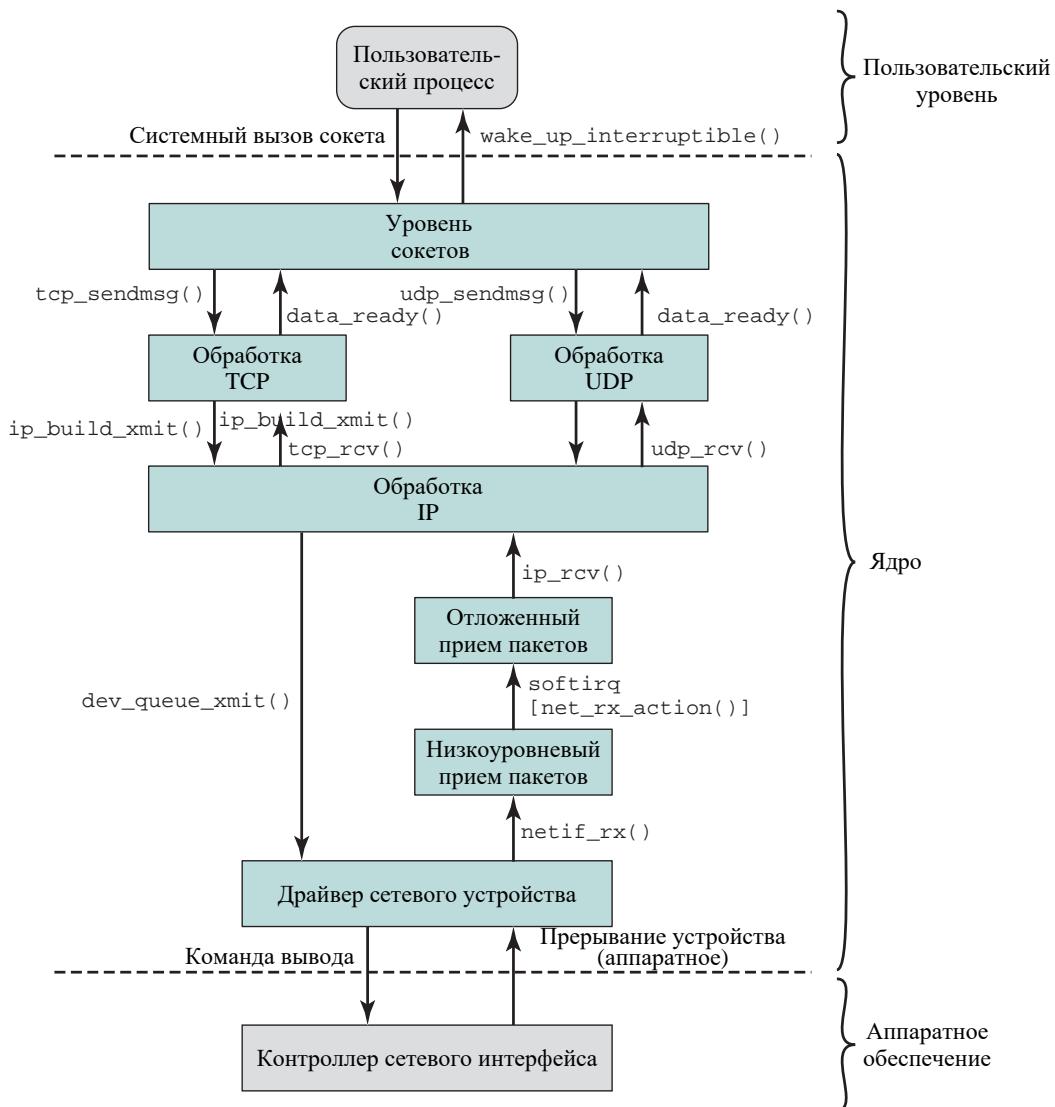


Рис. 17.7. Компоненты ядра Linux для обработки данных по протоколам TCP/IP

Сокеты реализуются в Linux как специальные файлы. Как упоминалось в главе 12, “Управление файлами”, в системах UNIX специальным является такой файл, который не содержит данные, но предоставляет механизм для отображения физических устройств на имена файлов. Для каждого нового сокета в ядре Linux создается новый индексный дескриптор в специальной файловой системе *sockfs*.

Взаимосвязи между различными модулями ядра Linux, задействованными в передаче и приеме блоков данных по протоколу TCP/IP, приведены на рис. 17.7. В остальной части этого раздела рассматриваются средства передачи и приема данных, доступные в Linux.

## Передача данных

Вызовы, описанные в разделе 17.3, применяются пользовательскими процессами для создания новых сокетов, установления соединений с удаленными сокетами, передачи и приема данных. Чтобы отправить данные, пользовательский процесс направляет их на запись в сокет следующим вызовом файловой системы:

```
write(sockfd, mesg, mesglen)
```

Здесь *mesglen* — длина буфера *mesg* в байтах.

Этот вызов приводит к вызову метода *write* файлового объекта, связанного с дескриптором файла *sockfd*, который указывает, настроен ли данный сокет для работы с TCP или UDP. Ядро выделяет соответствующие структуры данных и вызывает функцию уровня сокета *tcp\_sendmsg()* или *udp\_sendmsg()* для передачи данных модулю TCP или UDP соответственно. Модуль транспортного уровня выделяет структуру данных для заголовка TCP или UDP и выполняет функцию *ip\_build\_xmit()* для вызова модуля, обрабатывающего данные на уровне IP. Этот модуль составляет дейтаграмму для передачи и помещает ее в буфер передачи, выделенный для данного сокета. Затем модуль, действующий на уровне IP, выполняет функцию *dev\_queue\_xmit()* для постановки буфера сокета в очередь и последующей передачи данных через драйвер сетевого устройства. И как только драйвер сетевого устройства станет доступным, он передаст буферизованные пакеты.

## Прием данных

Прием данных является непредсказуемым событием, поэтому он подразумевает применение прерываний и отложенного выполнения функций. Когда поступает IP-дейтаграмма, контроллер сетевого интерфейса выдает аппаратное прерывание драйверу соответствующего сетевого устройства. Данное прерывание запускает процедуру обслуживания прерываний, обрабатывающую прерывание как часть модуля драйвера сетевого устройства. Этот драйвер выделяет буфер ядра для хранения входящего блока данных и передает данные из контроллера сетевого устройства в буфер. Затем драйвер выполняет функцию *netif\_rx()*, чтобы вызвать низкоуровневую процедуру приема пакетов. По существу, функция *netif\_rx()* размещает входящий блок данных в очереди, а затем выдает запрос на программное прерывание (*softirq*), чтобы в конечном счете обработать находящиеся в очереди данные. Действие, выполняемое при обработке программного прерывания, реализуется в функции *net\_rx\_action()*.

Как только программное прерывание *softirq* будет поставлено в очередь, обработка пакета приостановится до тех пор, пока ядро не выполнит функцию обработки прерыва-

ния softirq, т.е. не отреагирует на запрос на данное программное прерывание и не выполнит связанную с ним функцию (в данном случае — функцию `net_rx_action()`). В ядре можно выделить три момента, когда проверяется наличие ожидающих обработки программных прерываний softirq: после обработки аппаратного прерывания, при системном вызове из процесса на прикладном уровне и при планировании выполнения нового процесса.

Когда функция `net_rx_action()` выполняется, она извлекает пакет из очереди и передает его обработчику IP-пакетов, вызывая функцию `ip_rcv()`. Обработчик IP-пакетов обрабатывает IP-заголовок, а затем с помощью функции `tcp_rcv()` или `udp_rcv()` вызывает модуль обработки данных транспортного уровня. Этот модуль обрабатывает заголовок транспортного уровня и передает данные пользователю через интерфейс сокетов, вызывая функцию `wake_up_interruptible()`, которая выводит процесс приема данных из состояния ожидания.

## 17.5. Резюме

Функциональные возможности передачи данных, требующиеся для распределенных приложений, довольно сложны. Как правило, эти функциональные возможности реализуются в виде структурированного ряда модулей. Такие модули располагаются по вертикали в многоуровневом порядке, и на каждом уровне предоставляется отдельная порция требующихся функциональных возможностей, а более примитивные функции поручаются следующему по иерархии уровню. Подобная структура называется архитектурой протоколов.

Одна из причин применения такого типа структуры состоит в том, что она упрощает задачи проектирования и реализации. Разделение функций на отдельные модули, которые могут быть спроектированы и реализованы по отдельности, является стандартной практикой. Как только модуль спроектирован и реализован, его можно протестировать, а затем объединить и протестировать все модули вместе. Именно эта причина и вынудила поставщиков вычислительной техники разрабатывать оригинальные многоуровневые архитектуры протоколов. Характерным примером служит сетевая архитектура систем (SNA — Systems Network Architecture) от компании IBM.

Многоуровневую архитектуру можно также использовать для построения стандартизированного набора протоколов передачи данных. И в этом случае остаются преимущества модульного проектирования, а кроме того, многоуровневая архитектура оказывается вполне пригодной для разработки стандартов. Стандарты на протоколы могут разрабатываться одновременно на каждом уровне такой архитектуры. Благодаря этому работа над стандартами становится более выполнимой, а весь процесс их разработки ускоряется. Архитектура протоколов TCP/IP является стандартной и вполне пригодной для данной цели. Эта архитектура состоит из пяти уровней, на каждом из которых предоставляется определенная порция общих функций передачи данных, требующихся распределенным приложениям. Для каждого из этих уровней были разработаны отдельные стандарты. Работа над их разработкой продолжается, особенно на верхнем (прикладном) уровне, где по-прежнему определяются распределенные приложения.

## 17.6. Ключевые термины, контрольные вопросы и задачи

### Ключевые термины

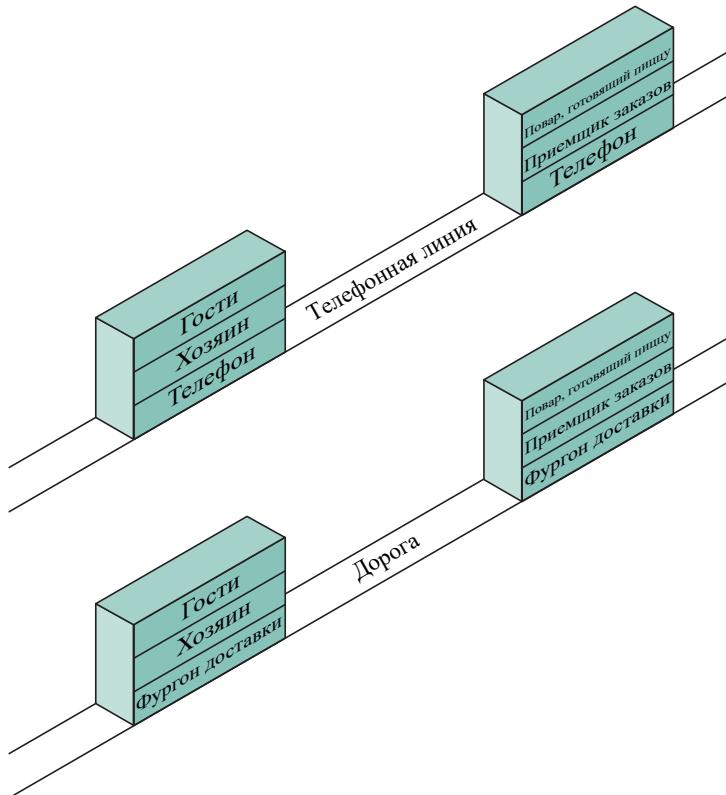
IP-адрес	Потоковая коммуникация	Протокол простого обмена электронной почтой (SMTP)
IP-дейтаграмма	Потоковый сокет	Протокол управления передачей (TCP)
TCP-сегмент	Прикладной уровень	Сетевой протокол
Архитектура протоколов	Протокол IP (Internet Protocol)	Сокет
Дейтаграммная коммуникация	Протокол передачи файлов (FTP)	Транспортный уровень
Дейтаграммный сокет	Протокол пользовательских дейтаграмм (UDP)	Уровень доступа к сети
Маршрутизатор		Физический уровень
Порт		

### Контрольные вопросы

- 17.1. Какова основная функция уровня доступа к сети?
- 17.2. Какие задачи выполняются на транспортном уровне?
- 17.3. Что такое протокол?
- 17.4. Что такое архитектура протоколов?
- 17.5. Что такое TCP/IP?
- 17.6. Каково назначение интерфейса сокетов?

### Задачи

- 17.1. Для решения этой задачи рассмотрим ситуацию, когда требуется заказать пиццу для гостей. Для описания процессов заказа и доставки пиццы можно воспользоваться моделями уровней, приведенными на рис. 17.8. Гость, по существу, заказывает пиццу хозяину, а тот передает этот заказ приемщику заказов в пиццерии, который делает заказ повару. Телефонная система предоставляет физические средства для передачи заказа от хозяина к приемщику заказов. Повар отдает готовую пиццу приемщику заказов с формой заказа, действующей в качестве “заголовка”. Приемщик заказов упаковывает пиццу в коробку с адресом доставки, все заказанные продукты погружаются в фургон для доставки, а дорога открывает физический путь для доставки заказов.



**Рис. 17.8.** Архитектура для задачи 17.1

- Премьер-министры Франции и Китая должны договориться по телефону, но ни один из них не знает языка другого. Более того, ни у одного из них нет рядом переводчика на язык другого. Но в то же время у них имеются переводчики на английский язык. Нарисуйте блок-схему, наглядно показывающую данную ситуацию аналогично той, которая приведена на рис. 17.8, и опишите взаимодействие на каждом уровне.
- Допустим, что переводчик премьер-министра Китая может переводить только на японский язык, у премьер-министра Франции имеется переводчик на немецкий язык, а в Германии есть переводчик с немецкого на японский язык. Нарисуйте новую блок-схему, отражающую такое положение вещей, и опишите гипотетический телефонный разговор.

- Перечислите главные преимущества многоуровневого подхода к организации сетевых протоколов.
- TCP-сегмент, состоящий из 1500 бит данных и 160 бит заголовка, посыпается на уровень протокола IP, где к нему присоединяются еще 160 бит очередного заголовка. Затем этот сегмент передается через две сети, в каждой из которых применяется 24-разрядный заголовок пакета. Максимальный размер пакета в целевой сети составляет 800 бит. Сколько битов, включая заголовки, доставляется протоколу сетевого уровня по месту назначения?

- 17.4.** Почему в TCP-заголовке имеется поле его длины, тогда как в UDP-заголовке оно отсутствует?
- 17.5.** В предыдущей версии спецификации протокола TFTP, описанной в документе RFC 783, было сказано следующее.
- Все пакеты, кроме тех, которые служат для окончания связи, подтверждаются по отдельности, если только не истечет время ожидания.*
- Это заявление было пересмотрено в новой версии и теперь гласит следующее.
- Все пакеты, кроме тех, которые дублируют пакеты подтверждения ACK, а также тех, которые служат для окончания связи, подтверждаются по отдельности, если только не истечет время ожидания.*
- Это изменение было сделано с целью устраниить затруднение под названием “Ученик чародея”. Поясните данное затруднение.
- 17.6.** Какой ограничивающий фактор в отношении времени требуется для передачи файла по протоколу TFTP?
- 17.7.** Пользователю узла UNIX требуется передать текстовый файл размером 4000 байт в узел Microsoft Windows. С этой целью он передает файл по протоколу TFTP, используя режим передачи netascii. И хотя было получено сообщение об успешной передаче файла, в узле Windows сообщается, что размер результирующего файла составляет 4050, а не 4000 байт, как у исходного файла. Подразумевает ли это различие в размерах файлов ошибку при передаче данных и почему?
- 17.8.** В спецификации протокола TFTP, описанной в документе RFC 1350, заявляется, что идентификаторы передачи (transfer identifier — TID) должны выбираться для соединения произвольно, чтобы вероятность выбрать один и тот же номер TID два раза подряд оказалась весьма малой. Какое затруднение может вызвать употребление одного и того же номера TID два раза подряд?
- 17.9.** Для повторной передачи потерянных пакетов по протоколу TFTP приходится хранить копию отсылаемых данных. Сколько пакетов данных необходимо хранить одновременно, чтобы реализовать такой механизм передачи данных?
- 17.10.** Как и в большинстве других протоколов, в протоколе TFTP никогда не посылает пакет ошибки в ответ на полученный пакет ошибки. Почему?
- 17.11.** Как упоминалось выше, для обработки потерянных пакетов в протоколе TFTP реализуется схема тайм-аута и повторной передачи. С этой целью при передаче пакета в удаленный узел устанавливается таймер повторной передачи. В большинстве реализаций протокола TFTP этот таймер устанавливается на фиксированное время около пяти секунд. Обсудите преимущества и недостатки установки фиксированного времени таймера повторной передачи.
- 17.12.** Схема тайм-аута и повторной передачи в протоколе TFTP подразумевает, что в конечном счете все пакеты данных будут приняты в целевом узле. Будут ли они при этом приняты неповрежденными? Почему?

**17.13.** Для каждой организации может быть доступна определенная совокупность особых услуг, которая зависит от конкретного поставщика услуг, их стоимости и оборудования в помещениях потребителя. Какие из подобных услуг доступны вам в вашем регионе?

**17.14.** Wireshark — это бесплатный анализатор пакетов, позволяющий перехватывать трафик в локальной сети. Он может применяться в самых разных операционных системах. Для этого необходимо установить драйвер перехвата пакетов WinPcap, который можно загрузить по адресу [www.wireshark.org](http://www.wireshark.org).

Переведя Wireshark в режим перехвата, запустите на выполнение TCP-приложение, действующее, например, по протоколу TELNET, FTP или HTTP (веб-браузер). Сумеете ли вы вычленить из перехваченного сетевого трафика следующую информацию?

- a. Адреса отправителя и получателя второго уровня (MAC).
- b. Адреса отправителя и получателя третьего уровня (IP).
- c. Адреса отправителя и получателя четвертого уровня (номера портов).

**17.15.** Программы перехвата или анализа пакетов могут быть весьма эффективными инструментальными средствами управления и обеспечения безопасности. Используя встроенную в них возможность фильтрации пакетов, можно проследить сетевой трафик по самым разным критериям, пренебрегая всем остальным. Воспользуйтесь встроенной в Wireshark возможностью фильтрации пакетов, чтобы сделать следующее.

- a. Перехватить только сетевой трафик, поступающий из вашего компьютера с соответствующим MAC-адресом.
- b. Перехватить только сетевой трафик, поступающий из вашего компьютера с соответствующим IP-адресом.
- c. Перехватить только передачи по протоколу UDP.

# ПРИЛОЖЕНИЕ 17.А. ПРОСТОЙ ПРОТОКОЛ ПЕРЕДАЧИ ФАЙЛОВ

В этом приложении представлен общий обзор стандартного простого протокола передачи файлов (TFTP — Trivial File Transfer Protocol), определенного в документе RFC 1350. При этом преследуется цель дать читателю ясное представление об элементах данного протокола. Протокол TFTP достаточно прост, чтобы служить кратким примером, хотя и включающим в себя существенные элементы из других, более сложных, протоколов.

## Введение в протокол TFTP

Протокол TFTP более прост, чем стандартный для Интернета протокол передачи файлов (FTP — File Transfer Protocol). В протоколе TFTP не обеспечивается управление доступом или идентификацией пользователей, и поэтому он пригоден только для каталогов с открытым доступом к файлам. Вследствие своей простоты протокол TFTP реализуется легко и компактно. Например, протокол TFTP применяется в некоторых бездисковых устройствах для загрузки встроенного в них программного обеспечения во время начальной загрузки. Протокол TFTP действует поверх протокола UDP. Объект TFTP начинает передачу данных с отправки запроса на чтение или запись в UDP-сегменте через порт назначения 69 целевой системы. Этот порт распознается целевым модулем UDP как идентификатор модуля TFTP. На протяжении всей передачи данных каждая участвующая в ней сторона пользуется идентификатором передачи (TID) как номером своего порта.

## Пакеты TFTP

Объекты TFTP обмениваются командами, ответами и файловыми данными в форме пакетов, каждый из которых переносится в теле UDP-сегмента. В протоколе TFTP поддерживаются пять типов пакетов (рис. 17.9), и первые два байта содержат код операции, идентифицирующий тип пакета.

- **Пакет RRQ.** Это пакет с запросом чтения, запрашивающий разрешение на передачу файла из другой системы. В этот пакет входит имя файла в виде последовательности байтов в коде ASCII<sup>2</sup>, завершающихся нулевым байтом. Нулевой байт служит для принимающего объекта TFTP признаком окончания имени файла. В состав данного пакета входит также поле режима, обозначающее, как следует интерпретировать файл данных: как символьную строку, состоящую из последовательности байтов в коде ASCII (режим netascii), или как исходные 8-разрядные байты данных (режим octet). В режиме netascii файл передается строками символов, каждая из которых завершается знаками перевода каретки и новой строки. Каждая система должна обязательно выполнять взаимное преобразование между собственным форматом символьных файлов и форматом TFTP.

<sup>2</sup> Сокращение "ASCII" означает "American Standard Code for Information Interchange" (Американский стандартный код для обмена информацией), т.е. стандартную кодировку, разработанную Американским национальным институтом стандартов для обозначения символов уникальным 7-битовым кодом, в котором дополнительный, восьмой, разряд служит для проверки на четность. Код ASCII равнозначен набору символов IRA (International Reference Alphabet — Международный эталонный алфавит), определенному в рекомендации T.50 МСЭ-Т.

- Пакет WRQ.** Это пакет с запросом на запись, запрашивающий разрешение на передачу файла в другую систему.
- Пакет данных.** Номера блоков в пакетах данных начинаются с единицы и увеличиваются на единицу с каждым новым блоком данных. Такое условие позволяет использовать в программе уникальные номера, чтобы отличать новые пакеты от дубликатов. Поле данных имеет длину от 0 до 512 байт. Если это поле оказывается длиной 512 байт, то блок данных не является последним. Если же его длина находится в пределах от 0 до 511 байт, это свидетельствует об окончании передачи данных.
- Пакет ACK.** Этот пакет служит для подтверждения приема пакета данных или пакета WRQ. Пакет ACK, подтверждающий пакет данных, содержит номер блока из подтверждаемого пакета данных, а пакет ACK, подтверждающий пакет WRQ, — нулевой номер блока.
- Пакет ошибки.** Этот пакет может служить для подтверждения приема пакета любого другого типа. Код ошибки представлен целым числом, обозначающим характер возникшей ошибки (табл. 17.1). Сообщение об ошибке предназначено для представления в удобочитаемом виде и поэтому должно быть составлено в коде ASCII. Как и все прочие символьные строки, сообщение об ошибке завершается нулевым байтом.

Все принятые пакеты, кроме рассматриваемых далее дубликатов ACK, а также пакеты, служащие для завершения передачи данных, должны быть непременно подтверждены. Любой пакет может быть подтвержден пакетом ошибок. Если ошибок нет, вступают в действие следующие соглашения.

2 байта	$n$ байт	1 байт	$n$ байт	1 байт
Код операции	Имя файла	0	Режим	0

#### Пакеты RRQ и WRQ

2 байта	2 байта	От 0 до 512 байт
Код операции	Номер блока	Данные

#### Пакет данных

2 байта	2 байта
Код операции	Номер блока

#### Пакет подтверждения ACK

2 байта	2 байта	$n$ байт	1 байт
Код операции	Код ошибки	Сообщение об ошибке	0

#### Пакет ошибки

Рис. 17.9. Форматы пакетов TFTP

Пакет WRQ или пакет данных подтверждается пакетом ACK. Когда посыпается пакет RRQ, другая сторона реагирует на него (в отсутствие ошибок), начиная передачу файла; таким образом, первый блок данных служит в качестве подтверждения пакета RRQ. Если только передача файла не завершена, за каждым пакетом ACK с одной стороны следует пакет данных с другой стороны, так что такой пакет данных служит в качестве подтверждения. Пакет ошибок может рассматриваться как подтверждение пакета любого другого типа в зависимости от конкретных обстоятельств.

**Таблица 17.1. Коды ошибок TFTP**

Код	Значение
0	Не определен; см. сообщение об ошибке, если таковое имеется
1	Файл не найден
2	Нарушение прав доступа
3	Диск заполнен или выделенное место занято
4	Некорректная операция TFTP
5	Неизвестный идентификатор передачи
6	Файл уже существует
7	Такого пользователя нет

Пакет данных, передаваемых по протоколу TFTP, приведен на рис. 17.10. Когда такой пакет передается вниз по иерархии протоколу UDP, этот протокол добавляет заголовок для формирования UDP-сегмента. Затем он передается протоколу IP, который добавляет IP-заголовок для формирования IP-дейтаграммы.

## Краткий обзор передачи данных

В качестве примера на рис. 17.11 демонстрируется простая операция передачи файла из системы А в систему В. Файл передается без ошибок, детали данной операции не рассматриваются.

Операция начинается с того, что модуль TFTP в системе А посыпает пакет WRQ модулю TFTP в системе В. Пакет WRQ переносится как тело UDP-сегмента и включает в себя имя файла (в данном случае — XXX), а также режим octet (необработанные данные). В UDP-заголовке целевой номер порта 69 говорит принимающему объекту UDP, что данное сообщение предназначено для приложения TFTP. Номер исходного порта является идентификатором TID, выбранным в системе А (в данном случае — 1511). Система В подготавливается к приему файла и отвечает пакетом ACK с нулевым номером блока. В UDP-заголовке целевой номер порта — 1511, что позволяет UDP-объекту в системе А направить входящий пакет модулю TFTP, идентификатор TID которого совпадает с идентификатором TID в пакете WRQ. Исходный порт представляет собой TID, выбранный в системе В для передачи файла (в данном случае — 1660).

После этого первоначального обмена пакетами следует передача файла. Она состоит из одного или нескольких пакетов данных, отправляемых системой А; прием каждого из них подтверждается системой В. Завершающий пакет данных содержит менее 512 байт, что свидетельствует об окончании передачи.

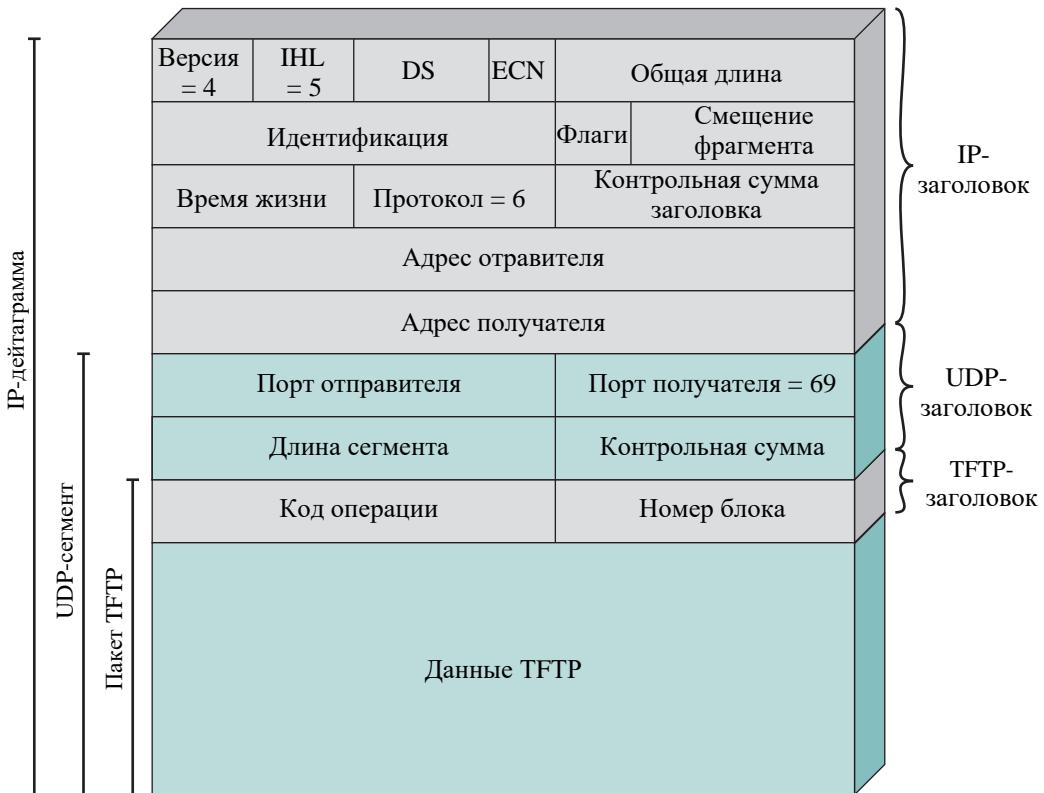


Рис. 17.10. Пакет TFTP

## Ошибки и задержки

Если протокол TFTP действует в сети или в Интернете, а не в канале прямой передачи данных, то передаваемые пакеты могут теряться. Поскольку протокол TFTP работает поверх протокола UDP, который не предоставляет гарантированную доставку пакетов, то в протоколе TFTP требуется некоторый механизм для обработки потерянных пакетов. Таким механизмом является установка тайм-аута до повторной передачи. Предположим, что система А отсылает системе В пакет, требующий подтверждения (т.е. любой пакет, кроме дублирующих пакетов ACK и пакетов, предназначенных для завершения передачи). Передав пакет, система А запускает таймер. Если время таймера истечет до получения подтверждения от системы В, система А снова передаст тот же самый пакет. Если исходный пакет действительно утерян, то будет повторно передана копия этого пакета, принятая системой В. Если же утерян не исходный, а подтверждающий пакет от системы В, то система В получит две копии одного и того же пакета от системы А и просто подтвердит прием обеих копий. Это не вызывает никакого конфликта благодаря применению номеров блоков. Единственным исключением из этого правила являются дублирующие пакеты ACK. Второй из них просто игнорируется.

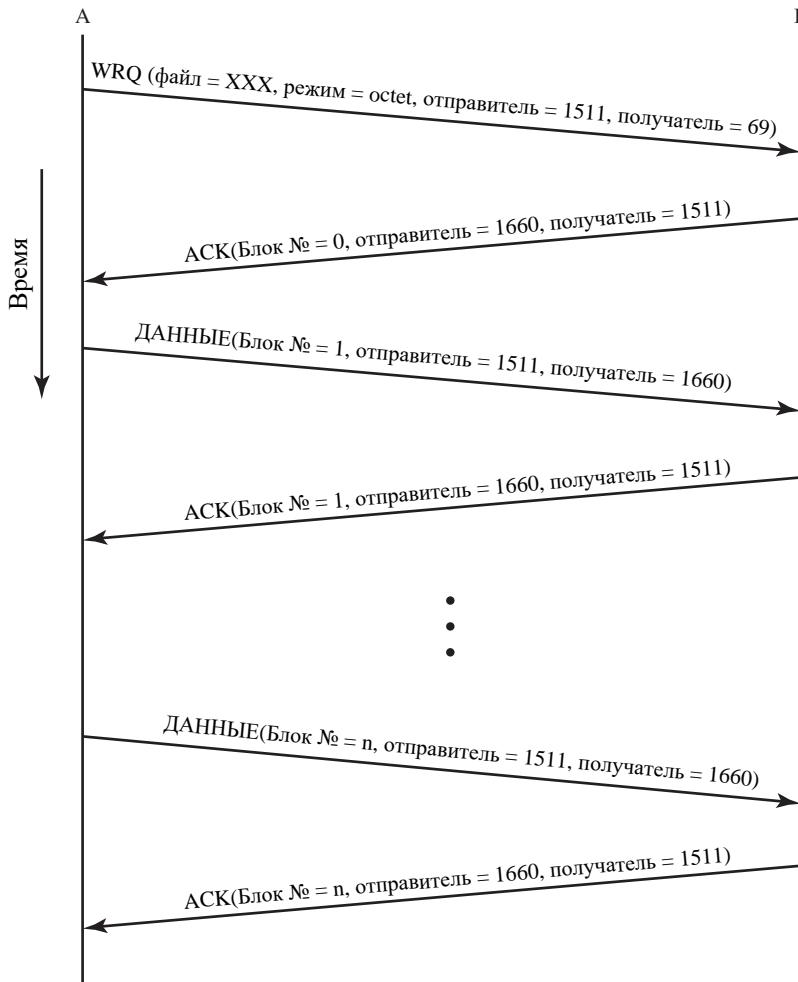


Рис. 17.11. Пример операции передачи файла по протоколу TFTP

## Синтаксис, семантика и синхронизация

Как упоминалось в разделе 17.1, основные свойства протокола можно разделить на категории синтаксиса, семантики и синхронизации. Эти категории легко обнаруживаются в протоколе TFTP. В частности, форматы различных пакетов TFTP определяют **синтаксис** протокола TFTP. **Семантика** данного протокола проявляется в определениях каждого типа пакета и в кодах ошибок. И наконец, последовательность обмена пакетами, применение номеров блоков данных и таймеров — все это особенности **синхронизации** в протоколе TFTP.



# ГЛАВА 18

# Распределенная обработка вычисления “клиент/сервер” и кластеры

В ЭТОЙ ГЛАВЕ...

## 18.1. Вычисления “клиент/сервер”

Что такое вычисления “клиент/сервер”

Приложения “клиент/сервер”

Приложения баз данных

Классы приложений “клиент/сервер”

Трехуровневая архитектура “клиент/сервер”

Согласованность содержимого файловых кешей

Промежуточное программное обеспечение

Архитектура промежуточного программного обеспечения

## 18.2. Распределенный обмен сообщениями

Надежность и ненадежность

Блокировка и неблокирующее выполнение

## 18.3. Вызов удаленных процедур

Передача параметров

Представление параметров

Привязка к архитектуре “клиент/сервер”

Синхронность и асинхронность

Объектно-ориентированные механизмы

## 18.4. Кластеры

- Конфигурации кластеров
- Вопросы проектирования операционных систем
  - Управление обработкой отказов
  - Распределение нагрузки
  - Распараллеливание вычислений
- Архитектура кластерных вычислительных систем
- Кластеры в сравнении с симметричной многопроцессорной обработкой

## 18.5. Кластерный сервер Windows

## 18.6. Кластеры Beowulf и Linux

- Функциональные средства Beowulf
- Программное обеспечение Beowulf

## 18.7. Резюме

## 18.8. Ключевые термины, контрольные вопросы и задачи

- Ключевые термины
- Контрольные вопросы
- Задачи

## УЧЕБНЫЕ ЦЕЛИ

- Представлять резюме главных особенностей клиент/серверных вычислений.
- Понимать основные принципы проектирования распределенного обмена сообщениями.
- Понимать основные принципы проектирования вызовов удаленных процедур.
- Понимать основные принципы проектирования кластеров.
- Описывать кластерные механизмы, действующие в Windows 7 и Beowulf.

Сначала в этой главе рассматриваются ключевые концепции, употребляемые в распределенном программном обеспечении, включая архитектуру “клиент/сервер”, обмен сообщениями и вызовы удаленных процедур. А затем в ней описывается приобретающая все большее и большее значение кластерная архитектура. Главы 17, “Сетевые протоколы”, и 18, “Распределенная обработка, вычисления «клиент/сервер» и кластеры”, завершаются описанием распределенных систем.

## 18.1. Вычисления “клиент/сервер”

Концепция клиент/серверных вычислений и связанные с ней понятия приобретают все большее и большее значение в IT-системах. Сначала в этом разделе описывается общий характер вычислений “клиент/сервер”, а затем обсуждаются альтернативные способы организации функций архитектуры “клиент/сервер”. Далее в этом разделе рассматривается вопрос согласованности файловых кешей, возникающий в связи с применением файловых серверов. И наконец, в этом разделе вводится понятие промежуточного программного обеспечения (middleware).

### Что такое вычисления “клиент/сервер”

Как и другие новые веяния в области вычислительной техники, вычисления “клиент/сервер” имеют собственную терминологию. Некоторые термины, зачастую обнаруживаемые в описаниях программных продуктов и приложений “клиент/сервер”, перечислены в табл. 18.1.

На рис. 18.1 предпринята попытка передать саму суть понятия “клиент/сервер”. Как подразумевает само понятие *среда “клиент/сервер”*, она состоит из клиентов и серверов. **Клиентами** обычно являются однопользовательские персональные компьютеры или рабочие станции, предоставляющие конечному пользователю удобный для работы интерфейс. На клиентской рабочей станции обычно предоставляется графический интерфейс, который считается самым удобным для пользователей, поскольку они могут работать с окнами, манипулируя мышью. Характерными примерами таких пользовательских интерфейсов служат Windows и Macintosh. Клиентские приложения специально приспособлены для того, чтобы ими можно было легко пользоваться, а к их числу относятся такие повсеместно известные инструментальные средства, как электронные таблицы.

**ТАБЛИЦА 18.1. ТЕРМИНОЛОГИЯ ВЫЧИСЛЕНИЙ ТИПА "КЛИЕНТ/СЕРВЕР"****Интерфейс прикладного программирования (API)**

Набор функций и вызовов программ, обеспечивающий взаимодействие клиентов и серверов.

**Клиент**

Подключенная к сети сторона, запрашивающая информацию, — как правило, персональный компьютер или рабочая станция, которая может запросить записи в базе данных и/или другую информацию на сервере.

**Промежуточное программное обеспечение**

Совокупность драйверов, API и другого программного обеспечения, улучшающего взаимодействие клиентского приложения с сервером.

**Реляционная база данных**

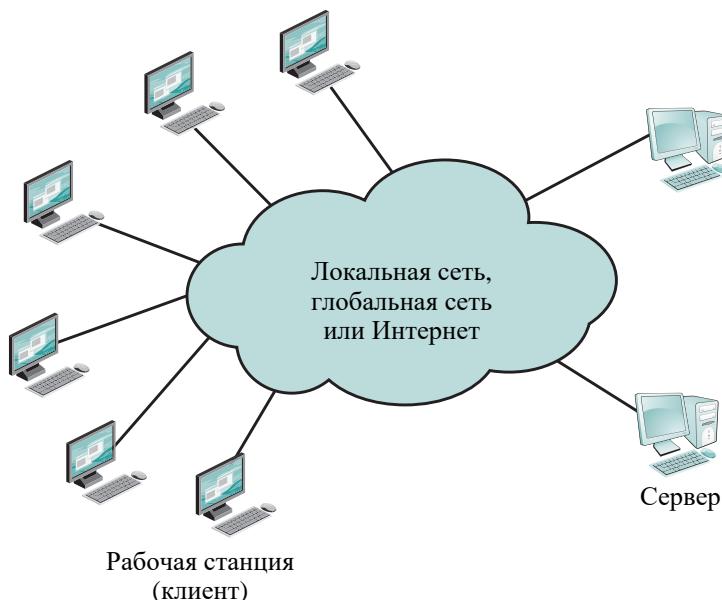
База данных, доступ к информации в которой ограничивается выбором строк, удовлетворяющих критериям поиска.

**Сервер**

Компьютер (зачастую высокопроизводительная рабочая станция), мини-ЭВМ или большая ЭВМ, на которой размещается информация, предназначенная для работы с подключаемыми к сети клиентами.

**Язык структурированных запросов (SQL)**

Язык, разработанный компанией IBM и стандартизованный ANSI для обращения к реляционным базам данных, их создания, обновления или запрашивания.

**Рис. 18.1. Типичная среда "клиент/сервер"**

Каждый сервер в среде “клиент/сервер” предоставляет клиентам ряд совместно используемых услуг. Наиболее распространенным в настоящее время типом сервера является сервер баз данных, обычно управляющий реляционной базой данных. Сервер позволяет многим клиентам совместно использовать общий доступ к одной и той же базе данных, а также пользоваться высокопроизводительной вычислительной системой для управления базой данных.

Помимо клиентов и серверов, третьей важной составляющей среды “клиент/сервер” является сеть. Вычисления “клиент/сервер”, как правило, являются распределенными. Пользователи, приложения и ресурсы являются распределенными в ответ на требования к предметной области и подключаются к одной локальной или глобальной сети или к объединениям сетей.

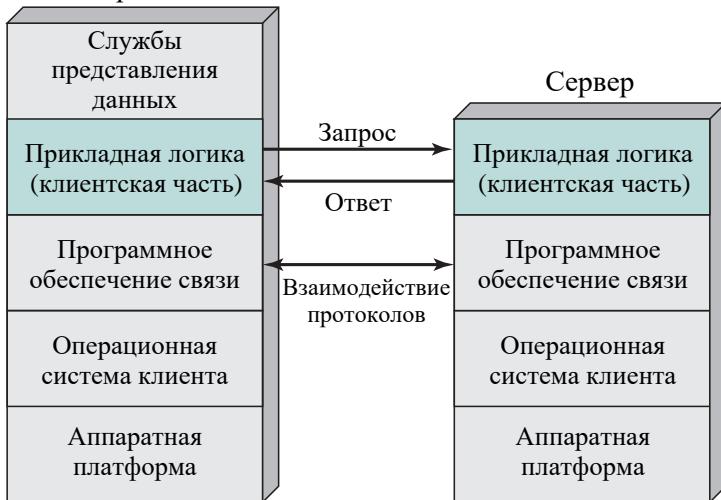
Чем же конфигурация “клиент/сервер” отличается от любого другого решения задачи распределенной обработки данных? Ниже перечислен ряд отличительных характеристик такой конфигурации, которыми она отличается от других типов распределенной обработки данных.

- Сильный акцент на перенос удобных для пользователя приложений в его систему. Это дает пользователю большие возможности контролировать синхронизацию и порядок пользования компьютером, а руководителям отделов организаций — своевременно реагировать на нужды своих подчиненных.
- Несмотря на то что приложения распределены, основной акцент все же делается на централизацию корпоративных баз данных и многие служебные функции и функции управления сетью. Это дает руководству организаций возможность сохранять полный контроль над общим объемом капиталовложений в вычислительные и информационные системы и обеспечивать их тесное взаимодействие. В то же время отдельные подразделения организаций избавляются от большей части издержек на эксплуатацию сложно устроенных вычислительных центров, и вместо этого могут выбрать практически любой тип машины и интерфейса для доступа к данным и прочей информации.
- Как у пользовательских организаций, так и у поставщиков имеется приверженность к открытым модульным системам. Это означает, что у пользователя появляется больше вариантов выбора программных продуктов и сочетания разнотипного оборудования от различных поставщиков.
- Подключение к сети является основой для работы “клиент/сервер”. Поэтому наивысший приоритет в организации и эксплуатации информационных систем отдается управлению сетью и безопасности.

## Приложения “клиент/сервер”

Ключевой особенностью архитектуры “клиент/сервер” является распределение задач прикладного уровня между клиентами и серверами. Общее представление архитектуры “клиент/сервер” приведено на рис. 18.2. Разумеется, основным программным обеспечением как на стороне клиента, так и на стороне сервера является операционная система, работающая на аппаратной платформе. Впрочем, платформы и операционные системы клиента и сервера могут различаться.

## Клиентская рабочая станция



**Рис. 18.2.** Типичная архитектура “клиент/сервер”

На самом деле в одной среде могут присутствовать разнотипные клиентские и серверные платформы и операционные системы. А поскольку отдельный клиент и сервер совместно пользуются одними и теми же протоколами передачи данных и поддерживают одни и те же приложения, то подобные низкоуровневые различия несущественны.

Именно коммуникационное программное обеспечение позволяет клиенту и серверу взаимодействовать. Характерным примером такого программного обеспечения служит набор протоколов TCP/IP. Безусловно, программное обеспечение, поддерживающее архитектуру “клиент/сервер” на уровне связи и операционной системы, предназначено служить основой для функционирования распределенных приложений. В идеальном случае конкретные функции, выполняемые приложением, могут быть распределены между клиентом и сервером таким образом, чтобы оптимизировать использование ресурсов. В одних случаях, в зависимости от потребностей приложений, большая часть прикладного программного обеспечения выполняется на сервере, тогда как в других случаях большая часть логики располагается на стороне клиента.

Важным фактором успешного развития среды “клиент/сервер” является порядок взаимодействия пользователя с системой в целом. Таким образом, решающее значение приобретает проектирование пользовательского интерфейса. В большинстве систем “клиент/сервер” сильный акцент делается на предоставление графического пользовательского интерфейса (GUI), который прост в применении, легко усваивается, но в то же время эффективен и гибок. Модуль услуг представления данных на клиентской рабочей станции можно рассматривать как ответственный за предоставление удобного пользовательского интерфейса для распределенных приложений, доступных в среде “клиент/сервер”.

### Приложения баз данных

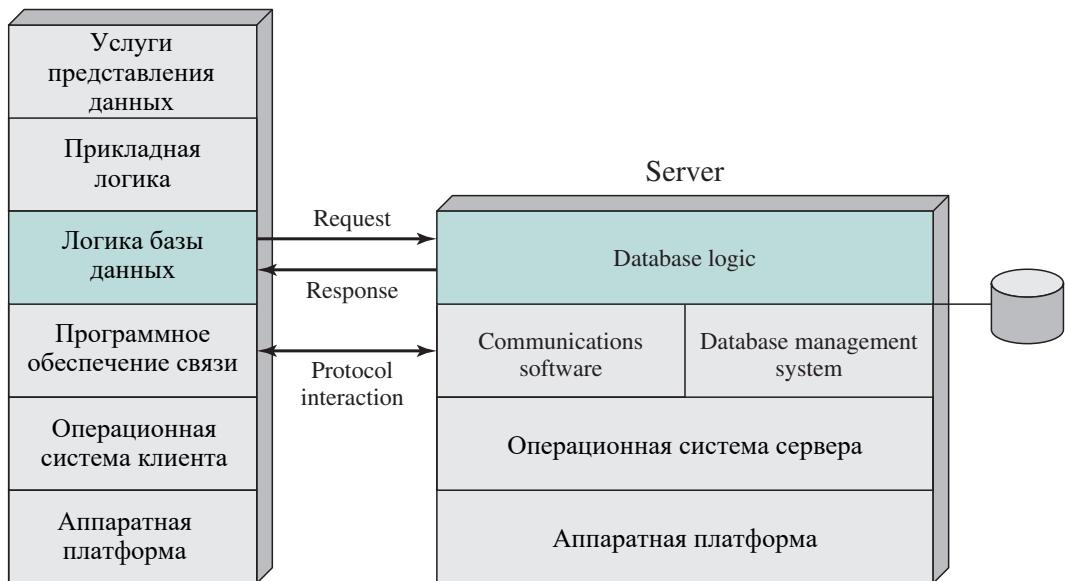
В качестве примера, демонстрирующего принцип распределения прикладной логики между клиентом и сервером, рассмотрим одно из самых распространенных семейств приложений типа клиент/сервер, в которых применяются реляционные базы данных. В такой среде сервер, по существу, выполняет функции сервера базы данных. Взаимо-

действие между клиентом и сервером осуществляется в форме транзакций, в которых клиент делает запрос к базе данных и получает от нее ответ.

На рис. 18.3 демонстрируется в общих чертах архитектура такой системы, в которой сервер отвечает за ведение базы данных. Для этой цели требуется программный модуль комплексной системы управления базой данных, а различные приложения, пользующиеся базой данных, могут быть размещены на клиентских машинах. Звеном, связующим клиента с сервером, служит программное обеспечение, позволяющее клиенту делать запросы на доступ к базе данных сервера. Распространенным примером такой логики служит язык структурированных запросов (SQL).

### Клиентская

#### рабочая станция



**Рис. 18.3.** Архитектура "клиент/сервер" для приложений баз данных

В архитектуре, показанной на рис. 18.3, предполагается, что вся прикладная логика, реализуемая в программном обеспечении для обработки больших массивов числовых данных или других видов их анализа, располагается на стороне клиента, тогда как сервер отвечает только за ведение базы данных. Пригодность такой конфигурации зависит от типа и назначения конкретного приложения. Предположим, что основная цель состоит в том, чтобы предоставить интерактивный доступ к базе данных для поиска в ней записей. На рис. 18.4, а показано, как может быть достигнута эта цель. Допустим, что сервер ведет базу данных с 1 миллионом записей, именуемых в терминологии баз данных строками, а пользователю требуется произвести поиск, чтобы получить в итоге не более чем несколько записей. Пользователь может искать эти записи по целому ряду критерии (например, записи, сделанные раньше 1992 года; записи, обозначающие отдельных жителей штата Огайо; записи, обозначающие конкретное событие или характеристику и т.д.). По первоначальному запросу клиент может получить от сервера ответ, указывающий на присутствие в базе данных 100 000 возможных записей, удовлетворяющих критерию поиска, а затем ввести дополнительные уточнения и ограничения.

чения в новый запрос. На этот раз возвратится ответ, указывающий на присутствие в базе данных 1000 возможных записей. И наконец, пользователь выдает третий запрос с дополнительными уточнениями, которые совпадают с единственной записью, которая и возвращается клиенту.



а) Желаемое применение архитектуры "клиент/сервер"



б) Неправильное применение архитектуры "клиент/сервер"

**Рис. 18.4.** Пример применения базы данных в архитектуре “клиент/сервер”

Упомянутое выше приложение вполне пригодно для архитектуры “клиент/сервер” по двум следующим причинам.

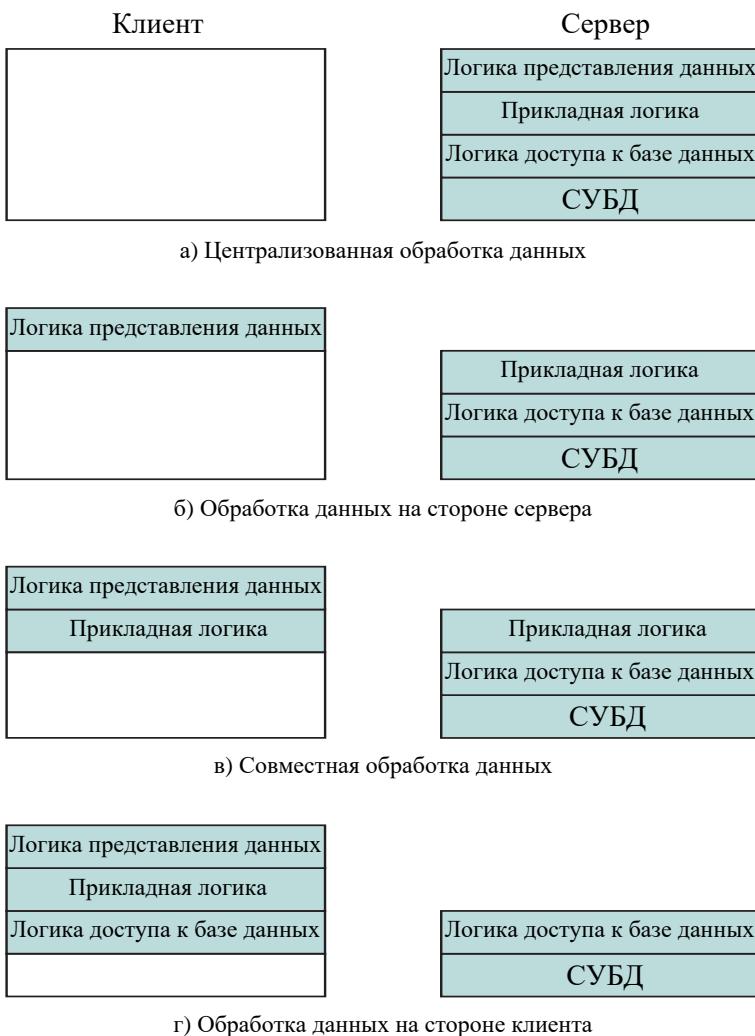
1. Для массовой сортировки и поиска информации в базе данных требуются большой диск или даже дисковый массив, быстродействующий процессор и архитектура высокоскоростного ввода-вывода. Такие емкости и вычислительные мощности не требуются (да и обходятся слишком дорого) для однопользовательской рабочей станции или персонального компьютера.
2. Для передачи файла с 1 миллионом записей на сторону клиента с целью поиска придется наложить на сеть непосильное бремя. Поэтому недостаточно, чтобы сервер мог лишь извлекать записи от имени клиента. Для поиска информации серверу потребуется логика доступа к базе данных, позволяющая искать информацию от имени клиента.

А теперь рассмотрим приведенный на рис. 18.4, б пример использования архитектуры “клиент/сервер” с той же самой базой данных на 1 миллион записей. В данном случае по единственному запросу через сеть передается 300 000 записей. Такое может произойти, например, в том случае, если пользователю требуется найти общий итог или среднее значение по какому-нибудь полю во многих записях или даже во всей базе данных.

Очевидно, что этот последний пример использовании архитектуры “клиент/сервер” неприемлем. В качестве выхода из подобного затруднения с целью сохранить все преимущества архитектуры “клиент/сервер” можно, например, перенести часть прикладной логики на сервер. Это означает, что сервер должен быть оснащен прикладной логикой для анализа, извлечения и поиска информации в базе данных.

### **Классы приложений “клиент/сервер”**

В общих рамках архитектуры “клиент/сервер” имеется целый спектр реализаций, по-разному распределяющих работу между клиентом и сервером. На рис. 18.5 представлены в общих чертах основные варианты приложений базы данных. Но в любом случае этот рисунок полезно проанализировать, чтобы дать ясное представление о возможных компромиссах. Ниже перечислены классы приложений типа “клиент/сервер”, приведенных на рис. 18.5.



**Рис. 18.5. Классы приложений “клиент/сервер”**

- **Централизованная обработка данных.** Такая обработка данных не является подлинным вычислением типа “клиент/сервер”, как подразумевает ее название. Напротив, *централизованная обработка данных* относится к традиционной среде больших ЭВМ, в которой буквально вся обработка выполняется на центральном узле, а пользовательский интерфейс зачастую предоставляется через простой, не-программируемый терминал. И даже если пользователь задействует мини-ЭВМ, роль его рабочей станции все равно ограничивается эмуляцией терминала.
- **Обработка данных на стороне сервера.** Это основной класс приложений типа “клиент/сервер”, в котором клиент в основном отвечает за предоставление графического пользовательского интерфейса, тогда как буквально вся обработка данных выполняется на сервере. Такая конфигурация является типичной для первых попыток реализовать архитектуру “клиент/сервер”, особенно в системах, действующих на уровне отдельных подразделений организации. Довод в пользу такой конфигурации состоит в том, что пользовательская рабочая станция лучше всего подходит для предоставления удобного пользовательского интерфейса и что базы данных и приложения можно легко сопровождать в центральных системах. И хотя пользователь получает в качестве преимущества лучший интерфейс, такой тип конфигурации обычно не допускает никаких выгод в производительности или коренных изменениях в конкретных функциях, реализующих логику предметной области, которые поддерживаются в системе.
- **Обработка данных на стороне клиента.** На другом краю спектра находятся приложения типа “клиент/сервер”, выполняющие всю обработку данных на стороне клиента, за исключением процедур проверки достоверности данных и остальных функций, которые реализуют логику доступа к базе данных и которые лучше выполнять на сервере. Как правило, некоторые из более изощренных функций, реализующих логику доступа к базе данных, располагаются на стороне клиента. Такая архитектура сегодня, вероятно, является наиболее распространенным подходом к реализации принципа “клиент/сервер”. Она предоставляет пользователю возможность задействовать приложения, подходящие для удовлетворения локальных потребностей.
- **Совместная обработка данных.** В такой конфигурации приложение, обрабатывающее данные, выполняется в оптимальном виде, выгодно используя сильные стороны машин как на стороне клиента, так и на стороне сервера, а также распределенный характер данных. Настроить и поддерживать такую конфигурацию сложнее, но в долгосрочной перспективе она может обеспечить повышение производительности труда пользователей и эффективности использования сети по сравнению с другими подходами к архитектуре “клиент/сервер”.

На рис. 18.5, в и г представлены конфигурации, в которых значительная доля нагрузки ложится на клиента. Эта модель, именуемая **толстым клиентом**, нашла широкое распространение благодаря таким инструментальным средствам для разработки приложений, как PowerBuilder от корпорации Sybase, Inc. и SQL Windows от Gupta Corp. Приложения, разрабатываемые с помощью подобных инструментальных средств, как правило, рассчитаны на применение в рамках подразделений организации. Главное преимущество модели толстого клиента заключается в том, что в ней выгодно используются вычислительные мощности настольных систем, а серверы освобождаются от бремени

обработки данных и поэтому работают более эффективно, а вероятность появления в их работе узких мест заметно уменьшается.

Впрочем, модели толстого клиента присущи свои недостатки. В частности, внедрение дополнительных функций быстро приводит к превышению обрабатывающих способностей настольных систем, вынуждая их владельцев прибегать к модернизации. Если такая модель выйдет за пределы подразделения, чтобы охватить многих пользователей, то организации придется развернуть локальные сети с высокой пропускной способностью для поддержки больших объемов данных, которыми обмениваются клиенты и серверы. И наконец, приложения, распределенные среди десятков или даже сотен настольных систем, нелегко сопровождать, обновлять или заменять.

На рис. 18.5, б представлен подход, называемый **тонким клиентом**. Такой подход в большей степени подражает традиционному централизованному подходу и зачастую служит путем переноса развивающихся общекорпоративных приложений из больших ЭВМ в распределенную среду.

### Трехуровневая архитектура “клиент/сервер”

Традиционная архитектура “клиент/сервер” состоит из двух уровней или ярусов: клиентского и серверного. Не менее распространена и трехуровневая архитектура “клиент/сервер” (рис. 18.6). В ней прикладное программное обеспечение распределено среди трех типов машин: пользовательской машины, сервера среднего уровня и внутреннего сервера (базы данных).



**Рис. 18.6.** Трехуровневая архитектура “клиент/сервер”

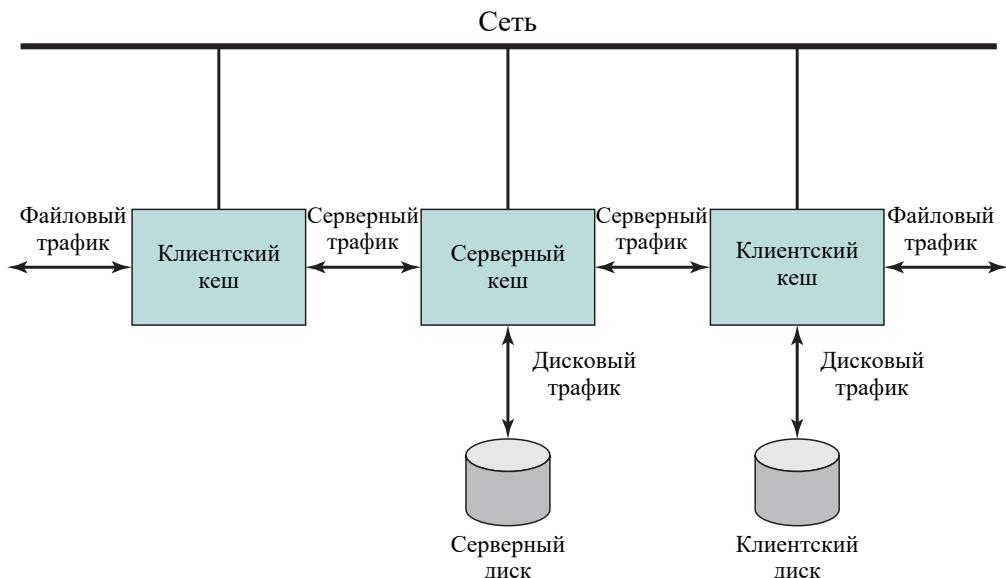
Пользовательской является упоминавшаяся ранее клиентская машина; в трехуровневой модели она, как правило, выполняет функции тонкого клиента. Машины среднего уровня, по существу, являются шлюзами между тонкими клиентами пользователей и разнообразными внутренними серверами баз данных. Машины среднего уровня способны преобразовывать сетевые протоколы и одни типы запросов базы данных в другие, а также объединять или интегрировать результаты, получаемые из разных источников данных. И наконец, машина среднего уровня может служить шлюзом между настольными клиентскими приложениями и устаревшими серверными приложениями, играя роль посредника между ними.

Взаимодействие сервера среднего уровня и сервера баз данных также следует модели клиент/сервер. Таким образом, система среднего уровня действует как в качестве клиента, так и в качестве сервера.

### **Согласованность содержимого файловых кешей**

При применении файлового сервера производительность файлового ввода-вывода может заметно упасть по сравнению с локальным доступом к файлам вследствие задержек, вносимых сетью. Для сокращения такого снижения производительности в отдельных системах могут применяться файловые кеши, где хранятся записи файлов, к которым недавно выполнялось обращение. В силу принципа локальности применение такого локального файлового кеша должно способствовать сокращению количества попыток доступа к удаленному серверу.

На рис. 18.7 наглядно показан типичный распределенный механизм кеширования файлов среди подключенных к сети рабочих станций.



**Рис. 18.7.** Распределенное кеширование файлов в ОС Sprite

Когда процесс осуществляет доступ к файлу, сначала делается запрос к кешу на рабочей станции данного процесса (так называемый “файловый трафик”). Если этот запрос не удовлетворен, он направляется на локальный диск, если искомый файл хранится там (так называемый “дисковый трафик”), или же на файловый сервер, где хранится искомый файл (так называемый “серверный трафик”). На сервере сначала опрашивается кеш, и, если файл в нем отсутствует, то осуществляется доступ к диску на сервере. Такой метод двойного кеширования применяется с целью сократить сетевой трафик и дисковый ввод-вывод.

Если кеши всегда содержат точные копии данных из удаленного источника, они называются **согласованными** (*consistent*). Кеши могут стать несогласованными, когда данные в удаленном источнике изменяются; их устаревшие копии в локальном кеше при этом не уничтожаются. Это может произойти, если клиент модифицирует файл, который кешируется другими клиентами. При этом возникают трудности на двух уровнях. Так, даже если клиент действует по правилу, предписывающему немедленную запись любых изменений обратно в файл на сервере, то любой другой клиент, имеющий в своем распоряжении кешированную копию соответствующей части файла, будет обладать устаревшими данными. Трудность усугубляется еще больше, если клиент задерживает запись изменений в файл на сервере. В таком случае устаревшая версия файла оказывается на самом сервере и по новым запросам чтения из файла могут быть получены устаревшие данные. Сложности, связанные с хранением в локальных кешах актуальных копий, отвечающих последним изменениям данных в удаленном источнике, получили название проблемы **согласованности содержимого кешей** (*cache consistency*).

В простейшем случае обеспечить согласованность содержимого кешей можно, воспользовавшись методами блокировки файлов, чтобы исключить тем самым одновременный доступ к файлу со стороны нескольких клиентов. Но при этом согласованность содержимого кешей обеспечивается за счет снижения производительности и гибкости. В операционной системе Sprite обеспечивается более эффективный подход с применением специальных возможностей [179, 181]. Файл может быть открыт для чтения и создания собственного клиентского кеша любым количеством удаленных процессов. Но когда сервер получает запрос на доступ к открытому файлу для записи данных, а другие процессы открыли этот файл, чтобы получить к нему доступ для чтения, сервер предпринимает два действия. Во-первых, он уведомляет записывающий процесс, что тот, несмотря на возможную поддержку собственного кеша, должен записать обратно в файл все измененные блоки данных немедленно после их обновления. Таких клиентов у сервера может быть не больше одного. Во-вторых, сервер уведомляет все процессы, открывшие файл для чтения, что этот файл больше не является кешируемым.

## Промежуточное программное обеспечение

Разработка программных продуктов на основе архитектуры “клиент/сервер” намного опередила работы по стандартизации всех особенностей распределенных вычислений: от физического уровня и вплоть до прикладного уровня. Такой недостаток стандартов затрудняет реализацию интегрированной конфигурации “клиент/сервер” масштаба предприятия от разных поставщиков. А поскольку большая доля выгоды от такой конфигурации связана с ее модуляризацией и способностью сочетать разные платформы и приложения с целью предоставить приемлемое техническое решение, то необходимо решить задачу их нормального взаимодействия.

Чтобы извлечь настоящие выгоды из архитектуры “клиент/сервер”, разработчики должны иметь в своем распоряжении набор инструментальных средств, обеспечивающих единообразный доступ к системным ресурсам на всех платформах. Это даст разработчикам возможность строить приложения, которые не только выглядят одинаково на разных персональных компьютерах и рабочих станциях, но и пользуются одним и тем же методом доступа к данным независимо от их местоположения.

Самый распространенный способ удовлетворить такое требование — воспользоваться стандартными интерфейсами и протоколами, занимающими промежуточное положение между приложением вверху и программным обеспечением связи и операционной системой внизу. Такие стандартизованные программные интерфейсы и протоколы обычно принято называть промежуточным программным обеспечением (*middleware*). С помощью стандартных программных интерфейсов нетрудно реализовать одно и то же приложение на разнотипных серверах и рабочих станциях. Выгоды для потребителей вполне очевидны, но и у поставщиков имеются мотивы для предоставления таких интерфейсов. Дело в том, что потребители приобретают приложения, а не серверы; они выбирают только те серверы, которые позволяют выполнять необходимые им приложения. Для связи различных серверных интерфейсов с клиентами, которым требуется к ним доступ, нужны стандартизованные протоколы.

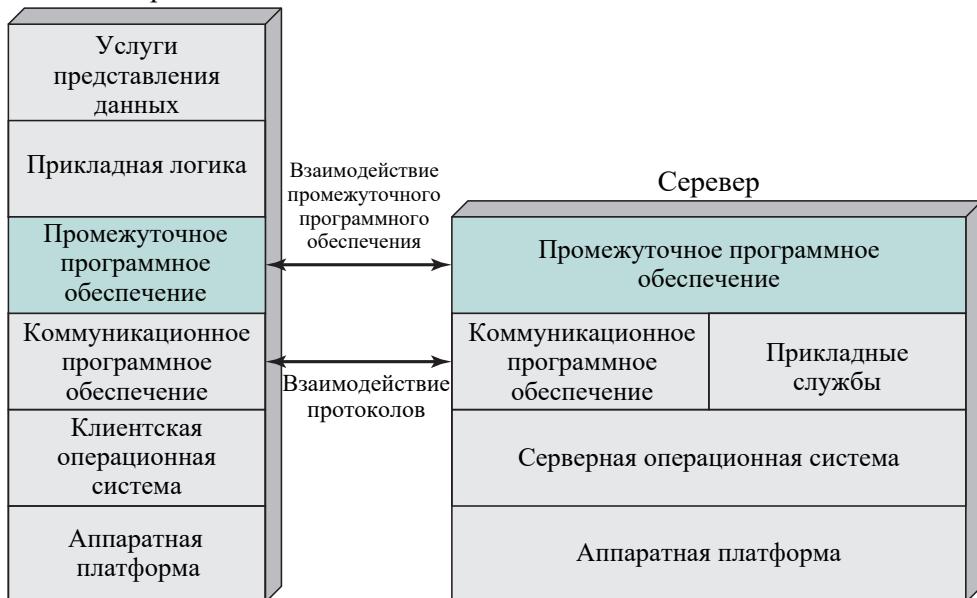
Существуют самые разные пакеты промежуточного программного обеспечения — от очень простых до весьма сложных. Общей для них является возможность скрыть сложности и несоответствия разных сетевых протоколов и операционных систем. Поставщики клиентского и серверного программного обеспечения обычно предоставляют целый ряд более распространенных пакетов промежуточного программного обеспечения в качестве дополнительных возможностей. Таким образом, пользователь может выбрать конкретную стратегию промежуточного программного обеспечения, а затем подобрать оборудование от различных поставщиков, поддерживающих такую стратегию.

### ***Архитектура промежуточного программного обеспечения***

На рис. 18.8 наглядно продемонстрирована роль промежуточного программного обеспечения в архитектуре “клиент/сервер”. Конкретная роль компонента промежуточного программного обеспечения зависит от вида выполняемых вычислений “клиент/сервер”. Если вернуться к рис. 18.5, то можно вспомнить, что имеется целый ряд классов приложений “клиент/сервер”, различающихся распределением их функций. Так или иначе, рис. 18.8 дает ясное общее представление об архитектуре промежуточного программного обеспечения.

Следует заметить, что имеются как клиентский, так и серверный компоненты промежуточного программного обеспечения. Основное назначение промежуточного программного обеспечения — разрешить приложению или пользователю на стороне клиента доступ к самым разным услугам на стороне сервера, не особенно беспокоясь о различиях среди серверов. В качестве примера применения промежуточного программного обеспечения в одной из прикладных областей может служить язык структурированных запросов (SQL), предназначенный для предоставления стандартизованных средств доступа к реляционной базе данных со стороны локального или удаленного пользователя или приложения. Но несмотря на то, что многие поставщики реляционных баз данных поддерживают язык SQL, они все же дополняют его собственными расширениями, что приводит к потенциальной несовместимости приложений.

## Клиентская рабочая станция



**Рис. 18.8.** Роль промежуточного программного обеспечения в архитектуре “клиент/сервер”

В качестве примера рассмотрим распределенную систему, предназначенную, среди прочего, для поддержки деятельности отдела кадров. Основные сведения о сотрудниках, включая фамилию, имя и отчество, а также адрес сотрудника, могут храниться в базе данных Gupta, тогда как сведения о зарплате — в базе данных Oracle. Когда пользователю данной системы из отдела кадров требуется доступ к конкретным записям, его не должно интересовать, в базе данных какого именно поставщика хранится требующаяся ему запись. Промежуточное программное обеспечение обеспечивает уровень программного обеспечения, допускающий единообразный доступ к подобным системам от разных поставщиков.

Поучительно рассмотреть роль промежуточного программного обеспечения с логической точки зрения (рис. 18.9), а не с точки зрения его реализации. Промежуточное программное обеспечение позволяет выполнить обещание распределенного характера вычислений “клиент/сервер”. Всю распределенную систему можно рассматривать как ряд приложений и ресурсов, доступных пользователям. При этом пользователей вообще не должно интересовать местоположение данных или приложений. Все приложения действуют через единообразный интерфейс прикладного программирования (API), а промежуточное программное обеспечение, врезающееся в промежуток между всеми клиентскими и серверными платформами, отвечает за маршрутизацию клиентских запросов соответствующему серверу.

Несмотря на наличие большого разнообразия промежуточного программного обеспечения, соответствующие продукты, как правило, основываются на одном из двух базовых механизмов: обмене сообщениями или вызовах удаленных процедур. Оба эти механизма рассматриваются в двух следующих разделах.

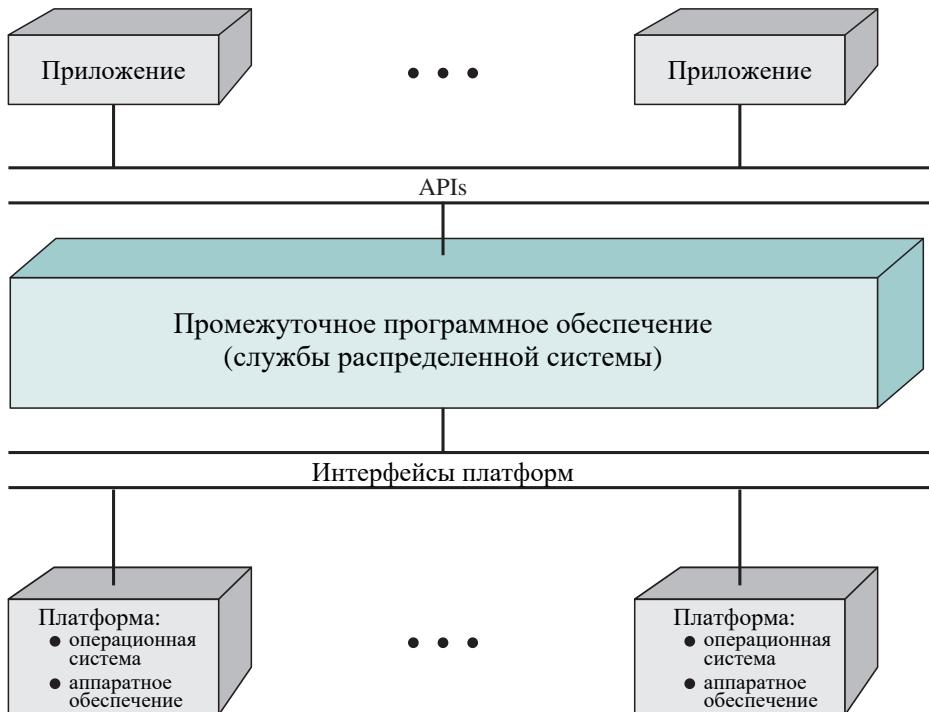
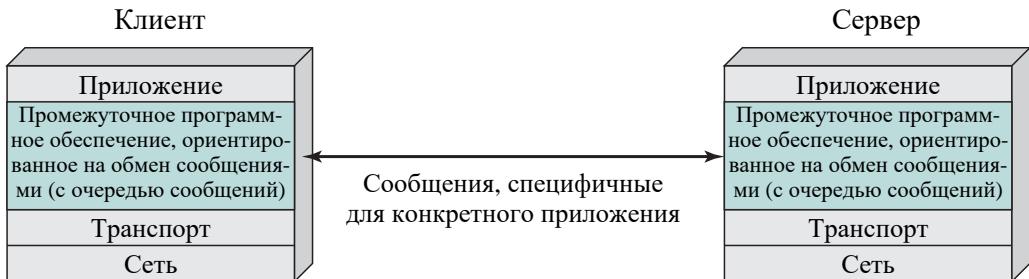


Рис. 18.9. Логическое представление промежуточного программного обеспечения

## 18.2. РАСПРЕДЕЛЕННЫЙ ОБМЕН СООБЩЕНИЯМИ

В системах распределенной обработки данных обычно принято, чтобы компьютеры не использовали совместно основную память, поскольку каждый из них представляет собой изолированную вычислительную систему. Следовательно, в таких системах нельзя применять методы межпроцессорного взаимодействия, полагающиеся на совместно используемую общую память (например, семафоры). Вместо них применяются методы, полагающиеся на механизм обмена сообщениями. В этом и следующем разделах будут рассмотрены два таких механизма. В первом из них применяется простой обмен сообщениями, как если бы они использовались в единой системе, а во втором — вызов удаленных процедур.

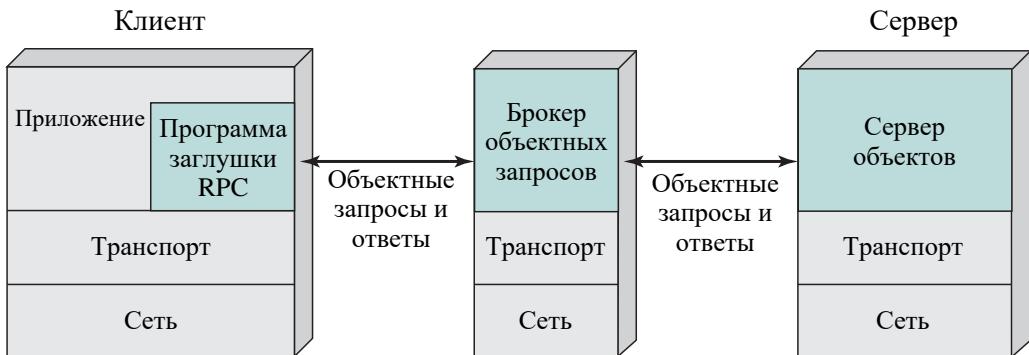
На рис. 18.10, *a* показано, каким образом с помощью обмена сообщениями реализуются функциональные возможности архитектуры “клиент/сервер”. Процессу клиента требуется некоторая служба (например, чтение содержимого файла или печать), и с этой целью он отправляет серверному процессу сообщение, содержащее запрос на обслуживание. Серверный процесс обрабатывает данный запрос и отправляет клиентскому процессу сообщение, содержащее ответ. Таким образом, для реализации обмена сообщениями в самой простой форме требуются лишь две функции: отправки и получения. В функции отправки указываются место назначения отправляемого сообщения, а также его содержимое. Функция получения выясняет, от кого именно получено сообщение (включая “всех”), а также предоставляет буфер для хранения входящего сообщения.



а) Промежуточное программное обеспечение, ориентированное на обмен сообщениями



б) Вызовы удаленных процедур

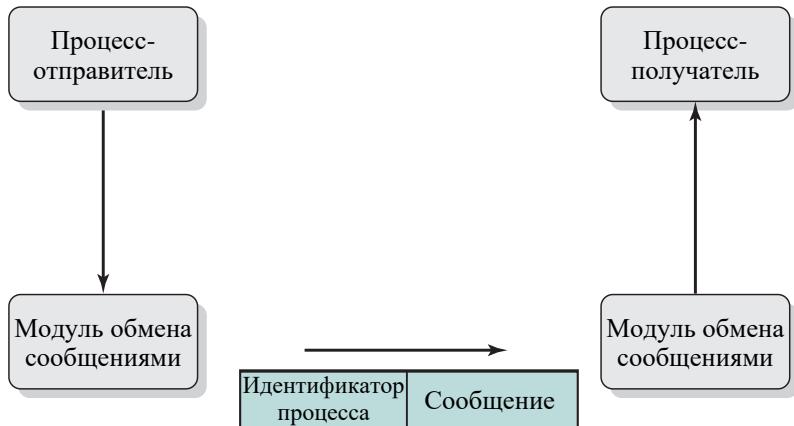


в) Брокер объектных запросов

**Рис. 18.10.** Механизмы ППО для обмена сообщениями

На рис. 18.11 показано, каким образом реализуется обмен сообщениями. Процессы пользуются услугами модуля обмена сообщениями. Запросы на обслуживание можно выразить в виде примитивов и параметров, где примитив обозначает выполняемую функцию, а параметры служат для обмена данными и управляющей информацией. Конкретная форма примитива зависит от программного обеспечения обмена сообщениями. Это может быть вызов процедуры или же само сообщение для процесса как часть операционной системы.

Примитив отправки используется в том процессе, в котором требуется отправить сообщение. Параметрами являются идентификатор целевого процесса и содержимое сообщения.



**Рис. 18.11.** Основные примитивы для обмена сообщениями

Модуль обмена сообщениями составляет блок данных, включающий в себя оба эти элемента. Этот блок данных отправляется на ту машину, на которой находится целевой процесс, с помощью средства связи некоторого вида (например, набора протоколов TCP/IP). Когда целевая система получает этот блок данных, средство связи направляет его модулю обмена сообщениями. В этом модуле анализируется поле идентификатора процесса, а сообщение сохраняется в буфере, выделенном для данного процесса.

В этом сценарии получающий процесс должен заявить о своей готовности получать сообщения, выделив буферную область и известив модуль обмена сообщениями с помощью примитива получения. При альтернативном подходе такого объявления не требуется. Вместо этого, получив сообщение, модуль обмена сообщениями отсылает его целевому процессу с помощью сигнала получения определенного вида, а затем делает полученное сообщение доступным в совместно используемом буфере.

С распределенным обменом сообщениями связано несколько вопросов проектирования, обсуждаемых в остальной части этого раздела.

## Надежность и ненадежность

Надежное средство обмена сообщениями — это средство, которое гарантирует их доставку. В таком средстве применяется надежный транспортный протокол или аналогичная логика, а также выполняются проверка ошибок, подтверждение, повторная передача и переупорядочение неупорядоченных сообщений. Поскольку доставка сообщения гарантируется, отправляющему процессу не обязательно знать, что оно доставлено. Тем не менее было бы полезно направить подтверждение обратно процессу-отправителю, чтобы дать ему знать, что доставка сообщения успешно произошла. Но в любом случае, если средству обмена сообщениями не удастся доставить сообщение (например, из-за постоянного отказа в сети или аварийного сбоя в целевой системе), об этом обязательно уведомляется отправляющий процесс.

С другой стороны, средство обмена сообщениями может просто отправить сообщение в сеть, но не уведомлять ни об удачном, ни о неудачном исходе его доставки. Такая альтернатива значительно упрощает обмен сообщениями и их обработку, а также сокращает накладные расходы обмена сообщениями. Те приложения, которым все же требуется подтверждение доставки сообщений, могут удовлетворить данное требование самостоятельно с помощью сообщений-запросов и ответов.

## Блокировка и неблокирующее выполнение

При использовании неблокирующих (асинхронных) примитивов текущий процесс в результате выдачи примитива отправки или получения не приостанавливается. Так, если процесс выдаст примитив отправки, операционная система возвратит управление данному процессу, как только сообщение будет поставлено в очередь на передачу или же будет создана его копия. Если копия сообщения не создана, то любые изменения, внесенные отправляющим процессом в сообщение до и даже во время его передачи, остаются на ответственности данного процесса. Как только сообщение будет передано или скопировано в надежное место для последующей передачи, произойдет прерывание отправляющего процесса с извещением о готовности буфера к новому использованию. Аналогично при выдаче неблокирующего примитива получения текущий процесс продолжает свое выполнение. Как только поступит сообщение, процесс будет прерван извещением об этом событии; в качестве альтернативы он может сам периодически опрашивать состояние получения.

Неблокирующие примитивы дают процессам возможность эффективно и гибко пользоваться средством обмена сообщениями. Недостаток такого подхода заключается в том, что программы, в которых применяются неблокирующие примитивы, с трудом поддаются тестированию и отладке. При этом невоспроизводимые, зависящие от временных характеристик последовательности действий могут создавать тонкие и сложные проблемы.

В качестве альтернативы можно воспользоваться блокирующими (синхронными) примитивами. Такой блокирующий примитив отправки сообщения не возвращает управление отправляющему процессу до тех пор, пока сообщение не будет передано (ненадежное обслуживание) или же пока сообщение не только будет передано, но и будет получено подтверждение об его доставке (надежное обслуживание). Блокирующий примитив получения не возвратит управление до тех пор, пока сообщение не будет размещено в выделенном буфере.

## 18.3. Вызов удаленных процедур

Вызов удаленных процедур является разновидностью основной модели обмена сообщениями. В настоящее время это широко принятый и весьма распространенный метод инкапсуляции коммуникации в распределенной системе. Суть метода состоит в том, чтобы дать программам на разных машинах возможность взаимодействовать, используя семантику вызовов и возвратов из простых процедур, как будто обе взаимодействующие программы находятся на одной машине. То есть вызов процедуры используется для доступа к удаленным службам. Распространенность такого подхода объясняется следующими преимуществами.

1. Вызов процедуры является широко принятой, общеупотребимой и вполне понятной абстракцией.
2. Применение вызовов удаленных процедур позволяет указывать удаленные интерфейсы как совокупность именованных операций с определенными типами. Такие интерфейсы могут быть точно документированы, а распределенные программы — статически проверены на ошибки несоответствия типов.

3. Наличие стандартизированного точно определенного интерфейса обеспечивает возможность автоматической генерации кода передачи данных для приложения.
4. Наличие стандартизированного точно определенного интерфейса обеспечивает возможность написания разработчиками клиентских и серверных модулей, которые могут быть перенесены с одного компьютера и операционной системы на другой компьютер и операционную систему с незначительной модификацией.

Механизм вызова удаленных процедур можно рассматривать как уточнение надежного блокирующего механизма обмена сообщениями. На рис. 18.10, б наглядно показана общая архитектура вызова удаленных процедур, а на рис. 18.12 данный механизм представлен более подробно. В вызывающей программе делается обычный вызов с параметрами на своей машине, например

CALL P(X, Y),

где

CALL — собственно вызов;  
 Р — имя процедуры;  
 X — передаваемые аргументы;  
 Y — возвращаемые значения.

Такой вызов может быть прозрачным для пользователя (а может и не быть), намеревающегося вызвать удаленную процедуру на некоторой другой машине. Фиктивная процедура-заглушка Р должна быть включена в адресное пространство вызывающей программы или же динамически связана с ним во время вызова. Эта процедура создает сообщение, указывающее вызываемую процедуру и включающее ее параметры, а затем отправляет данное сообщение удаленной системе и ожидает ответа. Как только ответ будет получен, процедура-заглушка возвратит управление вызывающей программе, предоставив ей полученные значения.

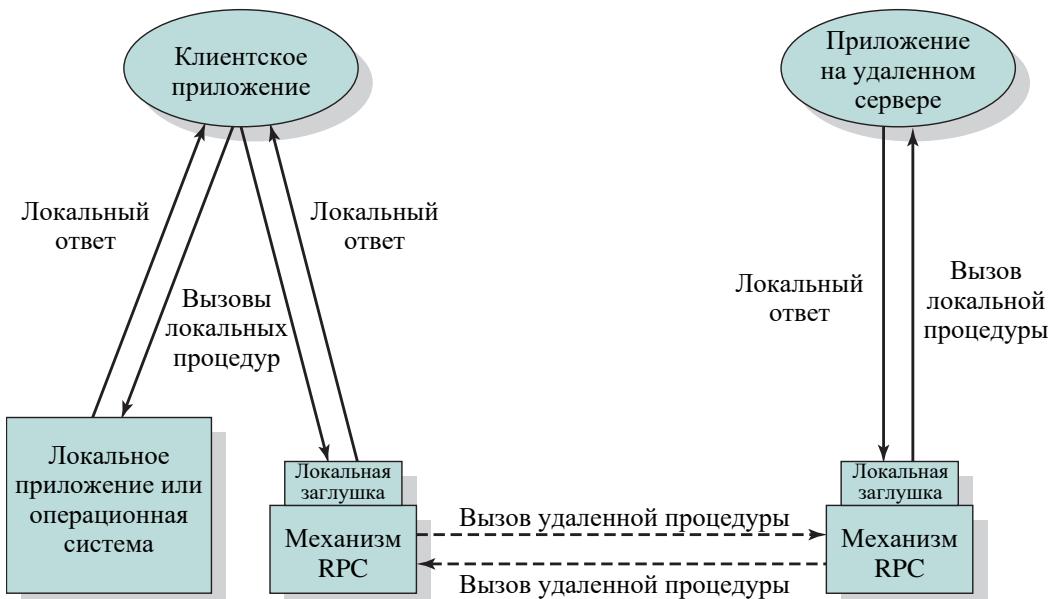


Рис. 18.12. Механизм вызова удаленных процедур

На удаленной машине действует другая процедура-заглушка, связанная с вызываемой процедурой. Как только поступит сообщение, оно будет проанализировано, и в конечном итоге будет сформирован локальный вызов `CALL P(X, Y)`. Эта удаленная процедура вызывается локально, поэтому обычные предположения о месте нахождения параметров, состоянии стека и прочем оказываются такими же, как и при исключительно локальном вызове процедуры. С вызовами удаленных процедур связано несколько вопросов проектирования, которые будут рассмотрены в остальной части этого раздела.

## Передача параметров

В большинстве языков допускается передавать параметры в виде значений (это так называемый вызов по значению) или же в виде указателей на то место, где содержится значение (это так называемый вызов по ссылке). При вызове удаленных процедур вызов по значению осуществляется следующим образом: параметры просто копируются в сообщение и посылаются удаленной системе. Намного труднее реализовать вызов по ссылке, поскольку для этого требуется уникальный уровень системы указатель на каждый объект. Издержки на реализацию такой возможности могут не стоить затраченных усилий.

## Представление параметров

Еще один вопрос связан с порядком представления параметров и результатов в сообщениях. Так, если вызываемая и вызывающая программы написаны на сходных языках программирования для выполнения на однотипных машинах в одной и той же операционной системе, то удовлетворить требование представления может быть не так уж и трудно. А если в этих областях проявляются отличия, то они, вероятнее всего, проявятся и в способах представления чисел и даже текста. Так, если применяется полноценная архитектура связи, то рассматриваемый здесь вопрос решается на представительском уровне. Однако издержки такой архитектуры ведут к проектированию средств вызова удаленных процедур, использующих собственные базовые средства связи в обход большей части упомянутой архитектуры. В этом случае ответственность за преобразование возлагается на средство вызова удаленных процедур (см., например, [88]).

Наилучший подход к решению данного вопроса состоит в предоставлении стандартизированного формата для таких распространенных объектов, как целые числа, числа с плавающей точкой, символы и символьные строки. Тогда вполне возможным становится взаимное преобразование платформенно-ориентированных параметров на любой машине и их стандартизированного представления.

## Привязка к архитектуре “клиент/сервер”

Привязка (*binding*) обозначает порядок установления взаимосвязи между удаленной процедурой и вызывающей программой. Привязка формируется, когда между двумя приложениями устанавливается логическое соединение и они подготавливаются к обмену командами и данными.

Неустойчивая привязка (*nonpersistent binding*) означает, что между двумя процессами устанавливается логическое соединение во время вызова удаленной процедуры и что это соединение разрывается, как только возвращаются значения. Поскольку для соединения требуется поддержание информации о состоянии на обоих концах соединения, то оно потребляет некоторые ресурсы. Неустойчивая привязка призвана сберегать эти

ресурсы. С другой стороны, вследствие издержек на установление соединений неустойчивая привязка оказывается непригодной для тех удаленных процедур, которые часто вызываются одной и той же вызывающей стороной.

При **устойчивой привязке** (persistent binding) соединение, устанавливаемое для вызова удаленной процедуры, поддерживается и после возврата из процедуры, чтобы им можно было воспользоваться для последующих вызовов удаленных процедур. Если же по истечении заданного периода времени в соединении не произойдет никаких действий, оно разрывается. Для приложений, выполняющих много повторяющихся вызовов удаленных процедур, устойчивая привязка поддерживает соединение, которое может быть использовано для последовательного ряда вызовов и возвратов из удаленных процедур.

## СИНХРОННОСТЬ И АСИНХРОННОСТЬ

Принципы синхронных и асинхронных вызовов удаленных процедур аналогичны принципам блокирующего и неблокирующего обмена сообщениями. Традиционный вызов удаленной процедуры является синхронным, требуя, чтобы вызывающий процесс ожидал до тех пор, пока вызываемый процесс не возвратит значение. Следовательно, **синхронный RPC** ведет себя, как вызов подпрограммы.

Синхронный RPC нетрудно понять и запрограммировать, поскольку его поведение вполне предсказуемо. Тем не менее он не позволяет в полной мере воспользоваться параллелизмом, присущим распределенным приложениям. Тем самым ограничивается вид взаимодействия, которое может быть у распределенного приложения, что в конечном счете приводит к снижению производительности.

Ради повышения гибкости были реализованы различные средства **асинхронного RPC**, позволяющие достичь большего параллелизма, сохранив в то же время узнаваемость и простоту вызова RPC [3]. При асинхронном RPC вызывающая сторона не блокируется, а ответы могут быть получены по мере надобности. Благодаря этому клиент может продолжить свое выполнение локально и параллельно с вызовом сервера.

Типичным примером применения асинхронного RPC служит предоставление клиенту возможности вызывать сервер неоднократно, чтобы он мог в какой-то момент направить по конвейеру ряд запросов, причем каждый из них со своим набором данных. Синхронизация клиента и сервера может быть достигнута одним из двух следующих способов.

1. Сначала приложения верхнего уровня — клиент и сервер — инициируют обмен, после чего в конце проверяется, что все запрошенные действия полностью выполнены.
2. Клиент может выдать сначала строку асинхронных RPC, а затем завершающий синхронный RPC. Сервер отреагирует на синхронный RPC лишь по окончании всей работы, запрошенной в предыдущих асинхронных RPC.

В одних схемах асинхронные RPC не требуют от сервера реагировать на них, и поэтому сервер не может отправить ответное сообщение. В других схемах требуется (или разрешается) отвечать на RPC, но вызывающей стороне не обязательно ждать ответа на вызов.

## ОБЪЕКТНО-ОРИЕНТИРОВАННЫЕ МЕХАНИЗМЫ

Объектно-ориентированная технология становится все более и более преобладающей в проектировании операционных систем, поэтому и разработчики систем “клиент/сервер” начали постепенно осваивать данный подход к проектированию. При таком подходе клиен-

ты и серверы организуют обмен сообщениями между объектами. Взаимодействие объектов может опираться на базовую структуру сообщений или вызовов RPC или же быть организованным прямо поверх объектно-ориентированных возможностей операционной системы.

Клиент, которому требуется обслуживание, отсылает запрос брокеру объектных запросов, который действует в качестве каталога всех удаленных служб, доступных в сети (см. рис. 18.10, в). Брокер вызывает подходящий объект и передает ему все необходимые данные. Затем удаленный объект обслуживает запрос, отвечая брокеру, который возвращает ответ клиенту.

Успех объектно-ориентированного подхода зависит от стандартизации объектного механизма. Но, к сожалению, в данной области имеется несколько соперничающих архитектур. Одной из них является объектная модель компонентов (COM — Component Object Model) от корпорации Microsoft, которая служит основанием для технологии связывания и встраивания объектов (OLE — Object Linking and Embedding). С ней соперничает общая архитектура брокера объектных запросов (CORBA — Common Object Request Broker Architecture), разработанная консорциумом Object Management Group и нашедшая широкую поддержку в данной отрасли, в том числе в IBM, Apple, Sun и у многих других поставщиков аппаратных и программных средств вычислительной техники.

## 18.4. КЛАСТЕРЫ

Кластеризация служит альтернативой симметричной многопроцессорной обработке (SMP) как подход, обеспечивающий высокую производительность и степень доступности, что особенно привлекательно для серверных приложений. Кластер можно определить как взаимосвязанную группу целых компьютеров, совместно работающих как единый вычислительный ресурс, который может создать иллюзию единственной машины. Термин *целый компьютер* (whole computer) означает систему, которая может действовать самостоятельно, помимо кластера; в литературе каждый компьютер в кластере обычно называется узлом (node).

В [29] перечислены четыре преимущества, которых позволяет добиться кластеризация. Их можно также рассматривать как цели или технические требования.

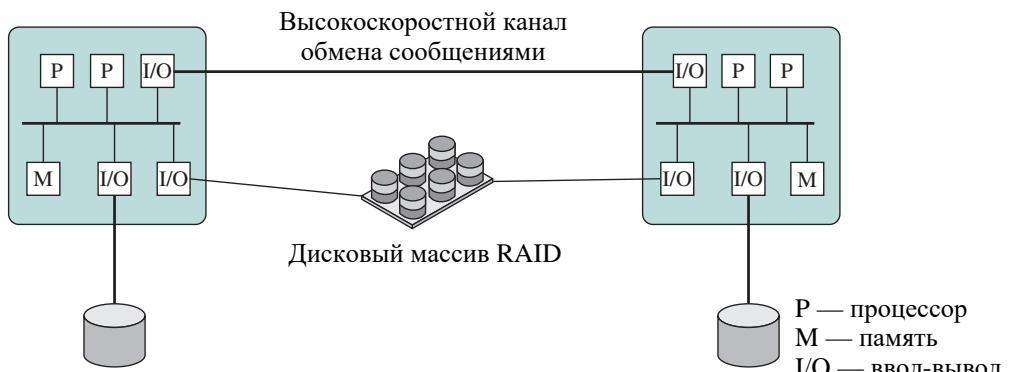
- **Абсолютная масштабируемость.** Имеется возможность создавать крупные кластеры, намного превосходящие по мощности даже самые большие автономные ЭВМ. Кластер может состоять из десятков или даже сотен машин, каждая из которых является многопроцессорной.
- **Постепенно наращиваемая масштабируемость.** Кластер конфигурируется таким образом, чтобы в него можно было вводить новые системы небольшими порциями. Следовательно, пользователь может начать со скромной системы и расширить ее, если потребуется наращивание мощностей, не прибегая к значительной модернизации, при которой небольшая существующая система заменяется более крупной.
- **Высокая степень доступности.** Каждый узел в кластере является автономным компьютером, и поэтому выход из строя одного узла не означает потерю обслуживания. Во многих продуктах отказоустойчивость поддерживается на программном уровне автоматически.
- **Хорошее отношение “цена/производительность”.** Используя типовые строительные блоки, можно собрать кластер вычислительной мощности, не меньшей, чем у одной большой ЭВМ, но по намного меньшей цене.

## Конфигурации кластеров

В литературе кластеры классифицируются самыми разными способами, но, вероятно, самая простая их классификация основывается на следующем критерии: пользуются ли компьютеры в кластере общим доступом к одним и тем же дискам. В качестве примера на рис. 18.13, а показан кластер, взаимодействие между двумя узлами которого осуществляется только через высокоскоростной канал связи, который может быть использован для обмена сообщениями с целью координировать действия в данном кластере. Таким каналом связи может служить локальная сеть, используемая совместно с другими компьютерами, не входящими в состав кластера, или же специально выделенное для взаимодействия узлов в кластере средство связи.



а) Автономный сервер без общего дискового массива



б) Совместно используемый дисковый массив

P — процессор  
M — память  
I/O — ввод-вывод

**Рис. 18.13.** Конфигурации кластеров

В последнем случае у одного или нескольких компьютеров в кластере будет канал связи с локальной или глобальной сетью для установления соединения между серверным кластером и удаленными клиентскими системами. Следует иметь в виду, что каждый компьютер, показанный на рис. 18.13, а, является многопроцессорным. Хотя это и не обязательное условие, оно способствует повышению производительности и степени доступности.

В простой классификации кластеров на рис. 18.13 показан и другой вариант — кластер с совместно используемым дисковым массивом. В этом случае обычно имеется канал свя-

зи для обмена сообщениями между узлами кластера, а кроме того — дисковая подсистема, непосредственно связанная с некоторыми компьютерами в кластере. Как показано на рис. 18.13, б, типичной дисковой подсистемой служит дисковый массив RAID. Поскольку в кластерах обычно применяются дисковые массивы RAID или некоторые другие технологии, реализующие дисковые массивы с избыточностью, то высокая степень доступности в них достигается благодаря тому, что функционирование нескольких компьютеров в кластере не нарушается наличием общего дискового массива как единственной точки отказа.

Более ясное представление о разнообразии подходов к кластеризации может дать рассмотрение их функциональных альтернатив. В официальной документации от компании Hewlett Packard [105] приведена удобная классификация методов кластеризации по рассматриваемым далее функциональным признакам (табл. 18.2).

**Таблица 18.2. Методы кластеризации, их преимущества и недостатки**

Метод кластеризации	Описание	Преимущества	Недостатки
<b>Пассивный резерв</b>	Вторичный сервер вступает в действие, если выходит из строя первичный сервер	Простота реализации	Высокая стоимость из-за того, что вторичный сервер недоступен для решения других задач
<b>Активный вторичный сервер</b>	Вторичный сервер также используется для решения задач обработки данных	Снижение затрат за счет того, что вторичный сервер также используется для обработки данных	Повышенная сложность
<b>Отдельные серверы</b>	У отдельных серверов имеются свои диски, а данные непрерывно копируются с первичного сервера на вторичный	Высокая степень доступности	Большие сетевые и серверные издержки из-за операций копирования
<b>Серверы, подключаемые к дискам</b>	Серверы подключены кабелями к одним и тем же дискам, но у каждого сервера имеются также собственные диски. Если один из серверов выйдет из строя, его диски перейдут в распоряжение другого сервера	Сокращение издержек на сеть и сервер за счет исключения операций копирования	Обычно требуется зеркальное отображение дисков или дисковый массив RAID для компенсации риска отказа дисков
<b>Серверы с общими дисками</b>	Несколько серверов одновременно разделяют общий доступ к дискам	Малые сетевые и серверные издержки. Снижение риска простоя из-за отказа дисков	Требуется программное обеспечение диспетчера блокировок. Обычно применяется вместе с зеркальным отображением дисков и дисковыми массивами RAID

Распространенный ранее метод под названием **пассивный резерв** состоит в том, что вся нагрузка при обработке данных ложится на один компьютер, тогда как другой компьютер остается неактивным, находясь в резерве, и вступает в действие только в случае отказа первого (основного) компьютера. Для координации работы обоих компьютеров активная (или первичная) система периодически отсылает специальное сообщение резервной (вторичной) системе. Если эти сообщения перестают поступать, резервный компьютер считает, что основной компьютер вышел из строя, и вступает в действие. При таком подходе повышается степень доступности, но не производительность. Если обе системы обмениваются только сообщениями об активности и не пользуются общими дисками, то вторичная система предоставляет функциональный резерв, но не имеет доступа к базам данных, находящимся под управлением первичной системы.

Пассивный резерв обычно не считается кластером. Термин “кластер” зарезервирован для обозначения множества взаимосвязанных компьютеров, активно выполняющих обработку данных и в то же время поддерживающих образ единой системы для внешнего мира. Такая конфигурация зачастую обозначается термином **активный вторичный сервер**. При этом можно выделить три категории методов кластеризации: отдельные серверы, без разделения ресурсов и с общей памятью.

При другом подходе к кластеризации каждый компьютер является **отдельным сервером со своими дисками**, но без общих дисков, разделяемых с другими системами (см. рис. 18.13, а). Такая конфигурация обеспечивает высокую производительность и степень доступности, хотя в данном случае требуется определенное программное обеспечение управления или планирования, чтобы распределять поступающие от клиентов запросы по серверам и тем самым уравновешивать нагрузку и достигать высокой эффективности их использования. Было бы также желательно, чтобы кластер был отказоустойчивым. Это означает, что если один компьютер выйдет из строя при выполнении приложения, другой компьютер в кластере может подхватить управление приложением и завершить его. Но для этого данные должны постоянно копироваться из одной системы в другую, чтобы в каждой системе были доступны текущие данные из других систем. Дополнительные издержки на такой обмен данными обеспечивают высокую степень доступности, но за счет снижения производительности.

Чтобы сократить издержки на обмен данными, большинство современных кластеров состоят из серверов, подключаемых к общим дискам (см. рис. 18.13, б). Одна из разновидностей такого подхода называется **без разделения ресурсов**, когда общие диски разбиваются на тома и каждый том принадлежит одному компьютеру. Если выйдет из строя один компьютер, кластер придется переконфигурировать таким образом, чтобы тома отказавшего компьютера стали принадлежать какому-то другому компьютеру.

Вполне возможна и такая конфигурация, когда несколько компьютеров одновременно пользуются одними и теми же дисками (это так называемый метод **общего диска**). При этом каждый компьютер имеет доступ ко всем томам на всех дисках. При таком подходе потребуется некоторое блокирующее средство, чтобы гарантировать одновременный доступ к данным только со стороны одного компьютера.

## Вопросы проектирования операционных систем

Для полноценной эксплуатации конфигурации кластерного оборудования требуются некоторые усовершенствования операционных систем для отдельных машин.

## Управление обработкой отказов

Порядок управления обработкой отказов в кластере зависит от применяемого метода кластеризации (см. табл. 18.2). В общем случае для обработки отказов могут быть выбраны два подхода — в виде высоконадежных или отказоустойчивых кластеров. В частности, высоконадежный кластер обеспечивает очень высокую вероятность нормальной работы всех имеющихся ресурсов. В случае отказа (например, если выйдет из строя узел или потеряется дисковый том) запросы, находящиеся в процессе обработки, будут потеряны. Если любой потерянный запрос повторится, он будет обработан другим компьютером кластера. Тем не менее кластерная операционная система не дает никаких гарантий относительно состояния частично выполненных транзакций, поэтому они должны обрабатываться на уровне приложений.

Отказоустойчивый кластер гарантирует постоянную доступность всех ресурсов. Это достигается благодаря применению избыточных совместно используемых дисков, а также механизмов отката незафиксированных транзакций и фиксации завершенных транзакций.

Функция переноса приложений и данных из отказавшей системы в альтернативную систему в кластере называется **преодолением отказа** (failover). С ней связана другая функция восстановления приложений и информационных ресурсов в исходной системе после устранения в ней неисправностей. Эта функция называется **восстановлением после отказа** (failback). Восстановление после отказа может быть автоматизировано, но это желательно лишь в том случае, когда неисправность действительно устранена и ее повторение маловероятно. В противном случае автоматическое восстановление может привести к постоянному переносу отказавших ресурсов с одних компьютеров на другие и обратно, что приведет к снижению производительности и проблемам с восстановлением.

## Распределение нагрузки

Кластер должен обладать эффективными возможностями распределять нагрузку среди имеющихся компьютеров, включая требование постепенно наращиваемой масштабируемости. Когда в кластер вводится новый компьютер, средство распределения нагрузки должно автоматически включать этот компьютер в приложения, предназначенные для планирования. Механизмы промежуточного программного обеспечения должны принимать во внимание, что службы могут располагаться в разных узлах кластера и переноситься из одного узла в другой.

## Распараллеливание вычислений

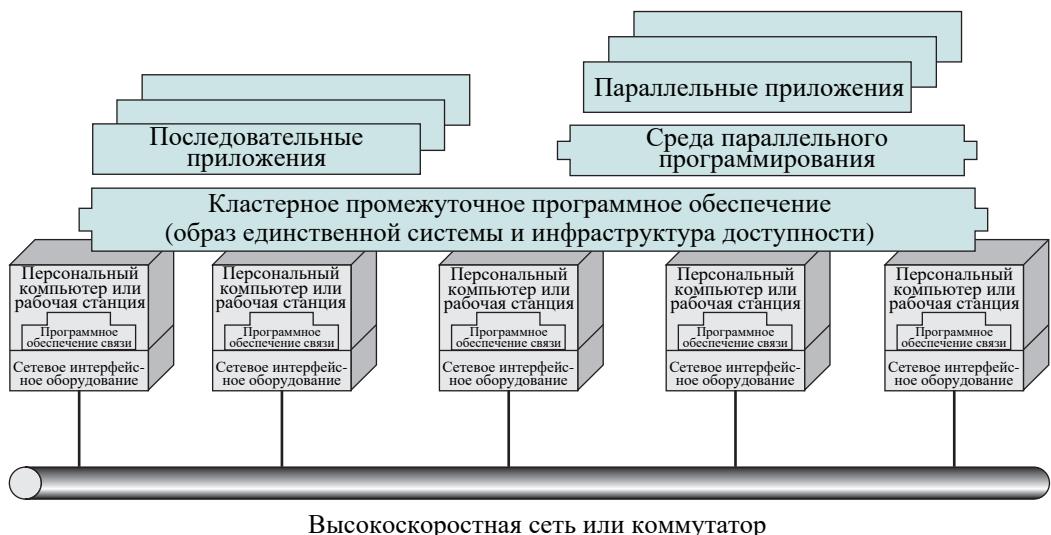
В некоторых случаях для эффективного применения кластера требуется параллельное выполнение программного обеспечения одного приложения. В [124] перечислены три общих подхода к решению этой задачи.

- **Распараллеливающий компилятор.** Такой компилятор во время компиляции определяет те части приложения, которые должны выполняться параллельно. Затем они распределяются и назначаются для выполнения разным компьютерам кластера. Производительность при этом зависит от характера решаемой задачи и качества самого компилятора.
- **Распараллеливание приложения.** При таком подходе программист пишет приложение с учетом его выполнения в кластере и пользуется обменом сообщениями для перемещения по мере надобности данных между узлами кластера. Хотя на программиста накладывается большое бремя, это может оказаться наилучшим подходом к эксплуатации кластеров для выполнения некоторых приложений.

- Параметрические вычисления.** Такой подход применяется в том случае, если в основу приложения положен алгоритм или программа, которая может быть выполнена многоократно, и каждый раз с разными начальными условиями или с разными параметрами. Хорошим примером может служить имитационная модель, которая выполняет самые разные сценарии, а затем подводит статистические итоги по полученным результатам. Чтобы сделать такой подход эффективным, требуются инструментальные средства параметрической обработки, с помощью которых можно организовывать, выполнять и управлять заданиями в запланированном порядке.

## Архитектура кластерных вычислительных систем

Типичная архитектура кластерной вычислительной системы приведена на рис. 18.14. Отдельные компьютеры подключены к некоторой высокоскоростной локальной сети или коммутаторному оборудованию, причем каждый компьютер способен действовать в такой архитектуре независимо от других компьютеров. Кроме того, на каждом компьютере устанавливается программное обеспечение нормальной работы кластера на уровне промежуточного программного обеспечения. Кластерное промежуточное программное обеспечение предоставляет пользователю унифицированный образ системы, называемый **образом единственной системы** (single-system image). Кроме того, промежуточное программное обеспечение может отвечать за высокую степень доступности, распределяя нагрузку и реагируя на отказы в отдельных компонентах. В [113] приведен следующий перечень желательных кластерных услуг и функций промежуточного программного обеспечения.



**Рис. 18.14.** Архитектура кластерной вычислительной системы

- **Единая точка входа.** Пользователь регистрируется в кластере, а не на отдельном компьютере.
- **Единая иерархия файлов.** Пользователю доступна единая иерархия каталогов с файлами, производная от одного и того же корневого каталога.
- **Единая точка управления.** Для управления и контроля над кластером имеется отдельный узел по умолчанию.
- **Единая организация виртуальной сети.** Из любого узла можно получить доступ к любой другой точке в кластере, несмотря на то что конкретная конфигурация кластера может состоять из нескольких взаимосвязанных сетей. При этом действует единая виртуальная сеть.
- **Единое пространство памяти.** Распределенная совместно используемая память позволяет программам совместно пользоваться общими переменными.
- **Единая система управления заданиями.** С помощью планировщика заданий кластера пользователь может передать задание на выполнение, не указывая конкретный компьютер, на котором оно должно быть выполнено.
- **Единый пользовательский интерфейс.** Общий графический интерфейс для всех пользователей независимо от рабочей станции, с которой они входят в кластер.
- **Единое пространство ввода-вывода.** Из любого узла может быть доступно любое периферийное или дисковое устройство ввода-вывода безотносительно к его физическому расположению.
- **Единое пространство процессов.** Для этого применяется единообразная схема идентификации процессов. Процесс в любом узле может создать любой другой процесс в удаленном узле или взаимодействовать с ним.
- **Создание контрольных точек.** Такая функция периодически сохраняет состояние процесса и промежуточные результаты вычислений, чтобы сделать возможным восстановление после отказа.
- **Перенос процессов.** Эта функция позволяет распределить нагрузку.

Четыре последних пункта в приведенном выше перечне способствуют повышению степени доступности кластера, а остальные пункты имеют отношение к предоставлению единого образа системы. Возвращаясь к рис. 18.14, следует заметить, что в состав кластера входят также программные инструментальные средства, позволяющие эффективно выполнять программы, допускающие параллельное их выполнение.

## Кластеры в сравнении с симметричной многопроцессорной обработкой

Кластеры и симметричные и многопроцессорные системы предоставляют конфигурацию с несколькими процессорами для поддержки приложений, пользующихся повышенным спросом. Оба сравниваемых здесь решения коммерчески доступны, хотя симметричная многопроцессорная обработка (SMP) появилась намного раньше.

Самая сильная сторона метода SMP состоит в том, что управлять симметричной многопроцессорной обработкой и конфигурировать ее проще, чем делать то же с кластером. SMP намного ближе к первоначальной модели однопроцессорной обработки, для которой написаны практически все приложения. Принципиальным изменением, требующим-

ся для перехода от однопроцессорной к симметричной многопроцессорной обработке, является функционирование планировщика. Еще одно преимущество симметричной многопроцессорной обработки заключается в том, что для нее обычно требуется меньше физического пространства и вычислительных мощностей по сравнению с кластером. Последнее преимущество SMP состоит в том, что программные продукты SMP вполне упрочились и устоялись.

Тем не менее в долгосрочной перспективе преимущества кластерного подхода, вероятнее всего, приведут к преобладанию кластеров на рынке высокопроизводительных серверов. Ведь кластеры намного превосходят системы SMP в отношении постепенно наращиваемой и абсолютной масштабируемости, а также в отношении степени доступности, поскольку все компоненты системы могут быть без особого труда сделаны в высшей степени резервируемыми.

## 18.5. КЛАСТЕРНЫЙ СЕРВЕР WINDOWS

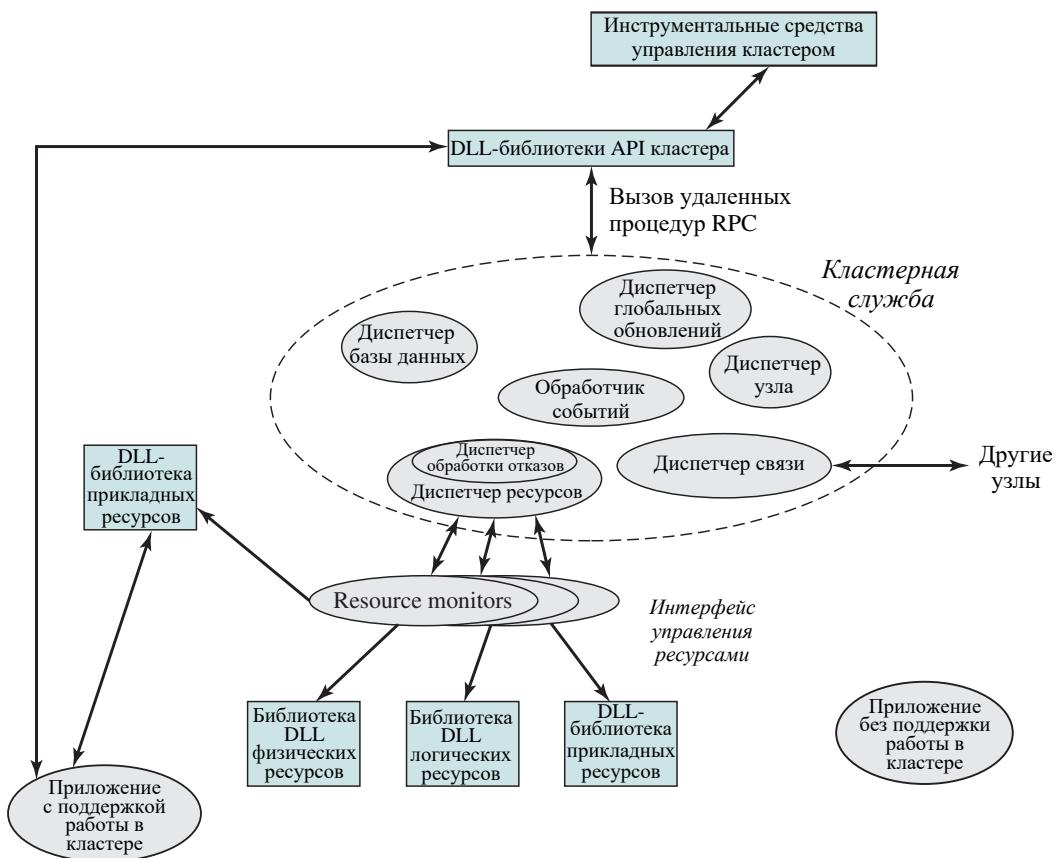
Для отказоустойчивой кластеризации в Windows служит кластер без разделения ресурсов, в котором каждый дисковый том и другие ресурсы одновременно принадлежат одной системе. В архитектуре кластеров Windows используются следующие концепции.

- **Кластерная служба.** Это комплект программного обеспечения в каждом узле, управляющий всеми характерными для кластера действиями.
- **Ресурс.** Это элемент, управляемый службой кластеров. Все ресурсы являются объектами, представляющими конкретные ресурсы в системе, включая такие аппаратные устройства, как накопители на дисках и сетевые адаптеры, а также логические элементы вроде логических дисковых томов, TCP/IP-адресов, целых приложений и баз данных.
- **Оперативная доступность.** Ресурс называется оперативно доступным в узле, когда он предоставляет услугу в данном конкретном узле.
- **Группа.** Это совокупность ресурсов, управляемых как единое целое. Обычно всем элементам, составляющим группу, требуется выполнять какое-нибудь конкретное приложение, а клиентской системе — подключаться к услуге, предоставляемой данным приложением.

Концепция *группы* имеет особое значение. Группа объединяет ресурсы в более крупные единицы, которыми легко управлять как для обработки отказов, так и для распределения нагрузки. Операции, выполняемые над группой (например, ее перенос в другой узел), автоматически воздействуют на все ресурсы данной группы. Ресурсы реализуются в виде динамически подключаемых библиотек (DLL) и управляются монитором ресурсов, который взаимодействует с кластерной службой через вызовы удаленных процедур и реагирует на команды кластерной службы для конфигурирования и перемещения групп ресурсов.

Компоненты кластеризации в Windows и их взаимосвязи в единой системе кластера приведены на рис. 18.15. В частности, *диспетчер узла* (node manager) отвечает за поддержание членства данного узла в кластере. Он периодически отсылает тактовые сообщения диспетчерам других узлов кластера. Когда диспетчер одного из узлов обнаруживает потерю тактовых сообщений от другого узла кластера, он рассыпает сообщение всем узлам кластера, заставляя всех членов узла обменяться сообщениями, чтобы сверить свои

представления о текущем членстве в кластере. Диспетчер узла, который не реагирует на сообщения, исключается из кластера, а его активные группы переносятся в один или несколько других активных узлов кластера.



**Рис. 18.15.** Блок-схема кластерного сервера в Windows

**Диспетчер базы данных конфигурации** поддерживает базу данных с информацией о конфигурации кластера, содержащую сведения о ресурсах и группах, а также о принадлежности групп узлам. Диспетчеры баз данных в каждом узле кластера совместно поддерживают согласованное представление конфигурационной информации. А программное обеспечение отказоустойчивых транзакций служит для гарантии согласованного и правильного внесения изменений в конфигурации всего кластера.

В диспетчере ресурсов и диспетчере обработки отказов принимаются все решения, касающиеся групп ресурсов, а также инициируются соответствующие действия на подобие запуска, сброса и обеспечения отказоустойчивости. Когда требуется обеспечить отказоустойчивость, диспетчеры обработки отказов в активном узле совместно согласуют распределение групп ресурсов из отказавшей системы в оставшихся активными системах. Когда система перезапускается после отказа, в диспетчере обработки отказов может быть принято решение переместить некоторые группы обратно в данную систему. В частности, любая группа может быть сконфигурирована с предпочтительным ее вла-

дельцем. Если этот владелец выйдет из строя и затем перезапустится, группа переместится обратно в узел при выполнении операции отката.

**Обработчик событий** соединяет все компоненты кластерной службы, выполняет общие операции обработки и управляет инициализацией кластерной службы. Диспетчер связи организует обмен сообщениями со всеми остальными узлами кластера, а диспетчер глобальных обновлений предоставляет услуги, которыми пользуются другие компоненты кластерной службы.

Корпорация Microsoft продолжает поставлять свои кластерные продукты на рынок, но ею также были разработаны решения для виртуализации как часть серверной операционной системы Windows Server 2008 R2, основанные на эффективном динамическом переносе виртуальных машин из одного гипервизора в другой на разных вычислительных системах. Динамический перенос виртуальных машин по сравнению с кластерным подходом дает немало выгод новым приложениям, в том числе упрощенное управление и большую гибкость.

## 18.6. КЛАСТЕРЫ BEOWULF И LINUX

Проект Beowulf был начат в 1994 году при содействии и в рамках проекта HPCC (NASA High Performance Computing and Communications — Высокопроизводительные вычислительные системы и средства связи в НАСА). Его целью было исследование истинного потенциала кластеризированных персональных компьютеров для выполнения важных вычислительных задач за пределами возможностей современных рабочих станций при минимальных затратах. Принятый в Beowulf подход широко реализован и, вероятно, является самой важной кластерной технологией из всех ныне доступных.

### ФУНКЦИОНАЛЬНЫЕ СРЕДСТВА Beowulf

Ниже перечислены основные возможности Beowulf [203].

- Компоненты широко представлены на массовом рынке.
- Специально выделенные процессоры вместо перехвата свободных циклов процессора на простаивающих рабочих станциях.
- Специально выделенная частная сеть (локальная, глобальная или обе).
- Отсутствие нестандартных компонентов.
- Простота тиражирования от многих поставщиков.
- Масштабируемый ввод-вывод.
- База свободно доступного программного обеспечения.
- Применение свободно доступных инструментальных средств распределенных вычислений, требующих минимальных изменений.
- Возвращение проектирования и усовершенствований сообществу разработчиков.

Несмотря на то что программное обеспечение Beowulf было реализовано на самых разных платформах, наиболее очевидным вариантом выбора платформы для него является Linux, и поэтому в большинстве реализаций Beowulf применяется кластер рабочих станций и/или персональных компьютеров, работающих под управлением Linux. Типичная конфигурация Beowulf приведена на рис. 18.16; здесь кластер состоит из ряда рабо-

чих станций, работающих под управлением операционной системы Linux, хотя и вполне возможно, что на разных аппаратных платформах. Вторичная память на каждой рабочей станции может быть сделана доступной для распределенных операций (обмена файлами, организации виртуальной памяти и прочих примеров применения). Узлы кластера (системы Linux) связаны между собой с помощью распространенного метода подключения к сети (как правило, Ethernet). Подключение к сети Ethernet может поддерживаться в форме единственного коммутатора Ethernet или ряда взаимосвязанных коммутаторов. Широко используемые сетевые адAPTERы Ethernet действуют на стандартных скоростях передачи данных 10 Мбит/с, 100 Мбит/с и 1 Гбит/с.

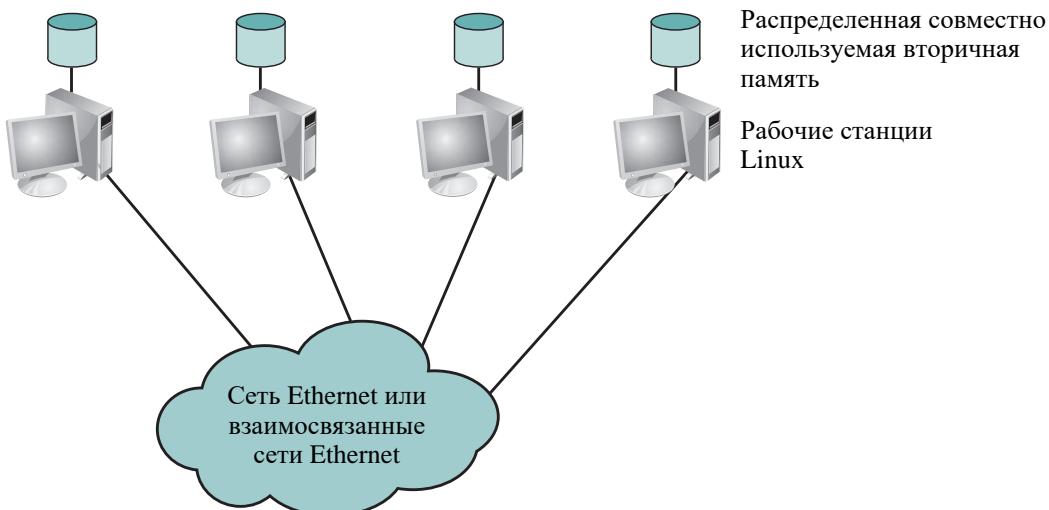


Рис. 18.16. Типичная конфигурация Beowulf

## Программное обеспечение Beowulf

Программная среда Beowulf реализуется в виде дополнения к коммерчески доступным или бесплатным базовым дистрибутивам Linux. Основным источником программного обеспечения с открытым исходным кодом служит веб-сайт Beowulf, расположенный по адресу [www.beowulf.org](http://www.beowulf.org), но свободно доступные версии инструментальных средств и утилит Beowulf предоставляются целым рядом других организаций.

В каждом узле кластера Beowulf выполняется своя копия ядра Linux, и поэтому он может функционировать как автономная система Linux. Для поддержки понятия кластера Beowulf были сделаны расширения ядра Linux, позволяющие отдельным узлам принимать участие в целом ряде глобальных пространств имен. Ниже приведены некоторые примеры системного программного обеспечения Beowulf.

- **Распределенное пространство процессов Beowulf (BPROC).** Этот пакет позволяет охватить пространством идентификаторов процессов несколько узлов в кластерной среде, а также предоставляет механизмы для запуска процессов в других узлах. Цель данного пакета — предоставить основные элементы, требующиеся для единого образа системы в кластере Beowulf. В пакете BPROC предоставляется механизм для запуска процессов в удаленных узлах даже без регистрации и для того, чтобы сделать все удаленные процессы видимыми в таблице процессов внешнего узла кластера.

- **Связывание каналов Ethernet в Beowulf.** Это механизм, соединяющий несколько недорогих сетей в единую логическую сеть с большей пропускной способностью. Чтобы воспользоваться единым сетевым интерфейсом, достаточно выполнить простую вычислительную задачу распределения пакетов по очередям на передачу данных, имеющимся в устройствах. Такой подход позволяет распределить нагрузку среди многих подключенных к сетям Ethernet рабочих станций Linux.
- **Pvmsync.** Это среда программирования, предоставляющая механизмы синхронизации и объекты общих данных для процессов в кластере Beowulf.
- **EnFusion.** Состоит из набора инструментальных средств для выполнения параметрических вычислений. Параметрические вычисления включают в себя выполнение программы как большого количества заданий, причем каждое из них с разными параметрами или начальными условиями. EnFusion эмулирует ряд пользователей-роботов на единой машине корневого узла, на которой каждый из них регистрирует одного из многих клиентов, образующих кластер. Каждое задание подготавливается к выполнению по особому сценарию, программируемому подходящим рядом начальных условий [124].

## 18.7. Резюме

Вычисления “клиент/сервер” служат ключом к пониманию истинного потенциала информационных систем и сетей, раскрыв который, можно значительно повысить производительность труда в организациях. При вычислениях “клиент/сервер” приложения распределяются среди пользователей, работающих на однопользовательских рабочих станциях и персональных компьютерах. В то же время ресурсы, которые могут и должны быть общими, поддерживаются в серверных системах, доступных всем клиентам. Таким образом, архитектура “клиент/сервер” сочетает в себе децентрализованные вычисления с централизованными.

Как правило, клиентская система предоставляет графический пользовательский интерфейс (GUI), дающий пользователю возможность относительно просто эксплуатировать самые разные приложения, прилагая минимум усилий для обучения. На серверах поддерживаются такие общие утилиты, как системы управления базами данных (СУБД). А отдельное приложение разделяется между клиентом и сервером таким образом, чтобы оптимизировать простоту использования и производительность.

Ключевым механизмом, который требуется в любой распределенной системе, является взаимодействие процессов. Обычно для этой цели используются два механизма. Механизм обмена сообщениями обобщает их применение в единой системе — в обмене сообщениями “клиент/сервер” применяются те же соглашения и правила, что и в единой системе. Другим механизмом является вызов удаленных процедур. С его помощью две программы на разных машинах вступают во взаимодействие, используя синтаксис и семантику вызовов и возвратов. При этом вызываемая и вызывающая программы ведут себя так, как будто они выполняются на одной и той же машине.

Кластер — это группа соединенных вместе целых компьютеров, действующих как единообразный вычислительный ресурс, создавая иллюзию одной машины. Термином *целый компьютер* обозначается система, способная работать самостоятельно и отдельно от кластера.

## 18.8. Ключевые термины, контрольные вопросы и задачи

### Ключевые термины

Восстановление после отказа	Клиент	Сервер
Вызов удаленных процедур (RPC)	Преодоление отказа	Согласованность файловых кешей
Графический пользовательский интерфейс (GUI)	Проект Beowulf	Сообщение
Интерфейс прикладного программирования интерфейс (API)	Промежуточное программное обеспечение	Толстый клиент
Кластер	Распределенный обмен сообщениями	Тонкий клиент

### Контрольные вопросы

- 18.1. Что такое вычисления “клиент/сервер”?
- 18.2. Чем вычисления “клиент/сервер” отличаются от любой другой формы распределенной обработки данных?
- 18.3. Какова роль архитектуры связи (например, набора протоколов TCP/IP) в среде “клиент/сервер”?
- 18.4. Приведите довод в пользу расположения приложений на стороне клиента, на стороне сервера или их разделения между клиентом и сервером.
- 18.5. Что собой представляют тонкий и толстый клиенты и каковы их принципиальные различия?
- 18.6. Приведите аргументы за и против стратегий тонкого и толстого клиентов.
- 18.7. Приведите довод в пользу трехуровневой архитектуры “клиент/сервер”.
- 18.8. Что такое промежуточное программное обеспечение?
- 18.9. Если имеются такие стандарты, как TCP/IP, то зачем нужно промежуточное программное обеспечение?
- 18.10. Перечислите некоторые преимущества и недостатки блокирующих и неблокирующих примитивов для обмена сообщениями.
- 18.11. Перечислите некоторые преимущества и недостатки устойчивой и неустойчивой привязки для вызовов удаленных процедур.
- 18.12. Перечислите некоторые преимущества и недостатки синхронных и асинхронных вызовов удаленных процедур.
- 18.13. Перечислите и кратко опишите четыре разных метода кластеризации.

### Задачи

- 18.1. Пусть  $\alpha$  — процентная доля кода программы, который может одновременно выполняться на  $n$  компьютерах в кластере, причем с разным набором параметров или начальных условий на каждом из них. Допустим, что оставшийся код программы должен быть выполнен последовательно на одном процессоре, а производительность каждого компьютера составляет  $x$  MIPS (т.е. миллионов операций в секунду).

- a. Выведите выражение для эффективной производительности в MIPS (через  $n$ ,  $\alpha$  и  $x$ ), когда система используется исключительно для выполнения данной программы.
- б. Если  $n = 16$ , а  $x = 4$  MIPS, то каково значение  $\alpha$ , которое позволит добиться производительности 40 MIPS?
- 18.2.** Прикладная программа выполняется в кластере, состоящем из девяти машин. Выполнение эталонной программы в этом кластере занимает время  $T$ . Кроме того, 25% времени  $T$  прикладная программа выполняется одновременно на всех девяти компьютерах, а остальное время — на одном компьютере.
- а. Рассчитайте эффективное повышение производительности при указанных выше условиях по сравнению с выполнением прикладной программы на одном компьютере. Кроме того, рассчитайте долю в процентах распараллеливаемого кода упомянутой выше программы, написанного и скомпилированного специально для применения в кластерном режиме.
- б. Допустим, вместо 9 компьютеров для выполнения распараллеливаемой части кода программы мы можем использовать 18 компьютеров. Рассчитайте повышение производительности, которое при этом достигается.
- 18.3.** Приведенная ниже программа на языке FORTRAN выполняется на компьютере, а ее параллельный вариант — в кластере, состоящем из 32 компьютеров.

```
L1:      DO 10 I = 1,1024
L2:      SUM(I) = 0
L3:      DO 20 J = 1, I
L4: 20   SUM(I) = SUM(I) + I
L5: 10   CONTINUE
```

Допустим, что для выполнения каждой из строк кода 2 и 4 требуется два машинных цикла, включая все операции процессора и обращения к памяти. Издержками на выполнение операторов цикла в строках кода 1, 3 и 5, а также на обращение к системе и разрешение конфликтов за ресурсы можно пренебречь.

- а. Каково общее время выполнения приведенной выше программы (в машинных тиках) на одном компьютере?
- б. Распределите шаги цикла I среди 32 компьютеров следующим образом: на первом компьютере выполняются 32 шага цикла (I принимает значения от 1 до 32), на втором компьютере — следующие 32 шага цикла и т.д. Каково время выполнения и коэффициент повышения производительности по сравнению с условиями, заданными в п. а)? (Имейте в виду, что вычислительная нагрузка, обусловленная циклом J, между компьютерами не балансируется.)
- в. Поясните, как модифицировать распараллеливание, чтобы упростить распределение всей вычислительной нагрузки среди 32 компьютеров для сбалансированного параллельного выполнения приведенной выше программы. Сбалансированная нагрузка означает одинаковое количество операций сложения, выполняемых каждым компьютером с учетом обоих циклов.
- г. Каково минимальное время выполнения, получающееся в результате выполнения приведенной выше программы на 32 компьютерах? Каково получающееся в итоге повышение производительности по сравнению с одним компьютером?

## ГЛАВА

# УПРАВЛЕНИЕ РАСПРЕДЕЛЕННЫМИ ПРОЦЕССАМИ

В ЭТОЙ ГЛАВЕ...

## 19.1. Перенос процессов

- Побудительные причины
- Механизмы переноса процессов
  - Инициирование переноса процессов
  - Что именно переносится?
  - Сообщения и сигналы
  - Сценарий переноса
- Согласования переноса процессов
- Выселение
- Вытесняющие переносы в сравнении с невытесняющими

## 19.2. Распределенные глобальные состояния

- Глобальные состояния и распределенные моментальные снимки
- Алгоритм распределенных моментальных снимков

## 19.3. Распределенное взаимное исключение

- Принципы распределенного взаимного исключения
- Упорядочение событий в распределенной системе
- Распределенная очередь
  - Первая версия алгоритма
  - Вторая версия алгоритма
- Метод передачи эстафеты

## 19.4. Распределенная взаимоблокировка

- Взаимоблокировка при распределении ресурсов
  - Предотвращение взаимоблокировки
  - Исключение взаимоблокировки
  - Обнаружение взаимоблокировки
- Взаимоблокировка при обмене сообщениями
  - Взаимное ожидание
  - Недоступность буферов сообщений

## 19.5. Резюме

## 19.6. Ключевые термины, контрольные вопросы и задачи

## УЧЕБНЫЕ ЦЕЛИ

- Пояснить перенос процессов.
- Понимать концепцию распределенных глобальных состояний.
- Анализировать распределенные алгоритмы взаимоисключений.
- Анализировать распределенные алгоритмы взаимоблокировок.

В этой главе исследуются основные механизмы, применяемые в распределенных операционных системах. Сначала в ней описывается перенос процессов, т.е. перемещение активного процесса с одной машины на другую. Затем в этой главе обсуждается вопрос координирования действий процессов в разных системах, когда каждый из них управляется локальными системными часами и когда возникает задержка в обмене информацией. И наконец, в этой главе рассматриваются два главных затруднения, возникающих при управлении распределенными процессами: взаимное исключение и взаимоблокировка.

### 19.1. ПЕРЕНОС ПРОЦЕССОВ

Перенос процессов означает перемещение значительной части состояния процесса с одного компьютера на другой для выполнения данного процесса на целевой машине. Интерес к данному понятию вырос из исследования методов распределения нагрузки среди многих подключенных к сети систем, хотя сегодня его применение выходит за рамки одной этой области.

В прошлом материал лишь немногих статей о распределении нагрузки основывался на практических реализациях переноса процессов, включавших в себя возможность вытеснить процесс на одной машине и повторно активизировать его в дальнейшем на другой машине. Как показал опыт, вытесняющий перенос процессов вполне возможен, хотя и с большими затратами и сложностями, чем предполагалось первоначально [11]. Эти затраты привели некоторых обозревателей к выводу, что перенос процессов непрактичен, но такие оценки оказались слишком пессимистичными. Новые реализации, в том числе и в коммерчески доступных программных продуктах, подогрели непрерывающийся интерес к переносу процессов и привели к новым разработкам в данной области. В этом разделе дается общий обзор переноса процессов.

#### Побудительные причины

Перенос процессов желателен в распределенных системах по целому ряду причин [123, 237], включая следующие.

- **Разделение нагрузки.** Перемещая процессы из сильно нагруженных в слабо нагруженные системы, можно равномерно распределить нагрузку и тем самым повысить общую производительность. Как показывают опытные данные, в результате такого переноса процессов можно добиться существенного повышения производительности [34, 147]. Тем не менее, разрабатывая алгоритмы распределения нагрузки, следует принять определенные меры предосторожности. Как указано в [72],

чем больше распределенным системам приходится обмениваться данными для равномерного распределения нагрузки, чем ниже становится производительность. Обсуждение этого вопроса со ссылками на другие исследования можно найти в [75].

- **Производительность связи.** Интенсивно взаимодействующие процессы могут быть перемещены в один и тот же узел, чтобы сократить затраты на обмен данными во время их взаимодействия. Кроме того, когда процесс производит анализ данных в некотором файле или ряде файлов, больших по размеру, чем сам процесс, то может быть выгоднее переместить процесс к данным, а не наоборот.
- **Доступность.** Длительные процессы, возможно, придется переместить, чтобы убедиться их от сбоев, которые можно предвидеть заранее, или от заранее запланированных простоев. Если операционная система предоставляет такое уведомление, то можно перенести процесс, который требуется продолжить, в другую систему или же убедиться, что он может быть перезапущен в текущей системе некоторое время спустя.
- **Реализация специальных возможностей.** Процесс можно переместить, чтобы выгодно воспользоваться особыми аппаратными или программными возможностями в конкретном узле.

## Механизмы переноса процессов

При проектировании средства переноса процессов приходится решать целый ряд вопросов, в том числе следующие.

- Кто инициирует перенос процессов?
- Какая часть процесса переносится?
- Что происходит с ожидающими своей очереди сообщениями и сигналами?

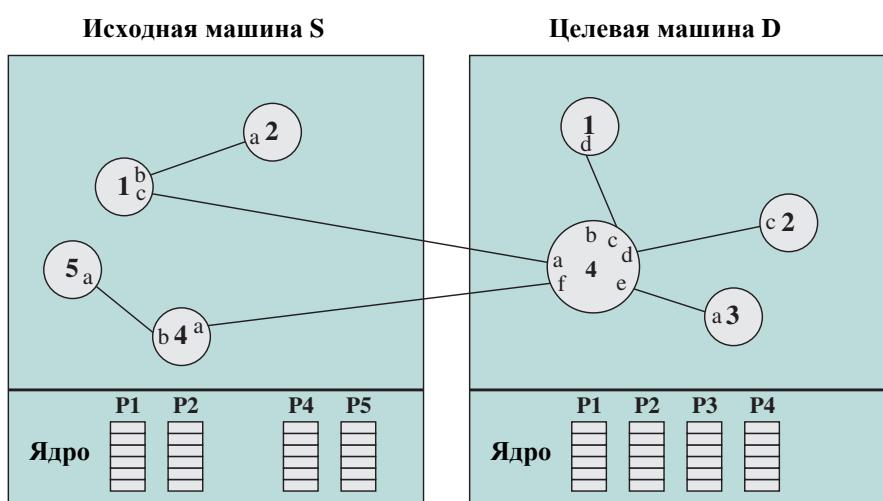
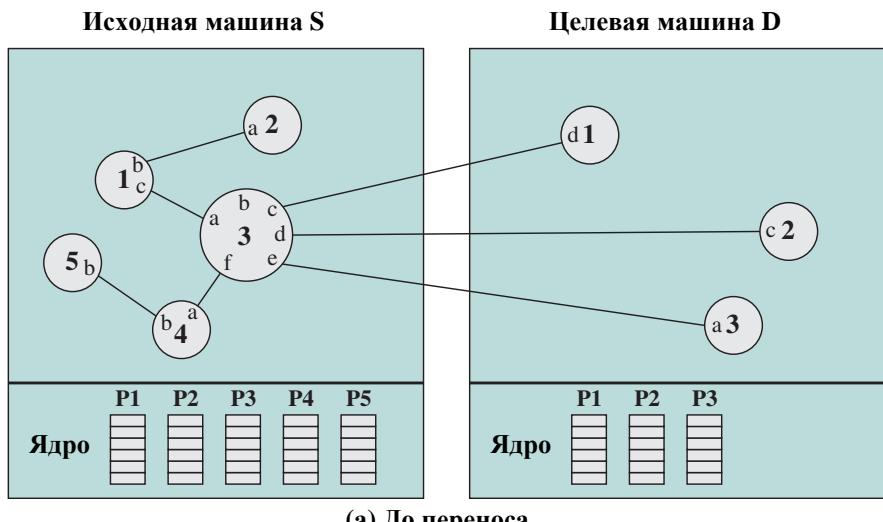
### Инициирование переноса процессов

Кто инициирует перенос процессов — зависит от цели, преследуемой средством подобного переноса. Так, если преследуется цель равномерно распределить нагрузку, то ответственность за принятие решения о моменте переноса возлагается на отдельный модуль операционной системы, постоянно контролирующий нагрузку системы. Такой модуль должен отвечать за вытеснение переносимого процесса или сигнализацию ему. Чтобы выяснить, куда именно должен быть перенесен процесс, такому модулю придется войти в общение с равноправными модулями в других системах для постоянного контроля их нагрузки. Если же преследуется цель достичь конкретных ресурсов, процесс по мере надобности может перемещаться самостоятельно. В последнем случае процессу должно быть известно о существовании распределенной системы. В первом же возможность переноса, как и существование многих систем, может быть процессу невидимой.

### Что именно переносится?

При переносе процесса его приходится уничтожать в исходной системе и воссоздавать в целевой системе. Это перемещение, а не тиражирование, поэтому должен быть перемещен образ процесса, состоящий как минимум из управляющего блока процесса. Кроме того, должны быть обновлены все связи между данным процессом и другими

процессами (например, для обмена сообщениями и сигналами). Все эти соображения наглядно показаны на рис. 19.1: процесс P3 переносится из исходной машины S, чтобы стать процессом P4 на целевой машине D. Все идентификаторы связей, поддерживаемых процессами и обозначенных прописными буквами на рис. 19.1, остаются такими же, как и прежде, а на операционную систему возлагается ответственность за перемещение управляющего блока процесса и обновление отображения связей. Перенос процесса с одной машины на другую происходит невидимо как для самого переносимого процесса, так и для его партнеров по связи.



**Рис. 19.1.** Пример переноса процесса

Перемещение управляющего блока процесса выполняется просто; вся трудность с точки зрения производительности связана с адресным пространством процесса и наличием любых файлов, открытых в переносимом процессе. Рассмотрим сначала адресное пространство процесса и допустим, что применяется схема виртуальной памяти (страничная организация или в дополнение к ней — сегментация). Как поясняется в [172], во внимание необходимо принять следующие стратегии.

- **Немедленно (полностью).** Это перемещение всего адресного пространства во время переноса процесса. Такой подход, определенно, самый ясный, поскольку в прежней системе не остается и следа процесса. Но если адресное пространство очень большое и если большая его часть процессу не нужна, то такой подход может оказаться излишне дорогостоящим. Первоначальные затраты на перенос могут быть порядка минут. Такой подход может быть использован в реализациях, представляющих средство перезапуска и установки контрольных точек, поскольку установить контрольные точки и осуществить перезапуск проще, если все адресное пространство локализовано.
- **Предварительное копирование.** Процесс продолжает выполняться в исходном узле, в то время как адресное пространство копируется в целевой узел. Страницы, изменившиеся в исходном узле за время операции предварительного копирования, приходится копировать второй раз. Такая стратегия позволяет сократить время, в течение которого при переносе процесс замораживается и не может выполняться.
- **Немедленно (“грязно”).** Это перемещение только тех страниц адресного пространства, которые находятся в главной памяти и модифицированы, а любые дополнительные блоки виртуального адресного пространства перемещаются только по требованию. Благодаря этому сводится к минимуму объем перемещаемых данных, но при этом требуется, чтобы исходная машина продолжала работу в течение срока жизни процесса для ведения записей в таблице страниц и/или сегментов, а также требуется поддержка удаленной страничной организации.
- **Копирование при обращении.** Это разновидность немедленной (“грязной”) стратегии, при которой страницы переносятся только при обращении к ним. Такая стратегия требует наименьших первоначальных затрат на перенос процессов: от нескольких десятков до сотен микросекунд.
- **Сброс.** В этом случае страницы процесса в основной памяти исходного узла очищаются путем сброса “грязных” страниц на диск. Затем страницы при необходимости оказываются доступными с диска, а не из оперативной памяти исходного узла. Такая стратегия избавляет исходный узел от необходимости хранить какие-либо страницы переносимого процесса в основной памяти, тут же освобождая блок памяти для использования в других процессах.

Если в процессе на целевой машине, вероятнее всего, будет использоваться лишь незначительная часть его адресного пространства, имеет смысл применение одной из трех последних среди перечисленных выше стратегий. Такое возможно, например, в том случае, когда процесс лишь временно переносится на другую машину для обработки файла, а после этого сразу же возвращается на исходную машину. С другой стороны, если происходит обращение к большей части адресного пространства процесса на целевой машине, то частичное перемещение блоков адресного пространства процесса может оказаться менее эффективным, чем простое перемещение всего адресного пространства

во время переноса процесса с использованием одной из двух первых перечисленных выше стратегий.

В большинстве случаев невозможно знать заранее, какая именно часть нерезидентного адресного пространства понадобится. Но если процессы структурированы в виде потоков выполнения и если основной единицей переноса является поток, а не весь процесс, то наиболее подходящей может оказаться стратегия, основанная на организации страниц в удаленном режиме. На самом деле такая стратегия практически обязательна, поскольку остальные потоки выполнения процесса остаются без внимания и им также требуется доступ к адресному пространству процесса. Перенос потоков выполнения реализован в операционной системе Emerald [122].

Аналогичные соображения применимы и к перемещению открытых файлов. Так, если файл первоначально находится в той же самой системе, где и переносимый процесс, и если этот файл заблокирован для исключительного доступа из данного процесса, то имеет смысл перенести файл вместе с процессом. Опасность состоит в том, что процесс может быть перенесен лишь временно, а файл не понадобится ему до тех пор, пока он не вернется обратно. Если же файл совместно используется несколькими распределенными процессами, то распределенный доступ к файлу должен поддерживаться без его перемещения.

Если разрешено кеширование, как, например, в системе Sprite (см. рис. 16.7), то оно вносит дополнительную сложность в процесс перемещения. Так, если файл открыт процессом для записи и при этом создается и переносится дочерний процесс, то файл должен быть открыт для записи в двух разных узлах. Алгоритм согласованности содержимого кешей, применяемый в системе Sprite, вынуждает сделать файл некешируемым на тех машинах, на которых выполняются оба процесса [68, 69].

## Сообщения и сигналы

Последний из перечисленных ранее вопросов, касающихся участия сообщений и сигналов, разрешается с помощью механизма временного хранения ожидающих своей очереди сообщений и сигналов во время операции переноса и последующего их направления по месту назначения. Может оказаться необходимой поддержка информации о пересылке в первоначальном месте, чтобы обеспечить гарантию того, что все сообщения и сигналы достигнут своего места назначения.

## Сценарий переноса

В качестве наглядного примера самопереноса рассмотрим средство, доступное в операционной системе AIX от компании IBM [263] — распределенной операционной системе UNIX. Аналогичное средство доступно в операционной системе LOCUS [194] (фактически система AIX основана на разработке LOCUS). Такое же средство перенесено под названием “TNC” в операционную систему OSF/I AD [272].

События в рассматриваемом здесь сценарии переноса происходят в следующем порядке.

1. Когда процесс принимает решение о своем переносе, он выбирает целевую машину и отсылает сообщение об удаленном выполнении задачи. Такое сообщение несет в себе часть образа процесса и сведения об открытом файле.
2. На принимающей стороне процесс ядра порождает дочерний процесс, передавая ему полученную информацию.

3. Новый процесс перемещает данные, среду, аргументы или информацию из стека по мере необходимости завершить свое выполнение. В частности, по мере необходимости код программы копируется, если он хранится на “грязных” страницах, или считывается из глобальной файловой системы, если он “чистый”.
4. Порождающий процесс получает сигнал о завершении переноса. Данный процесс отсылает сообщение об окончательном завершении новому процессу и самоуничтожается.

Аналогичная последовательность действий соблюдается и когда перенос инициирует другой процесс. Принципиальное отличие состоит в том, что переносимый процесс должен быть приостановлен, чтобы его можно было перенести в невыполняемом состоянии. Именно такая процедура соблюдается, например, в системе Sprite [68].

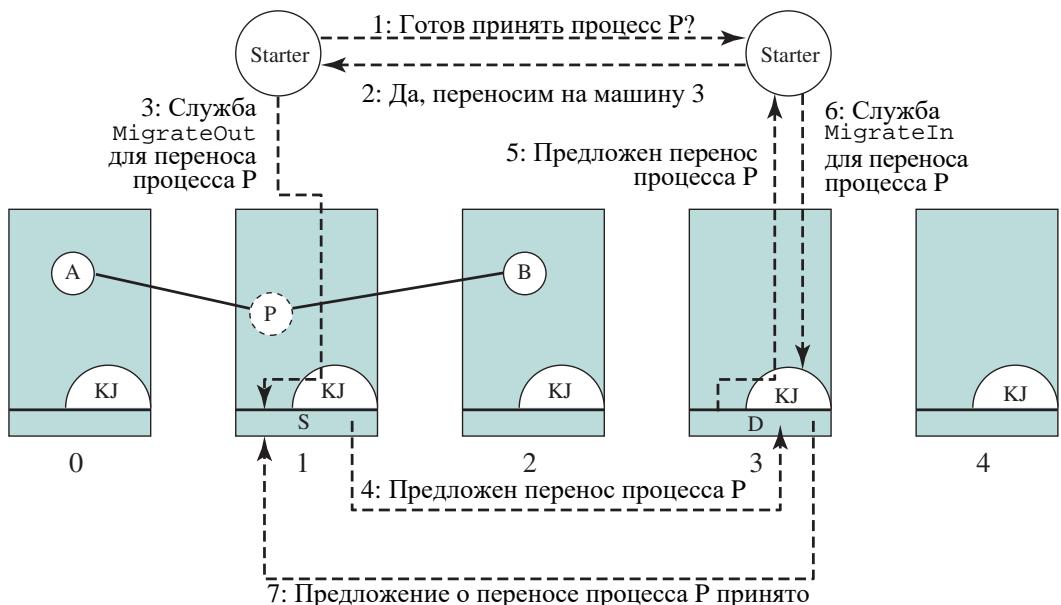
В упомянутом выше сценарии перенос является динамической операцией, выполняющей перемещение образа процесса в несколько этапов. Когда перенос инициируется другим процессом, то, в отличие от самопереноса, применяется другой подход, который состоит в копировании образа и всего адресного пространства процесса в файл, уничтожении процесса, копировании данного файла на другую машину посредством передачи файлов и последующего восстановления процесса из файла на целевой машине. Именно такой подход описан в [238].

## Согласования переноса процессов

Еще одна особенность переноса процессов связана с решением о переносе. В некоторых случаях такое решение принимается одним объектом. Так, если цель состоит в равномерном распределении нагрузки, то модуль, уравновешивающий нагрузку на систему, постоянно контролирует относительную нагрузку на разные машины и выполняет перенос, если возникает необходимость сохранить баланс нагрузки. Если же для доступа процесса к специальным средствам или крупным удаленным файлам применяется самоперенос, то решение о переносе принимает сам процесс. Тем не менее в некоторых системах выбранной целевой системе разрешается принимать участие в принятии решения о переносе процессов. Делается это, в частности, для того, чтобы сэкономить на времени реакции системы ради выгоды пользователей. Например, пользователь рабочей станции может испытывать неудобства от значительного ухудшения времени реакции, если процессы переносятся в его систему, несмотря на то что такой перенос служит более равномерному распределению нагрузки.

Пример механизма согласования переноса можно обнаружить в системе Charlotte [10, 80], где за соблюдение правила переноса, определяющего момент, процесс и место назначения переноса отвечает утилита Starter. Она представляет собой процесс, который отвечает также за долгосрочное планирование и выделение памяти. Таким образом, утилиты Starter координируют соблюдение правила переноса в трех указанных областях. Каждый процесс Starter может управлять кластером машин, своевременно принимая и непредвзято обрабатывая статические данные о нагрузке на систему из ядра каждой ее машины.

Решение о переносе должно быть совместно принято двумя процессами Starter: одним — в исходном узле, а другим — в целевом узле, как показано на рис. 19.2. При этом выполняются следующие действия.



**Рис. 19.2.** Согласование переноса процессов

1. Утилита Starter, управляющая исходной системой (S), решает, что процесс P должен быть перенесен в конкретную целевую систему (D). С этой целью она отсылает сообщение утилите Starter в целевой системе D, запрашивая перенос данного процесса.
2. Если утилита Starter в целевой системе D готова принять процесс, она отправляет обратно подтверждение.
3. Утилита Starter передает свое решение ядру системы S через вызов службы MigrateOut, если она действует в исходной системе S, или же отсылает сообщение процессу KernJob (KJ) в исходной системе S, если она действует на другой машине. Процесс KJ служит для преобразования сообщений, поступающих от удаленных процессов, в вызовы соответствующих служб.
4. Затем ядро исходной системы S предлагает перенести процесс в целевую систему D. В такое предложение входят статистические данные о процессе P, в том числе его возраст, а также степень нагрузки на процессор и канал связи.
5. Если целевой системе D недостает ресурсов, она может отклонить предложение. В противном случае ядро целевой системы D перешлет предложение утилите Starter, управляющей этой системой. В пересылаемое предложение входит та же самая информация, что и в предложение, поступившее от исходной системы S.
6. Стратегия принятия решения Starter связывается с целевой системой D через вызов службы MigrateIn.
7. Целевая система D сначала резервирует требующиеся ресурсы во избежание взаимоблокировок и нарушений в управлении потоком, а затем отсылает исходной системе S сообщение о принятии ее предложения.

На рис. 19.2 показаны также два других процесса А и В, имеющих открытые связи с процессом Р. Если следовать приведенной выше процедуре, машина 1, на которой находится исходная система S, должна отослать сообщение об обновлении связей обеим машинам, 0 и 2, чтобы сберечь связи процессов А и В с переносимым процессом Р. Сообщения об обновлении связей передают новый адрес каждой связи, удерживаемой процессом Р, и подтверждаются уведомляемыми ядрами в целях синхронизации. После этого сообщение, отправленное процессу Р по любой из его связей, будет отослано непосредственно целевой системе D. Такими сообщениями можно обмениваться одновременно только с описанными стадиями процедуры согласования переноса. И наконец, после этапа 7 и обновления всех связей исходная система S собирает весь контекст переносимого процесса Р в одном сообщении и отправляет его целевой системе D.

На машине 4 также действует система Charlotte, но она не принимает участия в переносе процесса Р, а следовательно, в данном случае с другими системами не общается.

## Выселение

Механизм согласования переноса позволяет целевой системе отказаться от приема переносимого процесса. Возможно, стоило бы также разрешить системе выселять перенесенный в нее процесс. Так, если рабочая станция пристаивает, на нее можно перенести один или несколько процессов. Но как только активизируется пользователь этой рабочей станции, перенесенные на нее процессы, возможно, придется выселить, чтобы обеспечить приемлемое время реакции.

Характерный пример возможности выселения процессов можно обнаружить в системе Sprite [68]. Это операционная система для рабочих станций, в которой каждый процесс выполняется в течение всего срока своего действия в отдельном узле, называемом исходным узлом данного процесса. Если процесс переносится, на целевой машине он становится инородным. Целевая машина в любой момент времени может выселить инородный процесс, который в этом случае придется перенести обратно в его исходный узел.

Ниже перечислены и кратко описаны механизмы выселения, применяемые в системе Sprite.

1. Процесс мониторинга в каждом узле, отслеживающий текущую нагрузку с целью определить момент, когда следует принять новые инородные процессы. Если текущий контроль обнаружит какие-либо действия на консоли рабочей станции, он инициирует процедуру выселения всех инородных процессов.
2. Если процесс выселяется, он переносится обратно в свой исходный узел. Процесс может быть перенесен снова, если для этого доступен другой узел.
3. Несмотря на то что для выселения всех процессов может потребоваться некоторое время, все процессы, помеченные для выселения, сразу же приостанавливаются. Разрешив выселяемому процессу выполниться в ожидании выселения, можно сократить время, в течение которого процесс остается замороженным, но при этом в процессе выселения вычислительные мощности, имеющиеся в узле, снижаются.
4. Все адресное пространство выселяемого процесса перемещается в исходный узел. Время выселения процесса и его обратного переноса в исходный узел может быть значительно сокращено, если извлечь хранящийся в памяти образ выселяемого процесса на его предыдущем узле. Правда, это вынуждает сторонний узел выделять ресурсы и обрабатывать запросы от выселяемого процесса дольше, чем требуется.

## Вытесняющие переносы в сравнении с невытесняющими

В этом разделе был рассмотрен вытесняющий перенос процессов, подразумевающий перемещение частично выполненного процесса или хотя бы такого процесса, создания которого было завершено. Более простую функцию выполняет невытесняющий перенос процессов, подразумевающий перемещение только тех процессов, которые еще не начали выполняться, а следовательно, не требуют переноса своего состояния. В обоих видах переноса процессов сведения о среде, в которой процесс будет выполняться, должны быть переданы удаленному узлу. К таким сведениям относятся текущий рабочий каталог пользователя, наследуемые процессом привилегии и ресурсы, подобные описаниям файлов.

## 19.2. РАСПРЕДЕЛЕННЫЕ ГЛОБАЛЬНЫЕ СОСТОЯНИЯ

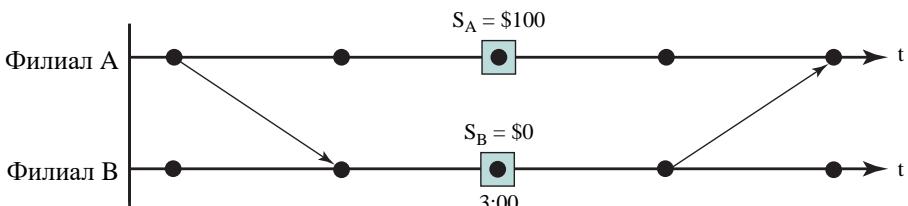
В этом разделе речь пойдет о распределенных глобальных состояниях.

### Глобальные состояния и распределенные моментальные снимки

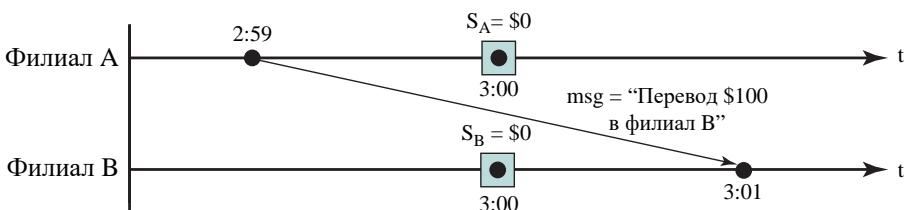
Все трудности параллелизма, возникающие в тесно связанной системе, такие как взаимное исключение, взаимоблокировка и голодание, проявляются и в распределенной системе. Стратегии проектирования подобных систем усложняются в связи с отсутствием глобального состояния системы. Это означает, что текущее состояние всех процессов в распределенной системе неизвестно ни операционной системе, ни любому отдельному процессу.

Чтобы узнать текущее состояние всех процессов в локальной системе, отдельный процесс должен получить доступ к находящимся в памяти управляющим блокам этих процессов. А сведения о состоянии удаленных процессов текущий процесс может получить только через сообщения, которые при этом представляют прошлое состояние удаленного процесса некоторое время назад. Это можно сравнить с положением дел в астрономии: наши знания о далеких звездах или галактиках основываются на свете и других электромагнитных волнах, приходящих из удаленных объектов в космосе и дающих картину этих объектов в далеком прошлом. Например, наши текущие знания об удаленном объекте, находящемся на расстоянии пяти световых лет от Земли, на самом деле имеют срок давности пять лет.

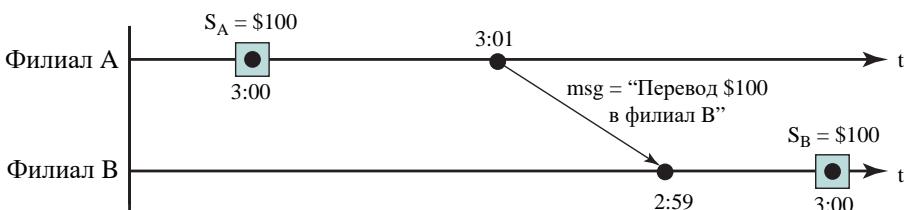
Временные задержки, обусловленные характером распределенных систем, еще больше усложняют все трудности, связанные с параллелизмом. Чтобы продемонстрировать это положение, рассмотрим пример, взятый из [8], в котором используются графы процессов и событий, приведенные на рис. 19.3 и 19.4. На этих графах горизонтальная линия для каждого процесса представляет собой ось времени; точка на линии — событие (например, внутреннее событие, отправка или прием сообщений); прямоугольник, охватывающий точку, — моментальный снимок состояния локального процесса, сделанный в данной точке; а стрелка — сообщение, которым обмениваются два процесса.



а) Итого \$100



б) Итого \$0



в) Итого \$200

Рис. 19.3. Пример определения глобальных состояний

В данном примере у одного клиента имеется счет в банке, распределенный между двумя филиалами. Чтобы определить итоговую сумму на счете клиента, банк должен выяснить сумму в каждом своем филиале. Допустим, что эта банковская операция выполняется ровно в 3 часа дня. На рис. 19.3, а приведен пример, в котором выявлен остаток на объединенном счете на сумму 100,00 долларов. Однако возможна ситуация, представленная на рис. 19.3, б, в которой остаток на счете переносится из филиала А в филиал В в момент запроса. В итоге на объединенном счете обнаруживается неверный остаток на сумму 0,00 долларов.

Это конкретное затруднение можно разрешить, проанализировав все сообщения по ходу переноса остатка на счет в момент наблюдения. Все денежные переводы со счета регистрируются в филиале А вместе с обозначением места назначения каждого перевода. Следовательно, в “состояние” счета в филиале А войдут текущий остаток и запись о денежных переводах. Анализируя оба счета, наблюдатель обнаружит денежный перевод, отправленный из филиала А на счет клиента в филиале В. А поскольку денежная сумма еще не поступила в филиал В, то она прибавляется к итоговому остатку на счете клиента. Любая сумма, которая была отправлена и получена, считается только один раз как часть остатка на принимающем счету.

Но и такая стратегия ненадежна, как показано на рис. 19.3, в. В данном примере системные часы в обоих филиалах банка не полностью синхронизированы. Состояние счета клиента в филиале А в 3 часа дня указывает остаток 100,00 долларов. Однако эта сумма впоследствии переводится в филиал В, когда системные часы в филиале А показывают время 3:01, а поступает эта сумма в филиал В, когда системные часы в нем показывают время 2:59. Следовательно, в момент наблюдения — ровно в 3:00 — сумма подсчитывается дважды.

Чтобы возникающие трудности стали понятнее и было легче сформулировать правильное решение, определим следующие термины.

- **Канал.** Такой канал существует между двумя процессами, если они обмениваются сообщениями. Канал можно рассматривать как путь или средство для передачи сообщений. Ради удобства каналы считаются односторонними. Так, если два процесса обмениваются сообщениями, для передачи сообщений в обоих направлениях потребуются два канала.
- **Состояние.** Состояние процесса представляет собой последовательность сообщений, отправленных и полученных по всем каналам, связанным с данным процессом.
- **Моментальный снимок.** Такой снимок записывает состояние процесса. Каждый моментальный снимок включает в себя запись всех сообщений, отправленных и полученных по всем каналам после того, как был сделан последний моментальный снимок.
- **Глобальное состояние.** Это объединенное состояние всех процессов.
- **Распределенный моментальный снимок.** Это совокупность моментальных снимков, приходящихся по одному на каждый процесс.

Проблема в том, что истинное глобальное состояние невозможно определить из-за промежутка времени, требующегося для передачи сообщения. Можно попытаться определить глобальное состояние, собрав моментальные снимки из всех процессов. Например, глобальное состояние, приведенное на рис. 19.4, а в момент получения моментальных снимков, показывает передачу одного сообщения по каналу  $\langle A, B \rangle$ , другого — по каналу  $\langle A, C \rangle$  и еще одного — по каналу  $\langle C, A \rangle$ . Сообщения 2 и 4 представлены в глобальном состоянии надлежащим образом, чего нельзя сказать о сообщении 3. Распределенный моментальный снимок показывает, что данное сообщение получено, хотя еще не отправлено.

Нам хотелось бы, чтобы в распределенном моментальном снимке было зарегистрировано согласованное глобальное состояние. Согласованным считается такое глобальное состояние, когда для каждого состояния процесса, регистрирующего получение сообщения, отправка этого сообщения зарегистрирована в состоянии того процесса, который его отправил. Наглядный пример согласованного глобального состояния приведен на рис. 19.4, б. Несогласованное глобальное состояние возникает в том случае, если принимающий процесс зарегистрировал получение сообщения, но соответствующий передающий процесс не зарегистрировал его отправку (см. рис. 19.4, а).

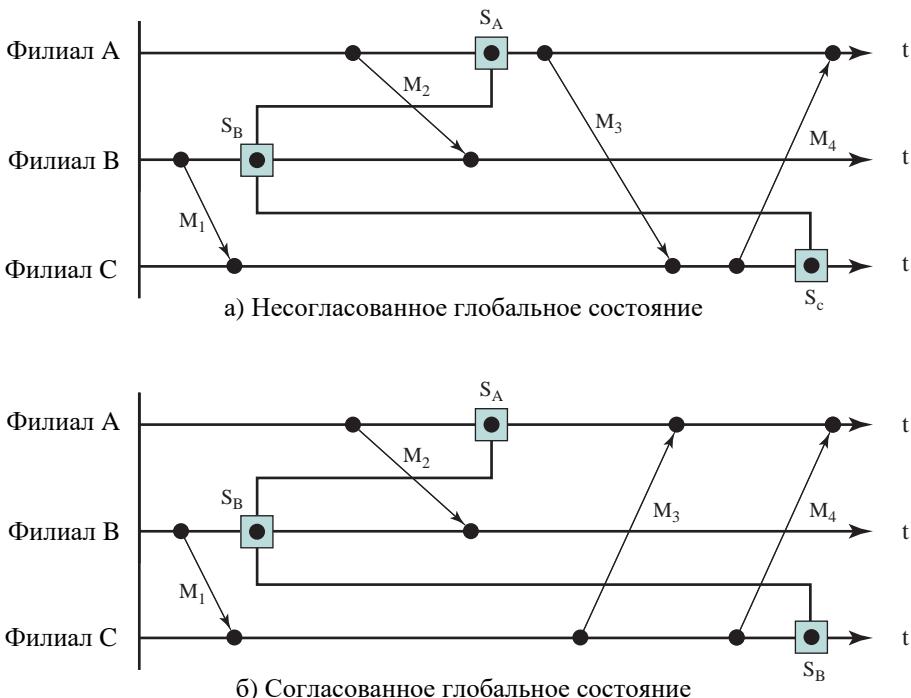


Рис. 19.4. Несогласованные и согласованные глобальные состояния

## Алгоритм распределенных моментальных снимков

В алгоритме распределенных моментальных снимков, регистрирующем согласованное глобальное состояние, предполагается, что сообщения доставляются в том порядке, в каком они отправляются, и что они не теряются. Этим требованиям удовлетворяет надежный транспортный протокол (например, TCP). В этом алгоритме применяется специальное управляющее сообщение, называемое **маркером** (marker).

Сначала некоторый процесс инициирует алгоритм, регистрируя свое состояние и отправляя маркер по всем исходящим каналам перед отправкой любых дополнительных сообщений. Затем каждый процесс  $p$  продолжается, как описано ниже. После первого приема маркера (скажем, от процесса  $q$ ) принимающий процесс  $p$  выполняет следующие действия.

1. Процесс  $p$  регистрирует локальное состояние  $S_p$ .
2. Процесс  $p$  регистрирует состояние входящего канала от процесса  $q$  к процессу  $p$  как пустое.
3. Процесс  $p$  передает маркер всем соседним процессам по всем исходящим каналам.

Все эти действия должны выполняться автоматически. Это означает, что процесс  $p$  не может ни отправлять, ни получать сообщения до тех пор, пока не будут выполнены все три этапа описанной выше процедуры.

Если процесс  $p$  получает маркер из другого входящего канала (скажем, от процесса  $r$ ) в любой момент после регистрации своего состояния, он выполняет следующее действие.

- Процесс  $p$  регистрирует состояние канала, входящего от процесса  $r$  к процессу  $p$ , как последовательность сообщений, полученных процессом  $p$  от процесса  $r$  с того момента, когда процесс  $p$  зарегистрировал свое локальное состояние  $S_p$ , и до того момента, когда он получил маркер от процесса  $r$ .

Рассматриваемый здесь алгоритм в конкретном процессе завершается, как только им будет получен маркер по каждому входящему каналу.

В [8] сделаны следующие наблюдения в отношении данного алгоритма.

1. Любой процесс может запустить данный алгоритм, отправив маркер. Даже если решение о запуске записи состояния будет принято независимо несколькими узлами, алгоритм завершится успешно.
2. Алгоритм завершится в течение конечного времени, если все сообщения, в том числе маркеры, будут доставлены в течение конечного времени.
3. Это распределенный алгоритм, в котором каждый процесс отвечает за регистрацию как своего состояния, так и состояния всех входящих каналов.
4. Как только все состояния будут зарегистрированы (т.е. алгоритм завершится во всех процессах), согласованное глобальное состояние, полученное по данному алгоритму, может быть собрано каждым процессом. С этой целью каждый процесс отправляет данные зарегистрированного состояния, а также пересыпает полученные им данные состояния по каждому исходящему каналу. В качестве альтернативы инициирующий процесс может опросить все остальные процессы, чтобы получить глобальное состояние.
5. Алгоритм не оказывает влияния на любой другой распределенный алгоритм, в котором участвуют данные процессы, и не подвергается влиянию любого другого распределенного алгоритма.

В качестве примера применения данного алгоритма, взятого из [21], рассмотрим ряд процессов, приведенных на рис. 19.5. Каждый процесс представлен на этом рисунке узлом графа, а каждый односторонний канал — линией между двумя узлами графа, направление которой обозначается стрелкой. Допустим, что выполняется алгоритм моментальных снимков и каждым процессом по каждому из девяти исходящих каналов отсылаются девять сообщений. Процесс 1 решает зарегистрировать глобальное состояние после отправки шести сообщений, а процесс 4 независимым образом решает зарегистрировать глобальное состояние после отправки трех сообщений. По завершении данного алгоритма моментальные снимки собираются из каждого процесса; полученные результаты приведены на рис. 19.6. В частности, процесс 2 отправляет четыре сообщения по каждому из двух исходящих каналов процессам 3 и 4, прежде чем зарегистрировать состояние. В то же время он получает четыре сообщения от процесса 1, прежде чем зарегистрировать свое состояние, оставив пятое и шестое сообщения связанными с каналом. Читателям остается лишь проверить согласованность полученного в итоге моментального снимка глобального состояния по следующему критерию: каждое отправленное сообщение было либо получено целевым процессом, либо зарегистрировано как передаваемое по каналу.

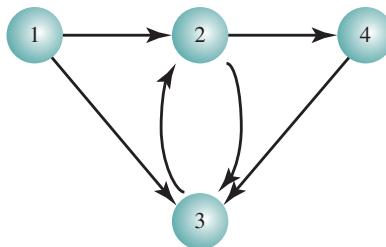


Рис. 19.5. Граф процессов и каналов

<b>Процесс 1</b> Исходящие каналы $2 \rightarrow 1, 2, 3, 4, 5, 6$ $3 \rightarrow 1, 2, 3, 4, 5, 6$ Входящие каналы	<b>Процесс 3</b> Исходящие каналы $2 \rightarrow 1, 2, 3, 4, 5, 6, 7, 8$ Входящие каналы $1 \leftarrow 1, 2, 3$ сохраняет 4, 5, 6 $2 \leftarrow 1, 2, 3$ сохраняет 4 $4 \leftarrow 1, 2, 3$
<b>Процесс 2</b> Исходящие каналы $3 \rightarrow 1, 2, 3, 4$ $4 \rightarrow 1, 2, 3, 4$ Входящие каналы $1 \leftarrow 1, 2, 3, 4$ сохраняет 5, 6 $3 \leftarrow 1, 2, 3, 4, 5, 6, 7, 8$	<b>Процесс 4</b> Исходящие каналы $3 \rightarrow 1, 2, 3$ Входящие каналы $2 \leftarrow 1, 2$ сохраняет 3, 4

Рис. 19.6. Пример получения моментального снимка

Алгоритм распределенных моментальных снимков является мощным и гибким инструментальным средством. С его помощью можно приспособить любой централизованный алгоритм к распределенной среде, поскольку основанием для любого централизованного алгоритма служит знание глобального состояния. К конкретным примерам применения данного алгоритма относится обнаружение взаимоблокировки и завершения процесса (см. в [21], [159]). С помощью данного алгоритма можно также предоставить контрольную точку для отката и восстановления при обнаружении сбоя в работе некоторого распределенного алгоритма.

## 19.3. РАСПРЕДЕЛЕННОЕ ВЗАЙМНОЕ ИСКЛЮЧЕНИЕ

Напомним, что в главах 5, “Параллельные вычисления: взаимоисключения и многозадачность”, и 6, “Параллельные вычисления: взаимоблокировка и голодание”, рассматривались вопросы, связанные с одновременным выполнением процессов, в ходе которого возникают два основных затруднения: взаимное исключение и взаимоблокировка. Основное внимание в этих главах было уделено способам разрешения этих затруднений в контексте одной системы с одним или несколькими процессорами, но с общей основной

памятью. Но когда приходится иметь дело с распределенной операционной системой и совокупностью процессов, не пользующихся общей основной памятью или системными часами, то возникают новые трудности и требуются новые способы их разрешения. Алгоритмы взаимного исключения и взаимоблокировки должны зависеть от обмена сообщениями и не могут зависеть от доступа к общей памяти. В этом и следующем разделах взаимное исключение и взаимоблокировка рассматриваются в контексте распределенной операционной системы.

## Принципы распределенного взаимного исключения

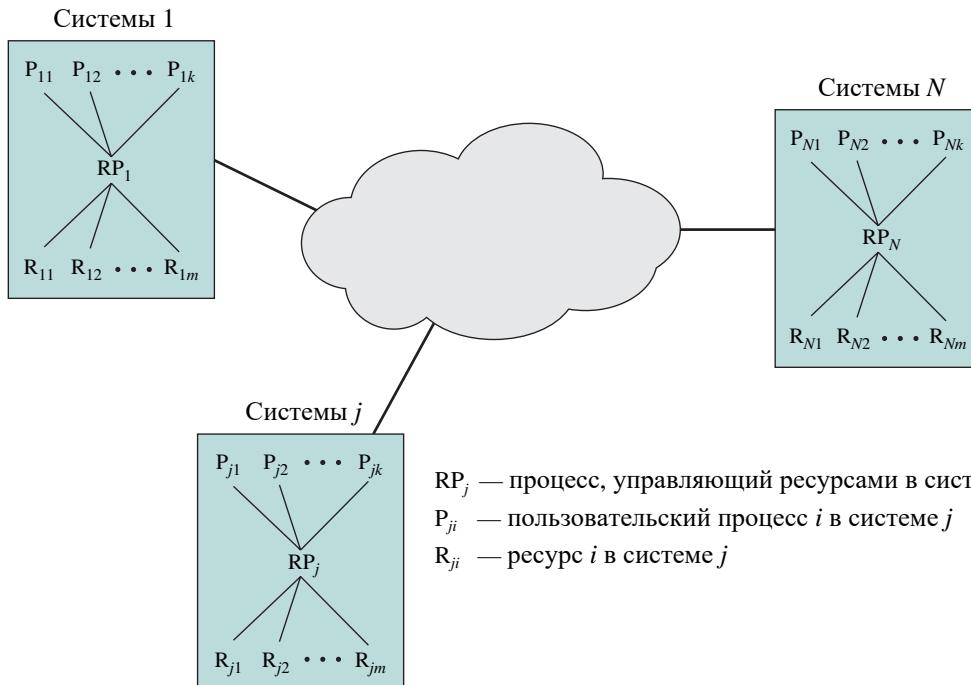
Когда два или более процессов конкурируют за использование системных ресурсов, возникает потребность в механизме для обеспечения взаимного исключения. Допустим, что двум (или более) процессам требуется доступ к единственному неразделяемому ресурсу, такому как принтер. В ходе выполнения каждый процесс будет отсылать команды устройству ввода-вывода, получать информацию о состоянии, передавать и/или принимать данные. Назовем такой ресурс критическим, а часть программы, в которой он используется, — критическим участком программы. Очень важно, что одновременно лишь одна программа может находиться в критическом участке. Чтобы понять и соблюсти это ограничение, нельзя полагаться только на операционную систему, поскольку детализированное требование может быть неочевидным. Так, например, в случае принтера было бы желательно, чтобы любой отдельный процесс мог управлять печатающим устройством, выводя на него целый файл. Ведь иначе выводимые данным процессом строки будут перемежаться со строками из соперничающего процесса.

Для успешного распараллеливания процессов требуется возможность определять критические участки и соблюдать взаимное исключение. Это основное требование к любой схеме параллельной обработки данных. Любое средство или функциональная возможность, обеспечивающая поддержку взаимного исключения, должно удовлетворять следующим требованиям.

1. Должно обеспечиваться взаимное исключение: войти в критический участок разрешается одновременно лишь одному из всех процессов, имеющих критические участки для одного и того же ресурса или общего объекта.
2. Процесс, останавливающийся в своем некритическом участке, не должен затрагивать другие процессы.
3. Процесс, требующий доступа к критическому участку, не может находиться в состоянии ожидания бесконечно: не должно быть ни взаимоблокировок, ни голодания.
4. Если ни один из процессов не находится в критическом участке, то любой процесс, которому требуется войти в свой критический участок, должен получить разрешение на вход без какой бы то ни было задержки.
5. Никаких предположений об относительном быстродействии или количестве процессов не делается.
6. Процесс остается в своем критическом участке только в течение конечного времени.

На рис. 19.7 представлена модель, с помощью которой можно исследовать разные подходы к взаимоисключению в распределенном контексте. Предполагается, что определенный ряд систем связан вместе через некоторый тип сетевого средства. Допускается также, что в каждой такой системе имеется некоторая функция (или процесс), действую-

шая на уровне операционной системы, которая отвечает за выделение ресурсов. Каждый такой процесс управляет целым рядом ресурсов и обслуживает ряд пользовательских процессов. Задача состоит в том, чтобы разработать алгоритм, с помощью которого все эти процессы могли бы совместно соблюдать взаимное исключение.



**Рис. 19.7.** Модель для решения задачи взаимного исключения при управлении распределенными процессами

Алгоритмы взаимного исключения могут быть централизованными или распределенными. В полностью **централизованном алгоритме** один узел служит для управления доступом ко всем общим объектам. Когда любому процессу требуется доступ к критическому ресурсу, он выдает запрос своему локальному процессу, управляющему ресурсами. Этот процесс, в свою очередь, отсылает сообщение с запросом управляющему узлу, который возвращает ответное (разрешающее) сообщение, когда совместно используемый объект становится доступным. Когда процесс завершит пользование ресурсом, он отправит управляющему узлу освобождающее сообщение. Такой централизованный алгоритм обладает двумя ключевыми свойствами.

1. Решения о выделении ресурсов принимает только управляющий узел.
2. Вся необходимая информация о ресурсах, а также о состоянии выделения ресурсов процессам, сосредоточена в управляющем узле.

Централизованный подход довольно прост, и при его применении легко понять, каким образом соблюдается взаимное исключение: управляющий узел не удовлетворит запрос на ресурс до тех пор, пока нужный ресурс не освободится. Тем не менее такой подход имеет ряд недостатков. Так, если управляющий узел выйдет из строя, то нару-

шится и механизм взаимного исключения, по крайней мере временно. Кроме того, для выделения и освобождения каждого ресурса приходится обмениваться сообщениями с управляющим узлом. Следовательно, управляющий узел в такой схеме взаимного исключения может стать узким местом.

Вследствие недостатков, присущих централизованным алгоритмам, был проявлен интерес к разработке распределенных алгоритмов. Полностью **распределенный алгоритм** характеризуется следующими свойствами.

1. Все узлы в среднем обладают равным количеством информации.
2. Каждый узел имеет лишь частичное представление обо всей системе в целом и должен принимать решения на основании этой информации.
3. Все узлы несут равную ответственность за окончательное решение.
4. Все узлы прилагают в среднем равные усилия, чтобы прийти к окончательному решению.
5. Выход узла из строя в общем случае не приводит к сбою всей системы в целом.
6. Не имеется общих системных часов, используемых для регулирования синхронизация событий.

Второе и шестое из перечисленных выше свойств требуют некоторого уточнения. Что касается второго свойства, то в некоторых распределенных алгоритмах требуется, чтобы вся информация, известная любому узлу, была сообщена всем остальным узлам. Но даже в этом случае в любой заданный момент времени часть этой информации будет находиться в состоянии передачи и еще не достигнет всех остальных узлов. Таким образом, информация в узле обычно не полностью актуальна из-за временных задержек в передаче сообщений, и в этом смысле она лишь частична.

Что же касается шестого свойства, то вследствие задержек в обмене данными между системами невозможно поддерживать системные часы, постоянно доступные всем системам. Более того, с технической точки зрения непрактично поддерживать одни центральные часы и точно синхронизировать с ними все локальные часы. Ведь со временем произойдет постепенное отклонение показаний различных локальных часов от центральных, что приведет к потере синхронизации.

Именно задержка в передаче наряду с отсутствием общих системных часов намного затрудняет разработку механизмов взаимного исключения в распределенной системе по сравнению с централизованной. Поэтому, прежде чем рассматривать некоторые алгоритмы для распределенного взаимного исключения, исследуем общий подход к преодолению недостатка несогласованности системных часов.

## Упорядочение событий в распределенной системе

В основу работы большинства распределенных алгоритмов взаимного исключения и взаимоблокировки положено временное упорядочение событий. Таким образом, главное ограничение состоит в отсутствии общих системных часов или средств синхронизации локальных часов. Задачу преодоления данного ограничения можно выразить следующим образом: допустим, что событие  $a$  произошло в системе  $i$  до (или после) события  $b$  в системе  $j$  и требуется, чтобы к такому выводу можно было бы согласованно прийти во всех системах, подключенных к сети. К сожалению, имеются две большие проблемы. Во-первых, событие, наступившее в одной системе, может наблюдаться в другой сист-

ме с некоторой задержкой. И во-вторых, отсутствие синхронизации приводит к отклонениям в показаниях часов в разных системах.

Для преодоления подобных трудностей Лэмпартом (Lamport) [141] был предложен метод, известный как присваивание отметок времени (timestamping), упорядочивающий события в распределенной системе без использования физических часов. Такой метод оказывается настолько эффективным и действенным, что применяется в подавляющем большинстве алгоритмов распределенного взаимного исключения и взаимоблокировки.

Прежде всего, необходимо дать определение понятию *события* (event). В конечном счете нас интересуют действия, происходящие в локальной системе, как, например, вхождение или оставление процессом его критического участка. Однако в распределенной системе процессы взаимодействуют посредством сообщений, и поэтому события имеет смысл связывать с сообщениями. Локальное событие можно очень легко привязать к сообщению. Например, процесс может отослать сообщение, когда ему потребуется войти в свой критический участок или же оставить его. Во избежание неоднозначности связем события только с отправкой, но не с получением сообщений. Таким образом, всякий раз, когда процесс передает сообщение, определяется событие, соответствующее моменту времени, когда сообщение покидает данный процесс.

Схема присваивания отметок времени предназначена для упорядочения событий, состоящих из передачи сообщений. В каждой подключенной к сети системе  $i$  поддерживается локальный счетчик  $C_i$ , действующий как системный таймер. И всякий раз, когда система передает сообщение, она увеличивает значение счетчика на 1. Сообщение отсылается в виде

$$(m, T_i, i),$$

где

$m$  — содержимое сообщения;

$T_i$  — отметка времени для данного сообщения, устанавливается равной  $C_i$ ;

$i$  — числовой идентификатор данной системы в распределенной системе.

Когда сообщение получено, в счетчике времени принимающей системы  $j$  устанавливается значение, на единицу большее максимума среди его текущего значения и входящей отметки времени:

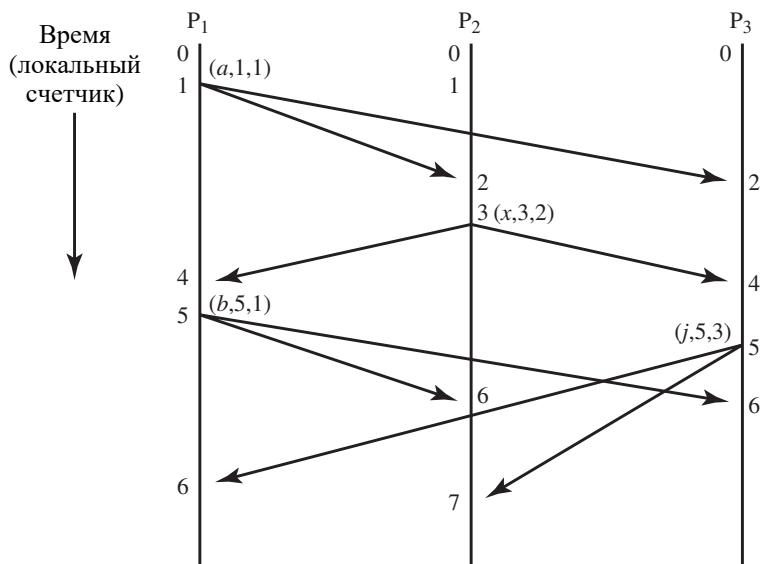
$$C_j = 1 + \max(C_j, T_i).$$

В каждой системе упорядочение событий определяется приведенными ниже правилами. Так, о сообщении  $x$  от системы  $i$  и сообщении  $y$  от системы  $j$  можно сказать, что сообщение  $x$  предшествует сообщению  $y$ , если выполняется одно из следующих условий.

1. Если  $T_i < T_j$  или
2. если  $T_i = T_j$  и  $i < j$ .

Момент времени, связанный с каждым сообщением, представляет собой отметку, передаваемую с сообщением, а упорядочение моментов времени определяется упомянутыми выше правилами. В частности, два сообщения с одинаковыми отметками времени упорядочиваются по номерам их систем. А поскольку эти правила применяются независимо от конкретной системы, такой подход позволяет избежать любых затруднений, возникающих в связи с отклонением показаний разных системных часов, по которым синхронизируется обмен данными между процессами.

Пример операции, выполняемой в соответствии с данным алгоритмом, приведен на рис. 19.8. В этом примере имеются три системы, каждая из которых представлена процессом, управляющим алгоритмом присваивания отметок времени. Процесс  $P_1$  начинается при нулевом значении счетчика времени. Чтобы передать сообщение  $a$ , он увеличивает значение счетчика времени на 1 и передает сообщение  $(a, 1, 1)$ , в котором первое числовое значение обозначает отметку времени, а второе — передающую (т.е. первую) систему. Это сообщение получают процессы во второй и третьей системах, и в обоих случаях вместо первоначального нулевого значения в локальном счетчике времени устанавливается значение  $1 + \max(0, 1) = 2$ .

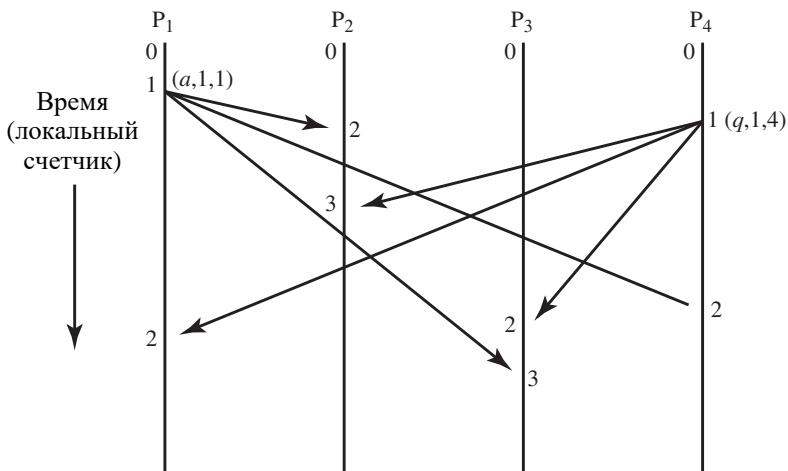


**Рис. 19.8.** Пример операции по алгоритму присваивания отметки времени

Процесс  $P_2$  передает следующее сообщение, увеличивая свой счетчик до 3. По получении этого сообщения процессы  $P_1$  и  $P_3$  увеличивают свои счетчики до 4. Затем процесс  $P_1$  передает сообщение  $b$ , а процесс  $P_3$  — сообщение  $j$  почти в один и тот же момент времени и с одной и той же отметкой времени. Это не приводит к конфликту в силу описанного выше принципа упорядоченности. После того как произойдут все эти события, сообщения будут одинаково упорядочены во всех системах, а именно — как  $\{a, x, b, j\}$ .

Данный алгоритм оказывается вполне работоспособным, несмотря на отличия во времени передачи сообщений между парами систем, как показано на рис. 19.9, где процессы  $P_1$  и  $P_4$  передают сообщения с одной и той же отметкой времени. Сообщение второй системе от процесса  $P_1$  поступает раньше, чем от процесса  $P_4$ , но в третьей системе все наоборот — сообщение от процесса  $P_4$  поступает первым. Тем не менее после получения всех сообщений всеми системами эти сообщения одинаково упорядочены во всех системах — следующим образом:  $\{a, q\}$ .

Следует, однако, иметь в виду, что упорядочение по такой схеме совсем не обязательно соответствует конкретной временной последовательности. Для алгоритмов, основанных на схеме присваивания отметок времени, совсем не важно, какое именно событие произошло первым. Важно лишь то, что все процессы, реализующие такой алгоритм, соглашаются относительно упорядочения событий.



**Рис. 19.9.** Еще один пример операции по алгоритму присваивания отметки времени

В обоих рассмотренных только что примерах каждое сообщение одним процессом отсылается всем остальным процессам. Если какие-либо сообщения отсылаются не таким способом, то некоторые системы принимают не все сообщения, и тогда нельзя добиться, чтобы полученные сообщения были одинаково упорядочены всеми системами. В таком случае возникает совокупность частичных упорядочений. Нас, однако, интересует главным образом применение отметок времени в распределенных алгоритмах для взаимного исключения и взаимоблокировки. В таких алгоритмах процесс обычно отсылает сообщение (со своей отметкой времени) всем остальным процессам, а отметки времени служат для выбора способа обработки сообщений.

## Распределенная очередь

### Первая версия алгоритма

Один из первых подходов к распределенному взаимному исключению основан на понятии распределенной очереди [141]. Такой алгоритм основан на следующих предположениях.

1. Распределенная система состоит из  $N$  узлов, уникально пронумерованных от 1 до  $N$ . Каждый узел содержит один процесс, который делает запросы для взаимно исключающего доступа к ресурсам от имени других процессов. Кроме того, данный процесс служит в качестве арбитра в разрешении перекрывающихся во времени входящих запросов от других узлов.
2. Сообщения, отсылаемые от одного процесса к другому, принимаются в том же порядке, в каком они отсылаются.
3. Каждое сообщение корректно доставляется по месту своего назначения в течение конечного периода времени.
4. Сеть полностью связная. Это означает, что каждый процесс может отсылать сообщения любому другому процессу непосредственно, не требуя промежуточного процесса для пересылки сообщения.

Второе и третье допущения могут быть осуществлены с помощью надежного транспортного протокола, например TCP (см. главу 13, “Встроенные операционные системы”).

Ради простоты здесь описывается алгоритм, рассчитанный на тот случай, когда каждая система управляет только одним ресурсом. Обобщение на случай многих ресурсов тривиально.

Рассматриваемый алгоритм пытается обобщить другой алгоритм, который работал бы по простому принципу в централизованной системе. Так, если бы ресурсом управлял центральный процесс, он мог бы поставить входящие запросы в очередь и обрабатывать их по принципу “первым пришел — первым обслужен”. Чтобы добиться того же в распределенной системе, каждая система должна иметь копию одной и той же очереди. Используя присваивание отметок времени, можно гарантировать, что все системы согласуют порядок, в котором обрабатываются запросы.

Но здесь возникает следующее затруднение: в связи с тем что для передачи сообщения по сети требуется некоторый конечный промежуток времени, возникает опасность, что две разные системы не придут к согласию относительно того, какой именно процесс должен находиться в голове очереди. Обратимся к рис. 19.9, на котором показан момент, когда сообщение  $a$  поступает в процесс  $P_2$ , а сообщение  $q$  — в процесс  $P_3$ , но оба сообщения все еще пребывают в состоянии передачи другим процессам. Таким образом, имеется период времени, в течение которого процессы  $P_1$  и  $P_2$  будут считать, что сообщение  $a$  находится в голове очереди, тогда как процессы  $P_3$  и  $P_4$  — что там находится сообщение  $q$ . Это может привести к нарушению требования взаимного исключения. Во избежание этого вводится следующее правило: для того чтобы процесс принял верное решение о выделении ресурса, основываясь на своей очереди, он должен получить сообщения от всех остальных систем, чтобы ни одно сообщение, отправленное раньше, чем то, которое находится в голове его собственной очереди, не находилось все еще в состоянии передачи. Это правило поясняется в п. 3, б описываемого далее алгоритма.

В каждой системе поддерживается структура данных для хранения последних сообщений, полученных от всех остальных систем (включая последнее сообщение, сгенерированное данной системой). Лэмпорт называет такую структуру данных очередью; на самом деле это массив, в котором для каждой системы выделен отдельный элемент. В любой момент времени элемент  $q[j]$  локального массива содержит сообщение от процесса  $P_j$ . Массив инициализируется следующим образом:

$$q[j] = (\text{Release}, 0, j) \quad j = 1, \dots, N^1.$$

В рассматриваемом алгоритме используются три типа сообщений.

- ( $\text{Request}, T, i$ ): запрос на доступ к ресурсу, сделанный процессом  $P_i$ .
- ( $\text{Reply}, T, j$ ): процесс  $P_j$  предоставляет доступ к ресурсу, которым он управляет.
- ( $\text{Release}, T, k$ ): процесс  $P_k$  освобождает ресурс, который был выделен ему ранее.

А вот описание самого алгоритма.

1. Когда процессу  $P_i$  требуется доступ к ресурсу, он генерирует сообщение ( $\text{Request}, T, i$ ), отмеченное текущим значением локального счетчика. Это сообщение помещается в элемент  $q[i]$  собственного массива и рассыпается всем остальным процессам.

<sup>1</sup> В сообщениях: release — освобождение, reply — ответ, request — запрос. — Примеч. пер.

2. Когда процесс  $P_i$  получает сообщение  $(\text{Request}, T_j, i)$ , он помещает его в свой массив в элемент  $q[i]$ . Если элемент массива  $q[j]$  не содержит сообщение-запрос, процесс передает сообщение  $(\text{Reply}, T_j, j)$  обратно процессу  $P_i$ . Это действие и реализует описанное выше правило, которое гарантирует, что никакое отправленное ранее сообщение-запрос не пребывает в состоянии передачи в момент принятия решения о выделении ресурса.
3. Процесс  $P_i$  получает доступ к ресурсу (т.е. входит в свой критический участок), если одновременно соблюдаются два следующих условия.
  - a) Сообщение-запрос от  $P_i$  в массиве  $q$  является самым ранним из всех сообщений-запросов; поскольку сообщения согласованно упорядочиваются всеми системами, данное условие разрешает доступ к ресурсу в любой момент времени одному и только одному процессу.
  - b) Все остальные сообщения в локальном массиве отправлены позднее, чем сообщение, находящееся в элементе массива  $q[i]$ . Это правило гарантирует, что процессу  $P_i$  известны все запросы, предшествовавшие текущему запросу.
4. Процесс  $P_i$  освобождает ресурс, генерируя сообщение  $(\text{Release}, T_j, i)$ , которое помещается в его собственный массив и передается всем остальным процессам.
5. Когда процесс  $P_i$  получает сообщение  $(\text{Release}, T_j, j)$ , он заменяет текущее содержимое элемента массива  $q[j]$  этим сообщением.
6. Когда процесс  $P_i$  получает сообщение  $(\text{Reply}, T_j, j)$ , он заменяет текущее содержимое элемента массива  $q[j]$  этим сообщением.

Нетрудно показать, что данный алгоритм обеспечивает взаимное исключение, действует беспристрастно, избегает взаимоблокировок и голодания.

- **Взаимное исключение.** Запросы на вход в критический участок обрабатываются в соответствии с упорядочением сообщений, налагаемым механизмом присваивания отметок времени. Как только процесс  $P_i$  решит войти в свой критический участок, в системе не может быть других сообщений-запросов, переданных прежде его собственного сообщения. Это действительно так, поскольку процесс  $P_i$  к тому времени должен был получить от всех других систем сообщения с более старой отметкой времени, чем его собственное сообщение-запрос. Уверенность в этом нам дает механизм ответных сообщений. Напомним, что обмен сообщениями между системами не может происходить не по порядку.
- **Беспристрастность.** Запросы удовлетворяются строго на основании упорядочения по отметкам времени. Следовательно, все процессы обладают равными возможностями.
- **Недопущение взаимоблокировки.** Упорядочение по отметкам времени согласованно поддерживается во всех системах, поэтому взаимоблокировка невозможна.
- **Отсутствие голодания.** Как только процесс  $P_i$  завершит выполнение своего критического участка и выйдет из него, он передаст освобождающее сообщение. В итоге во всех остальных системах сообщение-запрос от процесса  $P_i$  будет удалено, и благодаря этому некоторый другой процесс сможет войти в свой критический участок.

В качестве меры эффективности данного алгоритма заметим, что для гарантии взаимного исключения требуется  $3 \times (N-1)$  сообщений:  $(N-1)$  сообщений-запросов,  $(N-1)$  сообщений-ответов и  $(N-1)$  освобождающих сообщений.

### **Вторая версия алгоритма**

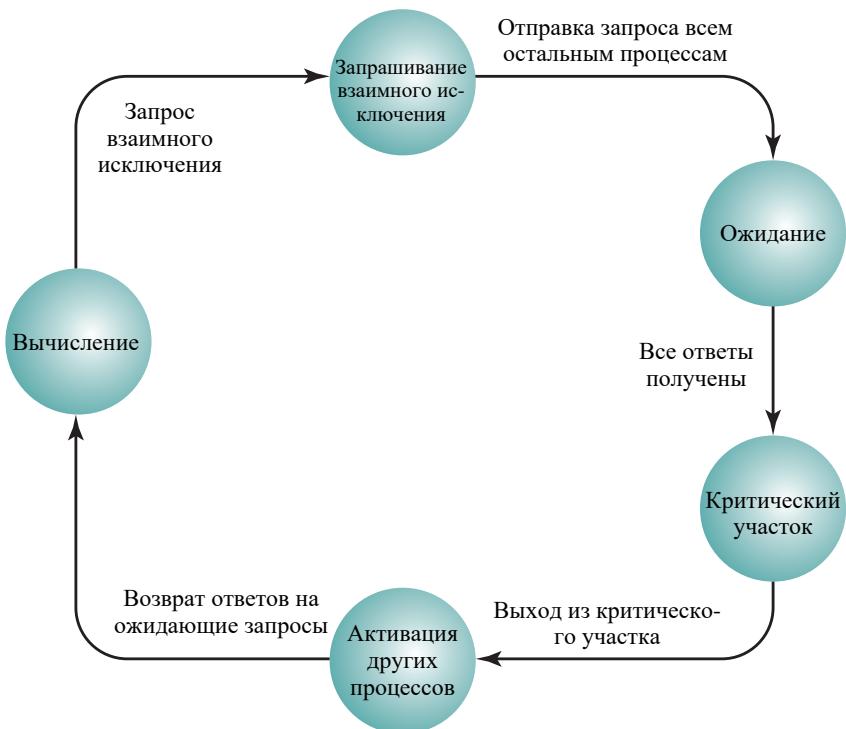
В [201] было предложено уточнение описанного выше алгоритма Лэмпорта. Это попытка оптимизировать первоначальную версию данного алгоритма, исключив из него освобождающие сообщения. Допущения остаются такими же, как и прежде, за исключением того, что сообщения, посланные одним процессом другому, не обязаны быть получены в том порядке, в котором они были отправлены.

Как и прежде, в каждой системе выполняется процесс, управляющий выделением ресурсов. Этот процесс поддерживает массив  $q$  и подчиняется следующим правилам.

1. Когда процессу  $P_i$  требуется доступ к ресурсу, он генерирует сообщение  $(\text{Request}, T_i, i)$ , помечает его текущим значением локального счетчика времени и отсылает всем остальным процессам, помещая также в элемент  $q[i]$  своего массива.
2. Когда процесс  $P_j$  принимает сообщение  $(\text{Request}, T_i, i)$ , он следует перечисленным далее правилам.
  - a) Если в настоящий момент процесс  $P_j$  находится в своем критическом участке, он откладывает отправку ответного сообщения.
  - б) Если процесс  $P_j$  не ожидает вход в свой критический участок, т.е. он не выдавал запрос, который все еще ожидает обработки, то он передает процессу  $P_i$  сообщение  $(\text{Reply}, T_j, j)$ .
  - в) Если процесс  $P_j$  ожидает входа в свой критический участок и если входящее сообщение следует после запроса от процесса  $P_i$ , он сохраняет это сообщение в элементе  $q[i]$  своего массива и откладывает отправку ответного сообщения.
  - г) Если процесс  $P_j$  ожидает входа в свой критический участок и если входящее сообщение предшествует запросу от процесса  $P_i$ , он сохраняет это сообщение в элементе  $q[i]$  своего массива и передает сообщение  $(\text{Reply}, T_j, j)$  процессу  $P_i$ .
3. Процесс  $P_i$  может получить доступ к ресурсу (т.е. войти в свой критический участок), как только получит ответное сообщение от всех других процессов.
4. Когда процесс  $P_i$  покидает свой критический участок, он освобождает ресурс, отсылая ответное сообщение на каждый ожидающий запрос.

Диаграмма переходов каждого процесса для данного алгоритма приведена на рис. 19.10.

Резюмируем: когда процессу требуется войти в свой критический участок, он отсылает сообщение-запрос с отметкой времени всем остальным процессам. И как только данный процесс получает ответ от всех остальных процессов, он может войти в свой критический участок. Когда процесс получает запрос от другого процесса, он в конечном итоге должен отправить соответствующий ответ. Если процессу не требуется войти в критический участок, он отправляет ответ сразу же. Если же ему требуется войти в свой критический участок, он сравнивает отметку времени своего запроса с отметкой времени последнего полученного им запроса. И если последняя отметка времени окажется более поздней, он отложит свой ответ, в противном случае отправляя его сразу же.



**Рис. 19.10.** Диаграмма перехода в разные состояния для алгоритма, предложенного в [201]

Данному алгоритму требуется  $2 \times (N-1)$  сообщений:  $(N-1)$  сообщений-запросов, чтобы обозначить намерение процесса  $P$ , войти в свой критический участок, и  $(N-1)$  сообщений-ответов, чтобы разрешить запрашиваемый им доступ.

Благодаря механизму присваивания отметок времени в данном алгоритме соблюдаются взаимное исключение, а также исключается взаимоблокировка. Чтобы убедиться в последнем, допустим противоположное: когда в состоянии передачи больше не пребывает ни одно сообщение, то возможна такая ситуация, когда каждый процесс передал запрос и не получил необходимый ему ответ. Но такая ситуация не может возникнуть, поскольку решение отложить ответ зависит от упорядочения запросов. Таким образом, имеется один запрос с самой ранней отметкой времени, по которому будут получены все требующиеся ответы, а следовательно, взаимоблокировка невозможна.

Голодание также исключается, поскольку запросы упорядочиваются. Они обслуживаются по порядку, и поэтому каждый запрос на определенной стадии становится самым ранним и будет обслужен.

## Метод передачи эстафеты

Ряд исследователей предложили совершенно иной подход к взаимному исключению, который подразумевает передачу эстафеты среди участвующих в ней процессов. Эстафета представляет собой объект, который удерживается в любой момент времени одним процессом. Тот процесс, который удерживает эстафету, может войти в свой критический

участок, не запрашивая разрешения. Когда же процесс покидает свой критический участок, он передает эстафету другому процессу.

В этом разделе рассматривается один из самых эффективных алгоритмов передачи эстафеты. Он был впервые предложен в [248]; логически равнозначное предложение появилось в [202]. Для реализации такого алгоритма требуются две структуры данных. В частности, эстафета, передаваемая от одного процесса к другому, фактически является массивом, в  $k$ -м элементе которого хранится отметка момента времени, когда данная эстафета в последний раз передавалась процессу  $P_k$ . Кроме того, в каждом процессе ведется массив запросов, в  $j$ -м элементе которого хранится момент времени, когда запрос в последний раз был получен от процесса  $P_j$ .

Процедура передачи эстафеты следующая. Первоначально эстафета произвольно назначается одному из процессов. Когда процессу требуется войти в свой критический участок, он может сделать это, если в настоящий момент владеет эстафетой. В противном случае он рассыпает запросное сообщение с отметкой времени всем остальным процессам и ожидает до тех пор, пока получит эстафету. Когда процесс  $P_j$  покидает свой критический участок, он должен передать эстафету какому-нибудь другому процессу. С этой целью он выбирает процесс для получения эстафеты, производя в порядке  $j+1, j+2, \dots, 1, 2, \dots, j-1$  поиск в массиве запросов первой записи запроса  $k$ , такой, чтобы отметка времени последнего запроса эстафеты от процесса  $P_k$  была больше, чем значение, записанное в эстафете, когда она в последний раз удерживалась процессом  $P_k$ .

Реализация рассматриваемого здесь алгоритма представлена на рис. 19.11 и состоит из двух частей. Первая часть, в которой используется критический участок, состоит из пролога, критического участка и эпилога; вторая часть содержит действие, которое должно быть выполнено после получения запроса. В переменной `clock` хранится значение локального счетчика времени, используемого для выполнения функции, присваивающей отметки времени. Операция `wait(access, token)` заставляет процесс ожидать до тех пор, пока будет получено сообщение о доступе, которое затем сохраняется в массиве переменных `token`.

Данному алгоритму требуется одно из двух:

- $N$  сообщений ( $N-1$  для рассылки запроса и 1 — для передачи эстафеты), когда запрашивающий процесс не удерживает эстафету;
- никаких сообщений, если процесс уже удерживает эстафету.

## 19.4. РАСПРЕДЕЛЕННАЯ ВЗАИМОБЛОКИРОВКА

Взаимоблокировка была определена в главе 6, “Параллельные вычисления: взаимоблокировка и голодание”, как перманентная блокировка ряда процессов, которые соперничают за системные ресурсы или обмениваются данными. Такое определение корректно как для единой системы, так и для распределенной. Как и взаимное исключение, взаимоблокировка вызывает в распределенной системе больше осложнений, чем в системе с общей памятью. Обработка взаимоблокировки в распределенной системе затруднена потому, что ни одному из узлов неизвестно текущее состояние всей системы в целом, а также потому, что каждый обмен сообщениями между процессами подразумевает непредсказуемую задержку.

```

if (!token_present) {
    clock++;           /* Пролог */
    broadcast (Request, clock, i);
    wait (access, token);
    token_present = true;
}

token_held = true;
<Критический участок>

token[i] = clock;      /* Эпилог */
token_held = false;
for (int j = i + 1; j < n; j++) {
    if (request(j) > token[j] && token_present) {
        token_present = false;
        send (access, token[j]);
    }
}
}

```

### a) Первая часть

```

if (received (Request, k, j)) {
    request (j) = max(request(j), k);
    if (token_present && !token_held)
        <Текст эпилога>;
}

```

### б) Вторая часть

Обозначения	
send(j, access, token)	Отправить из процесса $j$ сообщение с эстафетой о доступе типа $access$
broadcast(request, clock, i)	Отправить из процесса $i$ сообщение-запрос $request$ с отметкой времени $clock$ всем остальным процессам
received(request, t, j)	Получить от процесса $j$ сообщение-запрос $request$ с отметкой времени $t$

Рис. 19.11. Алгоритм передачи эстафеты для процесса  $P_j$

В литературе основное внимание получили два вида распределенной взаимоблокировки: возникающая при выделении ресурсов и возникающая при обмене сообщениями. При взаимоблокировке из-за ресурсов процессы пытаются получить доступ к таким ресурсам, как объекты данных в базе данных или ресурсы ввода-вывода на сервере. Взаимоблокировка осуществляется в случае, когда каждый процесс из некоторого множества процессов запрашивает ресурс, который захвачен другим процессом из того же множества. При взаимоблокировке обмена сообщениями ресурсами являются сообщения, которых ожидают процессы. Взаимоблокировка происходит в том случае, если каждый процесс из одного множества ожидает сообщение от другого процесса множества и в результате ни один из процессов вообще не отсылает ожидаемое сообщение.

## Взаимоблокировка при распределении ресурсов

Как упоминалось в главе 6, “Параллельные вычисления: взаимоблокировка и голодание”, взаимоблокировка возникает при распределении ресурсов в том случае, если удовлетворяются все перечисленные ниже условия.

- **Взаимное исключение.** Одновременно пользоваться ресурсом может лишь один процесс. Ни один процесс не может получить доступ к единице ресурса, выделенной другому процессу.
- **Удержание и ожидание.** Процесс может удерживать выделенные ему ресурсы, ожидая в то же время назначения других ресурсов.
- **Никакого вытеснения.** Ни один из ресурсов не может быть принудительно удален из удерживающего его процесса.
- **Циклическое ожидание.** Существует такой цикл процессов, в котором каждый из них удерживает хотя бы один ресурс, требующийся следующему в цикле процессу.

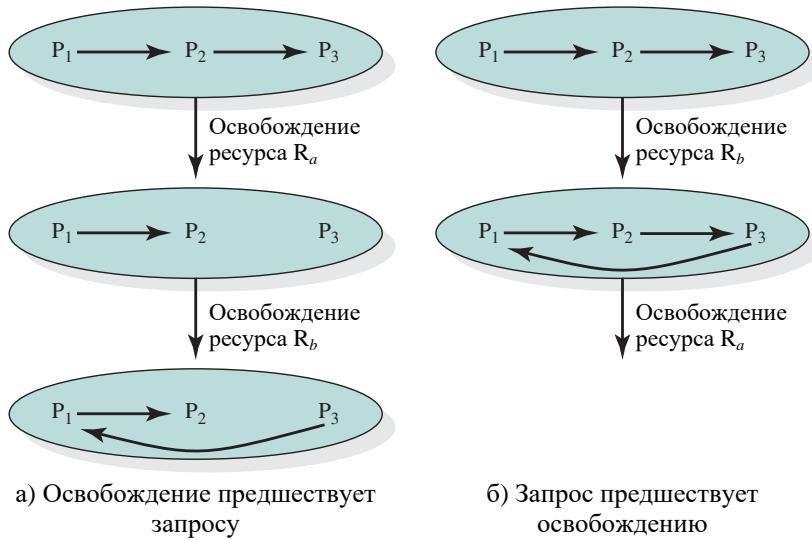
Цель алгоритма, работающего с взаимоблокировками, состоит в том, чтобы предотвратить образование циклического ожидания или обнаружить его фактическое или потенциальное возникновение. В распределенной системе ресурсы распределяются между различными системами, а доступ к ним регулируется управляющими процессами, которым глобальное состояние системы полностью не известно, и потому им приходится принимать решения, основываясь на локальной информации. Таким образом, возникает потребность в новых алгоритмах для исключения взаимоблокировки.

Одним из примеров трудностей, возникающих при распределенных взаимоблокировках, служит явление фантомной взаимоблокировки. Наглядный пример фантомной взаимоблокировки приведен на рис. 19.12, где  $P_1 \rightarrow P_2 \rightarrow P_3$ , означает, что процесс  $P_1$  остановлен в ожидании ресурса, удерживаемого процессом  $P_2$ , а тот, в свою очередь, — в ожидании ресурса, удерживаемого процессом  $P_3$ . Допустим, что вначале процесс  $P_3$  владеет ресурсом  $R_a$ , а процесс  $P_1$  — ресурсом  $R_b$ . Теперь допустим, что процесс  $P_3$  отсылает сначала первое сообщение, освобождая ресурс  $R_a$ , а затем — второе сообщение, запрашивая ресурс  $R_b$ . Если первое сообщение достигнет процесса, обнаруживающего циклическое ожидание, раньше второго, то в конечном итоге возникнет последовательность, приведенная на рис. 19.12, а, корректно отражающая потребности в ресурсах. Но если второе сообщение поступит до первого, то будет зарегистрирована взаимоблокировка, что показано на рис. 19.12, б. Это ложное срабатывание, а не настоящая взаимоблокировка, связанное с недостатком сведений о глобальном состоянии.

### Предотвращение взаимоблокировки

В главе 6, “Параллельные вычисления: взаимоблокировка и голодание”, рассматривались два метода предотвращения взаимоблокировки, которые могут быть использованы в распределенной среде и кратко описаны ниже.

1. **Циклическое ожидание** может быть предотвращено, если определить линейное упорядочение типов ресурсов. Так, если процессу выделены ресурсы типа R, он может в дальнейшем запросить ресурсы только тех типов, которые следуют по порядку после ресурсов типа R. Главный недостаток такого метода заключается в том, что ресурсы могут запрашиваться не в том порядке, в котором используются, а следовательно, могут удерживаться дольше, чем нужно.

**Рис. 19.12.** Фантомная блокировка

2. Возникновение условия удержания и ожидания может быть предотвращено, если потребовать, чтобы процесс запрашивал сразу все необходимые ему ресурсы, а также блокировать процесс до тех пор, пока все запросы не смогут быть обработаны одновременно. Такой метод оказывается неэффективным по двум причинам. Во-первых, процесс может быть остановлен надолго, ожидая обработки всех своих запросов, в то время как на самом деле он мог бы пользоваться лишь некоторыми ресурсами. И во-вторых, ресурсы, выделяемые процессу, могут оставаться неиспользуемыми в течение значительного периода времени, в то время как другим процессам в них будет отказано.

В обоих упомянутых выше методах требуется, чтобы процесс заранее определил свои потребности в ресурсах, что не всегда возможно. Примером тому служит приложение базы данных, в котором новые элементы могут добавляться динамически. В качестве примеров подхода, не требующего такого предвидения, рассмотрим два алгоритма, предложенные в [211]. Эти алгоритмы были разработаны в контексте работы базы данных, и потому далее речь пойдет о транзакциях, а не о процессах.

В обоих предложенных алгоритмах используются отметки времени. Каждая транзакция несет в течение всего срока своего существования отметку времени своего создания. Тем самым устанавливается строгое упорядочение транзакций. Так, если ресурс  $R$ , уже используемый в транзакции  $T_1$ , запрашивается в транзакции  $T_2$ , то конфликт разрешается сравнением их отметок времени. Благодаря такому сравнению предотвращается образование циклического ожидания. Авторами были предложены две разновидности этого основного алгоритма, называемые методами “ожидания-смерти” и “ранения-ожидания”.

Допустим, что в настоящий момент транзакция  $T_1$  удерживает ресурс  $R$ , а транзакция  $T_2$  выдает запрос. На рис. 19.13, а приведена реализация метода ожидания-смерти (wait-die method), применяемого распределителем ресурсов в системе ресурса  $R$ . Отметки времени обеих транзакций обозначены как  $e(T_1)$  и  $e(T_2)$ . Если транзакция  $T_2$  старше, она блокируется до тех пор, пока транзакция  $T_1$  не освободит ресурс  $R$ , выдав сообщение

об его освобождении, или же она будет “уничтожена” при запросе другого ресурса. Если транзакция T2 моложе, то она перезапускается, но с прежней отметкой времени.

Таким образом, при возникновении конфликта приоритет отдается более старшей транзакции. Поскольку уничтоженная транзакция восстанавливается со своей первоначальной отметкой времени, она становится более старшей, а потому ее приоритет растет. Все, что нужно, — это присвоить отметки времени транзакциям, запрашивающим свои ресурсы.

```
if (e(T2) < e(T1))
    halt_T2 ('wait');
else
    kill_T2 ('die');
```

а) Метод ожидания-смерти

```
if (e(T2) < e(T1))
    kill_T1 ('wound');
else
    halt_T2 ('wait');
```

б) Метод ранения-ожидания

**Рис. 19.13.** Методы предотвращения взаимоблокировок

**Метод ранения-ожидания** (wound-wait method) сразу же обрабатывает запрос от более старой транзакции, уничтожая более молодую транзакцию, пользующуюся требующимся ресурсом, как показано на рис. 19.13, б. В отличие метода ожидания-смерти, транзакции вообще не приходится ждать освобождения ресурса, используемого более молодой транзакцией.

### Исключение взаимоблокировки

Это метод, который позволяет динамически выяснить, может ли запрос на выделение заданного ресурса при его обработке привести к взаимоблокировке. Но, как указывается в [232], исключение распределенной взаимоблокировки непрактично по следующим причинам.

1. Каждый узел должен отслеживать глобальное состояние системы, а для этого требуются значительные издержки на хранение и обмен данными.
2. Процесс проверки безопасности глобального состояния должен быть взаимно исключающим, иначе два узла могут рассмотреть запрос ресурса из разных процессов и одновременно прийти к выводу, что запрос ресурса будет безопасен, в то время как такое решение в обоих узлах сразу же приведет к взаимоблокировке.
3. Для проверки безопасности состояния в распределенной системе с большим числом процессов и ресурсов требуются значительные издержки на обработку.

### Обнаружение взаимоблокировки

В этом случае процессам разрешается свободно получать ресурсы, как им угодно, а существование взаимоблокировки определяется *post factum*. Если же взаимоблокировка обнаружена, то выбирается один из *составляющих* ее процессов, и от него требуется освободить захваченные ресурсы, чтобы снять взаимоблокировку.

Трудности обнаружения распределенной взаимоблокировки состоят в том, что каждой системе известны лишь ее собственные ресурсы, тогда как во взаимоблокировку могут быть вовлечены ресурсы распределенные. Разрешить подобные трудности можно несколькими способами, в зависимости от характера управления системой: централизованного, иерархического или распределенного (табл. 19.1).

**Таблица 19.1. СТРАТЕГИИ ОБНАРУЖЕНИЯ РАСПРЕДЕЛЕННОЙ ВЗАИМОБЛОКИРОВКИ**

Централизованные алгоритмы		Иерархические алгоритмы		Распределенные алгоритмы	
Преимущества	Недостатки	Преимущества	Недостатки	Преимущества	Недостатки
<ul style="list-style-type: none"> <li>• Алгоритмы концептуально просты и легко реализуемы</li> <li>• Центральный узел обладает всей информацией и может оптимально разрешать взаимоблокировки</li> </ul>	<ul style="list-style-type: none"> <li>• Значительные издержки на обмен данными; каждый узел должен посыпать сведения о своем состоянии о центральному узлу</li> <li>• Уязвимы к отказам Центрального узла</li> </ul>	<ul style="list-style-type: none"> <li>• Неуязвимы к единой точке отказа</li> <li>• Действия по разрешению взаимоблокировок ограничены, если большая часть потенциальных взаимоблокировок относительно локализована</li> </ul>	<ul style="list-style-type: none"> <li>• Может быть трудно сконфигурировать систему так, чтобы локализовать большинство потенциальных взаимоблокировок в противном случае издержки могут фактически оказаться более существенными, чем при распределенном подходе</li> </ul>	<ul style="list-style-type: none"> <li>• Неуязвимы к единой точке отказа</li> <li>• Ни один из узлов не налагается бремя обнаружения взаимоблокировок</li> </ul>	<ul style="list-style-type: none"> <li>• Разрешение взаимоблокировок затруднено тем, что несколько узлов могут обнаружить одну и ту же взаимоблокировку и не знать, что в ней вовлечены другие узлы</li> <li>• Алгоритмы трудно разрабатывать из-за проблем синхронизации</li> </ul>

При централизованном управлении за обнаружение взаимоблокировки отвечает один узел. Все сообщения запросов и освобождений отсылаются центральному процессу, а также тому процессу, который управляет конкретным ресурсом. А поскольку полное представление о том, что происходит в системе, имеется только у центрального процесса, то именно он в состоянии обнаружить взаимоблокировку. Такой метод требует обильного обмена сообщениями и уязвим к отказам центрального узла. Кроме того, могут быть обнаружены фантомные взаимоблокировки.

При иерархическом управлении узлы организованы в древовидную структуру, в которой один узел служит в качестве корня дерева. В каждом узле, кроме листьев, собираются сведения о распределении ресурсов из всех дочерних узлов. Это позволяет обнаруживать взаимоблокировку на более низких уровнях, чем корневой узел. В частности, взаимоблокировка, в которую вовлечен ряд ресурсов, будет обнаружена узлом, являющимся общим предком для всех узлов, за ресурсы которых соперничают вступившие в конфликт объекты.

При распределенном управлении в обнаружении взаимоблокировки принимают участие все процессы. В общем случае это означает необходимость значительного обмена данными с отметками времени, а следовательно, и значительные издержки. В [199] перечисляется целый ряд методов, основанных на распределенном управлении, а подробное исследование каждого из них приведено в [58].

А теперь рассмотрим пример алгоритма обнаружения распределенной взаимоблокировки, описанного в [59] и [119]. Этот алгоритм применяется в распределенной СУБД, где каждый узел ведет часть базы данных и может инициировать транзакцию. У транзакции может быть хотя бы один ожидающий обработки запрос ресурса. Если же транзакции требуется больше одного объекта данных, то второй объект данных может быть запрошен только после предоставления первого объекта данных.

С каждым объектом данных  $i$  в узле связаны два параметра: уникальный идентификатор  $D_i$  и переменная  $\text{Locked\_by}(D_i)$ . Эта переменная имеет нулевое значение  $\text{nil}$ , если объект данных не заблокирован ни одной из транзакций. В противном случае ее значением является идентификатор блокирующей транзакции.

С каждой транзакцией  $j$  в узле связаны четыре параметра, перечисленные ниже.

- Уникальный идентификатор  $T_j$ .
- Переменная  $\text{Held\_by}(T_j)$ , которая получает значение  $\text{nil}$ , если транзакция  $T_j$  выполняется или находится в состоянии готовности. В противном случае значением этой переменной является транзакция, удерживающая объект, который запрошен транзакцией  $T_j$ .
- Переменная  $\text{Wait\_for}(T_j)$ , которая имеет значение  $\text{nil}$ , если транзакция  $T_j$  не ожидает никакой другой транзакции. В противном случае значением этой переменной является идентификатор транзакции, находящейся в начале упорядоченного списка заблокированных транзакций.
- Очередь  $\text{Request\_Q}(T_j)$ , содержащая все ожидающие обработки запросы объектов данных, удерживаемых транзакцией  $T_j$ . Каждый элемент такой очереди представлен в виде  $(T_k, D_k)$ , где  $T_k$  — запрашивающая транзакция, а  $D_k$  — объект данных, удерживаемый транзакцией  $T_j$ .

Предположим, например, что транзакция  $T_2$  ожидает объект данных, удерживаемый транзакцией  $T_1$ , которая, в свою очередь, ожидает объект данных, удерживаемый транзакцией  $T_0$ . В таком случае соответствующие параметры транзакций будут иметь приведенные ниже значения.

Транзакция	Wait_for	Held_by	Request_Q
$T_0$	nil	nil	$T_1$
$T_1$	$T_0$	$T_0$	$T_2$
$T_2$	$T_0$	$T_1$	nil

В данном примере подчеркивается различие  $\text{Wait\_for}(T_i)$  и  $\text{Held\_by}(T_i)$ . Ни один из процессов не может быть продолжен до тех пор, пока транзакция  $T_0$  не освободит объект данных, требующийся транзакции  $T_1$ , которая может затем выполниться и освободить объект данных, требующийся транзакции  $T_2$ .

На рис. 19.14 приведена реализация алгоритма, применяемого для обнаружения взаимоблокировки. Когда транзакция делает запрос на блокировку объекта данных, серверный процесс, связанный с этим объектом данных, принимает или отклоняет такой запрос. Если запрос не принят, серверный процесс возвращает идентификатор транзакции, удерживающей объект данных.

Когда запрашивающая транзакция принимает предоставленный ответ, она блокирует объект данных. В противном случае запрашивающая транзакция обновляет свою переменную  $\text{Held\_by}$  идентификатором транзакции, удерживающей объект данных. Она вносит свой идентификатор в очередь  $\text{Request\_Q}$  транзакции, удерживающей объект данных. Кроме того, она обновляет свою переменную  $\text{Wait\_for}$  идентификатором транзакции, удерживающей объект данных (если эта транзакция не находится в состоянии ожидания), или же идентификатором, хранящимся в переменной  $\text{Wait\_for}$  транзакции, удерживающей объект данных. Таким образом, в переменной  $\text{Wait\_for}$  устанавливается значение транзакции, которая в конечном итоге блокирует выполнение. И наконец, запрашивающая транзакция отсылает обновляющее сообщение всем транзакциям в своей очереди  $\text{Request\_Q}$ , чтобы модифицировать все переменные  $\text{Wait\_for}$ , которые затрагивает такое изменение.

Транзакция, получив обновляющее сообщение, обновляет свою переменную  $\text{Wait\_for}$ , чтобы отразить тот факт, что транзакция, которой она в конечном итоге дождалась, теперь заблокирована другой транзакцией. Затем она выполняет конкретные действия по обнаружению взаимоблокировки, проверяя, не ожидает ли она теперь один из процессов, в свою очередь, ожидающих ее. Если это не так, она пересыпает обновляющее сообщение; в противном случае транзакция отсылает очищающее сообщение транзакции, удерживающей запрашиваемый ею объект данных, выделяя каждый удерживаемый ею объект первой запрашивающей транзакции в ее очереди  $\text{Request\_Q}$  и ставя остальные запрашиваемые транзакции в очередь новой транзакции.

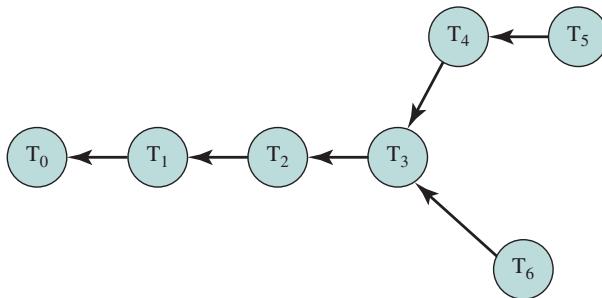
Пример работы данного алгоритма приведен на рис. 19.15. Когда транзакция  $T_0$  делает запрос объекта данных, удерживаемого транзакцией  $T_3$ , то образуется цикл. Транзакция  $T_0$  отсылает обновляющее сообщение, распространяющееся от транзакции  $T_1$  к транзакции  $T_2$  и далее к транзакции  $T_3$ . В этот момент транзакция  $T_3$  обнаруживает, что пересечение ее переменных  $\text{Wait\_for}$  и  $\text{Request\_Q}$  не пусто. Поэтому транзакция  $T_3$  отсылает очищающее сообщение транзакции  $T_2$ , чтобы удалить из своей очереди элемент  $\text{Request\_Q}(T_2)$ , а затем освобождает удерживаемые ею объекты данных, активируя транзакции  $T_4$  и  $T_6$ .

```

/* Объект данных Dj принимающий a lock_request(Ti) */
if (Locked_by(Dj) == null)
    send(granted);
else {
    send not granted to Ti;
    send Locked_by(Dj) to Ti
}
/* Транзакция Ti делает запрос на блокировку объекта данных Dj */
send lock_request(Ti) to Dj;
wait for granted/not granted;
if (granted) {
    Locked_by(Dj) = Ti;
    Held_by(Ti) = f;
}
else { /* Предполагаем, что объект данных Dj
используется транзакцией Tj */
    Held_by(Ti) = Tj;
    Enqueue(Ti, Request_Q(Tj));
    if (Wait_for(Tj) == null)
        Wait_for(Ti) = Tj ;
    else
        Wait_for(Ti) = Wait_for(Tj);
        update(Wait_for(Ti), Request_Q(Ti));
}
/* Транзакция Tj принимающая обновляющее сообщение */
if (Wait_for(Tj) != Wait_for(Ti))
    Wait_for(Tj) = Wait_for(Ti);
if (intersect(Wait_for(Tj), Request_Q(Tj)) = null)
    update(Wait_for(Ti), Request_Q(Tj));
else {
    DECLARE DEADLOCK;
    /* Инициирование разрешения взаимоблокировки следующим образом:
       Транзакция Tj выбирается как аварийно прерываемая
       Транзакция Tj освобождает все удерживаемые ею объекты данных */
    send_clear(Tj, Held_by(Tj));
    Выделение каждого объекта данных Di,
    удерживаемого транзакцией Tj,
    первой запрашивающей транзакции Tk в Request_Q(Tj);
    for (каждой транзакции Tn в Request_Q(Tj),
          запрашивающей объект данных Di,
          удерживаемый транзакцией Tj)
    {
        Enqueue(Tn, Request_Q(Tk));
    }
}
/* Транзакция Tk принимающая сообщение clear(Tj, Tk) */
Очистить кортеж Tj в качестве запрашивающей
транзакции from Request_Q(Tk);

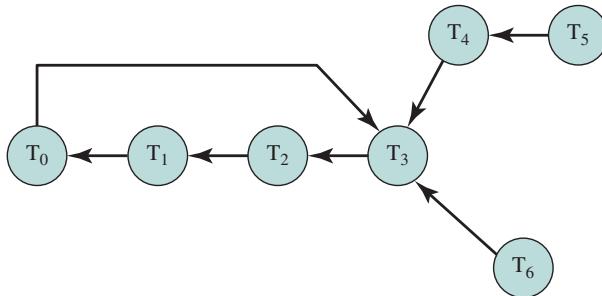
```

Рис. 19.14. Алгоритм обнаружения распределенной взаимоблокировки



Транзакция	Wait_for	Held_by	Request_Q
T <sub>0</sub>	nil	nil	T <sub>1</sub>
T <sub>1</sub>	T <sub>0</sub>	T <sub>0</sub>	T <sub>2</sub>
T <sub>2</sub>	T <sub>0</sub>	T <sub>1</sub>	T <sub>3</sub>
T <sub>3</sub>	T <sub>0</sub>	T <sub>2</sub>	T <sub>4</sub> , T <sub>6</sub>
T <sub>4</sub>	T <sub>0</sub>	T <sub>3</sub>	T <sub>5</sub>
T <sub>5</sub>	T <sub>0</sub>	T <sub>4</sub>	nil
T <sub>6</sub>	T <sub>0</sub>	T <sub>3</sub>	nil

а) Состояние системы перед запросом



Транзакция	Wait_for	Held_by	Request_Q
T <sub>0</sub>	T <sub>0</sub>	T <sub>3</sub>	T <sub>1</sub>
T <sub>1</sub>	T <sub>0</sub>	T <sub>0</sub>	T <sub>2</sub>
T <sub>2</sub>	T <sub>0</sub>	T <sub>1</sub>	T <sub>3</sub>
T <sub>3</sub>	T <sub>0</sub>	T <sub>2</sub>	T <sub>4</sub> , T <sub>6</sub> , T <sub>0</sub>
T <sub>4</sub>	T <sub>0</sub>	T <sub>3</sub>	T <sub>5</sub>
T <sub>5</sub>	T <sub>0</sub>	T <sub>4</sub>	NIL
T <sub>6</sub>	T <sub>0</sub>	T <sub>3</sub>	NIL

б) Состояние системы после запроса, направленного транзакцией T<sub>3</sub> транзакции T<sub>3</sub>

**Рис. 19.15.** Пример работы алгоритма обнаружения распределенной взаимоблокировки, реализация которого приведена на рис. 19.14

## Взаимоблокировка при обмене сообщениями

### Взаимное ожидание

В этом случае взаимоблокировка происходит при обмене сообщениями, когда каждая группа процессов ожидает сообщение от другого члена группы, но никаких сообщений в состоянии передачи нет.

Чтобы проанализировать эту ситуацию более подробно, определим множество зависимостей (dependence set — DS) от процесса. Так, если процесс  $P_i$  остановлен в ожидании сообщения, то множество его зависимостей  $DS(P_i)$  состоит из всех процессов, от которых он ожидает сообщение. Как правило, процесс  $P_i$  может продолжаться, если поступит любое из ожидаемых им сообщений. В альтернативной формулировке процесс  $P_i$  может продолжаться, когда поступят все ожидаемые им сообщения. Более типична первая ситуация, и поэтому здесь рассматривается именно она.

С учетом представленного определения взаимоблокировку в множестве процессов  $S$  можно описать следующим образом.

1. Все процессы в множестве  $S$  остановлены в ожидании сообщений.
2. Множество  $S$  содержит множество зависимостей всех процессов из множества  $S$ .
3. Ни одно из сообщений не пребывает в состоянии передачи между членами множества  $S$ .

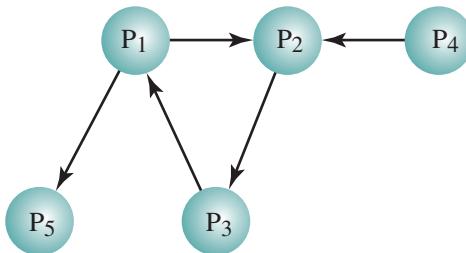
Любой процесс из множества  $S$  находится в состоянии взаимоблокировки, поскольку он может никогда не получить сообщение, которое его освободит.

В графическом представлении имеется отличие взаимоблокировки при обмене сообщениями от взаимоблокировки при выделении ресурсов. В случае выделения ресурсов взаимоблокировка возникает в том случае, когда в графе, отображающем зависимости процессов, имеется замкнутая петля, или цикл. В этом случае один процесс зависит от другого процесса, если последний удерживает ресурс, который требуется первому. Взаимоблокировка при обмене сообщениями возникает при условии, что все преемники любого члена множества  $S$  сами находятся в этом множестве.

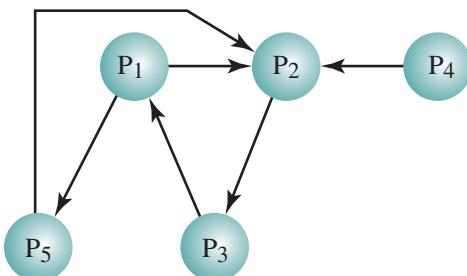
Упомянутое выше отличие наглядное показано на рис. 19.16, *a* показано, что процесс  $P_1$  ожидает сообщение от процесса  $P_2$  или  $P_5$ . Процесс  $P_5$  не ожидает никаких сообщений, поэтому может отправить сообщение процессу  $P_1$ , который, таким образом, освобождается. В итоге связи  $(P_1, P_5)$  и  $(P_1, P_2)$  удаляются. На рис. 19.16, *b* показано добавление зависимости: процесс  $P_5$  ожидает сообщение от процесса  $P_2$ , а тот — от процесса  $P_3$ , который, в свою очередь, ожидает сообщение от процесса  $P_1$ , а тот — от процесса  $P_2$ . Таким образом, возникает взаимоблокировка. Способом борьбы как с взаимоблокировкой при выделении ресурсов, так и с взаимоблокировкой при обмене сообщениями может быть предотвращение или обнаружение. Некоторые примеры таких мер борьбы с взаимоблокировкой приведены в [199].

### Недоступность буферов сообщений

Еще один способ возникновения взаимоблокировки при обмене сообщениями связан с выделением буферов для хранения сообщений, находящихся в состоянии передачи. Такого рода взаимоблокировка давно известна в сетях передачи данных с коммутацией пакетов. Рассмотрим сначала эту проблему в контексте сетей передачи данных, а затем — с точки зрения распределенной операционной системы.



а) Взаимоблокировка отсутствует



б) Взаимоблокировка

**Рис. 19.16.** Взаимоблокировка при обмене сообщениями

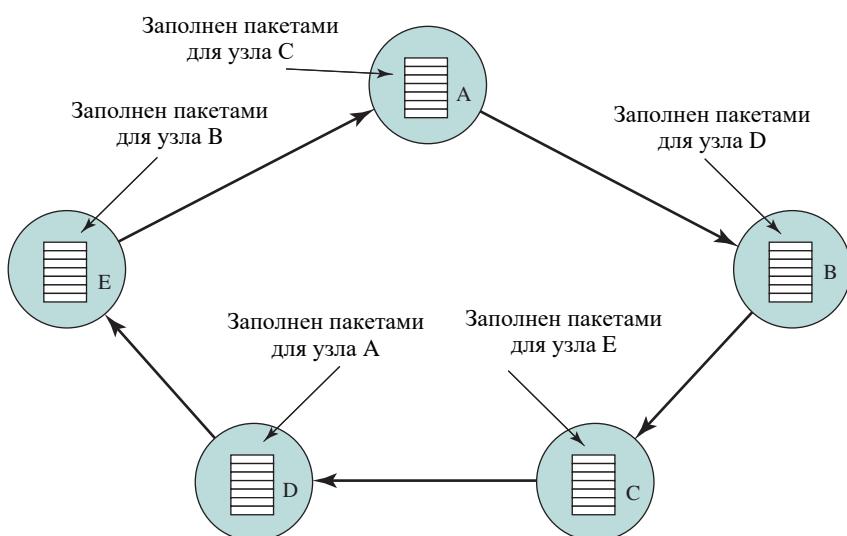
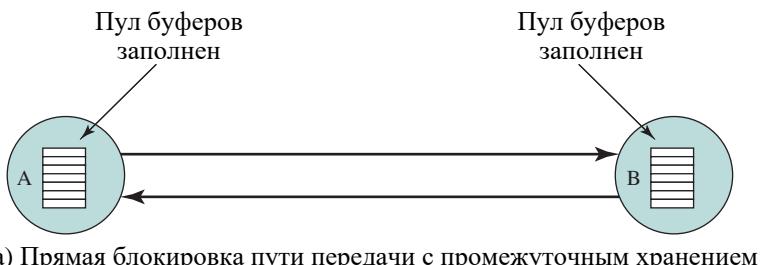
Простейшей формой взаимоблокировки в сети передачи данных является прямая блокировка пути передачи с промежуточным хранением, которая может возникнуть в том случае, если в узле коммутации пакетов применяется общий пул буферов, из которого буфера назначаются пакетам по требованию. На рис. 19.17, а показана ситуация, когда все буферное пространство в узле А занято пакетами, предназначенными для узла В. В узле В возникает та же ситуация, но для пакетов узла А. В результате ни один из этих узлов не может больше принимать пакеты, поскольку их буфера заполнены, так что ни один из них не может ни передавать, ни принимать данные по любому каналу связи.

Прямую блокировку пути передачи с промежуточным хранением можно предотвратить, не разрешая выделять все буфера одному каналу связи. Этого можно добиться, используя отдельные буфера фиксированного размера, по одному на каждый канал связи. Даже если используется общий пул буферов, взаимоблокировки можно избежать, если ни одному из каналов связи не разрешено захватывать все буферное пространство.

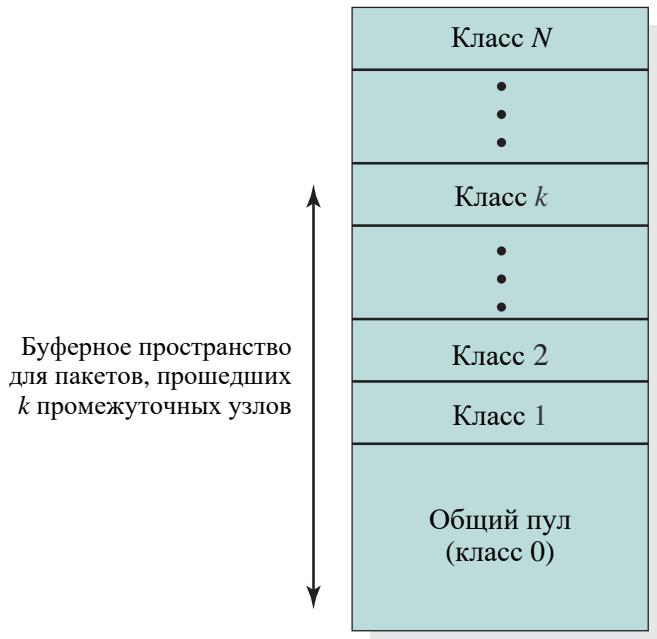
Более тонкая форма взаимоблокировки (косвенная блокировка пути передачи с промежуточным хранением) показана на рис. 19.17, б. В каждом узле очередь для соседнего узла в одном и том же направлении заполняется пакетами, предназначенными для следующего далее узла. Один из простых способов предотвратить такого рода взаимоблокировку — организовать иерархически структурированный пул буферов, показанный на рис. 19.18. Пул памяти на уровне 0 неограничен, в нем может быть сохранен любой входящий пакет. Начиная с уровня 1 до уровня  $N$  (где  $N$  — максимальное количество узлов на любом сетевом пути), буфера резервируются для тех пакетов, которые прошли до этого по меньшей мере  $k$  промежуточных узлов. Таким образом, в условиях большой

нагрузки буферы заполняются постепенно, от уровня 0 до уровня  $N$ . Если заполнены все буферы вплоть до уровня  $k$ , то поступающие пакеты, преодолевшие  $k$  или менее промежуточных узлов, отклоняются. Как показано в [91], такая стратегия исключает как прямые, так и косвенные блокировки путей передачи с промежуточным хранением.

Описанная выше проблема взаимоблокировки решается в контексте архитектуры связи и, как правило, на сетевом уровне. Схожая проблема может возникнуть и в распределенной операционной системе, где обмен сообщениями применяется для взаимодействия процессов. В частности, если операция отправки является неблокирующей, то для хранения исходящих сообщений требуется буфер. Буфер, предназначенный для хранения сообщений, отсылаемых от процесса X к процессу Y, можно рассматривать в качестве канала связи между процессами X и Y. Если емкость такого канала конечна (конечен размер буфера), то операция отправки может привести к приостановке процесса. Так, если имеется буфер размером  $n$  и в настоящий момент в состоянии передачи находятся  $n$  сообщений (т.е. они еще не получены целевым процессом), то выполнение дополнительной операции отправки приведет к блокировке отправляющего процесса до тех пор, пока не освободится место в буфере.

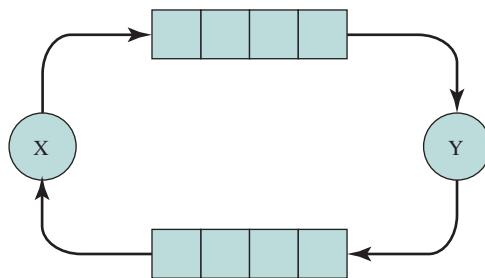


**Рис. 19.17.** Блокировка пути передачи с промежуточным хранением



**Рис. 19.18.** Структурированный пул буферов для предотвращения взаимоблокировки

На рис. 19.19 показано, каким образом применение конечных каналов может привести к взаимоблокировке. На рисунке имеются два канала, емкость каждого из которых рассчитана на четыре сообщения, причем один канал — от процесса X к процессу Y, а другой — от процесса Y к процессу X. Если в состоянии передачи по каждому каналу пребывают по четыре сообщения, и оба процесса, X и Y, попытаются передать следующие сообщения, прежде чем получить предыдущие, то оба они приостановятся, и в конечном счете возникнет взаимоблокировка.



**Рис. 19.19.** Взаимоблокировка при обмене сообщениями в распределенной системе

Если можно установить верхний предел для количества сообщений, которые пребывают в состоянии передачи между каждой парой процессов в системе, то очевидная стратегия предотвращения взаимоблокировки будет состоять в выделении стольких буферных ячеек, сколько требуется для всех имеющихся каналов. Но это может быть слишком расточительно и, очевидно, требует некоторого предвидения. Если требования

заранее неизвестны или если выделение буферных ячеек исходя из верхнего предела оказывается слишком расточительным, то для оптимизации такого выделения требуется некоторая методика оценивания. Можно показать, что такая задача в общем случае неразрешима, и поэтому в [17] предложен ряд эвристических стратегий для выхода из этого положения.

## 19.5. Резюме

В распределенной операционной системе может поддерживаться перенос процессов — перенос значительной части состояния процесса с одной машины на другую для выполнения процесса на целевой машине. Перенос процессов может быть использован для равномерного распределения нагрузки, повышения производительности сведением к минимуму действий по обмену данными, повышения степени доступности или для того, чтобы разрешить процессам доступ к специализированным удаленным средствам.

В распределенной системе зачастую важно установить сведения о глобальном состоянии, чтобы разрешить соперничество ресурсов и скординировать взаимодействие процессов. Вследствие переменности и непредсказуемости временной задержки в передаче сообщений приходится принимать меры, чтобы разные процессы согласовали порядок, в котором наступают события.

Управление процессами в распределенной системе включает в себя средства для соблюдения взаимного исключения и принятия мер борьбы с взаимоблокировкой. В обоих случаях подобные проблемы в распределенной системе оказываются более сложными, чем в одной системе.

## 19.6. Ключевые термины, контрольные вопросы и задачи

### Ключевые термины

Выселение	Канал	Перенос процессов
Вытесняющий перенос	Моментальный снимок	Распределенная взаимоблокировка
Глобальное состояние	Невытесняющий перенос	Распределенное взаимное исключение

### Контрольные вопросы

- 19.1. Приведите некоторые причины для реализации переноса процессов.
- 19.2. Как обращаться с адресным пространством процессов во время их переноса?
- 19.3. Каковы побудительные причины для вытесняющего и невытесняющего переноса процессов?
- 19.4. Почему невозможно определить истинное глобальное состояние?
- 19.5. Чем обеспечение распределенного взаимного исключения в централизованном алгоритме отличается от его обеспечения в распределенном алгоритме?
- 19.6. Определите два типа распределенной взаимоблокировки.

## Задачи

- 19.1. В подразделе, посвященном стратегиям переноса и входящем в раздел 19.1, описано правило сброса.
- Какую другую стратегию напоминает сброс с точки зрения его источника?
  - Какую другую стратегию напоминает сброс с точки зрения его места назначения?
- 19.2. Для рис. 19.9 утверждается, что во всех четырех процессах оба сообщения,  $a$  и  $q$ , упорядочены как  $\{a, q\}$ , несмотря на то что сообщение  $q$  поступает в процесс  $P_3$  первым. Выполните пошагово алгоритм, чтобы продемонстрировать истинность этого утверждения.
- 19.3. Имеются ли в алгоритме Лэмпорта такие условия, при которых процесс  $P_i$  может сам сохранить передачу ответного сообщения?
- 19.4. Задачи для алгоритма, предложенного в [201].
- Докажите, что он обеспечивает взаимное исключение.
  - Если сообщения поступают не в том порядке, в каком они отсылаются, то данный алгоритм не гарантирует выполнение критических участков по порядку их запросов. Возможно ли в таком случае голодание?
- 19.5. Применяется ли присваивание отметок времени в алгоритме взаимного исключения с передачей эстафеты для сброса счетчиков времени и коррекции отклонений, как это делается в алгоритмах распределенных очередей? Если не применяется, то какова функция отметок времени?
- 19.6. Для алгоритма взаимного исключения с передачей эстафеты докажите, что он:
- гарантирует взаимное исключение;
  - исключает взаимоблокировку;
  - действует беспристрастно.
- 19.7. Поясните, почему во второй строке кода на рис. 19.11, б нельзя просто написать `request(j)=t`.



# ГЛАВА 20

---

## Обзор вероятности и стохастических процессов

В ЭТОЙ ГЛАВЕ...

### 20.1. Основы теории вероятности

Определения вероятности

Аксиоматическое определение

Определение относительной частоты событий

Классическое определение

Условная вероятность и независимость

Теорема Байеса

### 20.2. Случайные переменные

Функции распределения и плотности

Важные виды распределений

Экспоненциальное распределение

Распределение Пуассона

Нормальное распределение

Множество случайных переменных

### 20.3. Элементарные понятия стохастических процессов

Статистика первого и второго порядка

Стационарные стохастические процессы

Спектральная плотность

Независимые приращения

Процесс броуновского движения

Пуассоновский и связанные с ним процессы

Эргодичность

### 20.4. Задачи

## УЧЕБНЫЕ ЦЕЛИ

- Понимать основные понятия теории вероятности.
- Понимать понятие случайной переменной.
- Понимать некоторые из основных понятий стохастических процессов.

Прежде чем приступить к исследованию анализа очередей, приведем краткий обзор основ вероятностных и стохастических процессов. Читатель, знакомый с данными предметами, может спокойно пропустить эту главу.

Сначала в этой главе представлено введение в некоторые элементарные понятия из теории вероятности и случайных переменных. Этот материал потребуется для изучения главы 21, “Анализ очередей”, посвященной анализу очередей. Затем в главе рассматриваются стохастические процессы, которые также имеют отношение к анализу очередей.

## 20.1. Основы теории вероятности

В этом разделе теория вероятности поясняется в самых общих чертах, но представленных здесь понятий должно быть достаточно для проработки материала остальной части этой главы.

### Определения вероятности

Вероятность имеет отношение к назначению числовых характеристик событиям. Так, вероятность  $\Pr[A]$  события  $A$  обозначается числом в пределах от 0 до 1, которое соответствует возможности наступления события  $A$ . В общем случае речь идет о проведении эксперимента и получении **результата**. При этом **событие  $A$**  является конкретным результатом или множеством результатов и этому событию присваивается определенная вероятность.

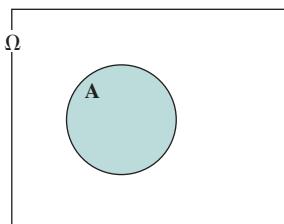
Дать точное обоснование концепции вероятности достаточно сложно. Разные приложения теории вероятности представляют ее в разном свете. Фактически имеется ряд разных определений вероятности. Здесь мы дадим три таких определения.

#### Аксиоматическое определение

Формальный подход к вероятности состоит в том, чтобы сформулировать ряд аксиом, определяющих меру вероятности, а из них вывести законы вероятности, которые можно применять для выполнения полезных вычислений. Аксиомы — это утверждения, которые должны быть приняты безоговорочно. Когда аксиомы приняты, можно доказать каждый из законов.

Аксиомы и законы находят применение в следующих понятиях из теории множеств. Так, **достоверное событие  $\Omega$**  представляет собой такое событие, которое происходит в каждом эксперименте. Оно состоит из совокупности, или **выборочного пространства** (пространства элементарных событий), всех возможных результатов. **Объединением**  $A \cup B$  двух событий,  $A$  и  $B$ , является такое событие, которое происходит в том случае, если происходит событие  $A$  или событие  $B$ . **Пересечением**  $A \cap B$ , иначе обозначаемым

как  $AB$ , событий  $A$  и  $B$  является такое событие, которое происходит в том случае, если происходят оба события, и  $A$ , и  $B$ . Если наступление одного из событий  $A$  и  $B$  исключает наступление другого события, то они называются **взаимоисключающими**, т.е. не существует результата, включающего в себя оба события  $A$  и  $B$ . Событие  $\bar{A}$ , называемое **дополнением** события  $A$ , является таким событием, которое происходит в том случае, если не происходит событие  $A$ , т.е. оно обозначает все результаты из выборочного пространства, не включенные в событие  $A$ . Эти понятия нетрудно представить наглядно с помощью диаграмм Венна, таких как приведенные на рис. 20.1, где затененная часть диаграммы соответствует выражению под диаграммой. Случай, когда события  $A$  и  $B$  не являются взаимно исключающими, показаны на рис. 20.1,  $\delta$  и  $\varepsilon$ , т.е. некоторые результаты являются частью событий  $A$  и  $B$ . Случай, когда события  $A$  и  $B$  оказываются взаимоисключающими, показаны на рис. 20.1,  $\delta$  и  $\varepsilon$ . Следует, однако, иметь в виду, что в этих случаях результатом пересечения двух событий оказывается пустое множество.



(a) A

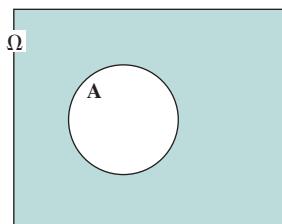
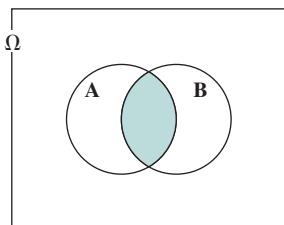
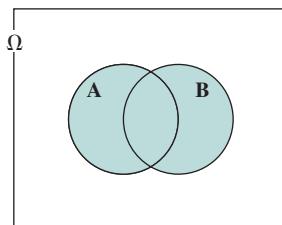
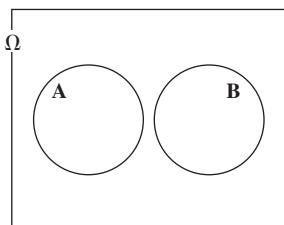
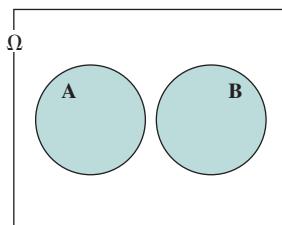
(б) NOT A  
 $\bar{A}$ (в) A AND B  
 $A \cap B$ (г) A OR B  
 $A \cup B$ (д) A AND B  
 $A \cap B$ (е) A OR B  
 $A \cup B$ 

Рис. 20.1. Диаграммы Венна

Ниже приведен общий ряд аксиом, применяемых для определения вероятности.

1.  $0 \leq \Pr[A] \leq 1$  для каждого события  $A$ .
2.  $\Pr[\Omega] = 1$ .
3.  $\Pr[A \cup B] = \Pr[A] + \Pr[B]$ , если события  $A$  и  $B$  взаимоисключающие.

Третья из перечисленных выше аксиом может быть расширена до многих событий. Например,

$$\Pr[A \cup B \cup C] = \Pr[A] + \Pr[B] + \Pr[C],$$

если  $A$ ,  $B$  и  $C$  — взаимоисключающие события. Следует, однако, иметь в виду, что в упомянутых выше аксиомах ничего не говорится о вероятностях, присваиваемых отдельным результатам или событиям.

На основании этих аксиом могут быть выведены многие законы. Ниже перечислены самые важные из них.

$$\Pr[\bar{A}] = 1 - \Pr[A]$$

$$\Pr[A \cap B] = 0, \text{ если } A \text{ и } B \text{ — взаимоисключающие события}$$

$$\Pr[A \cup B] = \Pr[A] + \Pr[B] - \Pr[A \cap B]$$

$$\begin{aligned} \Pr[A \cup B \cup C] &= \Pr[A] + \Pr[B] + \Pr[C] \\ &\quad - \Pr[A \cap B] - \Pr[A \cap C] - \Pr[B \cap C] \\ &\quad + \Pr[A \cap B \cap C] \end{aligned}$$

В качестве примера рассмотрим бросание одной игральной кости, которое может иметь шесть разных результатов. Достоверным является событие, которое наступает, если верхней оказывается любая из шести граней игральной кости. Объединением событий {четное} и {меньше 3} является событие {1 или 2 или 4 или 6}, а их пересечением — событие {2}. События {четное} и {нечетное} — взаимоисключающие. Если допустить, что каждый из шести результатов бросания игральной кости является равновероятным, и присвоить ему вероятность  $1/6$ , то нетрудно заметить, что все три рассматриваемые здесь аксиомы удовлетворяются. Упомянутые выше законы вероятности могут быть применены следующим образом.

$$\Pr\{\text{четное}\} = \Pr\{2\} + \Pr\{4\} + \Pr\{6\} = 1/2$$

$$\Pr\{\text{меньше 3}\} = \Pr\{1\} + \Pr\{2\} = 1/3$$

$$\begin{aligned} \Pr[\{\text{четное}\} \cup \{\text{меньше 3}\}] &= \Pr\{\text{четное}\} + \Pr\{\text{меньше 3}\} - \Pr\{2\} \\ &= 1/2 + 1/3 - 1/6 = 2/3 \end{aligned}$$

### **Определение относительной частоты событий**

При подходе относительной частоты событий используется следующее определение вероятности. Если эксперимент выполняется некоторое количество раз, то каждый такой раз называется **испытанием**. В каждом испытании наблюдается, наступит ли событие  $A$ . Вероятность события  $A$  является пределом

$$\Pr[A] = \lim_{n \rightarrow \infty} \frac{n_A}{n},$$

где  $n$  — количество испытаний, а  $n_A$  — количество наступлений события  $A$ .

Например, монету можно подбрасывать много раз. Если после очень большого числа подбрасываний отношение количества выпадений орла к общему количеству подбрасываний колеблется около величины 0,5, то можно допустить, что это правильная монета с равной вероятностью выпадений орла и решки.

### Классическое определение

Рассмотрим классическое определение вероятности. Пусть  $N$  — количество возможных результатов с ограничением равновероятности всех результатов, а  $N_A$  — количество результатов, при которых наступает событие  $A$ . И тогда вероятность события  $A$  определяется следующим образом:

$$\Pr[A] = \frac{N_A}{N}.$$

Так, если бросать одну игральную кость, то  $N = 6$ , причем три результата соответствуют событию {четное}, а следовательно,  $\Pr\{\text{четное}\} = 3/6 = 0,5$ . Рассмотрим более сложный пример, когда бросаются две игральные кости и требуется определить вероятность того, что сумма окажется равной 7. Если рассмотреть самые разные возможные суммы (2, 3, ..., 12), общее число которых равно 11, то можно прийти к неверному выводу, что вероятность равна 1/11. Важно не забывать о равновероятности результатов, и поэтому необходимо рассмотреть каждую комбинацию граней игральной кости, а также провести различие между первой и второй игральными kostями. Например, результат (3,4) следует учитывать отдельно от результата (4,3). При таком подходе получаются 36 равновероятных результатов, а благоприятными исходами оказываются шесть пар: (1,6), (2,5), (3,4), (4,3), (5,2) и (6,1). Таким образом, искомая вероятность —  $p = 6/36 = 1/6$ .

## Условная вероятность и независимость

Мы часто хотим знать условную вероятность, которая зависит от какого-либо события. Влияние условия состоит в том, что некоторые результаты из пространства элементарных событий удаляются. Какова, например, вероятность выпадения суммы 8 при бросании двух игральных костей, если известно, что грань хотя бы одной кости является четным числом? Рассуждать в этом случае можно следующим образом: одна игральная кость дает четное число, и сумма двух костей тоже четна, поэтому и вторая игральная кость должна дать четное число. Следовательно, из общего множества событий возможны три равновероятных результата: (2,6), (4,4) и (6,2); само общее множество событий имеет размер [36 — (количество событий, когда обе кости нечетны)] = 36 — 3 × 3 = 27. Поэтому окончательная вероятность равна 3/27 = 1/9.

Формально **условная вероятность** события  $A$  при условии события  $B$  (обозначается как  $\Pr[A|B]$ ) определяется следующим отношением:

$$\Pr[A|B] = \frac{\Pr[AB]}{\Pr[B]}.$$

Здесь предполагается, что вероятность  $\Pr[B]$  не равна нулю.

В рассматривавшемся примере  $A = \{\text{сумма равна } 8\}$ ,  $B = \{\text{четное число хотя бы на одной кости}\}$ . Количественно вероятность  $\Pr[AB]$  охватывает все результаты, при которых сумма равна 8 и хотя бы на одной кости выпадает четное число. Следовательно,  $\Pr[AB] = 3/36 = 1/12$ . Минутное размыщение должно привести вас к выводу, что  $\Pr[B] = 3/4$ . Теперь можно вычислить интересующую нас вероятность  $\Pr[A|B]$ :

$$\Pr[A|B] = \frac{1/12}{3/4} = \frac{1}{9},$$

что вполне согласуется с предыдущим рассуждением.

Два события называются **независимыми**, если  $\Pr[AB] = \Pr[A]\Pr[B]$ . Нетрудно видеть, что если события  $A$  и  $B$  независимы, то вероятность  $\Pr[A|B] = \Pr[A]$ , а вероятность  $\Pr[B|A] = \Pr[B]$ .

## Теорема Байеса

И в заключение этого раздела выведем один из самых важных результатов из теории вероятности, называемый теоремой Байеса. Прежде всего, нам требуется вывести формулу полной вероятности. Так, если имеется множество взаимно исключающих событий  $E_1, E_2, \dots, E_n$ , объединение которых охватывает все возможные результаты, а также имеется произвольное событие  $A$ , то можно показать, что

$$\Pr[A] = \sum_{i=1}^n \Pr[A|E_i] \Pr[E_i]. \quad (20.1)$$

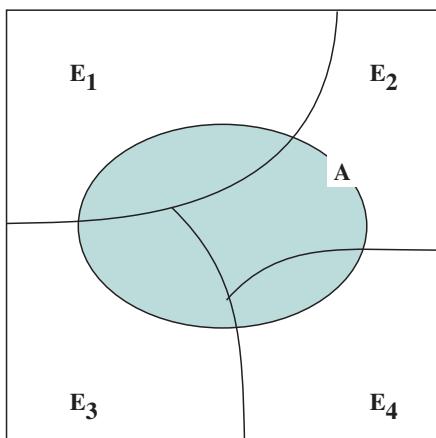
Теорема Байеса может быть сформулирована следующим образом:

$$\Pr[E_i | A] = \frac{\Pr[A | E_i] \Pr[E_i]}{\Pr[A]} = \frac{\Pr[A | E_i] \Pr[E_i]}{\sum_{j=1}^n \Pr[A | E_j] \Pr[E_j]}.$$

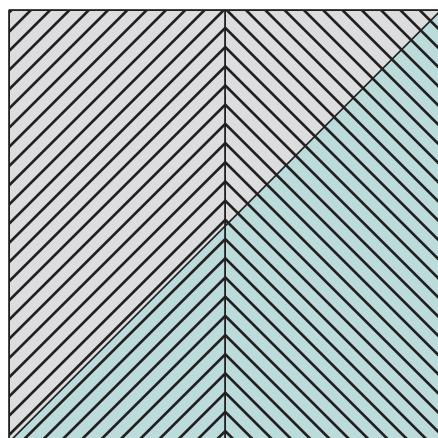
Понятия полной вероятности и теоремы Байеса наглядно показаны на рис. 20.2, а.

Теорема Байеса служит для вычисления *апостериорной вероятности*, т.е. вероятности, что нечто действительно произошло, если в его пользу имеются некоторые свидетельства. Допустим, что последовательность нулей и единиц передается по линии передачи с высоким уровнем помех. Пусть  $S0$  и  $S1$  обозначают события отправки в заданный момент времени нуля и единицы, а  $R0$  и  $R1$  — события получения нуля и единицы соответственно. Допустим также, что известны вероятности событий в источнике данных, а именно:  $\Pr[S1] = p$  и  $\Pr[S0] = 1-p$ . Теперь будем наблюдать, насколько часто в линии передачи возникают ошибки, когда отсылается нуль и когда отсылается единица. Вероятности таких событий вычисляются следующим образом:  $\Pr[R0|S1] = p_a$  и  $\Pr[R1|S0] = p_b$ . Если принимается нуль, то по теореме Байеса можно вычислить условную вероятность ошибки, как показано ниже, а именно — условную вероятность того, что была отправлена единица, а получен нуль:

$$\Pr[S1|R0] = \frac{\Pr[R0|S1] \Pr[S1]}{\Pr[R0|S1] \Pr[S1] + \Pr[R0|S0] \Pr[S0]} = \frac{p_a p}{p_a p + (1-p_b)(1-p)}.$$



а) Диаграмма, иллюстрирующая концепции



б) Пример

Рис. 20.2. Диаграмма для иллюстрации вводимых понятий

Приведенное выше уравнение наглядно показано на рис. 20.2, б, где выборочное пространство представлено единичным квадратом. Одна половина этого квадрата соответствует событию  $S_0$ , а другая — событию  $S_1$ , так что вероятность  $\Pr[S_0] = \Pr[S_1] = 0,5$ . Аналогично одна половина квадрата соответствует событию  $R_0$ , другая — событию  $R_1$ , так что вероятность  $\Pr[R_0] = \Pr[R_1] = 0,5$ . В области, представляющей событие  $S_0$ , четвертая часть площади соответствует событию  $R_1$ , так что вероятность  $\Pr[R_1|S_0] = 0,25$ . Столь же очевидны и остальные условные вероятности.

## 20.2. Случайные переменные

**Случайная переменная** является результатом отображения множества всех возможных событий из рассматриваемого выборочного пространства на вещественные числа, т.е. случайная переменная связывает с каждым событием вещественное число. Эта концепция иногда выражается с точки зрения эксперимента со многими возможными результатами; в каждом таком результате случайной переменной присваивается значение. Таким образом, значением случайной переменной оказывается произвольная величина. Дадим следующее формальное определение случайной переменной: случайная переменная  $X$  является функцией, присваивающей числовое значение каждому результату в выборочном пространстве и удовлетворяющей следующим условиям.

1. Множество  $\{X \leq x\}$  для каждого  $x$  является событием.
2.  $\Pr[X = \infty] = \Pr[X = -\infty] = 0$ .

Случайная переменная **непрерывна**, если она принимает несчетное бесконечное число отдельных значений. Если случайная переменная принимает конечное или счетное бесконечное число значений, она **дискретна**.

## ФУНКЦИИ распределения и плотности

Непрерывная случайная переменная  $X$  может быть описана с помощью **функции распределения вероятности**  $F(x)$  или **функции плотности вероятности**  $f(x)$ .

Функция распределения:  $F(x) = \Pr[X \leq x]$ ,  $F(-\infty) = 0$ ,  $F(\infty) = 1$ .

Функция плотности:  $f(x) = \frac{d}{dx} F(x)$ ,  $F(x) = \int_{-\infty}^x f(y) dy$ ,  $\int_{-\infty}^{\infty} f(y) dy = 1$ .

Распределение вероятности для дискретной случайной переменной характеризуется следующим образом:

$$P_X(k) = \Pr[X = k], \sum_{\forall k} P_X(k) = 1.$$

Зачастую нас интересует отдельная характеристика случайной переменной, а не ее распределение в целом, как показано в табл. 20.1.

**Таблица 20.1. ХАРАКТЕРИСТИКИ СЛУЧАЙНОЙ ПЕРЕМЕННОЙ**

<b>Среднее значение</b> (математическое ожидание, первый момент)	<p>Непрерывный случай: <math>E[X] = \mu_X = \int_{-\infty}^{\infty} xf(x) dx</math></p> <p>Дискретный случай: <math>E[X] = \mu_X = \sum_{\text{all } k} k \Pr[x = k]</math></p>
<b>Второй момент</b>	<p>Непрерывный случай: <math>E[X^2] = \int_{-\infty}^{\infty} x^2 f(x) dx</math></p> <p>Дискретный случай: <math>E[X^2] = \sum_{\forall k} k^2 \Pr[x = k]</math></p>
<b>Дисперсия</b>	$\text{Var}[X] = E[(X - \mu_X)^2] = E[X^2] - \mu_X^2$
<b>Стандартное отклонение</b>	$\sigma_X = \sqrt{\text{Var}[X]}$

Дисперсия и стандартное отклонение являются мерами разброса величины вокруг среднего значения. Так, большая дисперсия означает, что случайная переменная принимает больше значений, находящихся относительно дальше от среднего значения, чем при малой дисперсии. Нетрудно показать, что для любой константы  $a$  справедливо следующее:

$$E[aX] = aE[X]; \quad \text{Var}[aX] = a^2 \text{Var}[X].$$

Среднее значение иначе называется статистикой первого порядка, а второй момент и дисперсия — статистикой второго порядка. Статистика высшего порядка также может быть получена из функции плотности распределения вероятности.

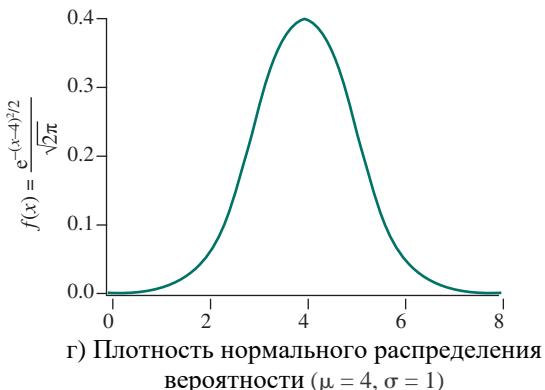
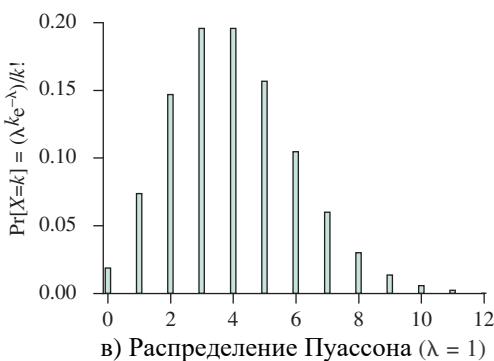
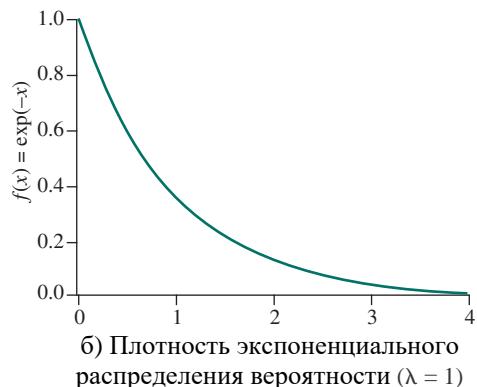
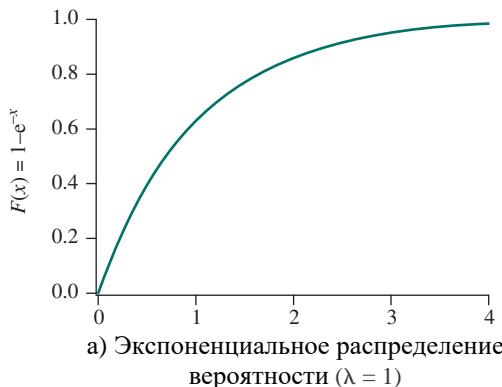
## Важные виды распределений

В анализе очередей важную роль играет несколько видов распределений, которые описываются ниже.

### Экспоненциальное распределение

Это распределение задается параметром  $\lambda > 0$  (рис. 20.3, *а* и *б*) и следующими функциями распределения и плотности:

$$F(x) = 1 - e^{-\lambda x} \quad f(x) = \lambda e^{-\lambda x} \quad x \geq 0$$



**Рис. 20.3.** Некоторые функции распределения вероятности

Экспоненциальное распределение обладает интересным свойством — его среднее значение равно его стандартному отклонению:

$$E[X] = \sigma_X = \frac{1}{\lambda}$$

При рассмотрении временного интервала, такого как время обслуживания, это распределение иногда упоминается как случайное распределение. Это связано с тем, что завершение уже начавшегося временного интервала равновероятно в любой момент времени.

Такое распределение играет важную роль в теории массового обслуживания (или очередей) потому, что зачастую можно считать, что время обслуживания сервером в системе массового обслуживания подчиняется этому распределению. Если речь идет о телефонном трафике, то время обслуживания определяется временем, в течение которого абонент работает с телефонным оборудованием. В сети с коммутацией пакетов время обслуживания определяется временем передачи данных (и потому оно пропорционально длине пакета). Трудно представить надежное теоретическое обоснование экспоненциального характера времени обслуживания, но зачастую оно оказывается очень близким к нему. Это хорошая новость, потому что это упрощает анализ очередей.

### Распределение Пуассона

Это еще один важный вид распределения вероятности (см. рис. 20.3, в) с параметром  $\lambda > 0$ , принимающим следующие значения в точках 0, 1, ...:

$$\Pr[X = k] = \frac{\lambda^k}{k!} e^{-\lambda}, \quad k = 0, 1, 2, \dots$$

$$\mathbb{E}[X] = \text{Var}[X] = \lambda \quad .$$

Если  $\lambda < 1$ , то вероятность  $\Pr[X = k]$  достигает максимума при  $k = 0$ . Если же параметр  $\lambda > 1$ , но не является целочисленным значением, то вероятность  $\Pr[X = k]$  достигает максимума при наибольшем целочисленном значении, не превосходящем  $\lambda$ . Если  $\lambda$  является положительным целым числом, то указанная вероятность имеет два максимума — при  $k = \lambda$  и  $k = \lambda - 1$ .

Распределение Пуассона также играет важную роль в анализе очередей, поскольку шаблон пуассоновского входящего потока позволяет выработать уравнения массового обслуживания, рассматриваемые в главе 21, “Анализ очередей”. К счастью, такое предположение о входящих потоках обычно оказывается верным.

Распределение Пуассона можно применить к скорости поступления следующим образом. Если элементы поступают в очередь согласно пуассоновскому процессу, это можно выразить следующим образом:

$$\Pr[\text{За время } T \text{ поступило } k \text{ элементов}] = \frac{(\lambda T)^k}{k!} e^{-\lambda T},$$

$$\mathbb{E}[\text{Количество поступивших за время } T \text{ элементов}] = \lambda T,$$

$$\text{Средняя скорость поступления элементов} = \lambda.$$

Потоки, подчиняющиеся распределению Пуассона, нередко называются потоками со случным поступлением, потому что вероятность поступления элемента в течение некоторого промежутка времени пропорциональна длительности этого промежутка и не зависит от того, сколько времени прошло с момента поступления последнего элемента. Это означает, что если элементы поступают согласно процессу Пуассона, то в любой момент времени вероятность поступления элемента такая же, как и в любой другой момент, и не зависит от моментов времени поступления других элементов.

Еще одним любопытным свойством пуассоновского процесса является его взаимосвязь с экспоненциальным распределением. Так, если рассмотреть промежутки времени  $T_a$  между поступлением элементов (именуемые интервалами между поступлениями), то можно обнаружить, что их величина количественно подчиняется экспоненциальному распределению, как показано ниже. Таким образом, средний временной интервал между моментами поступления, как и следовало ожидать, обратно пропорционален интенсивности входящего потока.

$$\Pr[T_a < t] = 1 - e^{-\lambda t}$$

$$\mathbb{E}[T_a] = \frac{1}{\lambda}$$

### Нормальное распределение

Нормальное распределение с параметрами  $\mu > 0$  и  $\sigma$  имеет показанные далее функции плотности (см. рис. 20.3,  $\varepsilon$ ) и распределения вероятности.

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}}e^{-(x-\mu)^2/2\sigma^2} \quad F(x) = \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^x e^{-(y-\mu)^2/2\sigma^2} dy$$

При этом

$$\mathbb{E}[X] = \mu, \quad \text{Var}[X] = \sigma^2$$

Важным результатом является *центральная предельная теорема*, которая утверждает, что распределение среднего значения большого количества независимых случайных переменных оказывается приблизительно нормальным, практически не завися от отдельных распределений; ключевыми требованиями к ним являются только конечность среднего значения и дисперсии. Центральная предельная теорема играет ключевую роль в статистике.

## Множество случайных переменных

При наличии двух или более случайных переменных нас зачастую интересует, насколько дисперсии одной из них отражаются в другой. И в этом разделе определяется ряд важных мер такой зависимости.

В общем случае для статистической характеристики случайных переменных требуется определение их функции плотности совместного распределения или функции совместного распределения вероятности.

**Распределение:**  $F(x_1, x_2, \dots, x_n) = \Pr[X_1 \leq x_1, X_2 \leq x_2, \dots, X_n \leq x_n]$

**Плотность:**  $f(x_1, x_2, \dots, x_n) = \frac{\partial^n}{\partial x_1 \partial x_2 \dots \partial x_n} F(x_1, x_2, \dots, x_n)$

**Дискретное распределение:**  $P(x_1, x_2, \dots, x_n) = \Pr[X_1 = x_1, X_2 = x_2, \dots, X_n = x_n]$

Для любых двух случайных переменных  $X$  и  $Y$  справедливо соотношение

Две непрерывные случайные переменные,  $X$  и  $Y$ , называются (статистически) **независимыми**, если  $F(x,y) = F(x)F(y)$ , а следовательно,  $f(x,y) = f(x)f(y)$ . Если случайные переменные  $X$  и  $Y$  дискретны, то они независимы, если  $P(x,y) = P(x)P(y)$ .

Для независимых случайных переменных справедливы следующие соотношения:

$$\begin{aligned} E[XY] &= E[X] \times E[Y], \\ \text{Var}[X+Y] &= \text{Var}[X] + \text{Var}[Y]. \end{aligned}$$

**Ковариация** (смешанный второй момент) двух случайных переменных,  $X$  и  $Y$ , определяется следующим образом:

$$\text{Cov}[X,Y] = E[(X - \mu_X)(Y - \mu_Y)] = E[XY] - E[X]E[Y].$$

Если дисперсии случайных переменных  $X$  и  $Y$  конечны, то конечна и их ковариация. Она может быть положительной, отрицательной или нулевой.

Для конечных дисперсий  $X$  и  $Y$  можно определить **коэффициент корреляции** случайных переменных  $X$  и  $Y$  следующим образом:

$$r(X,Y) = \frac{\text{Cov}(X,Y)}{\sigma_X \sigma_Y}. \quad (20.2)$$

Этот коэффициент можно рассматривать как меру линейной зависимости случайных переменных  $X$  и  $Y$ , нормализованную относительно величины изменчивости в этих переменных. При этом выполняется следующее соотношение:

$$-1 \leq r(X,Y) \leq 1.$$

Случайные переменные  $X$  и  $Y$  считаются **положительно коррелированными**, если  $r(X,Y) > 0$ , **отрицательно коррелированными**, если  $r(X,Y) < 0$ , и **некоррелированными**, если  $r(X,Y) = \text{Cov}(X,Y) = 0$ . Если случайные переменные  $X$  и  $Y$  являются независимыми, то они некоррелированные, и  $r(X,Y) = 0$ . Однако возможна ситуация, когда случайные переменные  $X$  и  $Y$  некоррелированные, но не являются независимыми (см. задачу 20.12).

Коэффициент корреляции служит мерой линейной связности двух случайных переменных. Так, если совместное распределение случайных переменных  $X$  и  $Y$  относительно сосредоточено вокруг прямой линии с положительным наклоном на двумерной координатной плоскости, то коэффициент  $r(X,Y)$  будет, как правило, близким к 1. Это указывает на то, что перемещение по оси  $X$  будет согласовано с перемещением на связанную величину в том же направлении по оси  $Y$ . А если же совместное распределение случайных переменных  $X$  и  $Y$  относительно сосредоточено вокруг прямой линии с отрицательным наклоном, то коэффициент  $r(X,Y)$  будет, как правило, близким к -1.

Нетрудно показать, что справедливо следующее соотношение:

$$\text{Var}(X+Y) = \text{Var}(X) + \text{Var}(Y) + 2\text{Cov}(X,Y).$$

Если случайные переменные  $X$  и  $Y$  имеют одинаковую дисперсию  $\sigma^2$ , то приведенное выше соотношение можно переписать следующим образом:

$$\text{Var}(X+Y) = 2\sigma^2(1 + r(X,Y)).$$

Если случайные переменные  $X$  и  $Y$  некоррелированы ( $r(X,Y)=0$ ), то

$$\text{Var}(X+Y) = 2\sigma^2.$$

Эти результаты нетрудно обобщить для большего количества случайных переменных. Рассмотрим, например, множество случайных переменных  $X_1, \dots, X_N$  с одинаковой дисперсией  $\sigma^2$ . В этом случае можно записать следующее соотношение:

$$\text{Var}\left(\sum_{i=1}^N X_i\right) = \sigma^2 \left( N + 2 \sum_i \sum_{j < i} r(i, j) \right).$$

Здесь  $r(i, j)$  — сокращенное обозначение коэффициента  $r(X_i, X_j)$ . Используя соотношение  $\text{Var}(X/N) = \text{Var}(X)/N^2$ , можно записать следующие уравнения для дисперсии выборочного среднего значения для множества случайных переменных:

$$\begin{aligned} \bar{X} &= \frac{1}{N} \sum_{i=1}^N X_i, \\ \text{Var}(\bar{X}) &= \frac{\sigma^2}{N} \left( 1 + \sum_i \sum_{j < i} r(i, j) \right). \end{aligned}$$

Если же случайные переменные  $X_i$  взаимно независимы, то мы получаем

$$\text{Var}(\bar{X}) = \frac{\sigma^2}{N}.$$

## 20.3. ЭЛЕМЕНТАРНЫЕ ПОНЯТИЯ СТОХАСТИЧЕСКИХ ПРОЦЕССОВ

**Стохастический процесс**, иначе называемый **случайным процессом**, представляет собой семейство случайных переменных  $\{x(t), t \in T\}$ , индексированных параметром  $t$  в некотором индексном множестве  $T$ . Как правило, индексное множество интерпретируется как измерение времени, а  $x(t)$  — функция от времени. **Непрерывным во времени стохастическим процессом** называется процесс, в котором временная переменная  $t$  изменяется непрерывно — как правило, по неотрицательной вещественной числовой оси  $\{x(t), 0 \leq t \leq \infty\}$ , хотя иногда это происходит и по всей вещественной числовой оси. А **дискретным во времени стохастическим процессом** называется такой процесс, в котором временная переменная  $t$  принимает дискретные значения — как правило, натуральные числа  $\{x(t), t = 1, 2, \dots\}$ , хотя иногда это могут быть целые числа в пределах от  $-\infty$  до  $+\infty$ .

Напомним, что случайная переменная определяется как функция, отображающая результат эксперимента в заданное значение. С учетом этого определения можно интерпретировать выражение  $x(t)$  несколькими описанными ниже способами (конкретная интерпретация выражения  $x(t)$  обычно ясна из контекста).

1. Как семейство временных функций (переменная  $t$ ; все возможные результаты).
2. Как единственную временную функцию (переменная  $t$ ; один результат).
3. Как случайную переменную (фиксированное значение временной переменной  $t$ ; все возможные результаты).
4. Как единственное число (фиксированное значение временной переменной  $t$ ; один результат).

А теперь несколько слов о терминологии. **Стохастическим процессом с непрерывными значениями** является такой процесс, в котором случайная переменная  $x(t)$  с фиксированным значением временной переменной  $t$  (см. выше третий способ интерпретации) принимает непрерывные значения, тогда как **стохастическим процессом с дискретными значениями** называется такой процесс, в котором произвольная переменная в любой момент времени  $t$  принимает конечное или счетное бесконечное число значений. Стохастический процесс с непрерывными значениями может принимать как непрерывные, так и дискретные значения, а стохастический процесс с дискретными значениями — только дискретные значения.

Подобно любой случайной переменной, случайная переменная  $x(t)$  для фиксированного значения временной переменной  $t$  может быть характеризована функциями распределения вероятности и его плотности. Для стохастического процесса с непрерывными значениями эти функции принимают следующий вид.

Функция распределения:  $F(x; t) = \Pr[x(t) \leq x]$ ,  $F(-\infty; t) = 0$ ,  $F(\infty; t) = 1$ .

Функция плотности:  $f(x; t) = \frac{\partial}{\partial x} F(x; t)$ ,  $F(x; t) = \int_{-\infty}^x f(y; t) dy$ ,  $\int_{-\infty}^{\infty} f(y; t) dy = 1$ .

Для стохастического процесса с дискретными значениями эти функции имеют следующий вид:

$$P_{x(t)}(k) = \Pr[x(t) = k], \quad \sum_{\forall k} P_{x(t)}(k) = 1.$$

При полном статистическом описании стохастического процесса необходимо принимать во внимание временную переменную. Используя первую интерпретацию из приведенного выше списка, стохастический процесс  $x(t)$  можно представить как состоящий из бесконечного числа случайных переменных: по одной для каждой временной переменной  $t$ . Чтобы полностью описать статистику такого процесса, придется определить функцию плотности совместного распределения случайных переменных  $x(t_1), x(t_2), \dots, x(t_n)$  для всех возможных временных промежутков выборки ( $t_1, t_2, \dots, t_n$ ). Но этот вопрос выходит за рамки нашего рассмотрения.

## Статистика первого и второго порядка

Среднее значение и дисперсия стохастического процесса определяются обычным способом, как показано ниже.

$$E[x(t)] = \mu(t) = \int_{-\infty}^{\infty} xf(x; t) dx \quad \text{непрерывные значения}$$

$$E[x(t)] = \mu(t) = \sum_{\forall k} k \Pr[x(t) = k] \quad \text{дискретные значения}$$

$$E[x^2(t)] = \int_{-\infty}^{\infty} x^2 f(x; t) dx \quad \text{непрерывные значения}$$

$$E[x^2(t)] = \sum_{\forall k} k^2 \Pr[x(t) = k] \quad \text{дискретные значения}$$

$$\text{Var}[x(t)] = \sigma_{x(t)}^2 = E[(x(t) - \mu(t))^2] = E[x^2(t)] - \mu^2(t).$$

Следует, однако, иметь в виду, что в общем случае среднее значение и дисперсия стохастического процесса являются функциями времени. Для описания стохастических процессов очень важно понятие **функции автокорреляции**  $R(t_1, t_2)$ , определяющей совместный момент случайных переменных  $x(t_1)$  и  $x(t_2)$ :

$$R(t_1, t_2) = E[x(t_1)x(t_2)].$$

Подобно представленной ранее корреляционной функции для двух случайных переменных, функция автокорреляции служит мерой взаимосвязи между двумя стохастическими процессами в два момента времени. Связанная с ней величина называется **автоковариацией** и определяется следующим образом:

$$C(t_1, t_2) = E[(x(t_1) - \mu(t_1))(x(t_2) - \mu(t_2))] = R(t_1, t_2) - \mu(t_1)\mu(t_2). \quad (20.3)$$

При этом следует иметь в виду, что дисперсия стохастического процесса  $x(t)$  задается следующим образом:

$$\text{Var}[x(t)] = C(t, t) = R(t, t) - \mu^2(t).$$

И наконец, **коэффициент корреляции** (см. уравнение 20.2)  $x(t_1)$  и  $x(t_2)$  называется **нормализованной функцией автокорреляции** стохастического процесса и может быть выражен следующим образом:

$$\rho(t_1, t_2) = \frac{E[(x(t_1) - \mu(t_1))(x(t_2) - \mu(t_2))]}{\sigma_1 \sigma_2} = \frac{C(t_1, t_2)}{\sigma_1 \sigma_2}. \quad (20.4)$$

К сожалению, в некоторых статьях и книгах функция автокорреляции обозначается как  $r(t_1, t_2)$ , поэтому читатель должен проявить внимательность при чтении.

## Стационарные стохастические процессы

**Стационарным стохастическим процессом** является такой процесс, в котором вероятностные характеристики процесса не изменяются (не являются функцией от времени). Имеются самые разные точные определения данного понятия, но наибольший интерес представляет концепция **стационарности в широком смысле**. Процесс является стационарным в широком смысле (или слабо стационарным), если его ожидаемое значение является константой, а его функция автокорреляции зависит только от разности времен:

$$\begin{aligned} E[x(t)] &= \mu, \\ R(t, t + \tau) &= R(t + \tau, t) = R(\tau) = R(-\tau) \quad \text{для всех } t. \end{aligned}$$

Из этих равенств можно вывести следующие:

$$\begin{aligned} \text{Var}[x(t)] &= R(t, t) - \mu^2(t) = R(0) - \mu^2, \\ C(t, t + \tau) &= R(t, t + \tau) - \mu(t)\mu(t + \tau) = R(\tau) - \mu^2 = C(\tau). \end{aligned}$$

Важное свойство функции автокорреляции  $R(\tau)$  состоит в том, что она измеряет степень зависимости одного момента времени в стохастическом процессе от других моментов времени. Так, если автокорреляционная функция  $R(\tau)$  быстро стремится по э

споненте к нулю по мере роста разности времени  $\tau$ , то стохастический процесс мало зависит от его экземпляров в другие, удаленные моменты времени. Такой стохастический процесс называется **процессом с кратковременной памятью**. Если функция автокорреляции  $R(\tau)$  остается существенной при больших значениях  $\tau$  (т.е. стремится к нулю медленнее, чем по экспоненте), то такой стохастический процесс называется **процессом с долговременной памятью**.

## Спектральная плотность

**Спектр мощности**, или **спектральная плотность** стационарного случайного процесса представляет собой преобразование Фурье его автокорреляционной функции:

$$S(w) = \int_{-\infty}^{\infty} R(\tau) e^{-jw\tau} d\tau,$$

где  $w$  — частота в радианах ( $w = 2\pi f$ ), а  $j = \sqrt{-1}$ .

Для детерминированной временной функции спектральная плотность задает распределение мощности сигнала по частотам. Для стохастического процесса спектральная плотность  $S(w)$  означает среднюю плотность мощности в частотных компонентах случайной переменной  $x(t)$  вблизи  $w$ . Напомним, что случайная переменная  $x(t)$ , а в данном случае — стохастический процесс, может быть интерпретирована как единственная функция от времени. В такой интерпретации временная функция, как и всякая другая функция от времени, состоит из суммы частотных составляющих, а ее спектральная плотность задает относительную мощность, вносимую каждой такой составляющей. Если представить стохастический процесс  $x(t)$  как семейство временных функций (переменная  $t$ ; все возможные результаты), то спектральная плотность задает среднюю мощность каждой частотной составляющей, усредненную по всем возможным времененным функциям случайной переменной  $x(t)$ .

Следующая формула обратного преобразования Фурье выражает временную функцию через ее преобразование Фурье:

$$R(\tau) = \frac{1}{2\pi} \int_{-\infty}^{\infty} S(w) e^{jw\tau} dw.$$

Если  $\tau = 0$ , то данная формула дает

$$\frac{1}{2\pi} \int_{-\infty}^{\infty} S(w) dw = R(0) = E[|x(t)|^2].$$

Таким образом, общая площадь под графиком  $S(w)/2\pi$  равна средней мощности процесса  $x(t)$ . Обратите также внимание, что

$$S(0) = \int_{-\infty}^{\infty} R(\tau) d\tau.$$

$S(0)$  обозначает постоянную составляющую спектра мощности и соответствует интегралу функции автокорреляции. Эта составляющая оказывается конечной лишь в том случае, если функция автокорреляции  $R(\tau)$  достаточно быстро убывает при  $\tau \rightarrow \infty$ , так чтобы интеграл от  $R(\tau)$  был конечным.

Спектр мощности можно выразить и для стохастического процесса, определяемого в дискретные моменты времени (дискретного во времени стохастического процесса). В этом случае мы имеем

$$S(w) = \sum_{k=-\infty}^{\infty} R(k) e^{-jkw}, \quad S(0) = \sum_{k=-\infty}^{\infty} R(k).$$

И вновь  $S(0)$  означает постоянную составляющую спектра мощности, которая соответствует бесконечной сумме функции автокорреляции. Эта составляющая оказывается конечной лишь в том случае, когда функция автокорреляции  $R(\tau)$  достаточно быстро убывает при  $\tau \rightarrow \infty$ , так чтобы сумма была конечной.

В табл. 20.2 приведены некоторые любопытные соотношения между функцией автокорреляции и спектральной плотностью мощности.

**Таблица 20.2. Функции автокорреляции и спектральные плотности**

Стационарный случайный процесс	Функция автокорреляции	Спектральная плотность
$X(t)$	$R_X(\tau)$	$S_X(w)$
$aX(t)$	$a^2 R_X(\tau)$	$a^2 S_X(w)$
$X'(t)$	$-d^2 R_X(\tau)/d\tau^2$	$w^2 S_X(w)$
$X^{(n)}(t)$	$(-1)^n d^{2n} R_X(\tau)/d\tau^{2n}$	$w^{2n} S_X(w)$
$X(t)\exp(jw_0 t)$	$\exp(jw_0 \tau) R_X(\tau)$	$S_X(w - w_0)$

## Независимые приращения

Говорят, что непрерывный по времени стохастический процесс  $\{x(t), 0 \leq t \leq \infty\}$  имеет независимые приращения, если  $x(0) = 0$ , и для всех выборов индексов  $t_0 < t_1 < \dots < t_n$   $n$  случайных переменных

$$x(t_1) - x(t_0), x(t_2) - x(t_1), \dots, x(t_n) - x(t_{n-1})$$

являются независимыми. Таким образом, величина “перемещения” в одном промежутке времени стохастического процесса не зависит от перемещения в любом другом не перекрывающемся с ним промежутке времени. Если, помимо этого, распределение  $x(t_2+h) - x(t_1+h)$  оказывается таким же, как и распределение  $x(t_2) - x(t_1)$  для всех вариантов выбора  $t_2 > t_1$  и каждого  $h > 0$ , то говорят, что у стохастического процесса имеются стационарные независимые приращения.

Отдельного упоминания заслуживают два свойства стохастических процессов со стационарными независимыми приращениями. Так, если у стохастического процесса  $x(t)$  имеются стационарные независимые приращения, а  $E[x(t)] = \mu(t)$  — непрерывная функция времени, то  $\mu(t) = a + bt$ , где  $a$  и  $b$  — константы. Также, если  $\text{Var}[x(t) - x(0)]$  — непрерывная функция от времени, то для всех значений  $s$  справедливо соотношение  $\text{Var}[x(s+t) - x(s)] = \sigma^2 t$ , где  $\sigma^2$  — константа.

В теории стохастических процессов центральная роль принадлежит процессу броуновского движения и пуассоновскому процессу с независимыми приращениями. Ниже представлено их краткое описание.

### Процесс броуновского движения

Броуновское движение является случайным движением микроскопических взвешенных частиц в жидкости и газе, обусловленным столкновениями с молекулами в окружающей их среде. Это физическое явление служит основанием для определения стохастического процесса броуновского движения, иначе называемого процессом Винера или Винера–Леви.

Рассмотрим функцию  $B(t)$ , обозначающую в броуновском движении смещение от начальной точки в одном измерении по истечении времени  $t$ . Рассмотрим также общее движение частицы в промежутке времени  $(s, t)$ , гораздо более продолжительном, чем промежуток времени между столкновениями. Величину  $B(t) - B(s)$  можно рассматривать как сумму большого количества малых смещений. Согласно центральной предельной теореме можно считать, что вероятность такой величины имеет нормальное распределение.

Если допустить, что среда находится в равновесии, то с полным основанием можно предположить, что общее смещение зависит только от длительности промежутка времени, а не от того момента, когда этот промежуток начинается. Это означает, что вероятность величины  $B(t) - B(s)$  должна быть такой же, как и величины  $B(t+h) - B(s+h)$  для любого значения  $h > 0$ . И наконец, если движение частицы полностью обусловлено частыми случайными столкновениями, то общие смещения в течение неперекрывающихся промежутков времени должны быть независимыми, а следовательно, процесс броуновского движения, обозначаемый функцией  $B(t)$ , обладает независимыми приращениями.

Принимая во внимание приведенные выше рассуждения, определим процесс броуновского движения  $B(t)$  как удовлетворяющий следующим условиям.

1. Процесс  $\{B(t), 0 \leq t \leq \infty\}$  имеет стационарные независимые приращения.
2. Случайная переменная  $B(t)$  имеет нормальное распределение для каждого значения  $t > 0$ .
3.  $E[B(t)] = 0$  для всех значений  $t > 0$ .
4.  $B(0) = 0$ .

Функция плотности распределения вероятности процесса броуновского движения имеет следующий вид:

$$f_B(x, t) = \frac{1}{\sigma \sqrt{2\pi t}} e^{-x^2/2\sigma^2 t}$$

Отсюда вытекает:

$$\text{Var}[B(t)] = t, \quad \text{Var}[B(t) - B(s)] = |t - s|.$$

Еще одной важной величиной является автокорреляция процесса  $B(t)$ , записываемая как  $R_B(t_1, t_2)$ . Эта величина выводится следующим образом. Сначала заметим, что для  $t_4 > t_3 > t_2 > t_1$  выполняется следующее соотношение:

$$\begin{aligned} E[(B(t_4) - B(t_3))(B(t_2) - B(t_1))] &= E[B(t_4) - B(t_3)] \times E[B(t_2) - B(t_1)] \\ &= (E[B(t_4)] - E[B(t_3)]) \times (E[B(t_2)] - E[B(t_1)]) \\ &= (0 - 0) \times (0 - 0) = 0. \end{aligned}$$

Первая строка приведенного выше уравнения верна, потому что оба промежутка времени не перекрываются, а следовательно, величины  $B(t_4) - B(t_3)$  и  $B(t_2) - B(t_1)$  независимы в силу допущения о независимости приращений. Напомним, что для независимых случайных переменных  $X$  и  $Y$  справедливо соотношение  $E[XY] = E[X]E[Y]$ . Теперь рассмотрим два промежутка времени,  $(0, t_1)$  и  $(t_1, t_2)$ , для  $0 < t_1 < t_2$ . Эти промежутки времени не перекрываются, поэтому

$$\begin{aligned} 0 &= E[(B(t_2) - B(t_1))(B(t_1) - B(0))] \\ &= E[(B(t_2) - B(t_1))B(t_1)] \\ &= E[B(t_2)B(t_1)] - E[B^2(t_1)] \\ &= E[B(t_2)B(t_1)] - \text{Var}[B(t_1)] \\ &= E[B(t_2)B(t_1)] - t_1. \end{aligned}$$

Следовательно,

$$R_B(t_1, t_2) = E[B(t_1)B(t_2)] = t_1, \text{ где } t_1 < t_2.$$

В общем случае автокорреляцию  $B(t)$  можно выразить следующим образом:

$$R_B(t, s) = \min[t, s].$$

Поскольку среднее значение  $B(t)$  равно нулю, его автоковариация оказывается такой же, как и автокорреляция. Таким образом,

$$C_B(t, s) = \min[t, s].$$

Для любых  $t \geq 0$  и  $\delta > 0$  приращение процесса броуновского движения  $B(t + \delta) - B(t)$  нормально распределено с нулевым средним значением и дисперсией  $\delta$ . Следовательно,

$$\Pr[(B(t + \delta) - B(t)) \leq x] = \frac{1}{\sqrt{2\pi\delta}} \int_{-\infty}^x e^{-y^2/2\delta} dy \quad (20.5)$$

Обратите внимание, что данное распределение зависит только от величины  $\delta$ , но не от величины  $t$ . Это вполне согласуется с тем, что процесс  $B(t)$  имеет стационарные приращения.

Одним из способов визуализации броуновского движения является предел дискретного по времени процесса. Рассмотрим частицу, совершающую случайное блуждание по прямой. В течение небольшого промежутка времени  $\tau$  частица случайным образом перемещается на небольшое расстояние  $\delta$  влево или вправо. Обозначим положение частицы в момент времени  $k\tau$  как  $X_\tau(k\tau)$ . Если положительные и отрицательные переходы равновероятны, то  $X_\tau((k+1)\tau)$  с равной вероятностью равно  $X_\tau(k\tau) + \delta$  или  $X_\tau(k\tau) - \delta$ . Если допустить, что  $X_\tau(0) = 0$ , то положение частицы в момент времени  $t$  представляет собой

$$X_t(t) = \delta(Y_1 + Y_2 + \dots + Y_{\lfloor t/\tau \rfloor}).$$

Здесь  $Y_1, Y_2, \dots$  — независимые случайные переменные, с равной вероятностью принимающие значение 1 или -1, а  $\lfloor t/\tau \rfloor$  означает наибольшее целочисленное значение, не превышающее  $t/\tau$ . Длину шага  $\delta$  удобно нормализовать как  $\sqrt{\tau}$ , так что

$$X_t(t) = \sqrt{\tau}(Y_1 + Y_2 + \dots + Y_{\lfloor t/\tau \rfloor}).$$

Согласно центральной предельной теореме для фиксированного  $t$  и достаточно малого  $\tau$  сумма в приведенном выше уравнении состоит из большого количества случайных переменных, а следовательно, распределение  $X_t(t)$  оказывается приблизительно нормальным с нулевым средним значением и дисперсией  $t\tau$ , поскольку независимая случайная переменная  $Y_i$  имеет нулевое среднее значение и единичную дисперсию. Также, для фиксированных  $t$  и  $h$  и достаточно малого  $\tau$  распределение  $X_t(t+h) - X_t(t)$  оказывается приблизительно нормальным с нулевым средним значением и дисперсией  $h\tau$ . Наконец, следует учесть независимость приращений  $X_t(t)$ . Таким образом,  $X_t(t)$  является дискретной по времени функцией, аппроксимирующей броуновское движение. Если разделить временную ось на более мелкие части, то аппроксимация окажется более точной. В пределе этот процесс броуновского движения становится непрерывным во времени.

### Пуассоновский и связанные с ним процессы

Вспомним, что распределение Пуассона для случайных по времени входящих потоков выражается следующей формулой:

$$\Pr[\text{За время } T \text{ поступило } k \text{ элементов}] = \frac{(\lambda T)^k}{k!} e^{-\lambda T}.$$

**Пуассоновский процесс подсчета** (Poisson counting process)  $\{N(t), t \geq 0\}$  можно определить следующим образом.

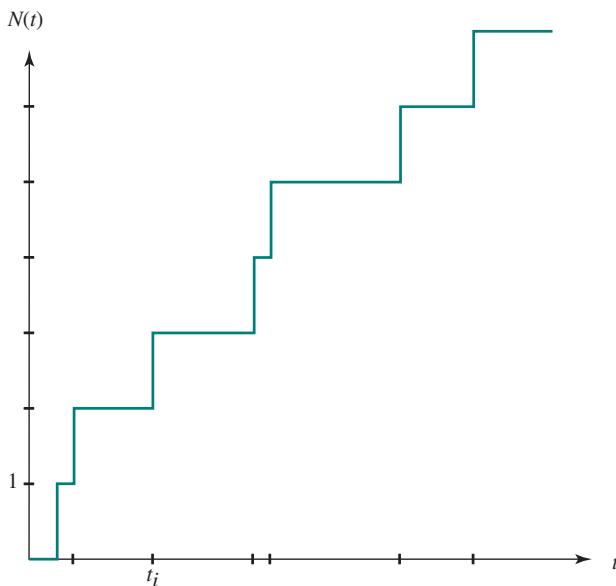
1. Процесс  $N(t)$  имеет независимые приращения.
2.  $N(0) = 0$ .
3. Величина приращения  $N(t_2) - N(t_1)$  при  $0 < t_1 < t_2$  равна количеству точек на интервале  $(t_1, t_2)$  и имеет распределение Пуассона со средним значением  $\lambda(t_2 - t_1)$ .

Тогда мы имеем следующие функции вероятности для пуассоновского процесса  $N(t)$ :

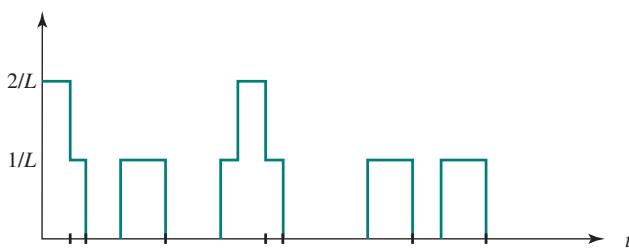
$$\Pr[N(t) = k] = \frac{(\lambda t)^k}{k!} e^{-\lambda t},$$

$$\mathbb{E}[N(t)] = \text{Var}[N(t)] = \lambda t.$$

Очевидно, что процесс  $N(t)$  — не стационарный, поскольку его среднее значение является функцией времени. В любой момент времени функция этого стохастического процесса (с одним результатом) имеет вид идущей вверх лестницы с единичной ступенькой, возникающей в случайные моменты времени  $t_j$ . Пример стохастического процесса  $N(t)$  с определенным результатом приведен на рис. 20.4, а.



а) Пуассоновский процесс подсчета



б) Пуассоновский процесс приращения

Рис. 20.4. Пуассоновские процессы

С пуассоновским процессом подсчета связан **пуассоновский процесс приращения** (Poisson increment process). В частности, для пуассоновского процесса подсчета  $N(t)$  со средним значением  $\lambda t$  и константой  $L > 0$  можно определить пуассоновский процесс приращения  $X(t)$  следующим образом:

$$X(t) = \frac{N(t+L) - N(t)}{L}.$$

$X(t)=k/L$ , где  $k$  — количество точек в интервале  $(t, t+L)$ . Процесс приращения, производный от процесса подсчета, представленного на рис. 20.4, а, показан на рис. 20.4, б. Для этого процесса справедливо следующее соотношение:

$$\mathbb{E}[X(t)] = \frac{1}{L} \mathbb{E}[N(t+L)] - \frac{1}{L} \mathbb{E}[N(t)] = \lambda.$$

При постоянном среднем значении процесс  $X(t)$  оказывается в широком смысле стационарным, а следовательно, имеет функцию автокорреляции  $R(\tau)$  от одной переменной. Можно показать, что эта функция может быть записана как

$$R(\tau) = \begin{cases} \lambda^2, & |\tau| > L \\ \lambda^2 + \frac{\lambda^2}{L} \left(1 - \frac{|\tau|}{L}\right), & |\tau| < L \end{cases} \quad (20.6)$$

Таким образом, корреляция оказывается наибольшей, когда два момента времени взаимно находятся в пределах интервала один от другого, а для больших временных отли-чий принимает небольшое постоянное значение.

## Эргодичность

Для стохастического процесса  $x(t)$  имеются две “усредняющие” функции, которые могут быть выполнены для определения средних значений по ансамблю и по времени.

Рассмотрим сначала **средние значения по ансамблю**. Для постоянного значения переменной  $t$ ,  $x(t)$  представляет собой единственную случайную переменную со средним значением, дисперсией и прочими свойствами распределения. Для заданного постоянно-го значения  $C$  переменной  $t$  существуют следующие меры.

$$E[x(C)] = \mu_x(C) = \int_{-\infty}^{\infty} xf(x; C) dx \quad \text{случай непрерывных значений}$$

$$E[x(C)] = \mu_x(C) = \sum_{\forall k} k \Pr[x(C) = k] \quad \text{случай дискретных значений}$$

$$\text{Var}[x(C)] = \sigma_{x(C)}^2 = E[(x(C) - \mu_x(C))^2] = E[x(C)^2] - \mu_x^2(C)$$

Каждая из этих величин вычисляется по всем значениям  $x(t)$  для всех возможных результатов. Ряд всех возможных результатов для заданной случайной переменной назы-вается *ансамблем*, а потому такие значения называют *средними по ансамблю*.

Для средних значений по времени рассмотрим единственный результат  $x(t)$  как единственную детерминированную функцию от  $t$ . При таком рассмотрении  $x(t)$  можно выяснить среднее значение данной функции во времени. Это **среднее значение по време-ни** обычно выражается следующим образом.

$$M_T = \frac{1}{2T} \int_{-T}^T x(t) dt \quad \text{случай непрерывных значений}$$

$$M_T = \frac{1}{T} \sum_{t=1}^T x(t) \quad \text{случай дискретных значений}$$

Следует, однако, иметь в виду, что  $M_T$  является случайной переменной, поскольку ее вычисление для единственной временной функции означает вычисление для единствен-ного результата.

Стационарный процесс является **эргодичным**, если средние значения по времени равны средним значениям по ансамблю. А поскольку  $E[x(t)]$  является для стационарного процесса константой, имеем:

$$E[M_T] = E[x(t)] = \mu.$$

Следовательно, можно утверждать, что стационарный процесс является эргодичным, если

$$\lim_{T \rightarrow \infty} \text{Var}(M_T) = 0.$$

Иными словами, по мере того как среднее по времени берется по все более и более длительным промежуткам времени, оно приближается к среднему значению по ансамблю.

Рассмотрение условий, при которых стохастический процесс является эргодичным, выходит за рамки данной книги, но нужные предположения сделаны. На самом деле предположение эргодичности имеет существенное значение для практически любой математической модели, применяемой к стохастическим процессам. Практическое значение эргодичности состоит в том, что в большинстве случаев недоступен ни ансамбль результатов, ни хотя бы более одного результата стохастического процесса. Следовательно, единственным средством оценивания проблематичных параметров стохастического процесса является анализ временной функции в течение длительного периода времени.

## 20.4. ЗАДАЧИ

- 20.1.** Вас пригласили принять участие в игре, в которой приз в ваше отсутствие равновероятно прячется в одной из трех коробок. Вернувшись в комнату, вы должны угадать, в какой именно коробке спрятан приз. Игра ведется в два этапа. Сначала вы указываете на одну из выбранных вами коробок. Как только вы это делаете, я открываю крышку одной из двух других коробок. Это всегда оказывается пустая коробка, поскольку мне заранее известно, где именно скрыт приз. Таким образом, приз должен быть либо в выбранной вами коробке, либо в другой неоткрытой коробке. Теперь вы вольны остаться при своем первоначальном выборе или указать на другую неоткрытую коробку. Вы выигрываете приз, если ваш окончательный выбор падет на коробку с ним. Каковая ваша наилучшая стратегия? Следует ли вам остаться при своем первоначальном выборе, выбрать другую коробку или же поступить как угодно, поскольку вероятность выигрыша одинакова в обоих случаях?
- 20.2.** Пациент проходит тест на некоторую болезнь, положительный результат которого указывает на то, что пациент болен. При этом известно следующее:
- Точность теста составляет 87%. Это означает, что если пациент болен, то в 87% случаев тест дает правильный результат; если пациент не болен, то тест дает правильный результат также в 87% случаев.
  - Распространенность данной болезни среди населения составляет 1%.
- Принимая во внимание положительный результат теста, насколько вероятно, что пациент действительно болен?
- 20.3.** Ночью произошло ДТП с участием такси, которое скрылось с места происшествия. Услуги такси в городе предоставляют две компании — “Зеленые” и “Синие”. При этом известно следующее:
- 85% всех таксомоторов принадлежат “Зеленым”, а 15% — “Синим”.
  - Свидетель происшествия опознал таксомотор “Синих”.
- Проверив надежность свидетельства при тех же обстоятельствах, что и в ночь происшествия, суд пришел к выводу, что свидетель верно опознает цвет таксомотора

в 80% случаев. Какова вероятность, что таксомотор, причастный к происшествию, принадлежал компании “Синих”, а не “Зеленых”?

- 20.4.** Парадокс дней рождения является известной в теории вероятности задачей, которую можно сформулировать следующим образом: каково минимальное значение  $K$ , чтобы хотя бы два человека из группы из  $K$  людей родились в один и тот же день с вероятностью больше 0,5? Пренебрегая датой 29 февраля и допуская равную вероятность каждого дня рождения, решите эту задачу.
- а. Определите  $Q(K)$  как вероятность отсутствия совпадающих дней рождений в группе из  $K$  людей. Выведите формулу для  $Q(K)$ . Указание: определите сначала число  $N$  разных способов, которыми можно получить  $K$  несовпадающих значений.
  - б. Определите  $P(K)$  как вероятность наличия хотя бы одного совпадения дней рождений в группе из  $K$  людей. Выведите формулу для  $P(K)$ . Каким должно быть минимальное значение  $K$ , чтобы получить  $P(K) > 0,5$ ? Здесь может помочь график распределения вероятности  $P(K)$ .
- 20.5.** Брошена пара игральных костей с вероятностью выпадения любого из чисел на каждой из них, равной 1/6. Допустим, что  $X$  — максимальное из двух чисел, выпавших на обеих игральных костях.
- а. Найдите распределение вероятности выпадения числа  $X$ .
  - б. Найдите математическое ожидание  $E[X]$ , дисперсию  $\text{Var}[X]$ , а также среднеквадратическое отклонение  $\sigma_X$ .
- 20.6.** Игрок бросает правильную кость. Если выпадает простое число, он выигрывает количество долларов, равное этому числу. Но если выпадает не простое число, он теряет количество долларов, равное этому числу.
- а. Обозначьте выигрыш или проигрыш игрока от одного бросания игральной кости как случайную переменную  $X$ . Укажите распределение вероятности случайной переменной  $X$ .
  - б. Честная ли данная игра (т.е.  $E[X] = 0$ )?
- 20.7.** В некоторой игре в кости игрок вносит сумму  $E$  в качестве взноса, выбирает число от одного до шести, а затем бросает три игральные кости. Если на всех трех костях выпадет выбранное игроком число, он получит сумму, в четыре раза превышающую его взнос. Если выбранное число выпадет на двух костях, он получит сумму, в три раза превышающую его взнос. Если выбранное число выпадет только на одной кости, он получит сумму, в два раза превышающую его взнос. Наконец, если выбранное число не выпадет ни на одной из костей, игрок не получит ничего. Пусть  $X$  — выигрыш игрока в одной партии этой игры, а кости правильные.
- а. Определите функцию распределения вероятности  $X$ .
  - б. Вычислите среднее значение  $E[X]$ .
- 20.8.** Среднее значение и дисперсия  $X$  равны 50 и 4 соответственно. Вычислите следующее.
- а. Среднее значение  $X^2$ .
  - б. Дисперсию и среднеквадратическое отклонение  $2X + 3$ .
  - в. Дисперсию и среднеквадратическое отклонение  $-X$ .

**20.9.** Непрерывная случайная переменная  $R$  имеет равномерную плотность распределения в пределах от 900 до 1100 и нулевую — вне указанного интервала. Найдите вероятность, что значение случайной переменной  $R$  находится в пределах от 950 до 1050.

**20.10.** Покажите, что чем больше (при прочих равных условиях) коэффициент корреляции двух случайных переменных, тем больше будет дисперсия их суммы и меньше дисперсия их разности.

**20.11.** Пусть каждая из случайных переменных  $X$  и  $Y$  имеет лишь два возможных значения 0 и 1. Докажите, что если случайные переменные  $X$  и  $Y$  не коррелируют, то они независимы.

**20.12.** Допустим, что имеется случайная переменная  $X$  со следующим распределением:  $\Pr[X=-1] = 0,25$ ;  $\Pr[X=0] = 0,5$ ;  $\Pr[X=1] = 0,25$ . Пусть  $Y = X^2$ .

а. Являются ли случайные переменные  $X$  и  $Y$  независимыми? Обоснуйте свой ответ.

б. Вычислите ковариацию  $\text{Cov}(X, Y)$ .

в. Являются ли случайные переменные  $X$  и  $Y$  не коррелиирующими? Обоснуйте свой ответ.

**20.13.** Искусственным примером стохастического процесса является детерминированный сигнал, выражаемый соотношением  $x(t)=g(t)$ . Определите среднее значение, дисперсию и автокорреляцию такого стохастического процесса.

**20.14.** Пусть  $x(t)$  — стохастический процесс со следующими параметрами:

$$\mu(t)=3, \quad R(t_1, t_2) = 9 + 4e^{-0.2|t_1 - t_2|}.$$

Определите среднее значение, дисперсию и ковариацию следующих случайных переменных:  $Z=x(5)$  и  $W=x(8)$ .

**20.15.** Пусть  $\{Z_n\}$  — множество не коррелирующих вещественных случайных переменных, каждая из которых имеет нулевое среднее значение и единичную дисперсию. Определим для констант  $\alpha_0, \alpha_1, \dots, \alpha_K$  скользящее среднее как

$$Y_n = \sum_{i=0}^K \alpha_i Z_{n-i}.$$

Покажите, что  $Y$  — стационарный процесс, и найдите его функцию автокорреляции.

**20.16.** Пусть

$$X_n = A \cos(n\lambda) + B \sin(n\lambda).$$

Здесь  $A$  и  $B$  — не коррелирующие случайные переменные, каждая со средним значением 0 и дисперсией 1. Покажите, что  $X$  — стационарный процесс со спектром, содержащим ровно одну точку.



# ГЛАВА 21

## АНАЛИЗ ОЧЕРЕДЕЙ

В ЭТОЙ ГЛАВЕ...

### 21.1. Простой пример поведения очередей

### 21.2. Цель анализа очередей

### 21.3. Модели очередей

Одноканальная система массового обслуживания

Параметры очереди

Демонстрация основных свойств

Характеристики модели

Многоканальная система массового обслуживания

Основные соотношения из теории массового обслуживания

Предположения

### 21.4. Одноканальные системы массового обслуживания

### 21.5. Многоканальные системы массового обслуживания

### 21.6. Примеры

Сервер базы данных

Вычисление процентилей

Сильно связанный мультипроцессор

Односерверный подход

Многосерверный подход

Задача построения многоканальной системы массового обслуживания

Одноканальная модель массового обслуживания

Многоканальная модель массового обслуживания

### 21.7. Очереди с приоритетами

### 21.8. Сети очередей

Разделение и объединение потоков трафика

Последовательные очереди

Теорема Джексона

Применение теоремы Джексона в сети с коммутацией пакетов

### 21.9. Другие модели систем массового обслуживания

### 21.10. Оценка параметров модели

Выборка

Ошибки выборки

### 21.11. Задачи

## УЧЕБНЫЕ ЦЕЛИ

- Понимать характеристическое поведение систем массового обслуживания.
- Пояснять ценность анализа очередей.
- Пояснять главные особенности одно- и многоканальных систем массового обслуживания.
- Анализировать модели одноканальных систем массового обслуживания.
- Анализировать модели многоканальных систем массового обслуживания.
- Описывать воздействие приоритетов на производительность системы массового обслуживания.
- Понимать основной принцип организации сетей массового обслуживания.
- Разбираться в вопросах, касающихся оценивания параметров модели массового обслуживания.

Анализ очередей, иначе называемый анализом систем массового обслуживания, является одним из самых важных инструментальных средств для тех, кто имеет отношение к компьютерному и сетевому анализу. С его помощью можно получить приблизительные ответы на многие вопросы, в том числе на следующие.

- Что происходит со временем извлечения информации из файла, когда возрастает уровень использования дискового ввода-вывода?
- Изменится ли время отклика, если удвоить быстродействие процессора и количество пользователей в системе?
- Если в алгоритм планирования процессов включить приоритеты, то как это повлияет на производительность?
- Какой алгоритм планирования работы дисков обеспечивает наилучшую в среднем производительность?

Вопросам, которые способен решить анализ очередей, нет числа, причем они касаются буквально каждой области вычислительной техники. Способность выполнить такой анализ является очень важным средством для тех, кто имеет отношение к данной области.

Несмотря на то что теория массового обслуживания имеет сложный математический аппарат, ее приложение к анализу производительности зачастую оказывается удивительно простым. Для этого достаточно знать элементарные статистические понятия (наподобие среднего значения и стандартного отклонения) и понимать азы теории массового обслуживания. Вооружившись этими знаниями, аналитик нередко в состоянии выполнить анализ очередей “на салфетке”, пользуясь готовыми таблицами и простыми компьютерными программами, занимающими всего лишь несколько строк кода.

Назначение этой главы — предоставить краткое практическое руководство по анализу очередей. В конце главы приведены ссылки на дополнительную литературу. Следует, однако, иметь в виду, что некоторые базовые понятия теории вероятности и математической статистики уже пояснялись в предыдущей главе.

## 21.1. ПРОСТОЙ ПРИМЕР ПОВЕДЕНИЯ ОЧЕРЕДЕЙ

Прежде чем вдаваться в подробности анализа очередей, рассмотрим грубый пример, дающий некоторое представление о данном предмете. Итак, рассмотрим веб-сервер, способный обработать отдельный запрос в среднем за 1 мс. Для простоты допустим, что веб-сервер действительно обрабатывает каждый запрос ровно 1 мс. Если запросы поступают через каждую миллисекунду (т.е. со скоростью 1000 запросов в секунду), то представляется вполне разумным заявить, что сервер способен справиться с такой нагрузкой.

Допустим, что запросы поступают с равномерной скоростью — ровно один запрос каждую миллисекунду. Как только запрос поступает, сервер сразу же его обрабатывает. Стоит серверу покончить с текущим запросом, как тотчас поступает новый запрос, и сервер берется за дело снова.

А теперь примем более реалистичный подход и допустим, что в среднем запросы поступают один раз в миллисекунду, но с некоторыми отклонениями. В течение любого заданного периода времени, равного 1 мс, может поступить один или несколько запросов (или вообще не поступить ни одного запроса), но в среднем они по-прежнему поступают один раз в миллисекунду. В этом случае здравый смысл, по-видимому, подсказывает, что сервер мог бы справиться с такой нагрузкой. В периоды полной загруженности, когда немало запросов поступает целыми пакетами, сервер может хранить ожидающие своей очереди запросы в буфере. Иными словами, поступающие запросы ставятся в очередь, ожидая обслуживания. А в периоды простоеов сервер может выбирать запросы из буфера, опустошая его. Но в этом случае может возникнуть следующий проектный вопрос: насколько большим должен быть буфер?

Поведение такой системы массового обслуживания в общих чертах представлено в табл. 21.1–21.3. Так, в табл. 21.1 предполагается, что средняя скорость поступления составляет 500 запросов в секунду, что соответствует половине пропускной способности сервера. Записи в данной таблице показывают количество запросов, поступающих в каждую секунду, количество запросов, обслуженных за эту секунду, а также количество запросов, ожидающих своей очереди в буфере по окончании секунды. Как видно из таблицы, за 50 секунд среднее содержимое буфера составляет 43 запроса, достигая пикового значения свыше 600 запросов.

**Таблица 21.1. Поведение очереди с нормализованной скоростью поступления запросов, равной 0,5**

Время	Вход	Выход	Очередь
0	0	0	0
1	88	88	0
2	796	796	0
3	1627	1000	627
4	51	678	0
5	34	34	0
6	966	966	0
7	714	714	0

Продолжение табл. 21.1

Время	Вход	Выход	Очередь
8	1276	1000	276
9	494	769	0
10	933	933	0
11	107	107	0
12	241	241	0
13	16	16	0
14	671	671	0
15	643	643	0
16	812	812	0
17	262	262	0
18	218	218	0
19	1378	1000	378
20	507	885	0
21	15	15	0
22	820	820	0
23	1253	1000	253
24	307	559	0
25	540	540	0
26	190	190	0
27	500	500	0
28	96	96	0
29	943	943	0
30	105	105	0
31	183	183	0
32	447	447	0
33	542	542	0
34	166	166	0
35	165	165	0
36	490	490	0
37	510	510	0
38	877	877	0
39	37	37	0
40	163	163	0
41	104	104	0

Окончание табл. 21.1

Время	Вход	Выход	Очередь
42	42	42	0
43	291	291	0
44	645	645	0
45	363	363	0
46	134	134	0
47	920	920	0
48	1507	1000	507
49	598	1000	105
50	172	277	0
<b>В среднем</b>	<b>499</b>	<b>499</b>	<b>43</b>

В табл. 21.2 средняя скорость поступления запросов возрастает до 95% пропускной способности сервера, т.е. до 950 запросов в секунду, а среднее содержимое буфера — до 1859 запросов. И это обстоятельство кажется удивительным, поскольку скорость поступления запросов возросла меньше чем в 2 раза, тогда как среднее содержимое буфера увеличилось больше чем в 40 раз. В табл. 21.3 средняя скорость поступления запросов возросла еще немного — до 99% пропускной способности сервера, и среднее содержимое буфера стало равным 2583 запросам. Таким образом, незначительное увеличение средней скорости поступления запросов приводит к увеличению среднего содержимого буфера почти на 40%.

**Таблица 21.2. Поведение очереди с нормализованной скоростью поступления запросов, равной 0,95**

Время	Вход	Выход	Очередь
0	0	0	0
1	167	167	0
2	1512	1000	512
3	3091	1000	2603
4	97	1000	1700
5	65	1000	765
6	1835	1000	1600
7	1357	1000	1957
8	2424	1000	3381
9	939	1000	3320
10	1773	1000	4093
11	203	1000	3296
12	458	1000	2754
13	30	1000	1784

## Окончание табл. 21.2

<b>Время</b>	<b>Вход</b>	<b>Выход</b>	<b>Очередь</b>
14	1275	1000	2059
15	1222	1000	2281
16	1543	1000	2824
17	498	1000	2322
18	414	1000	1736
19	2618	1000	3354
20	963	1000	3317
21	29	1000	2346
22	1558	1000	2904
23	2381	1000	4285
24	583	1000	3868
25	1026	1000	3894
26	361	1000	3255
27	950	1000	3205
28	182	1000	2387
29	1792	1000	3179
30	200	1000	2379
31	348	1000	1727
32	849	1000	1576
33	1030	1000	1606
34	315	1000	921
35	314	1000	235
36	931	1000	166
37	969	1000	135
38	1666	1000	801
39	70	871	0
40	310	310	0
41	198	198	0
42	80	80	0
43	553	553	0
44	1226	1000	226
45	690	916	0
46	255	255	0
47	1748	1000	748
48	2863	1000	2611
49	1136	1000	2747
50	327	1000	2074
<b>В среднем</b>	<b>948</b>	<b>907</b>	<b>1859</b>

**Таблица 21.3. Поведение очереди с нормализованной скоростью поступления запросов, равной 0,99**

Время	Вход	Выход	Очередь
0	0	0	0
1	174	174	0
2	1576	1000	576
3	3221	1000	2797
4	101	1000	1898
5	67	1000	965
6	1913	1000	1878
7	1414	1000	2292
8	2526	1000	3818
9	978	1000	3796
10	1847	1000	4643
11	212	1000	3855
12	477	1000	3332
13	32	1000	2364
14	1329	1000	2693
15	1273	1000	2966
16	1608	1000	3574
17	519	1000	3093
18	432	1000	2525
19	2728	1000	4253
20	1004	1000	4257
21	30	1000	3287
22	1624	1000	3911
23	2481	1000	5392
24	608	1000	5000
25	1069	1000	5069
26	376	1000	4445
27	990	1000	4435
28	190	1000	3625
29	1867	1000	4492
30	208	1000	3700
31	362	1000	3062
32	885	1000	2947

Окончание табл. 21.3

Время	Вход	Выход	Очередь
33	1073	1000	3020
34	329	1000	2349
35	327	1000	1676
36	970	1000	1646
37	1010	1000	1656
38	1736	1000	2392
39	73	1000	1465
40	323	1000	788
41	206	994	0
42	83	83	0
43	576	576	0
44	1277	1000	277
45	719	996	0
46	265	265	0
47	1822	1000	822
48	2984	1000	2806
49	1184	1000	2990
50	341	1000	2331
<b>В среднем</b>	<b>988</b>	<b>942</b>	<b>2583</b>

В данном приближенном примере продемонстрировано, насколько поведение системы массового обслуживания с организованной очередью может не согласовываться с нашей интуицией.

## 21.2. ЦЕЛЬ АНАЛИЗА ОЧЕРЕДЕЙ

Имеется немало случаев, когда в проекте важно отразить воздействие некоторых перемен — в связи с ожидаемым увеличением нагрузки на систему или намеченными изменениями в самом проекте. Допустим, что в организации поддерживается целый ряд терминалов, персональных компьютеров и рабочих станций, подключенных к локальной сети со скоростью передачи данных 100 Мбит/с. К этой сети планируется подключить дополнительное подразделение организации, расположенное в том же самом здании. Можно ли в таком случае увеличить существующую рабочую нагрузку на локальную сеть или все же лучше организовать вторую локальную сеть и мост между обеими сетями? Имеются и другие случаи, когда нужные средства отсутствуют, но необходимо создать проект системы, исходя из предполагаемой потребности в ней. Допустим, весь штат подразделения организации требуется оснастить персональными компьютерами, подключенными к файловому серверу через локальную сеть. Опираясь на опыт, накоп-

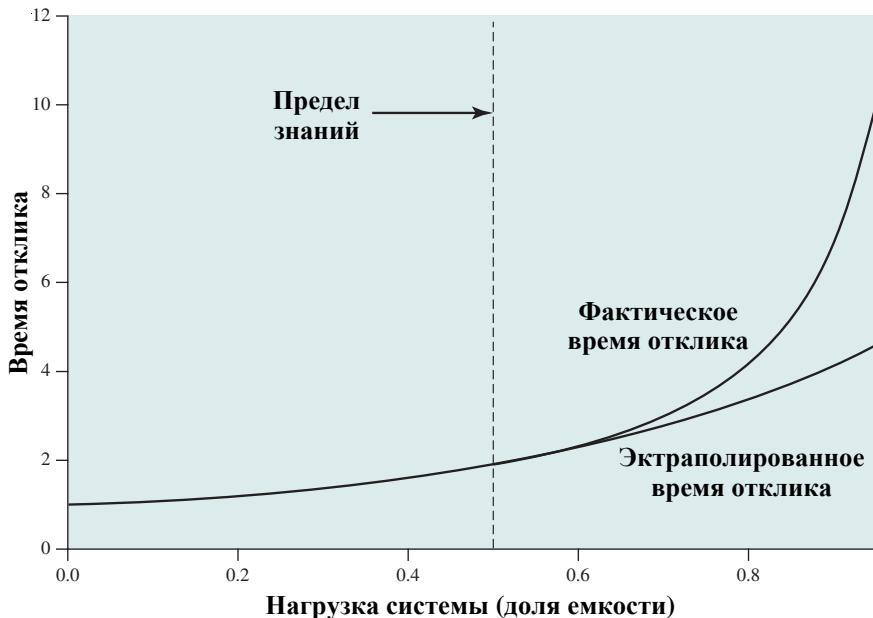
ленный в других подразделениях организации, можно оценить нагрузку каждого персонального компьютера на сеть и файловый сервер.

Предметом особой заботы является производительность системы. В приложении, действующем в интерактивном режиме или же в реальном времени, особый интерес представляет параметр системы, определяющий время ее отклика, а в других случаях принципиальным вопросом является пропускная способность системы. Но в любом случае проектные показатели производительности следует планировать на основании имеющихся сведений или оценок нагрузки для нового оборудования. И здесь возможен целый ряд подходов, перечисленных ниже.

1. Произвести анализ конкретных значений *post factum*.
2. Выполнить простое планирование, расширив имеющийся опыт до масштабов предполагаемой в будущем среды.
3. Разработать аналитическую модель на основе теории массового обслуживания.
4. Запрограммировать и запустить имитационную модель.

Первый из перечисленных выше подходов вообще нельзя рассматривать как возможный вариант, поскольку в этом случае придется ждать и наблюдать происходящее. Это приводит к тому, что пользователи системы оказываются неудовлетворенными, а покупки — неразумными. Более обещающим выглядит второй подход, при котором аналитик может стать на ту точку зрения, что запланировать заранее какую-нибудь потребность в будущем невозможно ни с какой степенью уверенности. Следовательно, нет никакого смысла пытаться выполнять какую-то конкретную процедуру моделирования, а вместо этого следует сделать грубые приблизительные оценки в качестве чернового планирования. Недостаток такого подхода заключается в том, что поведение большинства систем при изменяющейся нагрузке не отвечает интуитивно ожидаемому поведению, как уже упоминалось в предыдущем разделе. Если имеется среда, в которой совместно используется общее средство (например, сеть, канал передачи данных, система разделения времени), то производительность такой среды, как правило, должна находиться в экспоненциальной зависимости от увеличения степени потребности.

Наглядный тому пример приведен на рис. 21.1, где верхняя линия показывает, что, как правило, происходит со временем отклика на действия пользователей совместно используемого средства по мере увеличения нагрузки на него (нагрузка выражена в виде доли от пропускной способности). Так, если требуется обработать данные, вводимые с диска, способного передавать 1000 блоков данных в секунду, то нагрузка 0,5 соответствует скорости передачи 500 блоков данных в секунду, а время отклика представляет собой время, которое требуется для повторной передачи входящего блока данных. Нижняя линия отражает простое планирование, основанное на знании поведения системы вплоть до степени нагрузки порядка 0,5. Следует, однако, иметь в виду, что производительность системы на самом деле резко падает, когда нагрузка достигает степени около 0,8–0,9, несмотря на то что при простом планировании все выглядит весьма радужно.



**Рис. 21.1.** Сравнение планируемого и фактического времени отклика

Следовательно, требуется более точное инструментальное средство прогнозирования. Третий из перечисленных выше подходов состоит в том, чтобы воспользоваться аналитической моделью, выраженной в виде уравнений, решение которых может дать желаемые параметры системы (время отклика, пропускную способность и пр.). Для решения задач на уровне отдельного компьютера, операционной системы и сети, а на самом деле — многих практических задач, имеются аналитические модели, построенные по теории массового обслуживания и довольно хорошо согласующие теорию с практикой. Недостаток теории массового обслуживания заключается в том, что она вынуждает делать целый ряд упрощающих допущений, чтобы вывести уравнения для расчета представляющих интерес параметров.

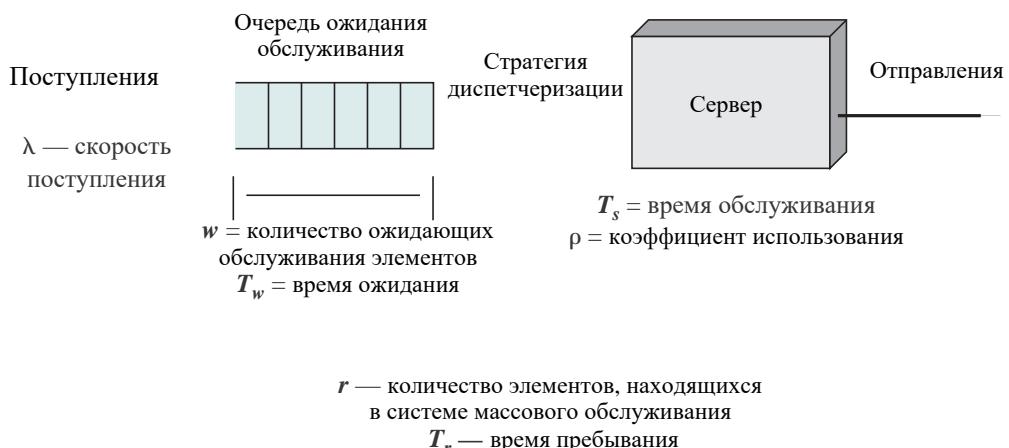
Последний из рассматриваемых здесь подходов состоит в построении имитационной модели. В этом случае, опираясь на достаточно эффективный и гибкий язык имитационного программирования, аналитик может довольно подробно смоделировать реальность, не делая многих допущений, требуемых теорией массового обслуживания. Но, как правило, имитационная модель не требуется или по крайней мере не рекомендуется на первой стадии анализа. С одной стороны, существующие измеренные и запланированные показатели будущей нагрузки на систему несут в себе определенную погрешность. Следовательно, какой бы совершенной ни была имитационная модель, ценность получаемых с ее помощью результатов ограничивается качеством исходных данных. С другой стороны, получаемые результаты нередко оказываются сходными с теми, которые дает тщательный имитационный анализ, несмотря на многие допущения, требуемые теорией массового обслуживания. Кроме того, для решения вполне определенной задачи анализ очередей может быть выполнен буквально в считанные минуты, тогда как имитация может занять целые дни, недели или еще больше времени на программирование и выполнение имитационной модели. Принимая во внимание все это, аналитику следует овладеть основами теории массового обслуживания.

## 21.3. МОДЕЛИ ОЧЕРЕДЕЙ

В этом разделе описываются разные модели очередей для систем массового обслуживания.

### Одноканальная система массового обслуживания

Простейшая система массового обслуживания приведена на рис. 21.2. Центральным элементом такой системы является сервер, обеспечивающий некоторое обслуживание поступающих в нее элементов. Если сервер простояивает, поступающий элемент обслуживается сразу же; в противном случае он ставится в очередь ожидания обслуживания.<sup>1</sup> Как только сервер завершит обслуживание элемента, последний отправляется далее. Если в очереди имеются ожидающие обслуживания элементы, они сразу же направляются серверу. В такой модели сервер может представлять собой все, что выполняет некоторую функцию или услугу для совокупности поступающих элементов. Например, процессор обслуживает процессы, канал связи предоставляет услуги передачи пакетов или фреймов данных, а устройство ввода-вывода — услуги чтения или записи запросов на ввод-вывод данных.



**Рис. 21.2.** Структура и параметры одноканальной системы массового обслуживания

#### Параметры очереди

На рис. 21.2 приведены также некоторые важные параметры, связанные с моделью очередей. В частности, элементы поступают в систему массового обслуживания с некоторой средней скоростью  $\lambda$  (элементов в секунду). К примерам поступающих элементов относятся пакеты, поступающие в маршрутизатор, вызовы, поступающие на АТС, и т.п. В любой заданный момент времени в очереди на обслуживания будет находиться определенное количество элементов (от нуля и больше). Среднее количество ожидающих об-

<sup>1</sup> Зачастую в литературе очередь ожидания обслуживания называется просто очередью; распространено также применение термина “очередь” ко всей системе массового обслуживания в целом. Поэтому под очередью здесь и далее, если не указано иное, подразумевается очередь ожидания обслуживания.

служивания элементов обозначается параметром  $w$ , а среднее время, в течение которого элементу приходится ожидать, — параметром  $T_w$ . Значение параметра  $T_w$  усредняется по всем поступающим элементам, включая и те, которые вообще не ожидают обслуживания. Сервер обслуживает поступающие элементы со средним временем обслуживания  $T_s$ . Это промежуток времени между направлением элемента серверу на обслуживание и отправкой обслуженного элемента из сервера. Коэффициент использования (utiлизации)  $\rho$  обозначает долю времени, в течение которого сервер занят обслуживанием. И наконец, в системе в целом применяются два параметра, а именно — среднее количество элементов, пребывающих в системе (включая обслуживаемый элемент, если таковой имеется, а также ожидающие обслуживания элементы), обозначаемое как  $r$ , и среднее время нахождения элемента в системе (включая время ожидания и обслуживания), обозначаемое как  $T_i$  и именуемое *средним временем пребывания*.<sup>2</sup>

Если считать, что емкость очереди бесконечна, то в системе не теряется ни один поступивший элемент; он просто задерживается в ней до тех пор, пока не будет обслужен. При таких условиях скорость отправки элементов оказывается равной скорости их поступления. По мере увеличения скорости поступления элементов, которая соответствует скорости трафика, проходящего через систему, коэффициент ее использования возрастает, а вместе с ним растет и вероятность перегрузки. В итоге очередь удлиняется, а время ожидания увеличивается. При  $\rho = 1$  сервер становится полностью загруженным, работая 100% времени. Сервер может справиться с поступающими элементами при условии, что коэффициент его использования меньше 100%. Как только сервер становится полностью загруженным, работая все время, скорость отправки элементов остается постоянной, как бы ни возрастала скорость их поступления. Таким образом, теоретически максимальная скорость поступления элементов, которую может обслужить система, выражается в виде соотношения

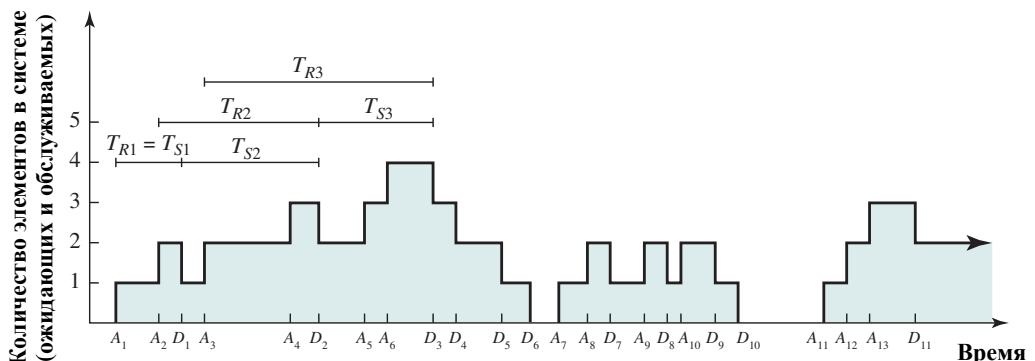
$$\lambda_{\max} = \frac{1}{T_s}.$$

Однако по мере приближения системы к полной загруженности очереди становятся очень большими, неограниченно разрастаясь при  $\rho = 1$ . Практические рекомендации наподобие требований ко времени отклика или к размеру буфера обычно ограничивают скорость поступления элементов в одноканальную систему массового обслуживания до 70–90% от теоретически максимальной величины.

### Демонстрация основных свойств

Полезно продемонстрировать процессы, задействованные в массовом обслуживании. Пример реализации процесса массового обслуживания приведен на рис. 21.3 в виде графика зависимости общего количества элементов в системе от времени. Затененные участки этого графика обозначают периоды времени, когда сервер занят обслуживанием. На оси времени помечены два вида событий: поступление элемента  $j$  в момент времени  $A_j$ , а также завершение обслуживания элемента  $j$  в момент времени  $D_j$ , когда элемент выходит из системы. Время пребывания элемента  $j$  в системе равно  $T_{Rj} = D_j - A_j$ , а конкретное время обслуживания элемента  $j$  обозначается как  $T_{Sj}$ .

<sup>2</sup> Этот параметр в ряде источников называется средним временем ожидания обслуживания, тогда как в других источниках среднее время ожидания обслуживания употребляется для обозначения среднего времени, затрачиваемого на ожидание в очереди ожидания обслуживания (перед собственным обслуживанием).



Для элемента  $i$ :

- $A_i$  — время поступления
- $D_i$  — время отправления
- $T_{Ri}$  — время пребывания
- $T_{Si}$  — время обслуживания

**Рис. 21.3.** Пример процесса массового обслуживания

В данном примере время пребывания  $T_{R1}$  полностью состоит из времени обслуживания  $T_{S1}$  первого элемента. Ведь когда первый элемент поступает в систему, она не занята, и потому сразу же приступает к его обслуживанию. Время пребывания  $T_{R2}$  состоит из времени ожидания вторым элементом своего обслуживания ( $D_1 - A_2$ ) и времени его обслуживания  $T_{S2}$ . Аналогично  $T_{R3} = (D_3 - A_3) = (D_3 - D_2) + (D_2 - A_3) = T_{S3} + (D_2 - A_3)$ . Однако элемент  $n$  может быть отправлен из системы до того, как поступит элемент  $n+1$  (например, когда  $D_6 < A_7$ ), так что общее выражение имеет следующий вид:

$$T_{Rn+1} = T_{Sn+1} + \text{MAX}[0, D_n - A_{n+1}].$$

### Характеристики модели

Прежде чем переходить к выводу любых аналитических уравнений для модели очередей, необходимо выбрать определенные характеристики этой модели. Ниже описаны типичные варианты выбора таких характеристик, обычно уместные в контексте передачи данных.

- **Количество элементов.** Мы считаем, что элементы поступают из настолько большого источника, что его можно считать бесконечным. Поэтому скорость поступления элементов в систему не зависит от количества поступивших элементов. Если же источник элементов конечный, то количество доступных для поступления элементов уменьшается на количество элементов, находящихся в настоящий момент в системе. Как правило, это приводит к пропорциональному сокращению скорости поступления элементов. Задачи организации сетей и серверов обычно можно решать в предположении бесконечного источника.
- **Длина очереди.** Мы считаем, что длина очереди бесконечна и, таким образом, может расти безгранично. Если же очередь конечна, элементы в системе могут быть потеряны; если очередь заполнена и в систему поступают дополнительные элементы, некоторые из элементов будут отвергнуты. На практике любая очередь конечна, но зачастую для анализа это отличие несущественно. Мы еще рассмотрим вкратце данный вопрос далее в этой главе.

- Стратегия диспетчеризации.** Когда сервер становится свободным, если обслуживания ожидает не один элемент, то необходимо принять решение, какой именно элемент будет направлен на обслуживание серверу следующим. В простейшем случае это можно сделать по принципу “первым пришел — первым обслужен” (или “первым пришел — первым ушел” (FIFO)). Обычно при употреблении термина *очередь* подразумевается именно такая стратегия. Имеется и иная возможность обслуживания — по принципу “последним пришел — последним ушел” (LIFO). А зачастую применяется стратегия диспетчеризации, основанная на относительном приоритете. Например, маршрутизатор может использовать информацию о качестве обслуживания (QoS) для того, чтобы отдать предпочтение обработке некоторых пакетов. О диспетчеризации на основе приоритетов речь пойдет далее. Еще одна стратегия диспетчеризации, встречающаяся на практике, основана на времени обслуживания. Например, планировщик процессов может организовать их диспетчеризацию, сначала обслуживая тот из них, который имеет наименьшее время обслуживания (обеспечивая как можно меньшее время ожидания наибольшему количеству процессов). К сожалению, стратегию, основанную на времени обслуживания, очень трудно смоделировать аналитически.

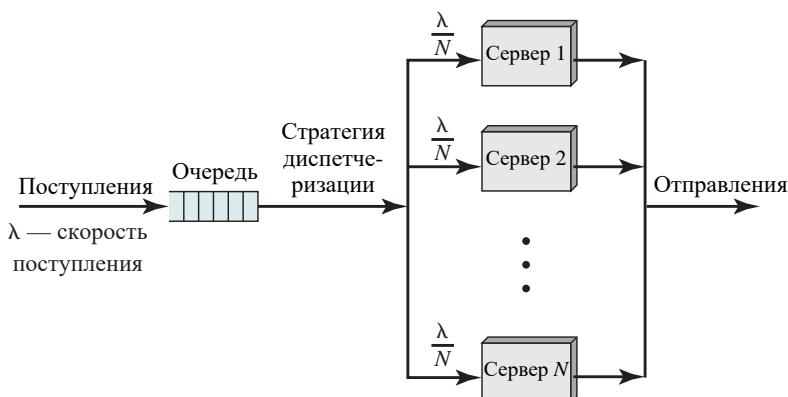
Обозначения, используемые на рис. 21.2, сведены в табл. 21.4, где представлены и другие полезные параметры. В частности, зачастую интерес представляет изменчивость различных параметров, которая характеризуется стандартным отклонением.

**Таблица 21.4. Основные параметры систем массового обслуживания**

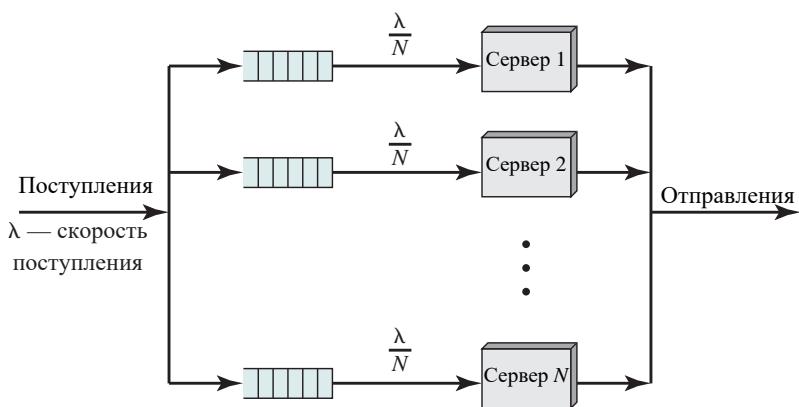
$\lambda$	Скорость поступления; среднее количество поступлений в секунду
$T_s$	Среднее время обслуживания каждого поступления; период времени обслуживания без учета времени ожидания в очереди
$\sigma_{Ts}$	Стандартное отклонение времени обслуживания
$\rho$	Коэффициент использования; доля времени, в течение которого сервер (или сервера) занят обслуживанием
$u$	Интенсивность трафика
$r$	Среднее количество элементов в системе, ожидающих и обслуживаемых
$R$	Количество элементов в системе, ожидающих и обслуживаемых
$T_r$	Среднее время пребывания элемента в системе
$T_R$	Время пребывания элемента в системе
$\sigma_r$	Стандартное отклонение $r$
$\sigma_{Tr}$	Стандартное отклонение $T_r$
$w$	Среднее количество элементов, ожидающих обслуживания
$\sigma_w$	Стандартное отклонение $w$
$T_w$	Среднее время ожидания всех элементов (в том числе обслуживаемые сразу же)
$T_d$	Среднее время ожидания элементов, вынужденных ожидать обслуживания
$N$	Количество серверов
$m_x(y)$	$y$ -й процентиль; значение $x$ , меньшее указанного, встречается у процентов раз

## Многоканальная система массового обслуживания

Обобщенное представление простой модели для нескольких серверов, совместно использующих общую очередь, показано на рис. 21.4, а. Если при поступлении элементов доступен хотя бы один сервер, элемент сразу же направляется ему на обслуживание. При этом считается, что все серверы одинаковы. Таким образом, если для обслуживания доступен не один сервер, выбор конкретного сервера для обслуживания элемента не оказывает никакого влияния на время обслуживания. Если все серверы заняты, формируется очередь. Как только один из серверов освобождается, элемент из очереди (согласно действующей стратегии диспетчеризации) направляется ему на обслуживание.



а) Многоканальная система массового обслуживания



б) Несколько одноканальных систем массового обслуживания

**Рис. 21.4.** Многоканальная система массового обслуживания в сравнении с несколькими одноканальными системами массового обслуживания

Все параметры, показанные на рис. 21.2 (кроме коэффициента использования), переносятся в многоканальную систему массового обслуживания с тем же самым смыслом. Если в такой системе имеется  $N$  одинаковых серверов, то параметр  $\rho$  обозначает коэффициент использования каждого сервера, а  $N\rho$  можно рассматривать как коэффициент

использования всей системы в целом. Этот последний параметр нередко обозначается термином *интенсивность трафика* (обозначается как  $u$ ). Таким образом, теоретический максимальный коэффициент использования равен  $N \times 100\%$ , а теоретическая максимальная скорость поступления равна

$$\lambda_{\max} = \frac{N}{T_s}.$$

Ключевые характеристики, обычно выбираемые для многоканальной системы массового обслуживания, соответствуют характеристикам, которые выбираются для одноканальной системы массового обслуживания. При этом предполагается бесконечность источника поступающих элементов и длины очереди, которая является единственной, совместно используемой всеми серверами в системе. Если не указано иное, стратегия диспетчеризации следует принципу FIFO. Если все серверы в многоканальной системе массового обслуживания предполагаются одинаковыми, то выбор конкретного сервера для элемента, ожидающего обслуживания, не оказывает никакого влияния на время обслуживания.

Для сравнения на рис. 21.4, б показана структура нескольких одноканальных систем массового обслуживания. Как мы увидим, это кажущееся малозначительным изменение в структуре оказывает значительное влияние на производительность.

## Основные соотношения из теории массового обслуживания

Чтобы продвинуться в рассмотрении данного предмета намного дальше, придется сделать ряд упрощающих допущений, хотя такие допущения и рисуют сделать описываемые здесь модели менее достоверными для различных реальных ситуаций. Правда, результаты зачастую получаются достаточно точными для целей планирования и проектирования.

Имеются, однако, некоторые соотношения, справедливые в общем случае и приведенные в табл. 21.5. Сами по себе эти соотношения не особенно полезны, хотя с их помощью можно найти ответы на некоторые основные вопросы. Рассмотрим в качестве примера шпиона из компании Burger King, пытающегося выяснить, сколько посетителей находятся в закусочной от компании McDonald's через дорогу. Он не может просидеть весь день в закусочной McDonald's, и поэтому ему придется искать ответ, основываясь только на своих наблюдениях за потоком входящих и выходящих в здание посетителей. За весь день он наблюдает, что в среднем ежечасно посещают закусочную 32 человека. Обращая внимание на некоторых людей, он обнаруживает, что в среднем посетитель проводит в закусочной 12 минут. Пользуясь формулой Литтла, соглядатай делает заключение, что в любой заданный момент времени в закусочной McDonald's находится в среднем 6,4 человека ( $6,4 = 32$  посетителя в час  $\times 0,2$  часа на посетителя).

**Таблица 21.5. Некоторые основные соотношения из теории массового обслуживания**

Общее соотношение	Одноканальная система	Многоканальная система
$r = \lambda T_r$ — формула Литтла	$\rho = \lambda T_s$	$\rho = \lambda T_s / N$
$w = \lambda T_w$ — формула Литтла	$r = w + \rho$	$u = \lambda T_s = \rho N$
$T_r = T_w + T_s$		$r = w + N\rho$

В данный момент было бы полезно дать интуитивное представление об уравнениях, приведенных в табл. 21.5. Что касается уравнения  $\rho = \lambda T_s$ , то при скорости поступления  $\lambda$  среднее время между поступлениями составляет  $1/\lambda = T$ . Если  $T > T_s$ , то в течение промежутка времени  $T$  сервер занят только время  $T_s$ , что дает коэффициент использования  $T_s/T = \lambda T_s$ . Применяя аналогичное рассуждение к многоканальной системе массового обслуживания, можно получить соотношение  $\rho = (\lambda T_s)/N$ .

Чтобы стала понятнее формула Литтла (Little), рассмотрим следующее рассуждение, опирающееся на опыт обслуживания одного элемента. Когда элемент поступает в систему, он обнаруживает, что своей очереди на обслуживание уже ожидает в среднем  $w$  элементов. Элемент, покидая очередь, чтобы быть обслуженным, оставляет за собой в среднем такое же количество  $w$  элементов в очереди. Чтобы понять, почему, заметим, что в то время как элемент ожидает обслуживания, очередь перед ним сокращается до тех пор, пока он окажется в ее голове. В то же время в систему поступают дополнительные элементы, которые становятся в очередь за ним. Когда же элемент покидает очередь, чтобы быть обслуженным, количество стоящих за ним в очереди элементов равно  $w$ , поскольку параметр  $w$  определяется как среднее количество ожидающих обслуживания элементов. Далее, среднее время ожидания элементом обслуживания равно  $T_w$ . Поскольку элементы поступают со скоростью  $\lambda$ , то можно заключить, что за время  $T_w$  должно поступить общее количество элементов  $\lambda T_w$ . Следовательно,  $w = \lambda T_w$ . Аналогичное рассуждение применимо и к соотношению  $r = \lambda T_r$ .

Если обратиться к последнему уравнению в первом столбце табл. 21.5, то нетрудно заметить, что время, которое элемент проводит в системе, состоит из суммы времени ожидания обслуживания и времени самого обслуживания элемента. Таким образом, в среднем  $T_r = T_w + T_s$ . Последние уравнения во втором и третьем столбцах 21.5 тоже совсем нетрудно обосновать. В любой момент времени количество элементов в системе состоит из суммы количеств ожидающих и обслуживаемых элементов. В одноканальной системе массового обслуживания среднее количество обслуживаемых элементов равно  $\rho$ . Следовательно, для такой системы  $r = w + \rho$ . Аналогично для  $N$ -канальной системы массового обслуживания  $r = w + N\rho$ .

## Предположения

Первостепенная задача анализа очередей формулируется следующим образом:  
для входных данных

- скорость поступления,
- время обслуживания,
- количество серверов

предоставить в качестве выходных данных следующие сведения:

- количество ожидающих элементов;
- время ожидания;
- количество элементов в системе;
- время нахождения в системе.

Что конкретно хотелось бы знать об этих выходных данных? Безусловно, их средние значения —  $(w, T_w, r, T_r)$ . Кроме того, было бы полезно знать кое-что об их изменчивости. Следовательно, было бы неплохо выяснить и стандартное отклонение каждого из них

$(\sigma_r, \sigma_{T_r}, \sigma_w, \sigma_{T_w})$ . Полезными могут оказаться и другие показатели. Например, чтобы спроектировать буфер, связанный с маршрутизатором или мультиплексором, целесообразно выяснить размер буфера, при котором вероятность его переполнения окажется меньше 0,001. Иными словами, каким должно быть значение  $N$ , чтобы выполнялось соотношение

$$\Pr[\text{количество ожидающих элементов} < N] = 0.999?$$

Чтобы найти ответы на подобные вопросы, в общем случае требуется полностью знать распределение вероятности промежутков времени между последовательными поступлениями элементов, а также времени обслуживания. Но даже при таком знании получаемые в результате формулы оказываются чрезмерно сложными. А потому, чтобы задача стала разрешимой, необходимо сделать ряд упрощающих предположений.

Самое важное из них касается скорости поступления. Мы предполагаем, что промежутки времени между поступлениями подчиняются экспоненциальному закону. Иными словами, количество поступлений за период времени  $t$  подчиняется распределению Пуассона, а это означает, что поступления происходят случайно и независимо одно от другого. Такое допущение делается почти всегда, ведь без этого большая часть анализа очередей оказывается непрактичной или по крайней мере достаточно трудной. При таком допущении оказывается, что многие полезные результаты могут быть получены, если известны всего лишь среднее значение и стандартное отклонение скорости поступления и времени обслуживания. Еще больше упростить дело и получить более подробные результаты можно, если допустить, что время обслуживания постоянно или подчиняется экспоненциальному закону.

Для подытоживания основных допущений, которые делаются при разработке модели очередей, было разработано так называемое *обозначение Кендалла*. Это обозначение  $X/Y/N$ , где  $X$  — распределение промежутков времени между поступлениями,  $Y$  — распределение времени обслуживания, а  $N$  — количество серверов. Ниже перечислены самые распространенные в теории массового обслуживания виды распределений.

$G$  — произвольное распределение промежутков времени между поступлениями или периодов времени обслуживания.

$GI$  — такое произвольное распределение промежутков времени между поступлениями с ограничением, что они взаимно независимы.

$M$  — отрицательное экспоненциальное распределение.

$D$  — детерминированные поступления или обслуживание фиксированной длительности.

Таким образом,  $M/M/1$  означает модель одноканальной системы массового обслуживания с пуассоновским распределением поступлений (экспоненциально распределенными промежутками времени между поступлениями) и экспоненциально распределенных периодов обслуживания.

## 21.4. Одноканальные системы массового обслуживания

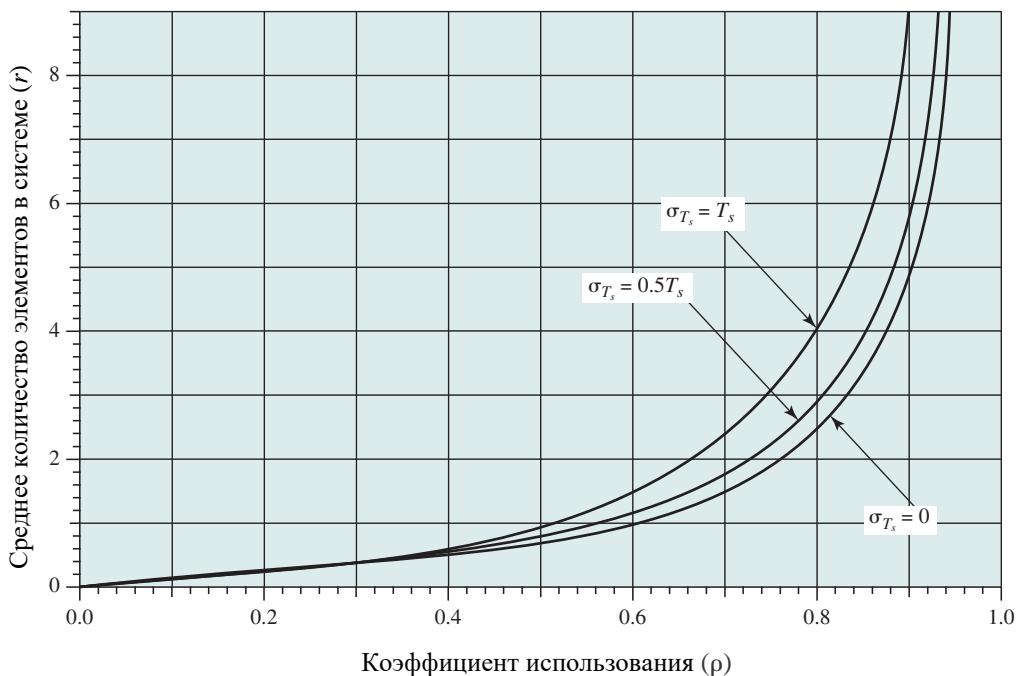
В табл. 21.6, *a* приведен ряд уравнений для проектирования одноканальных систем массового обслуживания по модели  $M/G/1$ .

**Таблица 21.6. Формулы для одноканальных систем массового обслуживания**

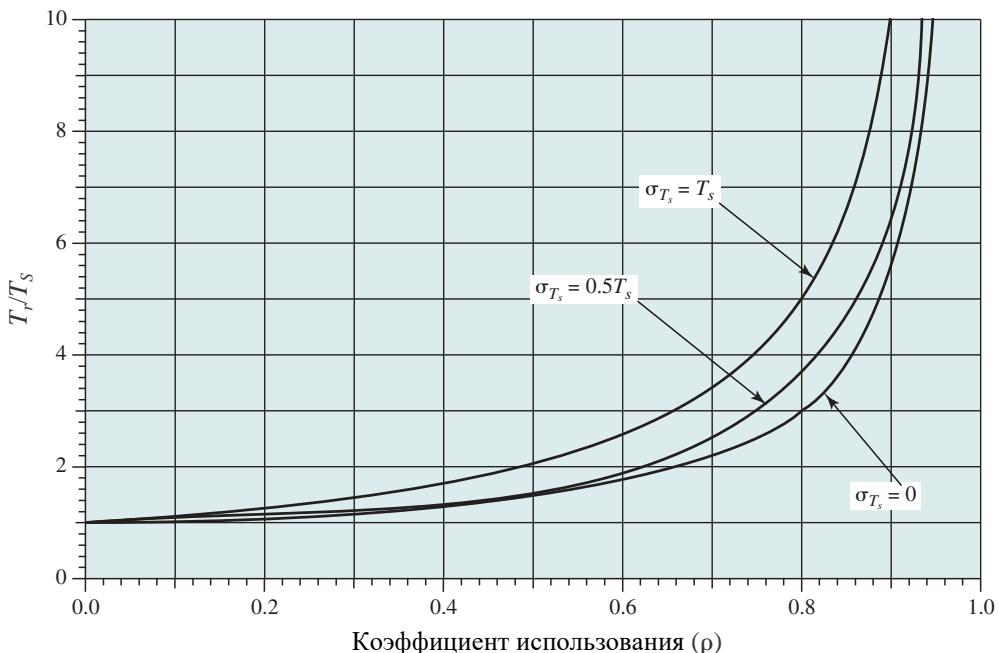
Допущения:	1. Скорость поступления имет распределение Пуассона. 2. Стратегия диспетчеризации не дает никаких преимуществ на основании времени обслуживания. 3. В формулах для стандартного отклонения предполагается диспетчеризация FIFO ('первым пришел – первым вышел'). 4. Ни один элемент из очереди не отвергается.	
(а) Произвольное время обслуживания (M/G/1)	<b>(б) Экспоненциальное распределение времени обслуживания (M/M/1)</b> $A = \frac{1}{2} \left[ 1 + \left( \frac{\sigma_{T_s}}{T_s} \right)^2 \right]$ $r = \rho + \frac{\rho^2 A}{1 - \rho}$ $w = \frac{\rho^2 A}{1 - \rho}$ $T_r = T_s + \frac{\rho T_s}{1 - \rho}$ $\sigma_r = \frac{\sqrt{\rho}}{1 - \rho}$ $Pr[R = N] = (1 - \rho) \rho^N$ $Pr[R \leq N] = \sum_{i=0}^N (1 - \rho) \rho^i$ $T_r = T_s + \frac{\rho T_s A}{1 - \rho}$ $T_w = \frac{\rho T_s}{1 - \rho}$ $Pr[T_R \leq T] = 1 - e^{-(1-\rho)wT_s}$ $m_{T_r}(y) = T_r \times \ln\left(\frac{100}{100 - y}\right)$ $m_{T_w}(y) = \frac{T_w}{\rho} \times \ln\left(\frac{100\rho}{100 - y}\right)$	
(в) Константное время обслуживания (M/D/1)	$r = \frac{\rho^2}{2(1 - \rho)} + \rho$ $w = \frac{\rho^2}{2(1 - \rho)}$ $T_r = \frac{T_s}{2(2 - \rho)}$ $T_r = \frac{\rho T_s}{2(1 - \rho)}$ $T_w = \frac{1}{2(1 - \rho)} \sqrt{\frac{3\rho^2}{2} + \frac{5\rho^3}{6} - \frac{\rho^4}{12}}$ $\sigma_r = \frac{T_s}{1 - \rho} \sqrt{\frac{\rho^2}{3} - \frac{\rho^2}{12}}$	

Это означает, что скорость поступления имеет распределение Пуассона, а время обслуживания — произвольное распределение. Определенное упрощение некоторых из основных выходных переменных можно выполнить с помощью масштабного множителя  $A$ . Ключевым фактором в масштабном множителе является отношение стандартного отклонения времени обслуживания к среднему значению. Никаких других сведений о времени обслуживания больше не требуется. Некоторый интерес представляют два особых случая. Так, если стандартное отклонение равно среднему значению, время обслуживания распределено экспоненциально ( $M/M/1$ ). Это простейший случай, наименее трудный для вычисления результатов. В табл. 21.6, б приведены упрощенные версии уравнений для определения стандартного отклонения параметров  $r$  и  $T_s$ , а также других представляющих интерес параметров. Другой интересный случай — когда стандартное отклонение времени обслуживания равно нулю, т.е. мы имеем постоянное время обслуживания ( $M/D/1$ ). Уравнения, соответствующие этому случаю, приведены в табл. 21.6, в.

На рис. 21.5 и 21.6 приведены графики зависимостей средней длины очереди и времени нахождения в системе от коэффициента использования для трех значений  $\sigma_{T_s}/T_s$ . Этот параметр называется **коэффициентом изменчивости** и дает нормализованный показатель изменчивости. Следует заметить, что наихудшая производительность демонстрируется экспоненциальным временем обслуживания, а наилучшая — постоянным временем обслуживания. Зачастую экспоненциальное время обслуживания можно рассматривать как наихудший случай, и поэтому анализ, основанный на таком предположении, дает консервативные результаты. Это совсем не плохо, поскольку для случая  $M/M/1$  имеются соответствующие таблицы, в которых можно быстро найти нужные значения.



**Рис. 21.5.** Среднее количество элементов в одноканальной системе массового обслуживания



**Рис. 21.6.** Среднее время пребывания в одноканальной системе массового обслуживания

Какие значения отношения  $\sigma_{T_s}/T_s$  наиболее вероятны? Их можно разбить на четыре категории.

- **Нуль.** Это редкий случай постоянного времени обслуживания. Например, если все передаваемые пакеты оказываются одинаковой длины, они подпадают под эту категорию.
- **Отношение меньше 1.** Такое отношение лучше, чем при экспоненциальном распределении, так что применение таблиц модели  $M/M/1$  консервативно даст несколько большие, “с запасом” длины очередей и времена, чем должны быть. Применение модели  $M/M/1$  позволит найти безопасные ответы на вопросы. Примером данной категории может быть приложение для ввода данных в некотором определенном виде.
- **Отношение близко к 1.** Такое отношение встречается очень часто и соответствует экспоненциальному, а по сути, случайному времени обслуживания. Рассмотрим длину сообщений на компьютерном терминале. Весь экран может быть заполнен 1920 символами, и в этих полных пределах может меняться длина сообщений. Примерами приложений, которые нередко относятся к данной категории, являются системы бронирования авиабилетов, поиска файлов, локальные сети с общим доступом, а также сети с коммутацией пакетов.
- **Отношение больше 1.** Если наблюдается именно такое отношение, то необходимо воспользоваться моделью  $M/G/1$  и не полагаться на модель  $M/M/1$ . Такое отношение нередко возникает при двухмодальном распределении с большим разбросом вершин. Примером данной категории является система, которая испытывает приток большого количества коротких и длинных сообщений, но малого количества сообщений средней длины.

Аналогичные соображения применимы и к скорости поступления. При пуассоновском поступлении промежутки времени между отдельными поступлениями распределены экспоненциально, а отношение стандартного отклонения и среднего значения равно 1. Если же наблюдается отношение, намного меньшее единицы, то поступления распределяются равномерно (без особой изменчивости). Предположение о пуассоновском распределении приведет к завышенной оценке длины очередей и задержек в обслуживании. Если же отношение больше единицы, наблюдается тенденция к кластеризации поступлений, и проблема перегрузки системы массового обслуживания становится более острой.

## 21.5. МНОГОКАНАЛЬНЫЕ СИСТЕМЫ МАССОВОГО ОБСЛУЖИВАНИЯ

В табл. 21.7 приведены формулы для некоторых ключевых параметров многоканальных систем массового обслуживания. При этом следует иметь в виду ограниченность предположений. Полезная информация была получена только для случая  $M/M/N$ , где экспоненциальное время обслуживания одинаково для всех  $N$  серверов.

Во всех уравнениях, приведенных в табл. 21.7, присутствует С-функция Эрланга (Erlang), выражающая вероятность занятости всех серверов в заданный момент времени. В равной степени это вероятность, что количество элементов в системе (ожидающих обслуживания и обслуживаемых) не меньше количества серверов. Данная функция имеет следующий вид:

$$C(N, \rho) = \frac{1 - K(N, \rho)}{1 - \rho K(N, \rho)}.$$

Здесь  $K$  — известный коэффициент Пуассона. А поскольку  $C$  — это вероятность, ее значение всегда находится в пределах от 0 до 1. Как было показано, это значение является функцией количества серверов и коэффициента их использования. Это выражение часто встречается в теории массового обслуживания, и для него составлены подробные таблицы значений, а также имеются готовые программы для его вычисления. Заметим, что для одноканальной системы массового обслуживания приведенное выше уравнение упрощается до  $C(1, \rho) = \rho$ .

## 21.6. ПРИМЕРЫ

Рассмотрим ряд примеров, дающих представление о применении упомянутых выше уравнений.

### Сервер базы данных

Рассмотрим локальную сеть, к которой подключено 100 персональных компьютеров и сервер, поддерживающий общую базу данных для приложения, делающего запросы к ней. Среднее время отклика сервера на запрос составляет 0,6 с, а стандартное отклонение оценивается как равное среднему значению. В моменты пиковой нагрузки частота поступления запросов через локальную сеть достигает величины 20 запросов в минуту.

**ТАБЛИЦА 21.7. ФОРМУЛЫ ДЛЯ МНОГОКАНАЛЬНЫХ СИСТЕМ МАССОВОГО ОБСЛУЖИВАНИЯ ( $M/M/N$ )**

Допущения	1. Скорость поступления подчиняется распределению Пуассона 2. Экспоненциально распределенное время обслуживания 3. Все серверы загружены одинаково 4. Среднее время обслуживания для всех серверов одинаково 5. Диспетчеризация FIFO ("первым пришел — первым ушел") 6. Ни один из элементов не отвергается из очереди
-----------	---

$$\text{Коэффициент Пуассона } K = \frac{\sum_{I=0}^{N-1} \frac{(N\rho)^I}{I!}}{\sum_{I=0}^N \frac{(N\rho)^I}{I!}}$$

С-функция Эрланга: вероятность занятости всех серверов  $C = \frac{1 - K}{1 - \rho K}$

$$r = C \frac{\rho}{1 - \rho} + N\rho \quad w = C \frac{\rho}{1 - \rho}$$

$$T_r = \left(\frac{C}{N}\right) \frac{T_s}{1 - \rho} + T_s \quad T_w = \left(\frac{C}{N}\right) \frac{T_s}{1 - \rho}$$

$$\sigma_{T_r} = \frac{T_s}{N(1 - \rho)} \sqrt{C(2 - C) + N^2(1 - \rho)^2}$$

$$\sigma_w = \frac{1}{1 - \rho} \sqrt{C\rho(1 + \rho - C\rho)}$$

$$\Pr[T_w > t] = Ce^{-N(1-\rho)t/T_s}$$

$$m_{T_w}(y) = \frac{T_s}{N(1 - \rho)} \ln\left(\frac{100C}{100 - y}\right)$$

$$T_d = \frac{T_s}{N(1 - \rho)}$$

Требуется найти ответы на следующие вопросы.

- Каково среднее время отклика, если пренебречь накладными расходами канала связи?
- Если время отклика 1,5 с считается максимальным приемлемым, то какой именно рост в процентах нагрузки возможен, прежде чем будет достигнут приемлемый максимум?
- Если коэффициент использования увеличится на 20%, то увеличится ли при этом время отклика больше или меньше чем на 20%?

Будем считать, что для построения рассматриваемой здесь системы применяется модель  $M/M/1$ , в которой сервером служит сервер базы данных. Не будем учитывать влияние локальной сети, полагая, что ее влияние на задержку ответов на запросы пре-небрежимо мало. Коеффициент использования данной системы вычисляется следующим образом:

$$\begin{aligned}\rho &= \lambda T_s \\ &= (20 \text{ поступлений в минуту})(0,6 \text{ с на передачу})/(60 \text{ с/мин}) \\ &= 0,2.\end{aligned}$$

Первое значение, среднее время отклика, нетрудно вычислить следующим образом:

$$\begin{aligned}T_r &= T_s / (1 - \rho) \\ &= 0,6 / (1 - 0,2) = 0,75 \text{ с.}\end{aligned}$$

Получить второе значение труднее. Ответа на второй вопрос в том виде, в каком он поставлен выше, на самом деле не существует, поскольку имеется ненулевая вероятность, что в какой-то момент времени отклика превысит 1,5 с при любом значении коэффициента использования. Вместо этого скажем, что было бы желательно, чтобы 90% всех откликов системы происходило меньше чем за 1,5 с. Тогда мы можем воспользоваться следующим уравнением из табл. 21.6, б:

$$\begin{aligned}m_{T_r}(y) &= T_r \times \ln\left(\frac{100}{100-y}\right), \\ m_{T_r}(90) &= T_r \times \ln(10) = \frac{T_s}{1-\rho} \times 2,3 = 1,5 \text{ с.}\end{aligned}$$

Если  $T_s = 0,6$ , то решение приведенного выше уравнения для  $\rho$  дает в итоге  $\rho = 0,08$ . В действительности, чтобы величина 1,5 была 90-м процентилем, коэффициент использования должен быть снижен с 20% до 8%.

Наконец, третий вопрос состоит в нахождении взаимосвязи между увеличением нагрузки и временем отклика. Коэффициент использования данной системы равен 0,2 и соответствует плоской части соответствующего графика, поэтому время отклика будет увеличиваться медленнее, чем коэффициент использования. В данном случае, если коэффициент использования системы увеличивается с 20% до 40%, что составляет увеличение на 100%, значение  $T_r$  возрастает с 0,75 до 1,0 с, что составляет увеличение только на 33%.

## Вычисление процентилей

Рассмотрим в качестве примера конфигурацию, в которой пакеты отсылаются из компьютеров по локальной сети системам, подключенным к другим сетям. Все эти пакеты должны пройти через маршрутизатор, соединяющий локальную сеть с глобальной сетью, а следовательно, с внешним миром. Проанализируем трафик, проходящий из локальной сети в глобальную сеть через маршрутизатор. Пакеты поступают со средней скоростью 5 пакетов в секунду, а средняя длина пакета составляет 144 октета. При этом предполагается, что длина пакета распределена по экспоненциальному закону. Скорость передачи данных по линии из маршрутизатора в глобальную сеть составляет 9600 бит/с. Требуется найти ответы на следующие вопросы.

1. Каково среднее время пребывания пакетов в маршрутизаторе?
2. Сколько в среднем пакетов находится в маршрутизаторе, включая ожидающие передачи и передающиеся в настоящий момент (если таковые имеются)?
3. То же, что и в вопросе 2, но для 90-го процентиля.
4. То же, что и в вопросе 2, но для 95-го процентиля.

$\lambda = 5$  пакетов в секунду

$$T_s = (144 \text{ октета} \times 8 \text{ Бит/октет}) / 9600 \text{ бит/с} = 0,12 \text{ с}$$

$$\rho = \lambda T_s = 5 \times 0,12 = 0,6$$

$$T_r = T_s / (1 - \rho) = 0,3 \text{ с} \quad \text{Среднее время нахождения в системе}$$

$$r = \rho / (1 - \rho) = 1,5 \text{ пакета} \quad \text{Среднее количество элементов в системе}$$

Чтобы получить процентили, воспользуемся следующим уравнением из табл. 21.6, б:

$$\Pr[R = N] = (1 - \rho) \rho^N.$$

Для вычисления  $y$ -го процентиля длины очереди запишем предыдущее уравнение в следующем виде:

$$\frac{y}{100} = \sum_{k=0}^{m_r(y)} (1 - \rho) \rho^k = 1 - \rho^{1+m_r(y)}.$$

Здесь  $m_r(y)$  обозначает максимальное количество пакетов, находящихся в очереди в течение  $y$  процентов всего времени ожидания, т.е. такое количество  $m_r(y)$ , при котором значение, меньшее  $R$ , встречается  $y$  процентов всего времени ожидания. Приведенная выше форма записи позволяет определить процентиль для любой длины очереди. Но в данном случае требуется сделать обратное: найти величину  $m_r(y)$  по заданному  $y$ -му процентилю. С этой целью следует прологарифмировать обе части приведенного выше уравнения:

$$m_r(y) = \frac{\ln\left(1 - \frac{y}{100}\right)}{\ln \rho} - 1.$$

Если величина  $m_r(y)$  окажется дробной, нужно взять следующее по порядку большее целочисленное значение, а если она отрицательна, то установить ее равной нулю. В рассматриваемом примере требуется найти величины  $m_r(90)$  и  $m_r(95)$  при  $\rho = 0,6$ .

$$m_r(90) = \frac{\ln(1 - 0,90)}{\ln(0,6)} - 1 = 3,5$$

$$m_r(95) = \frac{\ln(1 - 0,95)}{\ln(0,6)} - 1 = 4,8$$

Таким образом, 90% всего времени в очереди находится меньше 4 пакетов, а 95% всего времени — меньше 5 пакетов. Если проектировать буфер по критерию 95-го процентиля, то буфер должен быть достаточен для хранения как минимум 5 пакетов.

## СИЛЬНО СВЯЗАННЫЙ МУЛЬТИПРОЦЕССОР

Рассмотрим применение многих сильно связанных мультипроцессоров в единой вычислительной системе. Одно из проектных решений имеет непосредственное отноше-

ние к равномерному распределению процессов по отдельным процессорам. Так, если процесс назначается одному процессору от начала и вплоть до его завершения, то для каждого процессора организуется отдельная краткосрочная очередь. И в этом случае один процессор может простоять с пустой очередью, тогда как у другого процессора имеется ряд незавершенных процессов. Во избежание подобной ситуации может быть организована общая очередь. Все процессы направляются в одну общую очередь и планируются для выполнения любым доступным процессором. Таким образом, процесс может выполняться в течение своего срока жизни на разных процессорах в разное время.

Попробуем выяснить, как добиться повышения производительности, используя общую очередь. Рассмотрим в качестве примера систему, состоящую из пяти процессоров, в которой каждому процессу, находящемуся в состоянии выполнения, предоставляется среднее время процессора, равное 0,1 с. Допустим, что наблюдаемое стандартное отклонение времени обслуживания составляет 0,094 с. Такое стандартное отклонение близко к среднему значению, и поэтому предположим, что время обслуживания распределено экспоненциально. Допустим также, что процессы поступают на обслуживание в состоянии готовности к выполнению со скоростью 40 процессов в секунду.

### Односерверный подход

Если процессы распределены среди процессоров равномерно, то нагрузка на каждый процессор составляет  $40/5 = 8$  процессов в секунду. Следовательно,

$$\rho = \lambda T_s = 8 \times 0,1 = 0,8.$$

Теперь можно легко вычислить время пребывания в системе по следующей формуле:

$$t_r = \frac{T_s}{1-\rho} = \frac{0,1}{0,2} = 0,5 \text{ с.}$$

### Многосерверный подход

Предположим, что для всех процессоров организована единая очередь готовых к выполнению процессов, а совокупная скорость поступления составляет 40 процессов в секунду. Коэффициент использования системы по-прежнему равен 0,8 ( $\lambda T_s/M$ ). Чтобы вычислить время пребывания в системе по формуле из табл. 21.7, необходимо сначала вычислить С-функцию Эрланга. В отсутствие программы для определения данного параметра его можно найти в соответствующей таблице. Так, если коэффициент использования системы из пяти серверов равен 0,8, то  $C = 0,544$ . Подставив эту величину в формулу, можно вычислить время пребывания в системе:

$$T_r = (0,1) + \frac{(0,544)(0,1)}{5(1-0,8)} = 0,1544 .$$

Таким образом, благодаря применению многоканальной системы массового обслуживания среднее время пребывания сократилось с 0,5 до 0,1544 с (т.е. больше чем в три раза). Если принять во внимание только время ожидания, то его величина (0,0544 с) в многоканальной системе массового обслуживания сократилась в семь раз по сравнению с аналогичной величиной 0,4 с в одноканальной системе массового обслуживания.

Вы можете не быть знатоком теории массового обслуживания, но вы должны отчетливо представлять себе, как это раздражает — стоять в очереди в кассу, когда рядом находятся две свободные.

## Задача построения многоканальной системы массового обслуживания

Проектная фирма предоставляет каждому из своих аналитиков персональный компьютер, и все они подключены через локальную сеть к серверу базы данных. Имеется также дорогостоящая автономная рабочая станция, предназначенная для решения задач проектирования особого назначения. В течение обычного 8-часового рабочего дня инженеров этой рабочей станцией пользуются 10 инженеров, проводя в среднем по 30 минут за сеанс работы на ней.

### Одноканальная модель массового обслуживания

Инженеры жалуются своему начальству, что им приходится долго ждать возможности воспользоваться рабочей станцией (зачастую более часа), и поэтому они просят установить больше рабочих станций. Эта просьба удивляет начальство, поскольку коэффициент использования рабочей станции составляет всего лишь  $5/8$  ( $10 \times 1/2 = 5$  часов из 8 рабочих). Чтобы убедить начальство в справедливости своей просьбы, один из инженеров выполняет анализ очередей. При этом он делает обычные предположения о бесконечном источнике элементов, случайному их поступлению и экспоненциальному распределении периодов времени обслуживания, — все, что представляется вполне разумным для грубых подсчетов. Пользуясь уравнениями из табл. 21.5 и 21.6, б, инженер получает приведенные ниже параметры проектируемой системы массового обслуживания.

$T_w = \frac{\rho T_s}{1 - \rho} = 50 \text{ мин}$	Среднее время ожидания инженером возможности воспользоваться рабочей станцией
$m_{T_w} = \frac{T_w}{\rho} \times \ln(10\rho) = 146,6 \text{ мин}$	90-й процентиль времени ожидания
$\lambda = \frac{10}{8 \times 60} = 0,021 \text{ инженера/мин}$	Скорость поступления элементов (в данном случае — скорость обращения инженеров к рабочей станции)
$w = \lambda T_w = 1,0416 \text{ инженера}$	Среднее количество ожидающих своей очереди инженеров

Приведенные цифры ясно показывают, что инженерам действительно приходится ждать в среднем целый час, чтобы воспользоваться рабочей станцией, а в 10% всех случаев — больше двух часов. И даже при наличии в приведенных выше оценках существенной погрешности (скажем, порядка 20%) время ожидания все равно оказывается чрезмерно долгим. Более того, если инженер не выполняет никакой полезной работы, ожидая своей очереди для того, чтобы воспользоваться рабочей станцией, то каждый рабочий день теряется чуть больше одного инженерного человека-дня.

### Многоканальная модель массового обслуживания

Инженеры убедили начальство в необходимости установить больше рабочих станций. Им хотелось бы, чтобы время ожидания не превышало 10 минут, а для 90-го процентиля — 15 минут. Менеджеры при этом рассуждают следующим образом: если доступа к рабочей станции приходится ждать 50 минут, то для снижения данного показателя до 10 минут потребуется пять рабочих станций.

Инженеры приступают к работе, чтобы определить, сколько же рабочих станций им требуется. Имеются два варианта действий: установить дополнительные рабочие станции в том же помещении, где находится первая (многоканальная система массового обслуживания), или же распределить их по разным помещениям на разных этажах здания (несколько одноканальных систем массового обслуживания). Рассмотрим сначала многоканальный вариант со второй рабочей станцией в том же помещении. Допустим, что добавление новой рабочей станции для сокращения времени ожидания не оказывает никакого влияния на скорость поступления элементов на обслуживание (в данном случае — по 10 инженеров за рабочий день). В таком случае доступное время обслуживания составит 16 часов в течение 8-часового рабочего дня при требовании 5 часов (10 инженеров  $\times$  0,5 ч), а коэффициент использования —  $5/16 = 0,3125$ . Используя уравнения из табл. 21.7, можно получить приведенные ниже параметры описанной системы массового обслуживания.

$C(2, \rho) = C(2; 0,3125) = 0,1488$	Вероятность занятости обоих серверов
$T_w = \frac{CT_s}{N(1-\rho)} = 3,247 \text{ мин}$	Среднее время, в течение которого инженер ожидает возможности воспользоваться рабочей станцией
$m_{T_w}(90) = \frac{T_s}{2(1-\rho)} \ln(10C) = 8,67 \text{ мин}$	90-й percentile времени ожидания
$w = \lambda T_w = 0,07 \text{ инженера}$	Среднее количество ожидающих своей очереди инженеров

При такой организации массового обслуживания вероятность, что инженеру, которому требуется воспользоваться рабочей станцией, придется ждать, составит меньше 0,15; а в среднем время ожидания оказывается чуть больше трех минут при 90-м percentile времени ожидания меньше девяти минут. Несмотря на сомнения начальства, организация с двумя рабочими станциями вполне удовлетворяет проектным требованиям.

Все инженеры располагаются на двух этажах здания, поэтому начальство задается вопросом — не лучше ли установить по одной рабочей станции на каждом этаже? Если допустить, что сетевой трафик почти равномерно распределяется среди двух рабочих станций, то в результате образуются две очереди на обслуживание с моделью  $M/M/1$ , в каждой из которых находится  $\lambda$  из пяти инженеров в течение 8-часового рабочего дня. Это дает следующие показатели обслуживания.

$\rho = \lambda T_s = 0,3125$	Коэффициент использования одного сервера
$T_w = \frac{\rho T_s}{1-\rho} = 13,46 \text{ мин}$	Среднее время, в течение которого инженер ожидает возможности воспользоваться рабочей станцией
$m_{T_w}(90) = \frac{T_w}{\rho} \times \ln(10\rho) = 49,73 \text{ мин}$	90-й percentile времени ожидания
$w = \lambda T_w = 0,142 \text{ инженера}$	Среднее количество ожидающих своей очереди инженеров

Эта производительность значительно хуже, чем в многоканальной модели массового обслуживания, и не удовлетворяет проектным критериям. В табл. 21.8 подытожены результаты установки четырех и пяти отдельных рабочих станций. Заметим, что в этом случае для достижения проектной цели требуется пять отдельных рабочих станций (по сравнению всего лишь с двумя рабочими станциями в одном помещении).

**Таблица 21.8. Сводка результатов вычислений в примере многоканальной системы массового обслуживания**

Количество рабочих станций	Модель системы	$\rho$	$T_w$	$m_{T_w}(90)$
1	$M/M/1$	0,625	50	146,61
2	$M/M/2$	0,3125	3,25	8,67
3	Несколько $M/M/1$	0,3125	13,64	49,73
4	Несколько $M/M/1$	0,15625	5,56	15,87
5	Несколько $M/M/1$	0,125	4,29	7,65

## 21.7. ОЧЕРЕДИ С ПРИОРИТЕТАМИ

До сих пор нами рассматривались очереди, в которых обслуживание элементов осуществляется по принципу “первым пришел — первым вышел”. Но имеется немало таких архитектур сетевых и операционных систем, в которых желательно использовать приоритеты. Приоритеты могут назначаться самыми разными способами, в том числе и по типу сетевого трафика. Если же среднее время обслуживания разных типов сетевого трафика оказывается одинаковым, то общие уравнения для определения параметров такой системы массового обслуживания не меняются, хотя наблюдаемая производительность при разных категориях сетевого трафика будет отличаться.

Здесь важно рассмотреть один случай, когда приоритет назначается по среднему времени обслуживания. Зачастую элементы с более короткими предполагаемыми периодами времени обслуживания получают приоритет над элементами с более продолжительными периодами времени обслуживания. Например, маршрутизатор может присвоить потоку речевых пакетов более высокий приоритет, чем потоку пакетов данных. Ведь речевые пакеты, как правило, оказываются намного более короткими, чем пакеты данных. Благодаря такой схеме повышается производительность при обработке высокоприоритетного сетевого трафика.

В табл. 21.9 приведены формулы, полученные в предположении двух классов приоритетов с разным временем обслуживания. Эти формулы и получаемые по ним результаты нетрудно обобщить на любое число категорий приоритетов.

**Таблица 21.9. Формулы для одноканальных систем массового обслуживания с двумя категориями приоритетов**

- Допущения**
1. Скорость поступления подчиняется пуассоновскому распределению
  2. Элементы с приоритетом 1 обслуживаются прежде элементов с приоритетом 2
  3. Диспетчеризация элементов с равным приоритетом использует принцип FIFO (“первым пришел — первым вышел”)
  4. Обслуживание ни одного из элементов не прерывается
  5. Никакие элементы не покидают очередь

**а) Общие формулы**

$$\begin{aligned}\lambda &= \lambda_1 + \lambda_2 \\ \rho_1 &= \lambda_1 T_{s1}; \quad \rho_2 = \lambda_2 T_{s2} \\ \rho &= \rho_1 + \rho_2 \\ T_s &= \frac{\lambda_1}{\lambda} T_{s1} + \frac{\lambda_2}{\lambda} T_{s2} \\ T_r &= \frac{\lambda_1}{\lambda} T_{r1} + \frac{\lambda_2}{\lambda} T_{r2}\end{aligned}$$

**б) Экспоненциальное время обслуживания**

$$\begin{aligned}w_1 &= \frac{\rho_1(\rho_1 T_{s1} + \rho_2 T_{s2})}{T_{s1}(1 - \rho_1)} \\ w_2 &= w_1 \frac{\lambda_2}{\lambda_1(1 - \rho)} \\ T_{r1} &= T_{s1} + \frac{\rho_1 T_{s1} + \rho_2 T_{s2}}{1 - \rho_1} \\ T_{r2} &= T_{s2} + \frac{T_{r1} - T_{s1}}{1 - \rho}\end{aligned}$$

Чтобы увидеть результат назначения приоритетов, рассмотрим простой пример потока данных, состоящего из сочетания длинных и коротких пакетов, передаваемых узлом коммутации пакетов, причем оба эти типа пакетов поступают с одинаковой скоростью. Предположим, что длина и тех, и других пакетов распределена экспоненциально, причем средняя длина длинных пакетов в 10 раз больше, чем коротких. Пусть скорость передачи данных по каналу связи составляет 64 Кбит/с, а средние длины коротких и длинных пакетов составляют 80 и 800 октетов соответственно. Время обслуживания длинных и коротких пакетов равно 0,01 и 0,1 с соответственно, а скорость поступления каждого типа пакетов составляет 8 пакетов в секунду. Для того чтобы короткие пакеты не задерживались длинными пакетами, назначим коротким пакетам более высокий приоритет. Тогда мы получим следующие результаты.

$$\begin{aligned}\rho_1 &= 8 \times 0,001 = 0,08 & \rho_2 &= 8 \times 0,1 = 0,8 & \rho &= 0,88 \\ T_{r1} &= 0,01 + \frac{0,08 \times 0,01 + 0,8 \times 0,1}{1 - 0,08} = 0,098 \text{ с} \\ T_{r2} &= 0,1 + \frac{0,098 - 0,01}{1 - 0,88} = 0,833 \text{ с} \\ T_r &= 0,5 \times 0,098 + 0,5 \times 0,833 = 0,4655 \text{ с}\end{aligned}$$

Таким образом, пакеты с более высоким приоритетом обслуживаются намного лучше, чем пакеты с более низким приоритетом.

## 21.8. Сети очередей

В распределенной среде изолированные очереди, к сожалению, являются не единственной задачей, которую приходится решать аналитику. Нередко анализируемая задача состоит из нескольких взаимосвязанных очередей. Подобная ситуация наглядно показана на рис. 21.7, где узлы обозначают очереди, а соединяющие их линии — поток трафика.

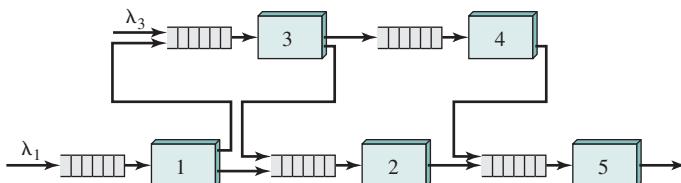


Рис. 21.7. Пример сети очередей

Два элемента такой сети резко усложняют применение рассматривавшихся ранее методов:

- разделение и объединение трафика, показанное на рис. 21.7 в узлах 1 и 5;
- наличие последовательных очередей (тандем), как в узлах 3 и 4 на рис. 21.7.

Для анализа общих задач массового обслуживания не разработано ни одного точного метода с упомянутыми выше элементами. Но если поток имеет распределение Пуассона, а периоды времени обслуживания — экспоненциальное распределение, то на этот случай существует простое и точное решение. В этом разделе сначала рассматриваются два показанных выше элемента, а затем представлен подход к анализу очередей.

### Разделение и объединение потоков трафика

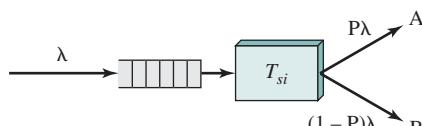
Предположим, что трафик поступает в очередь с единичной средней скоростью и что имеются два пути, А и В, по которым может быть отправлен элемент (рис. 21.8, а). Как только элемент обслужен, он покидает очередь по пути А с вероятностью  $P$ , а по пути В — с вероятностью  $(1 - P)$ . В общем случае распределение потоков трафика А и В будет отличаться от распределения входящего трафика. Но если распределение входящего трафика является пуассоновским, то оба выходящих потока также имеют пуассоновское распределение со средними скоростями  $P\lambda$  и  $(1 - P)\lambda$ .

Подобная ситуация возникает и при объединении трафика (рис. 21.8, б). Если объединяются два пуассоновских процесса со средними скоростями  $\lambda_1$  и  $\lambda_2$ , получающийся в результате поток также имеет пуассоновское распределение со средней скоростью  $\lambda_1 + \lambda_2$ .

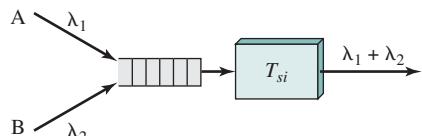
Оба приведенных результата можно обобщить для разделения или объединения более чем двух потоков.

### Последовательные очереди

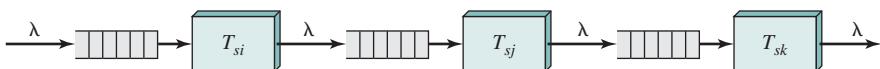
На рис. 21.8, в приведен пример целого ряда одноканальных очередей, действующих последовательно. Входом каждой последующей очереди, кроме первой, является выход предыдущей очереди.



а) Разделение трафика



б) Объединение трафика



в) Простая последовательная очередь

**Рис. 21.8. Элементы сети очередей**

В предположении, что входом каждой очереди является пуассоновский поток, выходом каждой очереди также является пуассоновский поток, статистически идентичный входному, если каждая очередь имеет экспоненциальное распределение и обладает бесконечной емкостью. Когда такой поток поступает в следующую очередь, задержки во второй очереди оказываются такими же, как и в случае, если направить исходный трафик в обход первой очереди непосредственно во вторую очередь. Таким образом, очереди действуют независимо и могут быть проанализированы по одной, а средняя общая задержка последовательной системы массового обслуживания равна сумме средних задержек на каждой стадии.

Этот результат можно распространить и на тот случай, когда некоторые (или все) узлы в последовательной системе являются многоканальными системами массового обслуживания.

## Теорема Джексона

Для анализа сети очередей (или массового обслуживания) можно воспользоваться теоремой Джексона (Jackson). Эта теорема основана на следующих трех предположениях.

- Сеть массового обслуживания состоит из  $m$  узлов, в каждом из которых предоставляется независимая экспоненциально распределенная услуга.
- Элементы поступают из внешней системы в любой из узлов в соответствии с распределением Пуассона.
- Как только элемент обслужен в узле, он либо немедленно направляется в один из других узлов с фиксированной вероятностью, либо вообще выходит из системы.

Согласно теореме Джексона в такой сети очередей каждый узел является независимой системой массового обслуживания, в которой входной пуассоновский поток определяется

принципами разделения, объединения и последовательного обслуживания. Следовательно, каждый узел можно проанализировать отдельно от других узлов, используя модель  $M/M/1$  или  $M/M/N$ , а результаты могут быть объединены с использованием обычновенных статистических методов. Средние задержки в каждом узле могут быть просуммированы с задержками в производной системе, но при этом ничего нельзя сказать о более высоких моментах системных задержек (например, о стандартном отклонении).

Теорема Джексона оказывается привлекательной для применения в сетях с коммутацией пакетов. В частности, сеть с коммутацией пакетов можно смоделировать как сеть очередей, каждый пакет которой представляет собой отдельный элемент. Допустим, что каждый пакет передается отдельно и в каждом узле с коммутацией пакетов на пути от источника к месту назначения пакет ставится в очередь на передачу следующему узлу. Обслуживание пакетов в очереди фактически является передачей пакета и пропорционально длине пакета.

Недостаток такого подхода заключается в том, что нарушается первое условие теоремы Джексона, а именно — распределение услуг совсем не обязательно является независимым. Длина пакета остается одной и той же в каждом канале передачи данных, и поэтому процесс их поступления в каждую очередь коррелирует с процессом обслуживания. Тем не менее Клейнрок (Kleinrock) показал в [134], что усреднение влияния разделения и объединения в предположении независимых периодов времени обслуживания дает неплохую аппроксимацию.

## Применение теоремы Джексона в сети с коммутацией пакетов<sup>3</sup>

Рассмотрим сеть с коммутацией пакетов, состоящую из узлов, соединенных каналами передачи данных, причем каждый узел действует в качестве интерфейса для нуля или большего количества подключенных систем, каждая из которых функционирует в качестве источника и места назначения сетевого трафика. Внешняя рабочая нагрузка на такую сеть может быть рассчитана следующим образом:

$$\gamma = \sum_{j=1}^N \sum_{k=1}^N \gamma_{jk},$$

где

$\gamma$  — общая нагрузка (количество пакетов, передаваемых в секунду);

$\gamma_{jk}$  — рабочая нагрузка между источником  $j$  и местом назначения  $k$ ;

$N$  — общее количество источников и мест назначения.

Пакет между источником и местом назначения может пройти не один канал связи, поэтому общая внутренняя нагрузка будет выше предложенной.

$$\lambda = \sum_{i=1}^L \lambda_i,$$

где

$\gamma$  — общая нагрузка на все каналы связи в сети;

$\gamma_i$  — нагрузка на канал связи  $i$ ;

$L$  — общее количество каналов связи.

---

<sup>3</sup> Обсуждаемый здесь материал основывается на разработке в [134].

Внутренняя нагрузка будет зависеть от фактического пути, проходимого пакетами по сети. Допустим, что алгоритм маршрутизации задается таким образом, чтобы нагрузку на отдельные каналы связи  $\lambda$ , можно было определить из расчетной нагрузки  $\gamma_{jk}$ . По этим параметрам для любой конкретной маршрутизации можно определить среднее количество каналов связи, через которые должен пройти пакет. Немного подумав, можно убедиться, что средняя длина всех путей прохождения пакетов определяется следующим образом:

$$E[\text{количество каналов связи на пути}] = \frac{\lambda}{\gamma}.$$

Теперь цель состоит в том, чтобы определить среднюю задержку  $T$ , испытываемую пакетом, проходящим по сети. С этой целью удобно применить формулу Литтла (см. табл. 21.5). Для каждого канала связи в сети среднее количество ожидающих и обслуживаемых элементов может быть получено по формуле

$$r_i = \lambda_i T_{ri}.$$

Здесь  $T_{ri}$  — еще не определенная задержка обслуживания в каждой очереди. Предположим, что мы просуммировали эти величины. Это дает нам среднее общее количество пакетов, ожидающих обслуживания во всех очередях сети. Оказывается, что формула Литтла пригодна для применения и для совокупных величин.<sup>4</sup> Следовательно, количество пакетов, ожидающих обслуживания в сети, может быть выражено как  $\gamma T$ . Объединив то и другое, можно определить среднюю задержку как

$$T = \frac{1}{\gamma} \sum_{i=1}^L \lambda_i T_{ri}.$$

Чтобы определить значение  $T$ , необходимо определить значения отдельных задержек  $T_{ri}$ . Поскольку предполагается, что каждая очередь может рассматриваться как независимая модель  $M/M/1$ , это нетрудно сделать следующим образом:

$$T_{ri} = \frac{T_{si}}{1 - \rho_i} = \frac{T_{si}}{1 - \lambda_i T_{si}}.$$

Время обслуживания  $T_{si}$  для канала  $i$  представляет собой просто отношение средней длины пакетов в битах ( $M$ ) к скорости передачи данных по каналу связи в битах в секунду ( $R_i$ ). Тогда

$$T_{ri} = \frac{\frac{M}{R_i}}{1 - \frac{M\lambda_i}{R_i}} = \frac{M}{R_i - M\lambda_i}.$$

Собрав все упомянутые выше элементы вместе, можно вычислить среднюю задержку пакетов в сети по формуле

$$T = \frac{1}{\gamma} \sum_{i=1}^L \frac{M\lambda_i}{R_i - M\lambda_i}.$$

---

<sup>4</sup> По сути, данное утверждение основано на том факте, что сумма средних величин является средним значением суммы.

## 21.9. ДРУГИЕ МОДЕЛИ СИСТЕМ МАССОВОГО ОБСЛУЖИВАНИЯ

Основное внимание в этой главе было уделено одному типу модели систем массового обслуживания. Фактически же имеется целый ряд таких моделей, основанных двух ключевых факторах:

- порядок обслуживания блокированных элементов;
- количество источников трафика.

Если элемент поступает на сервер и обнаруживает его занятым или же поступает в многоканальную систему массового обслуживания и обнаруживает занятыми все ее серверы, такой элемент считается блокированным. Блокированные элементы могут быть обработаны самыми разными способами. Во-первых, такой элемент может быть поставлен в очередь, ожидая освобождения сервера. В литературе по телефонному трафику такой способ обслуживания называется *задержкой потерянных вызовов* (*lost calls delayed*), хотя на самом деле вызов не теряется. Альтернативой является отсутствие очереди, что приводит к двум предположениям о действиях элемента. Элемент может ожидать в течение некоторого произвольного периода времени, а затем попытаться снова получить обслуживание. Такой способ называется *сбросом потерянных вызовов* (*lost calls cleared*). Если же элемент пытается получить обслуживание многократно, не делая пауз, то такой способ называется *удержанием потерянных вызовов* (*lost calls held*). Для решения большинства задач вычислительной техники и передачи данных наиболее подходящей является модель задержки потерянных вызовов. Модель сброса потерянных вызовов обычно оказывается наиболее пригодной для работы телефонной станции.

Вторым ключевым элементом модели трафика является предположение о конечности или бесконечности количества источников. В модели с бесконечным количеством источников предполагается фиксированная частота поступлений, а в модели с конечным количеством источников частота поступлений зависит от количества уже задействованных источников трафика. Так, если каждый источник  $L$  генерирует поступления с частотой  $\lambda/L$ , то, если система массового обслуживания не занята, скорость поступления оказывается равной  $\lambda$ . Но если в конкретный момент времени в обслуживании находятся  $K$  источников, то скорость поступления в этот момент равна  $\lambda(L-K)/L$ . Модели с бесконечным числом источников более просты в обращении. Предположение бесконечного количества источников трафика вполне разумно, если их количество по крайней мере в 5–10 раз превышает пропускную способность системы.

## 21.10. ОЦЕНКА ПАРАМЕТРОВ МОДЕЛИ

Чтобы выполнить анализ очередей, необходимо оценить значения входных параметров, в частности, среднее значение и стандартное отклонение скорости поступления и времени обслуживания. Если анализируется новая система, ее оценка может быть основана на оценке оборудования и рассмотрении преобладающих режимов его работы. Тем не менее нередко оказывается, что существующая система доступна для изучения. Например, совокупность терминалов, персональных компьютеров и узлов может быть связана в здании с помощью прямых подключений и мультиплексоров, и желательно заменить это оборудование локальной сетью. Чтобы развернуть такую сеть в масштабах здания, можно измерить нагрузку, порождаемую в настоящий момент каждым устройством.

## Выборка

Измерения производятся в форме выборок. Конкретный параметр (например, скорость передачи пакетов, формируемый терминалом, или длина пакетов) оценивается путем наблюдения количества пакетов, формируемых в течение определенного периода времени.

Самой важной для оценивания величиной является среднее значение. Во многих уравнениях из табл. 21.6 и 21.7 это единственная величина, которую требуется оценить. Такая оценка обозначается как среднее значение выборки  $\bar{X}$  и вычисляется следующим образом:

$$\bar{X} = \frac{1}{N} \sum_{i=1}^N X_i,$$

где

$N$  — размер выборки;

$X_i$  —  $i$ -й элемент выборки.

Важно отметить, что само среднее значение выборки является случайной переменной. Так, если взять выборку из некоторого состава элементов, и вычислить ее среднее значение и сделать это многократно, то вычисленные значения будут различаться. Поэтому можно говорить о среднем значении и его стандартном отклонении или даже о целом распределении вероятности среднего значения выборки. Чтобы различать эти понятия, распределение вероятности первоначальной случайной переменной  $X$  обычно называется *исходным распределением*, а распределение вероятности среднего значения выборки  $\bar{X}$  — *распределением среднего выборки*.

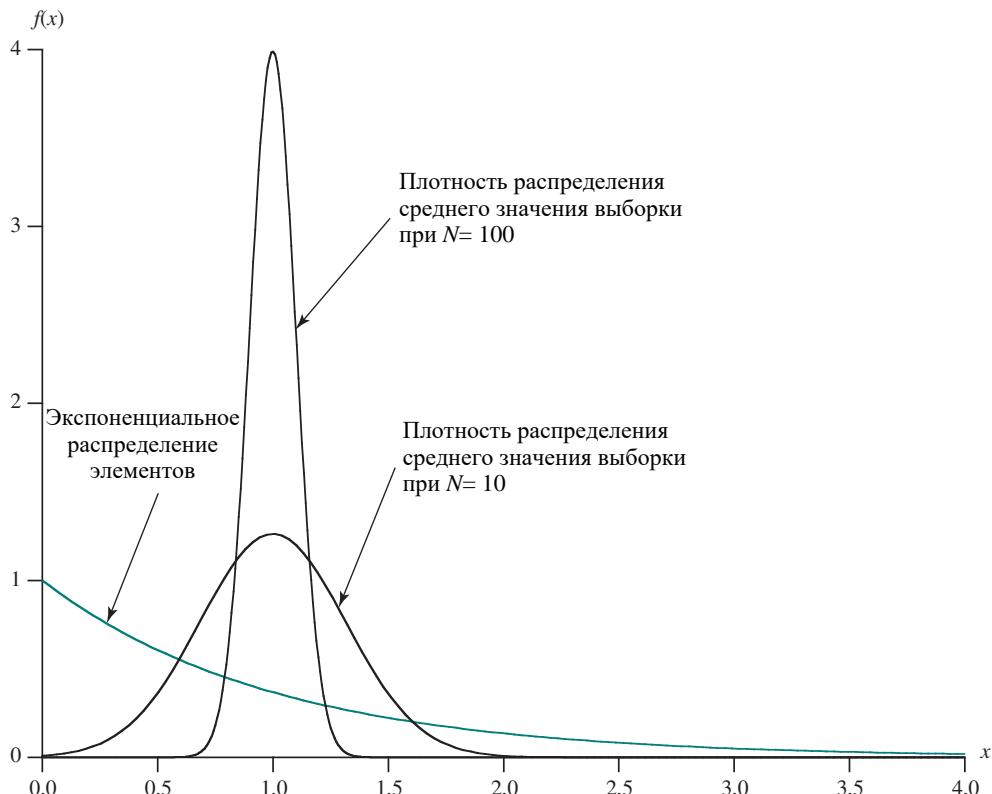
Среднее значение выборки  $\bar{X}$  и его дисперсия вычисляются по следующим формулам:

$$E[\bar{X}] = E[X] = \mu,$$

$$\text{Var}[\bar{X}] = \frac{\sigma_X^2}{N}.$$

Таким образом, если вычисляется среднее значение выборки, то ожидаемое ее значение оказывается таким же, как и у исходной случайной переменной, а дисперсия предполагаемого среднего значения выборки уменьшается по мере увеличения размера выборки  $N$ . Это демонстрируется на рис. 21.9, на нем показано исходное экспоненциальное распределение со средним значением  $\mu = 1$ . Это может быть распределение периодов времени обслуживания сервером или промежутков времени между поступлениями в пуссоновском процессе. Если для оценки среднего значения  $\mu$  используется выборка размером 10, то ожидаемое значение действительно равно  $\mu$ , но фактическое значение может вполне отличаться от него на целых 50%. Если же используется выборка размером 100, то разброс возможных вычисляемых значений существенно уже, и поэтому можно ожидать, что фактическое среднее значение любой такой выборки окажется намного более близким к  $\mu$ .

Определенное выше среднее значение выборки используется непосредственно для оценивания времени обслуживания сервером. Что касается скорости поступления, то для ее оценки можно осуществить наблюдение промежутков времени для  $N$  последовательных поступлений и сначала вычислить среднее значение выборки, а затем — оцениваемую скорость поступления.



**Рис. 21.9.** Средние значения выборки для экспоненциального распределения

Эквивалентный и более простой подход состоит в том, чтобы произвести оценку по следующей формуле:

$$\bar{\lambda} = \frac{N}{T}.$$

Здесь  $N$  — количество элементов, наблюдаемых в течение периода времени  $T$ .

Для большей части анализа очередей требуется лишь оценка среднего значения. Но в некоторых важных уравнениях требуется также оценка дисперсии исходной случайной переменной  $\sigma_X^2$ . Дисперсия выборки вычисляется следующим образом:

$$S^2 = \frac{1}{N-1} \sum_{i=1}^N (X_i - \bar{X})^2.$$

Ожидаемая величина  $S^2$  имеет нужное нам значение:

$$E[S^2] = \sigma_X^2.$$

Дисперсия выборки  $S^2$  зависит от исходного распределения и, в общем случае, трудна для вычисления. Но, как и следовало ожидать, дисперсия выборки  $S^2$  уменьшается по мере увеличения размера выборки  $N$ .

Все понятия, рассмотренные в этом разделе, подытожены в табл. 21.10.

Таблица 21.10. Статистические параметры выборки

	Состав элементов	Среднее значение выборки	Дисперсия выборки
Случайная переменная	$X$	$\bar{X} = \frac{1}{N} \sum_{i=1}^N X_i$	$S^2 = \frac{1}{N-1} \sum_{i=1}^N (X_i - \bar{X})^2$
Ожидаемое значение	$E[X] = \mu$	$E[\bar{X}] = \mu$	$E[S^2] = \sigma_X^2$
Дисперсия	$\text{Var}[X] = E[(X - \mu)^2] = \sigma_X^2$	$\text{Var}[\bar{X}] = \frac{\sigma_X^2}{N}$	

## Ошибки выборки

При оценке на основании выборки таких значений, как среднее и стандартное отклонение, мы оставляем область вероятности и входим в область статистики. Этот сложный вопрос не будет здесь исследоваться, за исключением некоторых комментариев.

Вероятностный характер оцениваемых значений является источником ошибки, называемой **ошибкой выборки** (*sampling error*). В общем случае чем больше размер сделанной выборки, тем меньше стандартное отклонение среднего значения выборки или другой величины, а следовательно, такая оценка, вероятнее всего, окажется ближе к фактическому значению. Сделав некоторые вполне обоснованные допущения о характере проверяемой случайной и произвольности процедуры выборки, в действительности можно определить вероятность того, что среднее значение выборки или его стандартное отклонение находится на некотором удалении от фактического среднего значения или стандартного отклонения. Об этом понятии нередко свидетельствуют представления результатов выборки. Например, результаты социологического опроса обычно сопровождаются комментариями наподобие “результаты опроса с достоверностью 99% отклоняются от истинного значения не более чем на 5%”.

Тем не менее имеется еще и **систематическая погрешность** (*bias*) — другой источник ошибок, который намного менее часто оценивается не специалистами. Так, если проводится социологический опрос только среди членов определенной общественно-экономической группы, то его результаты совсем не обязательно представляют все население. В контексте обмена данными выборка, сделанная в одно время суток, может не отражать информацию о подобной деятельности в другое время суток. Если речь идет о проектировании системы, способной выдерживать пиковую нагрузку, которую она, вероятнее всего, будет испытывать, то сетевой трафик следует наблюдать в течение того времени суток, когда максимальная нагрузка наиболее вероятна.

## 21.11. ЗАДАЧИ

- 21.1. В разделе 21.3 приведено интуитивное обоснование формулы Литтла. Приведите аналогичное обоснование соотношения  $r = \lambda T_r$ .

**21.2.** На рис. 21.3 показано количество находящихся в системе элементов как функция от времени. Эту функцию можно рассматривать как разность процессов поступления и отправки:  $n(t) = a(t) - d(t)$ .

- Покажите на одном графике, что функции  $a(t)$  и  $d(t)$  приводят к функции  $n(t)$ , показанной на рис. 21.3.
- Используя график из п. а, разработайте интуитивное обоснование формулы Литтла. *Указание:* рассмотрите область между двумя ступенчатыми функциями, вычисляемыми сначала сложением вертикальных прямоугольников, а затем добавлением горизонтальных прямоугольников.

**21.3.** Владелец магазина замечает, что в его магазин приходит в среднем 18 человек в час, а находятся в нем обычно 8 покупателей. Какова средняя продолжительность времени, которое каждый покупатель проводит в магазине?

**21.4.** Программа, имитирующая многопроцессорную систему, начинает выполняться и завершается, когда в очереди отсутствуют задания. Эта программа сообщает, что среднее количество заданий за время имитации системы — 12,356; средняя скорость поступления — 25,6 задания в минуту, а средняя задержка выполнения задания — 8,34 минуты. Насколько верно была сымитирована такая система?

**21.5.** В разделе 21.3 предоставляется интуитивное обоснование одноканального соотношения  $\rho = \lambda T_s$ . Выработайте аналогичное обоснование многоканального соотношения  $\rho = \lambda T_s / N$ .

**21.6.** Если скорость поступления в очередь, организованную как система массового обслуживания модели  $M/M/1$ , составляет 2 клиента в минуту, а скорость обслуживания — четыре клиента в минуту, то сколько в среднем клиентов пребывают в системе и сколько их в среднем находятся на стадии обслуживания?

**21.7.** Каков коэффициент использования очереди, организованной как система массового обслуживания модели  $M/M/1$ , в которой в среднем ожидают четырех человек?

**21.8.** Средняя продолжительность транзакции в банкомате, установленном в супермаркете, составляет две минуты, а клиенты пользуются банкоматом в среднем каждые пять минут. Какова средняя продолжительность времени ожидания и использования банкомата клиентом? Сколько в среднем клиентов ожидают своей очереди на обслуживание банкоматом? Предполагается модель  $M/M/1$  системы массового обслуживания.

**21.9.** Сообщения для отправки по каналу связи со скоростью 9600 бит/с поступают в случайному порядке. Коэффициент использования такой системы массового обслуживания составляет около 70%, а средняя длина сообщения — 1000 октетов. Определите среднее время ожидания сообщений постоянной длины, а также длины, подчиняющейся экспоненциальному распределению.

**21.10.** Через коммутатор сообщений проходят сообщения, имеющие три разные длины. Для 70% сообщений продолжительность обслуживания равна 1 мс, для 20% — 3 мс и для 10% — 10 мс. Вычислите среднее время пребывания сообщений в коммутаторе, а также среднее количество сообщений в нем, если сообщения поступают со средней скоростью:

- а. одно сообщение за 3 мс;
- б. одно сообщение за 4 мс;
- в. одно сообщение за 5 мс.

**21.11.** Сообщения поступают в центр коммутации по отдельному каналу связи в виде пуассоновского потока со средней скоростью 180 сообщений в час. Длина сообщения распределена экспоненциально и в среднем равна 14 400 символам, а скорость передачи данных по каналу связи — 9600 бит/с.

- а. Каково среднее время ожидания сообщения в центре коммутации сообщений?
- б. Сколько сообщений в среднем ожидают передачи в центре коммутации?

**21.12.** Зачастую потоки, входящие в систему массового обслуживания, независимы и произвольны, но поступают группами. Средние задержки ожидания для такого типа входящих потоков оказываются большими, чем для пуассоновских потоков. Это демонстрируется в данной задаче на простом примере. Предположим, что элементы поступают в очередь группами фиксированной длины из  $M$  элементов. Поступление таких групп имеет распределение Пуассона и происходит со средней скоростью  $\lambda/M$ , так что скорость поступления отдельных элементов равна  $\lambda$ . Время обслуживания каждого элемента равно  $T_s$  со стандартным отклонением  $\sigma T_s$ .

- а. Если рассматривать пакеты как крупные элементы, то каковы среднее значение и дисперсия времени обслуживания группы и каково среднее значение времени ожидания обслуживания группы?
- б. Каково среднее значение времени ожидания обслуживания элементом, когда начнется обслуживание его группы? Предположим, что элемент может равновероятно находиться на любой из  $M$  позиций в группе. Каково общее среднее значение времени ожидания элемента?
- в. Проверьте достоверность результатов, полученных в п. б, показав, что при  $M=1$  результаты сводятся к результатам для системы массового обслуживания модели  $M/G/1$ . Насколько варьируются результаты при  $M>1$ ?

**21.13.** Рассмотрим единственную очередь с постоянным временем обслуживания 4 с и пуассоновским потоком, входящим со скоростью 0,20 элементов в секунду.

- а. Найдите среднее значение и стандартное отклонение длины очереди.
- б. Найдите среднее значение и стандартное отклонение времени пребывания в системе.

**21.14.** Рассмотрим узел ретрансляции фреймов, обслуживающий пуассоновский поток входящих фреймов, которые должны быть переданы по исходящему каналу связи со скоростью 1 Мбит/с. Поток состоит из двух типов фреймов. Оба типа фреймов имеют одинаковое экспоненциальное распределение длины фреймов со средним значением 1000 бит.

- а. Предполагаем, что приоритеты не используются. Совокупная скорость поступления фреймов обоих типов равна 800 фреймам в секунду. Каково среднее время пребывания в системе ( $T_r$ ) для всех фреймов?
- б. Теперь предположим, что два типа фреймов имеют разные приоритеты, причем скорость поступления первого типа равна 200 фреймам в секунду, а второго типа — 600 фреймам в секунду. Рассчитайте среднее время пребывания фреймов первого типа, фреймов второго типа и общее среднее время их пребывания в системе.

- в. Повторите расчет для п. б при  $\lambda_1 = \lambda_2 = 400$  фреймов в секунду.
- г. Повторите расчет для п. б при  $\lambda_1 = 600$  фреймов в секунду и  $\lambda_2 = 200$  фреймов в секунду.

**21.15.** Протокол MLP (Multilink Protocol) является частью протокола X.25. Аналогичное средство применяется и в архитектуре IBM SNA (System Network Architecture — системная сетевая архитектура). Согласно протоколу MLP между двумя сетевыми узлами имеется ряд каналов передачи данных, которые используются как объединенный ресурс для передаваемых пакетов, независимо от номера виртуального канала. Когда пакет передается протоколу MLP для передачи, для выполнения конкретного задания может быть выбран любой доступный канал связи. Так, если две локальные сети в разных местах соединены парой мостов, то между мостами может быть несколько двухточечных соединений для повышения пропускной способности и доступности.

Принятый в MLP подход требует дополнительных издержек на обработку и передачу фреймов по сравнению с простым одноканальным протоколом. Для протокола также нужен специальный MLP-заголовок. Альтернативой данному подходу служит выделение каждому поступающему пакету одного исходящего канала связи в порядке круговой очереди. Благодаря этому обработка упрощается; однако какое влияние это оказывает на производительность?

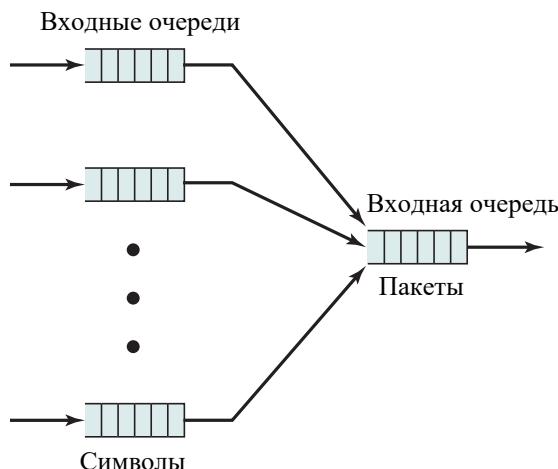
Рассмотрим конкретный пример. Допустим, что два сетевых узла связаны пятью каналами передачи данных со скоростью 9600 бит/с; средний размер экспоненциально распределенных по размеру пакетов составляет 100 октетов, а скорость их поступления — 48 пакетов в секунду.

- а. Вычислите коэффициент использования ( $\rho$ ) и среднее время пребывания в системе ( $T_s$ ) для одноканальной системы массового обслуживания.
- б. Для многоканальной системы массового обслуживания значение С-функции Эрланга равно 0,554. Определите среднее время пребывания в системе ( $T_s$ ).

**21.16.** Дополнением к стандарту X.25 коммутации пакетов служит ряд стандартов, X.3, X.28 и X.29, в которых определен модуль сборки-разборки пакетов (packet assembler-disassembler — PAD). PAD предназначен для подключения асинхронных терминалов к сети с коммутацией пакетов. Каждый терминал, подключенный к модулю PAD, по одному отсылает символы, которые сначала буферизуются в PAD, а затем собираются в X.25-пакет, передаваемый далее по сети с коммутацией пакетов. Длина буфера равна максимальному размеру поля данных для X.25-пакета. Такой пакет формируется из собранных символов и передается всякий раз, когда заполняется буфер, принимается специальный управляющий символ наподобие символа новой строки или истекает время ожидания. Для решения этой задачи можно пренебречь двумя последними условиями. Модель системы массового обслуживания для PAD приведена на рис. 21.10, где первая очередь моделирует задержку символов, ожидающих составления в пакет. Когда эта очередь заполняется, она сразу же опорожняется. Вторая очередь моделирует задержку пакетов, ожидающих передачи. В этой задаче используются следующие обозначения:

- $\lambda$  — скорость пуассоновского входящего потока символов, поступающих из каждого терминала;
- $C$  — скорость передачи данных в выходном канале связи, в символах в секунду;
- $M$  — количество символов данных в пакете;
- $H$  — количество служебных символов в пакете;
- $K$  — количество терминалов.

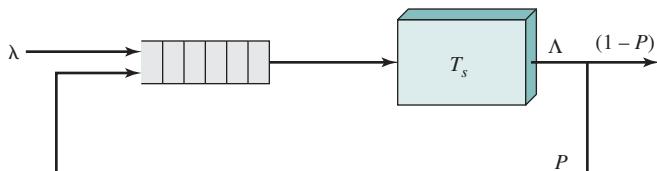
- Определите среднее время ожидания символа во входной очереди.
- Определите среднее время ожидания пакета в выходной очереди.
- Определите среднее время между моментом, когда символ покидает терминал, и моментом, когда он покидает модуль PAD. Представьте результат в виде графика функции от нормализованной нагрузки.



**Рис. 21.10.** Модель системы массового обслуживания для модуля сборки и разборки пакетов (PAD)

21.17. Часть  $P$  трафика от одного экспоненциального сервера подается обратно на вход, как показано на рис. 21.11, где  $\Lambda$  обозначает пропускную способность системы, т.е. скорость выходящего из сервера трафика.

- Определите пропускную способность системы, коэффициент использования сервера и среднее время пребывания в системе для однократного прохода через сервер.
- Определите среднее количество проходов элемента через систему массового обслуживания, а также общее среднее время пребывания элемента в данной системе.



**Рис. 21.11.** Очередь с обратной связью

# ПРИЛОЖЕНИЕ A

---

## ВОПРОСЫ ПАРАЛЛЕЛЬНОСТИ

В ЭТОМ ПРИЛОЖЕНИИ...

### A.1. Состояния гонки и семафоры

- Постановка задачи
- Первая попытка
- Вторая попытка
- Третья попытка
- Четвертая попытка
- Правильное решение

### A.2. Задача о парикмахерской

- Неполное решение задачи о парикмахерской
- Полное решение задачи о парикмахерской

### A.3. Задачи

## A.1. СОСТОЯНИЯ ГОНКИ И СЕМАФОРЫ

Хотя определение состояния гонки, представленное в разделе 5.1, кажется простым, опыт показал, что студентам обычно сложно обнаружить состояние гонки в своих программах. Цель данного раздела, основанного на [37]<sup>1</sup>, — пошагово рассмотреть ряд примеров с использованием семафоров, которые должны помочь прояснить эту тему.

### Постановка задачи

Предположим, что есть два процесса, **А** и **В**, каждый из которых состоит из ряда параллельных потоков. Каждый поток включает бесконечный цикл, в котором выполняется обмен сообщениями с потоком из другого процесса. Каждое сообщение состоит из целого числа, помещаемого в совместно используемый глобальный буфер. Есть два требования.

1. После того как поток A1 процесса **А** делает сообщение доступным некоторому потоку B1 в **В**, поток A1 может продолжаться только после получения сообщения от B1. Точно так же, после того как B1 делает сообщение доступным для A1, он может продолжаться только после получения сообщения от A1.
2. Как только поток A1 делает сообщение доступным, он должен убедиться, что никакие другие потоки в **А** не перезапишут глобальный буфер, прежде чем сообщение будет получено потоком из **В**.

В оставшейся части этого раздела мы покажем четыре попытки реализации описанной схемы с использованием семафоров, каждая из которых может привести к состоянию гонки. И наконец, мы покажем правильное решение данной задачи.

### Первая попытка

Рассмотрим следующий подход.

```
semaphore a = 0, b = 0;
int buf_a, buf_b;

thread_A(...)

{
    int var_a;
    ...
    while (true) {
        ...
        var_a = ...;
        semSignal(b);
        semWait(a);
        buf_a = var_a;
        var_a = buf_b;
        ...
    }
}

thread_B(...)

{
    int var_b;
    ...
    while (true) {
        ...
        var_b = ...;
        semSignal(a);
        semWait(b);
        buf_b = var_b;
        var_b = buf_a;
        ...
    }
}
```

<sup>1</sup> Я признателен профессору Чин-Куанг Шену (*Ching-Kuang Shene*) из Мичиганского технологического университета за разрешение использовать этот пример.

Это простой протокол квитирования (“рукопожатия”). Когда поток A1 в А готов к обмену сообщениями, он посыпает сигнал потоку в В и затем ожидает готовности потока B1 в В. Как только от B1 возвращается сигнал, который принимает А путем выполнения semWait(a), поток A1 предполагает, что B1 готов к обмену, и выполняет его. B1 ведет себя аналогично, и обмен происходит независимо от того, какой именно поток готов первым.

Эта попытка может привести к состоянию гонки. Например, рассмотрим такую последовательность действий (ось времени направлена в таблице вертикально вниз).

Thread A1	Thread B1
semSignal(b)	
semWait(a)	
	semSignal(a)
	semWait(b)
buf_a = var_a	
var_a = buf_b	
	buf_b = var_b

В показанной последовательности A1 достигает semWait(a) и блокируется. B1 достигает semWait(b) и не блокируется, но переключается до того, как сможет обновить свой buf\_b. Тем временем A1 выполняется и считывает buf\_b до того, как эта переменная получит требуемое значение. В этот момент buf\_b может иметь значение, предоставленное ранее другим потоком или предоставленное B1 в предыдущем обмене сообщениями. Это — состояние гонки.

Более тонкое состояние гонки можно увидеть, если в А и В активны по два потока. Рассмотрим такую последовательность действий.

Thread A1	Thread A2	Thread B1	Thread B2
semSignal(b)			
semWait(a)			
		semSignal(a)	
		semWait(b)	
	semSignal(b)		
	semWait(a)		
		buf_b = var_b1	
			semSignal(a)
buf_a = var_a1			
	buf_a = var_a2		

В этой последовательности действий потоки A1 и B1 пытаются обмениваться сообщениями и работают с помощью соответствующих инструкций семафоров. Однако сразу после того, как выполняются два сигнала `semWait` (в потоках A1 и B1), запускается поток A2 и выполняются `semSignal(b)` и `semWait(a)`, что приводит к выполнению потоком B2 `semSignal(a)`, а это приводит к освобождению потока A2 от `semWait(a)`. В этот момент либо A1, либо A2 может обновить `buf_a`, и мы получаем состояние гонки. Изменяя последовательность выполнения инструкций потоками, мы можем легко найти и другие условия гонки.

**Извлеченный урок:** когда переменная используется несколькими потоками, при отсутствии надлежащей защиты с помощью взаимоисключений может возникнуть состояние гонки.

## Вторая попытка

В этой попытке мы используем семафор для защиты совместно используемой переменной. Ее цель — гарантировать, что доступ к `buf_a` и `buf_b` является взаимоисключающим. Программа выглядит следующим образом.

<pre>semaphore a = 0, b = 0; mutex = 1; int buf_a, buf_b;  thread_A(...)  {     int var_a;     . .     while (true) {         . .         var_a =...;         semSignal(b);         semWait(a);         semWait(mutex);         buf_a = var_a;         semSignal(mutex);         semSignal(b);         semWait(a);         semWait(mutex);         var_a = buf_b;         semSignal(mutex);         . . .;     } }</pre>	<pre>thread_B(...)  {     int var_b;     . .     while (true) {         . .         var_b =...;         semSignal(a);         semWait(b);         semWait(mutex);         buf_b = var_b;         semSignal(mutex);         semSignal(a);         semWait(b);         semWait(mutex);         var_b = buf_a;         semSignal(mutex);         . . .;     } }</pre>
--	--

Прежде чем поток может обменяться сообщением, он следует тому же протоколу квитирования, что и в первой попытке. Семафор `mutex` защищает `buf_a` и `buf_b` в попытке гарантировать, что обновление предшествует чтению. Но эта защита не адекватна поставленной задаче. Как только оба потока завершат первый этап установления связи, значения семафоров — и `a`, и `b` — оба равны 1.

Могут возникнуть три ситуации.

1. Два потока, скажем, A1 и B1, завершают первое рукопожатие и переходят ко второму этапу обмена.
2. Вторая пара потоков начинает первый этап.
3. Один поток из текущей пары продолжает работу и обменивается сообщениями с новичком из другой пары.

Каждая из ситуаций может приводить к состоянию гонки. Пример состояния гонки на основе третьей ситуации приведен ниже.

Thread A1	Thread A2	Thread B1
semSignal(b)		
semWait(a)		semSignal(a)
		semWait(b)
buf_a = var_a1		buf_b = var_b1
	semSignal(b)	
	semWait(a)	
		semSignal(a)
		semWait(b)
	buf_a = var_a2	

В этом примере, после того как A1 и B1 прошли первое рукопожатие, они оба обновляют соответствующие глобальные буферы. Затем A2 инициирует первую стадию рукопожатия. Следуя ей, B1 инициирует вторую стадию рукопожатия. В этот момент A2 обновляет buf\_a до того, как B1 сможет получить значение, помещенное в buf\_a потоком A1. Это — состояние гонки.

**Извлеченный урок:** защита единственной переменной может оказаться недостаточной, если использование этой переменной является частью длинной последовательности выполнения. Защищайте всю последовательность выполняемых действий.

## Третья попытка

В этой попытке мы хотим расширить критический участок, включив в него весь обмен сообщениями (каждый из двух потоков обновляет один из двух буферов и читает другой буфер). Одного семафора недостаточно, потому что это может привести к взаимоблокировке, когда каждая сторона ждет другую. Программа выглядит следующим образом.

```

semaphore already = 1, adone = 0, bready = 1 bdone = 0;
int buf_a, buf_b;

thread_A(...)
{
    int var_a;
    ...
    while (true) {
        ...
        var_a = ...;
        semWait(already);
        buf_a = var_a;
        semSignal(adone);
        semWait(bdone);
        var_a = buf_b;
        semSignal(already);
        ...
    }
}

thread_B(...)
{
    int var_b;
    ...
    while (true) {
        ...
        var_b = ...;
        semWait(bready);
        buf_b = var_b;
        semSignal(bdone);
        semWait(adone);
        var_b = buf_a;
        semSignal(bready);
        ...
    }
}

```

Семафор `already` предназначен для того, чтобы никакой другой поток в **A** не мог обновить `buf_a`, пока один поток из **A** входит в критический участок. Семафор `adone` предназначен для того, чтобы ни один поток из **B** не пытался читать `buf_a` до его обновления. Аналогичные утверждения относятся к семафорам `bready` и `bdone`. Однако эта схема не препятствует состоянию гонки. Рассмотрим такую последовательность событий.

<b>Thread A1</b>	<b>Thread B1</b>
<code>buf_a = var_a</code>	
<code>semSignal(adone)</code>	
<code>semWait(bdone)</code>	
	<code>buf_b = var_b</code>
	<code>semSignal(bdone)</code>
	<code>semWait(adone)</code>
<code>var_a = buf_b;</code>	
<code>semSignal(already)</code>	
<code>...loop back...</code>	
<code>semWait(already)</code>	
<code>buf_a = var_a</code>	
	<code>var_b = buf_a</code>

В этой последовательности и A1, и B1 входят в свои критические участки, размещают свои сообщения и достигают второго ожидания. Затем A1 копирует сообщение из B1 и оставляет его критический участок. На этом этапе A1 может вернуться в свою программу, сгенерировать новое сообщение и поместить его в buf\_a, как показано в последовательности выполнения выше. Другая возможность состоит в том, что в этой же точке другой поток A может сгенерировать сообщение и положить его в buf\_a. В любом случае сообщение теряется и возникает состояние гонки.

**Извлеченный урок:** если у нас есть несколько сотрудничающих групп потоков, гарантированное для одной группы взаимное исключение может не препятствовать вмешательству потоков из других групп. Кроме того, если один и тот же поток повторно входит в критический участок, нужно должным образом обрабатывать моменты сотрудничества между потоками.

## Четвертая попытка

Третья попытка не смогла обеспечить требование, чтобы поток оставался в своем критическом участке, пока другой поток не извлечет сообщение. Вот как выглядит попытка достичь этой цели.

<pre> semaphore already = 1, adone = 0, bready = 1 bdone = 0; int buf_a, buf_b;  thread_A(...)  {     int var_a;     ...     while (true) {         ...         var_a =....;         semWait(bready);         buf_a = var_a;         semSignal(adone);         semWait(bdone);         var_a = buf_b;         semSignal(already);         ...     } } </pre>	<pre> thread_B(...)  {     int var_b;     ...     while (true) {         ...         var_b =....;         semWait(already);         buf_b = var_b;         semSignal(bdone);         semWait(adone);         var_b = buf_a;         semSignal(bready);         ...     } } </pre>
--	---

В этом случае первый поток в **A**, чтобы войти в свой критический участок, уменьшает bready до 0. Никакой последующий поток из **A** не может пытаться обмениваться сообщениями до тех пор, пока поток из **B** не завершит обмен сообщениями и увеличит bready до 1. И этот подход тоже может привести к состоянию гонки, например, при такой последовательности.

Thread A1	Thread A2	Thread B1
semWait (bready)		
buf_a = var_a1		
semSignal (adone)		
		semWait (aready)
		buf_b = var_b1
		semSignal (bdone)
		semWait (adone)
		var_b = buf_a
		semSignal (bready)
	semWait (bready)	
	...	
	semWait (bdone)	
		var_a2 = buf_b

В этой последовательности потоки A1 и B1 входят в соответствующие критические участки для обмена сообщениями в указанном порядке. Поток B1 извлекает свое сообщение и сигнализирует с помощью bready. Это позволяет другому потоку из A, A2, войти в его критический участок. Если A2 быстрее, чем A1, то A2 может получить сообщение, предназначенное для A1.

**Извлеченный урок:** если семафор для взаимного исключения освобожден не его владельцем, может возникнуть состояние гонки. В этой четвертой попытке семафор блокируется потоком в A, а затем разблокируется потоком в B. Это рискованная практика программирования.

## Правильное решение

Читатель может заметить, что проблема, рассматриваемая в этом разделе, — вариант задачи об ограниченном буфере, и она может быть решена методом, аналогичным обсуждавшемуся в разделе 5.4. Самый простой подход — использовать два буфера: один — для сообщений **B-к-A**, а второй — для сообщений **A-к-B**. Размер каждого буфера должен быть равен единице. Чтобы понять причины этого подхода, следует учесть, что не существует никаких предположений об упорядочении освобождения потоков примитивом синхронизации. Если в буфере более одного слота, мы не можем гарантировать, что сообщения будут корректно сопоставлены. Например, B1 может получить сообщение от A1, а затем отправить сообщение в A1. Но если буфер имеет несколько слотов, то другой поток в A может извлечь сообщение из слота, предназначенного для A1.

Используя тот же базовый подход, что и в разделе 5.4, мы можем разработать следующую программу.

```

semaphore notFull_A = 1, notFull_B = 1;
semaphore notEmpty_A = 0, notEmpty_B = 0;
int buf_a, buf_b;

thread A(...)
{
    int var_a;
    ...
    while (true) {
        ...
        var_a = ...;
        semWait(notFull_A);
        buf_a = var_a;
        semSignal(notEmpty_A);
        semWait(notEmpty_B);
        var_a = buf_b;
        semSignal(notFull_B);
        ...
    }
}

thread B(...)
{
    int var_b;
    ...
    while (true) {
        ...
        var_b = ...;
        semWait(notFull_B);
        buf_b = var_b;
        semSignal(notEmpty_B);
        semWait(notEmpty_A);
        var_b = buf_a;
        semSignal(notFull_A);
        ...
    }
}

```

Чтобы убедиться, что это решение работает, нужно проверить выполнение трех условий.

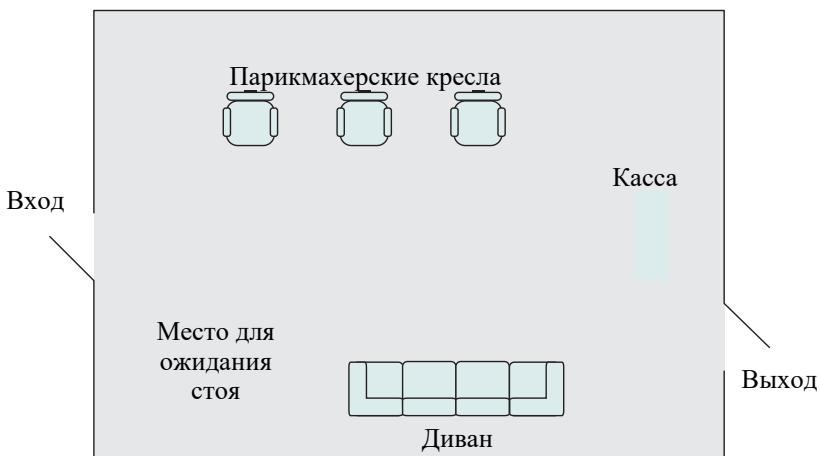
1. Участок обмена сообщениями является взаимоисключающим в пределах группы потоков. Поскольку начальное значение `notFull_A` равно 1, только один поток в **A** может пройти через `semWait (notFull_A)`, пока обмен не будет завершен, о чем сообщает поток в **B**, выполняющий `semSignal (notFull_A)`. Аналогичные рассуждения применимы и к потокам в **B**. Таким образом, это условие выполняется.
2. Как только два потока входят в свои критические участки, они обмениваются сообщениями без вмешательства других потоков. Никакой другой поток в **A** не может войти в свой критический участок, пока поток в **B** не завершит полностью обмен, и никакой другой поток в **B** не сможет войти в свой критический участок, пока поток в **A** не завершит полностью обмен. Таким образом, и это условие выполняется.
3. После того как один поток выходит из своего критического участка, ни один поток из той же группы не может прорваться и испортить существующее сообщение. Это условие выполняется, потому что в каждом направлении используется буфер с одним слотом. Как только поток в **A** выполнит `semWait (notFull_A)` и войдет в свой критический участок, никакой другой поток в **A** не сможет обновить `buf_a`, пока соответствующий поток в **B** не получит значение из `buf_a` и не вызовет `semSignal (notFull_A)`.

**Извлеченный урок:** следует тщательно рассмотреть решения известных задач, поскольку правильное решение рассматриваемой задачи может оказаться вариантом решения уже известной задачи.

## A.2. ЗАДАЧА О ПАРИКМАХЕРСКОЙ

В качестве другого примера использования семафоров для реализации параллельных вычислений рассмотрим простую задачу о парикмахерской.<sup>2</sup> Этот пример весьма поучителен, так как задача, возникающая при попытке обеспечить простой доступ в парикмахерскую, сродни тем, которые возникают в реальных операционных системах.

В нашей парикмахерской есть три кресла, три парикмахера и зал ожидания, в котором четыре клиента могут разместиться на диване, а остальные — стоя (рис. А.1). Правила пожарной безопасности ограничивают общее количество клиентов внутри помещения 20 людьми. В нашем примере мы предполагаем, что всего парикмахерская должна обслужить 50 клиентов.



**Рис. А.1.** Парикмахерская

Клиент не может войти в парикмахерскую, если она полностью заполнена другими клиентами. Оказавшись внутри, клиент либо присаживается на диван, либо стоит, если последний занят. Когда парикмахер освобождается, к нему отправляется наиболее долго ожидающий клиент с дивана; если имеются стоящие клиенты, то тот из них, кто ожидает дольше других, присаживается на диван. По окончании стрижки принять плату может любой парикмахер, но поскольку касса в парикмахерской лишь одна, плата принимается в один момент времени только от одного клиента. Рабочее время парикмахера разделяется на стрижку, принятие платы от клиента и сон в своем кресле в ожидании очередного клиента.

### Неполное решение задачи о парикмахерской

На рис. А.2 показана реализация задачи о парикмахерской с использованием семафоров; предполагается, что все очереди семафоров обрабатываются по принципу “первым вошел — первым вышел”.

<sup>2</sup> Я признателен профессору Ральфу Хильцеру (Ralph Hilzer) из Университета штата Калифорния в Чико за предоставление этой задачи.

```

/* program barbershop1 */
semaphore max_capacity = 20;
semaphore sofa = 4;
semaphore barber_chair = 3;
semaphore coord = 3;
semaphore cust_ready = 0, finished = 0, leave_b_chair = 0, payment= 0,
receipt = 0;

void customer ()
{
    semWait(max_capacity);
    enter_shop();
    semWait(sofa);
    sit_on_sofa();
    semWait(barber_chair);
    get_up_from_sofa();
    semSignal(sofa);
    sit_in_barber_chair();
    semSignal(cust_ready);
    semWait(finished);
    leave_barber_chair();
    semSignal(leave_b_chair);
    pay();
    semSignal(payment);
    semWait(receipt);
    exit_shop();
    semSignal(max_capacity)
}

void main()
{
    parbegin (customer, . . . 50 times, . . . customer, barber, barber,
              barber, cashier);
}

```

```

void barber()
{
    while (true)
    {
        semWait(cust_ready);
        semWait(coord);
        cut_hair();
        semSignal(coord);
        semSignal(finished);
        semWait(leave_b_chair);
        semSignal(barber_chair);
    }
}

void cashier()
{
    while (true)
    {
        semWait(payment);
        semWait(coord);
        accept_pay();
        semSignal(coord);
        semSignal(receipt);
    }
}

```

**Рис. A.2.** Неполное решение задачи о парикмахерской

Основное тело программы активизирует процессы 50 клиентов, трех парикмахеров и одного кассира. Рассмотрим теперь назначение различных синхронизирующих операторов.

- **Вместимость парикмахерской и дивана.** Вместимость парикмахерской и дивана управляется семафорами `max_capacity` и `sofa` соответственно. Каждый раз при попытке клиента войти в парикмахерскую значение семафора `max_capacity` уменьшается на 1; когда клиент покидает парикмахерскую, оно увеличивается. Если парикмахерская заполнена, то процесс клиента приостанавливается функцией `semWait(max_capacity)`. Точно так же операции `semWait` и `semSignal` окружают попытки сесть на диван и встать с него.

- Емкость парикмахерских кресел.** В наличии имеются три парикмахерских кресла, и следует обеспечить их корректное использование. Семафор `barber_chair` гарантирует такое одновременное обслуживание не более трех клиентов, чтобы один клиент не оказался на коленях у другого. Клиент не поднимется с дивана до тех пор, пока не окажется свободным хотя бы одно кресло (вызов `semWait(barber_chair)`), а каждый парикмахер сообщает о том, что его кресло освободилось (вызов `semSignal(barber_chair)`). Справедливый доступ к парикмахерским креслам гарантируется организацией очередей семафоров: клиент, который первым блокирован в очереди, первым же и приглашается на стрижку. Заметим, что если в процедуре клиента вызов `semWait(barber_chair)` разместить после `setSignal(sofa)`, то каждый клиент будет только присаживаться на диван, после чего немедленно вскакивать и занимать стартовую позицию у кресла, создавая излишнюю толкотню и мешая работать парикмахеру.
- Размещение клиента в кресле.** Семафор `cust_ready` обеспечивает подъем спящего парикмахера, сообщая ему о новом клиенте. Без этого семафора парикмахер никогда не отыходил бы и приступал бы к стрижке сразу после того, как очередной клиент покинет кресло. При отсутствии клиента в этот момент парикмахер стриг бы воздух.
- Удержание клиента в кресле.** Если уж клиент оказался в кресле, он должен отсидеть там до окончания стрижки, о чем просигнализирует семафор `finished`.
- Ограничение количества клиентов в креслах.** Семафор `barber_chair` предназначен для ограничения количества клиентов в креслах — их не должно быть более трех. Однако одного семафора `barber_chair` для этого недостаточно. Клиент, который не получит процессорное время непосредственно после того, как парикмахер сообщит о завершении работы над его прической (вызов `semSignal(finished)`), останется в кресле (например, впав в транс или глубоко задумавшись о задаче о парикмахерской), в то время как в этом же кресле будет стараться устроиться новый клиент. Для решения данной задачи используется семафор `leave_b_chair`, который не позволяет парикмахеру пригласить нового клиента до тех пор, пока предыдущий не покинет кресло. В задачах в конце приложения мы обнаружим, что даже эта предосторожность может не остановить клиентов.
- Оплата стрижки.** Естественно, с деньгами надо быть особенно осторожными. Кассир должен быть уверен, что каждый клиент, покидая парикмахерскую, сперва расплатится, а каждый клиент, оплатив стрижку, должен получить чек. Это достигается передачей денег кассиру из рук в руки — каждый клиент, покинув кресло, оплачивает услуги парикмахера, после чего дает знать об этом кассиру (вызов `semSignal(payment)`) и дожидается получения кассового чека (вызов `semWait(receipt)`). Кассир осуществляет прием платежей, ожидая сигнала о платеже, принимая деньги, а затем сообщая об этом. Здесь следует постараться избежать ряда программных ошибок. Если вызов `semSignal(payment)` выполняется непосредственно перед вызовом `pay()`, то клиент может оказаться прерванным в этот момент, и кассир будет пытаться принять не переданные деньги. Еще более серьезная ошибка происходит при обмене строк `semSignal(payment)` и `semWait(receipt)`. Это может привести к взаимоблокировке всех клиентов и кассира их операциями `semWait`.

- Координация действий кассира и парикмахера.** В целях экономии средств парикмахерская не нанимает отдельного кассира. Это действие выполняет парикмахер, когда не стрижет клиента. Для того чтобы обеспечить выполнение парикмахером в один момент времени только одной функции, используется семафор coord.

Назначение каждого семафора программы указано в табл. А.1.

**Таблица А.1. НАЗНАЧЕНИЕ СЕМАФОРОВ В КОДЕ НА РИС. А.2**

Семафор	Операция semWait	Операция semSignal
max_capacity	Клиент ожидает возможности войти в парикмахерскую	Клиент, покидающий парикмахерскую, сигнализирует об этом ожидающему входа
sofa	Клиент ожидает возможности сесть на диван	Клиент, встающий с дивана, сигнализирует об этом ожидающему возможности сесть на диван
barber_chair	Клиент ожидает, пока освободится кресло	Парикмахер сигнализирует, что кресло свободно
cust_read	Парикмахер ожидает, пока клиент сядет в кресло	Клиент сигнализирует парикмахеру, что он уже сел в кресло
finished	Клиент ожидает окончания стрижки	Парикмахер сигнализирует клиенту, что стрижка окончена
leave_b_chair	Парикмахер ожидает, пока клиент покинет кресло	Клиент сигнализирует парикмахеру о том, что он встал с кресла
payment	Кассир ожидает оплаты услуг клиентом	Клиент сигнализирует кассиру о том, что он оплатил стрижку
receipt	Клиент ожидает кассовый чек	Кассир сигнализирует о том, что оплата принята
coord	Ожидание освобождения парикмахера для стрижки или для выполнения обязанностей кассира	Сигнал об освобождении парикмахера

Процесс кассира можно устраниТЬ, внеся функцию оплаты в процедуру парикмахера. Каждый парикмахер будет последовательно стричь и принимать плату. Однако при наличии одного кассового аппарата необходимо ограничить доступ к функции accept\_pay() одним парикмахером в каждый момент времени. Этого можно добиться, рассмотрев функцию как критический участок и оградив ее соответствующим семафором.

## Полное решение задачи о парикмахерской

Несмотря на приложенные усилия (см. рис. А.2), у нас остались определенные сложности. Решение одной из проблем содержится в оставшейся части раздела, а решение прочих остается читателю в качестве упражнений (см. задачу А.3).

В листинге на рис. А.2 имеется проблема, которая может привести к некорректной работе с клиентами. Предположим, что в настоящий момент в парикмахерских креслах сидят три клиента. Они, скорее всего, заблокированы вызовами semWait (finished) и в соответствии с организацией очереди будут деблокированы в том порядке, в котором садились в кресла. Но что если один из парикмахеров очень быстро работает (или один

из клиентов лысый)? Получится ситуация, когда одного клиента будут вытаскивать из кресла и заставлять платить за незаконченную прическу, в то время как другого, полностью постриженного, будут держать в кресле силой. Эта проблема решается добавлением новых семафоров, как показано в коде на рис. А.3.

```

/* program barbershop2 */
semaphore max_capacity = 20;
semaphore sofa = 4;
semaphore barber_chair = 3, coord = 3;
semaphore mutex1 = 1, mutex2 = 1;
semaphore cust_ready = 0, leave_b_chair = 0, payment = 0, receipt = 0;
semaphore finished [50] = {0};
int count;

void customer()
{
    int custnr;
    semWait(max_capacity);
    enter_shop();
    semWait(mutex1);
    custnr = count;
    count++;
    semSignal(mutex1);
    semWait(sofa);
    sit_on_sofa();
    semWait(barber_chair);
    get_up_from_sofa();
    semSignal(sofa);
    sit_in_barber_chair();
    semWait(mutex2);
    enqueue1(custnr);
    semSignal(cust_ready);
    semSignal(mutex2);
    semWait(finished[custnr]);
    leave_barber_chair();
    semSignal(leave_b_chair);
    pay();
    semSignal(payment);
    semWait(receipt);
    exit_shop();
    semSignal(max_capacity)
}

void main()
{ count := 0;
  parbegin (customer, . . . 50 times, . . . customer, barber, barber,
            barber, cashier);
}

```

<b>void</b> customer() { int custnr; semWait(max_capacity); enter_shop(); semWait(mutex1); custnr = count; count++; semSignal(mutex1); semWait(sofa); sit_on_sofa(); semWait(barber_chair); get_up_from_sofa(); semSignal(sofa); sit_in_barber_chair(); semWait(mutex2); enqueue1(custnr); semSignal(cust_ready); semSignal(mutex2); semWait(finished[custnr]); leave_barber_chair(); semSignal(leave_b_chair); pay(); semSignal(payment); semWait(receipt); exit_shop(); semSignal(max_capacity) }	<b>void</b> barber() { int b_cust; while (true) { semWait(cust_ready); semWait(mutex2); dequeue1(b_cust); semSignal(mutex2); semWait(coord); cut_hair(); semSignal(coord); semSignal(finished[b_cust]); semWait(leave_b_chair); semSignal(barber_chair); } }  <b>void</b> cashier() { while (true) { semWait(payment); semWait(coord); accept_pay(); semSignal(coord); semSignal(receipt); } }
--	--

**Рис. А.3.** Полное решение задачи о парикмахерской

Мы присваиваем каждому пользователю уникальный номер (как если бы при входе в парикмахерскую каждый клиент получал номерок). Семафор `mutex1` обеспечивает защиту доступа к глобальной переменной `count`, гарантируя уникальность номера каждого клиента. Семафор `finished` превратился в массив из 50 семафоров. Когда клиент садится в кресло, он выполняет инструкцию `semWait(finished[custnr])`, ожидая собственный семафор. По окончании стрижки парикмахер выполняет инструкцию `signal(finished[b_cust])`, отпуская из кресла того клиента, которого он стриг.

Нам остается рассмотреть, каким образом парикмахер узнает номер клиента. Клиент помещает свой номер в очередь `enqueue1` непосредственно перед тем, как сообщить парикмахеру о готовности при помощи семафора `cust_ready`. Когда парикмахер готов стричь клиента, `dequeue1(b_cust)` удаляет номер клиента из очереди и помещает его в локальную переменную парикмахера `b_cust`.

## A.3. ЗАДАЧИ

- A.1.** Ответьте на следующие вопросы, относящиеся к полному решению задачи о парикмахерской (см. рис. A.3).
- Требует ли код, чтобы парикмахер, который заканчивает стрижку клиента, принимал оплату стрижки клиентом?
  - Всегда ли парикмахер использует одно и то же парикмахерское кресло?
- A.2.** В приведенном “полном” решении задачи о парикмахерской на рис. A.3 остается ряд вопросов. Измените программу так, чтобы устранить описываемые ниже проблемы.
- Кассир может принять оплату от одного клиента и освободить другого, если ждут возможности заплатить два или более клиентов. К счастью, как только клиент представляет платеж, нет никакого способа его вернуть, так что в конечном итоге в кассовом аппарате оказывается нужное количество денег. Тем не менее желательно отпустить клиента, как только у него взяты деньги.
  - Семафор `exit_b_chair` предположительно предотвращает множественный доступ к одному креслу парикмахера. К сожалению, это ему удается не всегда. Например, предположим, что все три парикмахера закончили стрижку и заблокированы вызовом `semWait(leave_b_chair)`. Двое из клиентов находятся в состоянии прерванного выполнения незадолго до того, как покинуть кресло парикмахера. Третий клиент покидает кресло и выполняет `semSignal(leave_b_chair)`. Какой парикмахер освободится? Так как очередь `leave_b_chair` представляет собой FIFO, освобождается тот парикмахер, который был заблокирован первым. Тот ли это парикмахер, который стриг волосы просигнализированного клиента? Может быть. Но может быть и не так. Если это не так, то новый клиент придет и сядет на колени клиенту, который только собирается вставать.
  - Программа требует, чтобы клиент сначала сел на диван, даже если кресло пустое. Конечно, это незначительная проблема, но ее исправление делает и так несколько запутанный код еще более запутанным. Тем не менее попробуйте справиться с этой проблемой.



## ПРИЛОЖЕНИЕ

Б

# ПРОЕКТЫ В ОБЛАСТИ ПРОГРАММИРОВАНИЯ И ОПЕРАЦИОННЫХ СИСТЕМ

В ЭТОМ ПРИЛОЖЕНИИ...

### Программный проект 1 — разработка оболочки

- Требования к проекту
- Представление проекта
- Требуемая документация

### Программный проект 2 — диспетчер процессов HOST

- Диспетчер процессов с четырехуровневым приоритетом
- Ограничения ресурсов
- Выделение памяти
- Процессы
- Список диспетчеризации
- Требования к проекту
- Практические результаты
- Представление проекта

Многие преподаватели считают, что реализация теоретических концепций на практике или исследовательские проекты имеют решающее значение для ясного понимания студентами концепций операционной системы. Без проектов студентам может быть трудно понять некоторые основные абстракции операционных систем и взаимодействия между их компонентами. Хороший пример концепции, которую многим студентам оказывается трудно освоить, — это семафоры. Проекты проясняют концепции, представленные в книге, дают учащемуся лучшее понимание того, как различные части операционных систем сочетаются между собой, могут мотивировать учащихся и придавать им уверенность в том, что они способны не только понимать, но и реализовывать детали операционных систем.

Такие проекты и вспомогательные материалы к ним для читателей оригинального издания книги доступны в Центре ресурсов для преподавателей (Instructor's Resource Center — IRC). Здесь для ознакомления представлена пара примеров таких проектов.

## **Б.1. ПРОГРАММНЫЙ ПРОЕКТ 1 — РАЗРАБОТКА ОБОЛОЧКИ**

Оболочка (shell), или интерпретатор командной строки, — это базовый пользовательский интерфейс операционной системы. Ваш первый проект состоит в том, чтобы написать простую оболочку — myshell — со следующими свойствами.

1. Оболочка должна поддерживать следующие внутренние команды.
  - a. cd <directory> — смена текущего каталога по умолчанию на <directory>. Если аргумент <directory> отсутствует, вывести текущий каталог. Если каталог отсутствует, вывести соответствующее сообщение об ошибке. Эта команда должна также соответствующим образом изменять переменную среды PWD.
  - b. clr — очистка экрана.
  - c. dir <directory> — вывод содержимого каталога <directory>.
  - d. environ — вывод всех переменных среды.
  - e. echo <comment> — вывод на экран <comment>, после которого выполняется переход на новую строку (множественные пробелы/табуляции могут быть сокращены до единственного пробела).
  - f. help — вывод руководства пользователя с использованием фильтра more.
  - g. pause — приостановка операций оболочки до нажатия клавиши <Enter>.
  - h. quit — выход из оболочки.
  - i. Среда оболочки должна содержать переменную shell=<pathname>/myshell, где <pathname>/myshell — полный путь к выполнимому файлу оболочки (не “прошитый” путь к вашему каталогу, а тот, откуда была выполнена оболочка).
2. Все прочие входные данные командной строки интерпретируются как вызовы программ, которые должны выполняться оболочкой с использованием механизмов fork и exec как собственные дочерние процессы. Программы должны выполняться в среде, содержащей переменную parent=<pathname>/myshell, где <pathname>/myshell такие же, как описано выше, в п. 1.и.

3. Оболочка должна быть в состоянии получать данные командной строки из файла. То есть, если оболочка вызывается с аргументом командной строки

```
myshell batchfile
```

предполагается, что файл `batchfile` содержит набор командных строк для обработки оболочкой. По достижении конца файла должно быть выполнено завершение работы оболочки. Очевидно, что если оболочка вызывается без аргумента командной строки, то она запрашивает ввод от пользователя через приглашение на дисплее.

4. Оболочка должна поддерживать перенаправление ввода-вывода для `stdin` и/или `stdout`, т.е. командная строка

```
programname arg1 arg2 < inputfile > outputfile
```

должна выполнить программу `programname` с аргументами `arg1` и `arg2` с заменой файлового потока `stdin` файлом `inputfile`, а файлового потока `stdout` — файлом `outputfile`. Перенаправление `stdout` должно также быть возможным для внутренних команд `dir`, `environ`, `echo` и `help`.

При перенаправлении вывода символ `>` должен приводить к созданию `outputfile`, если такого не существует, и его усечению, если он имеется. При перенаправлении `>>` файл `outputfile` создается, если он еще не существует, а если существует, выполняется дозапись в конец файла.

5. Оболочка должна поддерживать фоновое выполнение программ. Амперсанд (`&`) в конце командной строки указывает, что оболочка должна вернуться к командной строке сразу после запуска данной программы.
6. Приглашение командной строки должно содержать путь к текущему каталогу.

*Примечание.* Можно считать, что все аргументы командной строки (включая символы перенаправления `<`, `>` и `>>` и символ фонового выполнения `&`) будут отделены от других аргументов командной строки пробельными символами — одним или несколькими пробелами и/или символами табуляции.

## Требования к проекту

1. Разработайте простую оболочку командной строки, которая удовлетворяет вышеуказанным критериям, и реализуйте ее на конкретной платформе UNIX.
  2. Напишите простое руководство, описывающее, как использовать оболочку. Руководство должно содержать достаточно подробностей, чтобы новичок в UNIX мог его использовать. Например, вы должны объяснить концепции перенаправления ввода-вывода, программной среды и фонового выполнения программ. Руководство ДОЛЖНО быть названо `readme` и должно быть простым текстовым документом, который может быть прочитан стандартным текстовым редактором.
- В качестве примера требуемых глубины и типа описания вы можете рассмотреть онлайн-руководства для `csh` и `tcsh` (`man csh`, `man tcsh`). Эти оболочки, естественно, обладают гораздо большей функциональностью, чем разрабатываемая вами, а потому ваши руководства не обязательно должны быть такими большими.

Вы НЕ должны включать в руководство инструкции по сборке, в том числе списки файлов или исходный код. Это должно быть руководство оператора, а не руководство разработчика.

3. Исходный код **ДОЛЖЕН** быть тщательно прокомментирован и соответствующим образом структурирован, чтобы ваши коллеги могли легко понимать и поддерживать ваш код.
4. Детали процедур передачи работы оговариваются с преподавателем.
5. Передаваемая на проверку работа должна содержать только файл (или файлы) исходного кода, включая `makefile` и файл `readme`. Никакая выполняемая программа не должна быть включена. Проверяющий должен выполнить построение вашей оболочки из предоставленного исходного кода. Если представленный код не компилируется, работа считается невыполненной.
6. `makefile` **ДОЛЖЕН** генерировать бинарный файл `myshell`. Вот пример простейшего `makefile`

```
# Joe Citizen, s1234567 - Operating Systems Project 1
# CompLab1/01 tutor: Fred Bloggs
myshell: myshell.c utility.c myshell.h
        gcc -Wall myshell.c utility.c -o myshell
```

Программа `myshell` генерируется с помощью простой команды `make` в приглашении командной строки.

*Примечание.* Четвертая строка в приведенном выше примере `makefile` **ДОЛЖНА** начинаться с символа табуляции.

7. В показанном выше примере в представленном каталоге должны быть файлы

```
makefile
myshell.c
utility.c
myshell.h
readme
```

## Представление проекта

`makefile` обязателен. Все файлы в вашем представлении будут скопированы в один и тот же каталог, поэтому не включайте в свой `makefile` конкретные пути. `makefile` должен включать все зависимости, которые строят вашу программу. Если в проект включена библиотека, ваш `makefile` должен также собирать библиотеку.

**Не сдавайте преподавателю никакие бинарные или объектные файлы.** Все, что требуется, — ваш исходный код, `makefile` и файл `readme`. Протестируйте свой проект, скопировав исходный код в пустой каталог и скомпилировав его с помощью команды `make`.

При проверке используется сценарий оболочки, который копирует ваши файлы в тестовый каталог, удаляет все существующие ранее файлы `myshell`, `*.a` и/или `*.o`, выполняет команду `make`, копирует набор тестовых файлов в тестовый каталог, а затем выполняет оболочку со стандартным набором тестовых сценариев с использованием `stdin` и аргументов командной строки.

## Требуемая документация

Оцениваются как исходный текст, так и руководство `readme`. В обязательном порядке код должен быть тщательно комментирован. Руководство пользователя может быть представлено в любом формате по вашему выбору (в пределах ограничения, что он должен корректно отображаться простым текстовым редактором). Руководство должно содержать достаточно подробностей, чтобы новичок в UNIX мог использовать вашу оболочку. Например, вы должны объяснить концепции перенаправления ввода-вывода, программной среды и фонового выполнения программ. Файл руководства **ДОЛЖЕН** быть назван `readme` (строчными буквами и без расширения `.txt`).

## Б.2. ПРОГРАММНЫЙ ПРОЕКТ 2 — ДИСПЕТЧЕР ПРОЦЕССОВ HOST

Hypothetical Operating System Testbed (HOST, тестовый стенд гипотетической операционной системы) — это многопрограммная система диспетчера процессов с четырехуровневым приоритетом, работающим в рамках ограниченных доступных ресурсов.

### Диспетчер процессов с четырехуровневым приоритетом

Диспетчер работает с четырьмя уровнями приоритетов.

1. Процессы реального времени должны запускаться немедленно в порядке поступления (первым пришел — первым обслужен), вытесняя любые другие процессы, работающие с более низким приоритетом. Эти процессы выполняются до полного их завершения.
2. Обычные пользовательские процессы выполняются трехуровневым диспетчером со снижением приоритета (см. рис. Б.1). Основной квант времени диспетчера составляет одну секунду. Это также значение для кванта времени планировщика со снижением приоритета.

Диспетчер должен поддерживать две очереди подачи заданий — реального времени и с пользовательским приоритетом — из списка заданий на диспетчеризацию. Список проверяется на каждом такте диспетчера, и “поступившие” задания переносятся в соответствующую очередь. Затем эти очереди проверяются; все задания реального времени выполняются до конца, вытесняя любые другие задания, запущенные в данный момент.

Очередь заданий реального времени должна быть пустой перед тем, как будет снова активирован диспетчер со снижением приоритета с более низким приоритетом. Любые задания с пользовательским приоритетом в очереди заданий пользователя, которые могут выполняться в пределах доступных ресурсов (памяти и устройств ввода-вывода), переносятся в соответствующую приоритету очередь. Нормальная работа очереди со снижением приоритета принимает все задания с наивысшим уровнем приоритета и снижает приоритет после каждого кванта времени. Однако данный диспетчер имеет возможность принимать и задания с более низким приоритетом, вставляя их в соответствующую очередь. В случае, когда все задания принимаются с самым низким приоритетом, это позволяет эмулировать простой циклический диспетчер (рис. Б.2).

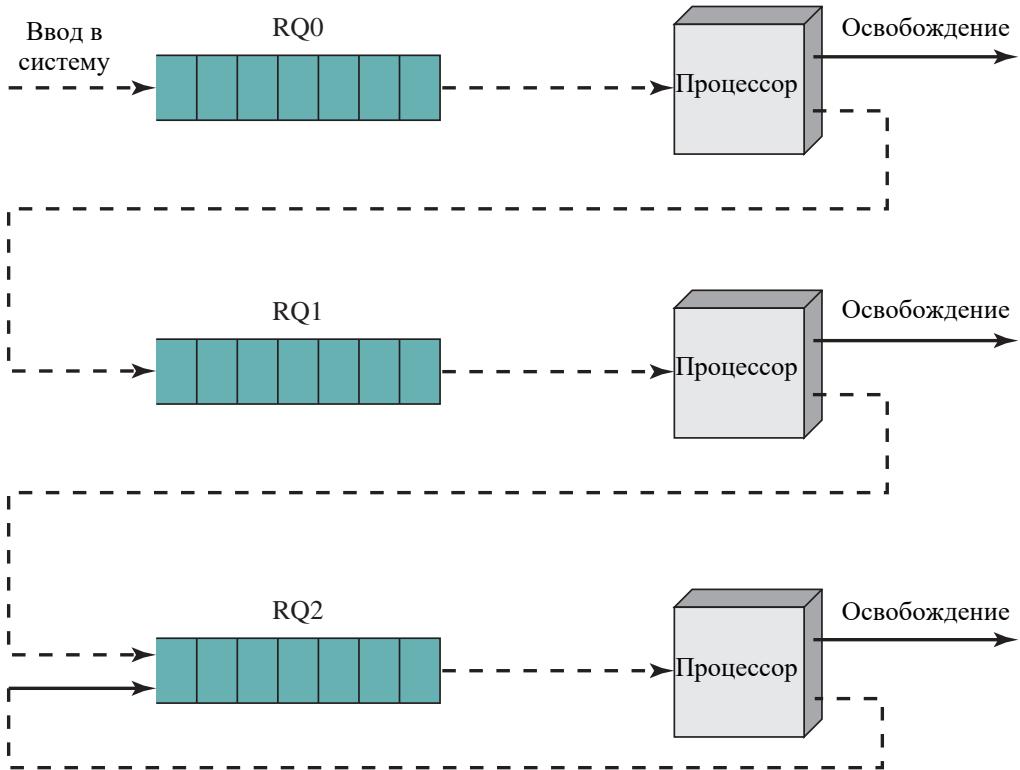


Рис. Б.2

Когда все “готовые” задания с более высоким приоритетом завершены, диспетчер со снижением приоритета возобновляет работу, начиная или продолжая выполнение процесса в начале непустой очереди с наивысшим приоритетом. На следующем такте текущее задание приостанавливается (или завершается, а его ресурсы освобождаются), если имеются другие “готовые” задания с тем же или более высоким приоритетом. Логически поток имеет вид, показанный на рис. Б.3 (и обсуждается далее в задании проекта).

## Ограничения ресурсов

Операционная система HOST имеет следующие ресурсы.

- Два принтера
- Сканер
- Модем
- Два CD-привода
- 1024 Мбайт памяти, доступной всем процессам

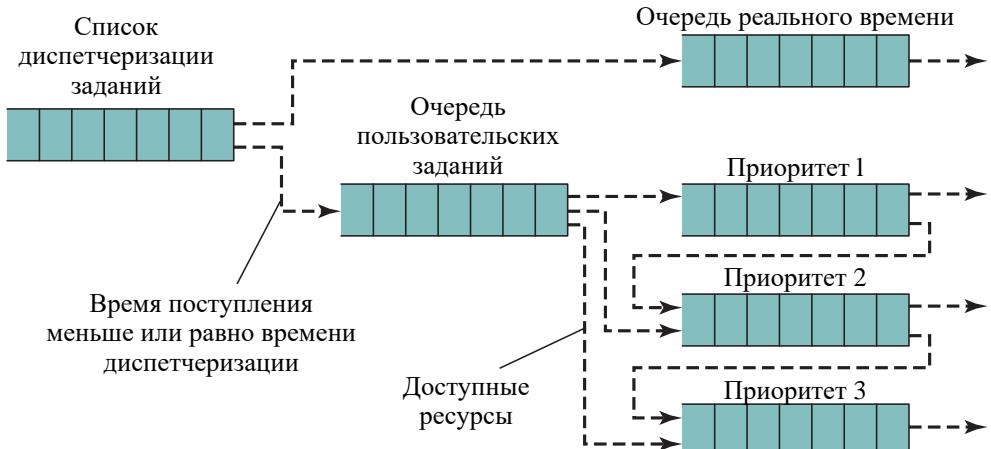


Рис. Б.3

Процессы с низким приоритетом могут использовать любой из этих ресурсов (или их все), но при передаче процесса диспетчер HOST уведомляется о том, какие ресурсы будет использовать этот процесс. Диспетчер гарантирует, что каждый запрошенный ресурс будет доступен этому процессу исключительно в течение всего его времени жизни в очередях “готовых к запуску” процессов: от первоначальной передачи из очереди заданий в очереди с приоритетом 1–3 до завершения процесса, включая промежуточные кванты времени простоя.

Процессам реального времени не требуются какие-либо ресурсы ввода-вывода (принтер, сканер, модем и CD-привод), но, очевидно, требуется память; в этом случае требование к памяти для заданий реального времени всегда будет составлять 64 Мбайт или менее.

## Выделение памяти

Каждому процессу должен быть назначен **непрерывный** блок памяти, который должен оставаться закрепленным за процессом в течение всего жизненного цикла последнего.

Необходимо оставить достаточное количество непрерывной зарезервированной памяти для того, чтобы процессы реального времени не блокировались в процессе выполнения — 64 Мбайт для выполнения задания в реальном времени, оставляя 960 Мбайт для совместного использования “активными” пользовательскими заданиями.

Аппаратное обеспечение HOST не поддерживает использование виртуальной памяти, так что свопинг памяти на диск невозможен. Невозможна и страничная организация памяти.

В рамках этих ограничений может использоваться любая подходящая схема распределения памяти с переменными размерами выделяемых фрагментов (первый подходящий, очередной подходящий, система двойников и т.д.).

## Процессы

Процессы в HOST моделируются диспетчером путем создания нового процесса для каждого диспетчерируемого процесса. Такой процесс является обобщенным (generic) (исходный текст: `sigtrap.c`), который можно использовать для любого процесса с приоритетом. Фактически он работает с очень низким приоритетом, в течение одной секунды находится в спящем состоянии и отображает следующее:

1. сообщение, указывающее идентификатор процесса при его запуске;
2. регулярно, раз в секунду, повторяемое сообщение о работе процесса;
3. сообщение об изменении состояния процесса — о приостановке, возобновлении, завершении.

Процесс, если он не был прерван вашим диспетчером, завершается через 20 с сам по себе. Вывод процесса осуществляется с использованием случайно генерированной цветовой схемы для каждого уникального процесса, поэтому отдельные “срезы” процессов могут быть легко различимыми. Используйте этот процесс, а не собственный.

Жизненный цикл процесса выглядит следующим образом.

1. Процесс передается во входную очередь диспетчера с помощью начального списка процессов, который определяет время поступления, приоритет, требуемое процессорное время (в секундах), размер блока памяти и другие запрашиваемые ресурсы.
2. Процесс является “готовым” к запуску, когда он является “прибывшим” и все необходимые для него ресурсы доступны.
3. Все ожидающие задания реального времени передаются для выполнения на основе принципа “первым пришел — первым обслужен”.
4. Если для пользовательского процесса с более низким приоритетом доступно достаточно количество ресурсов и памяти, то процесс переносится в соответствующую очередь в блоке диспетчера со снижением приоритета, а оставшиеся индикаторы ресурсов (список памяти и устройства ввода-вывода) обновляются.
5. Когда задание запущено (с помощью `fork` и `exec`), диспетчер перед выполнением `exec` выводит параметры задания (идентификатор процесса, приоритет, оставшееся время процессора (в секундах), расположение в памяти и размер блока, а также запрошенные ресурсы).
6. Процесс реального времени может выполняться до тех пор, пока не истечет его время и диспетчер завершит его, отправив ему сигнал `SIGINT`.
7. Заданию пользователя с низким приоритетом разрешается выполняться в течение одного такта диспетчера (одна секунда), прежде чем оно будет приостановлено (`SIGTSTP`) или прекращено (`SIGINT`), если его время истекло. При приостановке уровень приоритета задания понижается (если это возможно), и оно ставится в соответствующую очередь, как показано на рис. Б.1 и Б.3. Чтобы сохранить синхронизацию вывода между вашим диспетчером и дочерним процессом, диспетчер должен дождаться ответа процесса на сигнал `SIGTSTP` или `SIGINT`, прежде чем продолжить работу (`waitpid(p->pid, &status, WUNTRACED)`). Чтобы соот-

ветствовать последовательности производительности, указанной при сравнении политик планирования (см. рис. 9.5), если другой процесс не ожидает (перезапуска), пользовательское задание не следует приостанавливать и перемещать на уровень с более низким приоритетом.

8. При условии, что в очереди не ожидают выполнения задания с более высоким приоритетом реального времени, запускается или перезапускается процесс с наивысшим приоритетом в очередях со снижением приоритета (SIGCONT).
9. Когда процесс завершается, используемые им ресурсы возвращаются диспетчеру для перераспределения для процессов, запускаемых в дальнейшем.
10. Когда в списке диспетчеризации больше нет процессов, диспетчер завершает работу.

## Список диспетчеризации

Список диспетчеризации — это список процессов, которые должен обработать диспетчер. Список содержится в текстовом файле, указанном в командной строке, т.е.

```
>hostd dispatchlist
```

Каждая строка списка описывает один процесс со следующими данными в виде списка с разделителями-запятыми:

```
<Время поступления>, <приоритет>, <процессорное время>,
<память в мегабайтах>, <принтеры>, <сканеры>, <модемы>, <CD>
```

Таким образом, список

```
12, 0, 1, 64, 0, 0, 0
12, 1, 2, 128, 1, 0, 0, 1
13, 3, 6, 128, 1, 0, 1, 2
```

описывает следующее.

**Задание 1:** поступление в момент времени 12, приоритет — 0 (реального времени), требует 1 с процессорного времени и 64 Мбайт памяти; устройства ввода-вывода не нужны.

**Задание 2:** поступление в момент времени 12, приоритет — 1 (пользовательское задание с наивысшим приоритетом), требует 2 с процессорного времени и 128 Мбайт памяти, один принтер и один CD-привод.

**Задание 3:** поступление в момент времени 13, приоритет — 3 (пользовательское задание с наименшим приоритетом), требует 6 с процессорного времени и 128 Мбайт памяти, один принтер, один модем и два CD-привода.

Текстовый файл представления может быть любой длины и содержать до 1000 заданий. Он завершается символом конца строки, за которым следует маркер конца файла.

Входные списки диспетчера для проверки работы отдельных функций диспетчера описаны ниже в проектном задании. Эти списки почти наверняка станут основой тестов, которые будут применены к вашему диспетчеру во время проверки. Очевидно, что ваша реализация диспетчера будет проверяться и с иными, более сложными комбинациями.

## Требования к проекту

1. Разработайте диспетчер, удовлетворяющий представленным выше критериям.
  - a. Опишите и обсудите, какие алгоритмы распределения памяти вы могли бы использовать, и обоснуйте свой окончательный выбор.
  - b. Опишите и обсудите структуры, используемые диспетчером для организации очередей, диспетчеризации и распределения памяти и других ресурсов.
  - c. Опишите и обоснуйте общую структуру своей программы, описав различные модули и их основные функции (ожидается описание функционального интерфейса).
  - d. Обсудите, почему используется такая многоуровневая схема диспетчеризации, сравнивая ее со схемами, используемыми "реальными" операционными системами. Опишите недостатки схемы, предложите возможные ее улучшения. Включите в свои обсуждения схемы памяти и распределения ресурсов.

Официальный проектный документ должен содержать подробные обсуждения, описания и аргументы. Он должен быть представлен отдельно в виде физического бумажного документа. Проектный документ НЕ должен включать какой-либо исходный код.

2. Реализуйте диспетчер на языке программирования С.
3. Исходный код **ДОЛЖЕН** быть тщательно прокомментирован и соответствующим образом структурирован, чтобы ваши коллеги могли легко понимать и поддерживать ваш код.
4. Детали процедур передачи работы оговариваются с преподавателем.
5. Передаваемая на проверку работа должна содержать только файл (или файлы) исходного кода, заголовочные файлы и makefile. Никакая выполняемая программа не должна быть включена. Проверяющий должен выполнить построение вашей оболочки из предоставленного исходного кода. Если представленный код не компилируется, работа считается невыполненной.
6. makefile должен генерировать бинарный файл hostd. Вот пример простейшего makefile.

```
# Joe Citizen, s1234567 - Operating Systems Project 2
# CompLab1/01 tutor: Fred Bloggs
hostd: hostd.c utility.c hostd.h
gcc hostd.c utility.c -o hostd
```

Программа hostd генерируется с помощью простой команды make в приглашении командной строки.

*Примечание.* Четвертая строка в приведенном выше примере makefile **ДОЛЖНА** начинаться с символа табуляции.

## Практические результаты

1. Исходные файлы, заголовочные файлы, makefile.
2. Проектная документация, описанная выше.

## Представление проекта

makefile обязателен. Все файлы в вашем представлении будут скопированы в один и тот же каталог, поэтому не включайте в свой makefile конкретные пути. makefile должен включать все зависимости, которые строят вашу программу. Если в проект включена библиотека, ваш makefile должен также собирать библиотеку.

**Не сдавайте преподавателю никакие бинарные или объектные файлы.** Все, что требуется, — ваш исходный код и makefile. Протестируйте свой проект, скопировав исходный код в пустой каталог и скомпилировав его с помощью команды make.

При проверке используется сценарий оболочки, который копирует ваши файлы в тестовый каталог, удаляет все существовавшие ранее файлы myshell, \*.a и/или \*.o, выполняет команду make, копирует набор тестовых файлов в тестовый каталог, а затем выполняет оболочку со стандартным набором тестовых сценариев с использованием stdin и аргументов командной строки.



## ПРИЛОЖЕНИЕ



# ДОПОЛНИТЕЛЬНЫЕ ВОПРОСЫ ПАРАЛЛЕЛЬНОСТИ

В ЭТОМ ПРИЛОЖЕНИИ...

## **В.1. Регистры процессора**

Регистры, доступные пользователю

Управляющие регистры и регистры состояния

## **В.2. Выполнение команд функций ввода-вывода**

### **В.3. Технологии ввода-вывода**

Программируемый ввод-вывод

Ввод-вывод, управляемый прерываниями

Прямой доступ к памяти

## **В.4. Вопросы аппаратной производительности в многоядерных системах**

Увеличение степени параллелизма

Энергопотребление

## B.1. РЕГИСТРЫ ПРОЦЕССОРА

В процессоре имеется набор регистров, представляющих собой область памяти быстрого доступа, но намного меньшей емкости, чем основная память. Регистры процессора выполняют две функции.

- **Регистры, доступные пользователю.** Эти регистры позволяют программисту сократить число обращений к основной памяти, оптимизируя использование регистров с помощью машинного языка или ассемблера. В состав языков высокого уровня входят оптимизирующие компиляторы, построенные на алгоритмах, которые, в частности, позволяют определить, какие переменные следует заносить в регистры, а какие — в основную память. Некоторые языки высокого уровня, такие как С, предоставляют программисту возможность предложить компилятору хранить те или иные данные в регистрах.
- **Регистры управления и регистры состояния.** Используются в процессоре для контроля над выполняемыми операциями; с их помощью привилегированные программы операционной системы могут контролировать ход выполнения других программ.

Для разделения регистров на эти две категории не существует определенного признака. Например, на одних машинах оператор имеет возможность следить за состоянием программного счетчика, а на других — нет. Однако такое разделение удобно при дальнейшем рассмотрении.

### Регистры, доступные пользователю

К доступным регистрам пользователь может обращаться с помощью команд машинного языка. К этим регистрам, как правило, имеют доступ все программы — как приложения, так и системные. Обычно среди доступных регистров есть регистры данных, адресные регистры и регистры кода условия.

**Регистры данных.** Программист может применять их в различных целях. В ряде случаев они имеют общее назначение и могут использоваться любой машинной командой для операций с данными. Однако зачастую при этом накладываются определенные ограничения. Например, некоторые регистры предназначены для операций над числами с плавающей точкой, в то время как остальные — для хранения целых чисел.

**Адресные регистры.** В них заносятся адреса команд и данных в основной памяти; в этих регистрах может быть записана только часть адреса, использующаяся при вычислении полного или эффективного адреса. Рассмотрим следующие примеры.

- **Индексный регистр.** Используется в обычном режиме адресации, когда адрес получается в результате сложения содержимого индексного и базового регистров.
- **Сегментный регистр.** При сегментной адресации память разделяется на блоки (сегменты), состоящие из различного количества машинных слов<sup>1</sup>. Адрес ячейки памяти складывается из адреса сегмента и смещения относительно начала сегмента.

<sup>1</sup> Универсального определения термина “слово” не существует. В общем случае *слово* — это упорядоченный набор байтов или битов, которые представляют собой обычную единицу хранения, передачи или обработки информации в данном компьютере. Как правило, если процессор имеет набор команд фиксированной длины, то длина команды равна длине слова.

Имея представление о таком режиме адресации, легче будет усвоить материал, изложенный в главе 7, “Управление памятью”, в которой обсуждаются методы управления памятью. При этом режиме адресации базовый адрес сегмента (его начало) хранится в одном из регистров. Таких регистров может быть несколько; например, один — для операционной системы (т.е. использующийся при выполнении процессором кода операционной системы), другие — для выполняющихся в данный момент приложений.

- **Регистр стека.** При стековой адресации<sup>2</sup> выделяется специальный регистр, в котором размещен указатель на вершину стека. Этот режим адресации позволяет использовать некоторые команды, в которых отсутствует поле адреса, например push и pop.

В некоторых машинах вызов процедуры или подпрограммы приводит к автоматическому сохранению содержимого всех доступных пользователю регистров, чтобы по возвращении их можно было восстановить. Процедура сохранения и восстановления, являющаяся составной частью команды вызова и возврата, выполняется процессором. Такой подход позволяет процедурам использовать регистры независимо друг от друга. В других машинах операция сохранения содержимого регистров, выполняемая при вызове процедуры, является обязанностью программиста, который должен включить в программу необходимые команды. Таким образом, в зависимости от типа процессора функция сохранения и восстановления может выполняться либо аппаратно, либо программно.

## Управляющие регистры и регистры состояния

Для контроля над работой процессора используются различные регистры. В большинстве машин эти регистры в основном недоступны пользователю. Некоторые из них могут быть доступны для машинных команд, выполняемых в так называемом режиме управления или режиме ядра.

Конечно, у разных типов машин разная организация регистров; для их классификации также используется различная терминология. Здесь приводится довольно полный список типов регистров и дается их краткое описание. Кроме упомянутых в главе 1, “Обзор компьютерной системы”, регистров MAR, MBR, I/OAR и I/OBR (см. рис. 1.1), важными для выполнения команд являются следующие регистры.

- **Счетчик команд (program counter — PC).** Содержит адрес команды, которая должна быть выбрана из памяти.
- **Регистр команд (instruction register — IR).** Содержит последнюю выбранную из памяти команду.

В состав всех процессоров входит также регистр (или набор регистров), известный как **регистр слова состояния программы** (program status word — PSW). В нем, как правило, содержатся коды условий и другая информация о состоянии, например бит разрешения/запрета прерываний или бит режима “системный/пользовательский”.

---

<sup>2</sup> Стек размещается в основной памяти в виде последовательности ячеек. Он похож на стопку бумаг, в которой листы с данными можно брать и кладь только сверху. Более подробно стек рассматривается в приложении П, “Управление процедурами”.

**Коды условий** (известные также как *флаги*) — это последовательность битов, устанавливаемых или сбрасываемых процессором в зависимости от результата выполненных операций. Например, в результате выполнения арифметического действия может получиться положительное число, отрицательное, нуль, либо может произойти переполнение. В дополнение к сохранению полученного значения в памяти или регистре в результате арифметических операций устанавливаются также соответствующие коды условий. Впоследствии они могут быть проверены условной операцией ветвления. Биты кодов условий группируются в один или несколько регистров (обычно они составляют часть регистра управления). Вообще говоря, есть машинные команды, позволяющие прочитать содержимое этих битов с помощью явных обращений к регистрам; однако изменять их содержимое явным образом нельзя, поскольку эти биты предназначены для отображения результатов выполнения команд.

В машинах, в которых используются различные виды прерываний, может быть предусмотрено несколько регистров прерываний с указателями на каждую программу обработки прерываний. Если при реализации некоторых функций (например, вызова процедуры) используется стек, процессор должен иметь регистр — указатель стека. Для аппаратного обеспечения управления памятью (см. главу 7, “Управление памятью”) нужны свои регистры. И наконец, регистры часто используются при управлении операциями ввода-вывода.

На устройство и организацию управляющих регистров и регистров состояния влияет несколько факторов. Одним из них является поддержка операционной системы. Различные виды управляющей информации используются операционной системой по-разному. Если разработчик процессора имеет ясное представление об операционной системе, которая будет работать с этим процессором, он сможет так спланировать организацию регистров, чтобы обеспечить аппаратную поддержку ряда возможностей, таких как защита памяти или переключение пользовательских программ.

Еще одним ключевым конструкторским решением является распределение управляющей информации между регистрами и памятью. Общепринятым подходом является выделение для нее нескольких первых сотен или тысяч слов памяти. Конструктор должен решить, какая часть информации будет находиться в более дорогих, но более быстрых регистрах, а какая часть — в более дешевой, но медленной основной памяти.

## В.2. ВЫПОЛНЕНИЕ КОМАНД ФУНКЦИЙ ВВОДА-ВЫВОДА

Этот раздел дополняет информацию из раздела 1.3.

Процессор может не только читать данные из памяти и записывать их в нее, обращаясь по адресу к определенной ячейке, но также читать и записывать данные в устройство ввода-вывода. Таким образом, устройство ввода-вывода (например, контроллер диска) обменивается данными с процессором. При этом процессор должен идентифицировать устройство, которое будет управляться определенным устройством ввода-вывода. Из команд ввода-вывода можно сформировать такие же последовательности, как показанные на рис. 1.4 последовательности команд обращения к памяти.

Иногда желательно, чтобы обмен данными с памятью выполнялся непосредственно устройством ввода-вывода, а процессор в это время выполнял другие задания. В этом

случае процессор передает устройству ввода-вывода полномочия для чтения из памяти и записи в память, что позволяет освободить процессор. Во время такой передачи данных устройство ввода-вывода читает или записывает команды в память, принимая на себя ответственность за этот обмен. Эта операция, известная под названием прямого доступа к памяти (direct memory access — DMA), рассматривается в разделе 1.7.

## B.3. Технологии ввода-вывода

Возможны три метода выполнения операций ввода-вывода:

- программируемый ввод-вывод;
- ввод-вывод с использованием прерываний;
- прямой доступ к памяти (direct memory access — DMA).

### Программируемый ввод-вывод

Когда процессору при выполнении программы встречается команда, связанная с вводом-выводом, он выполняет ее, передавая соответствующие команды модулю ввода-вывода. При программируемом вводе-выводе это устройство выполняет требуемое действие, а затем устанавливает соответствующие биты в регистрах состояния ввода-вывода. Модуль ввода-вывода больше не посылает процессору никаких сигналов, в том числе сигналов прерываний. Таким образом, ответственность за периодическую проверку состояния модуля ввода-вывода несет процессор; он должен производить проверку до тех пор, пока операция ввода-вывода не завершится.

При использовании такого метода процессор отвечает за извлечение из основной памяти данных, предназначенных для вывода, и за размещение в ней данных, поступивших с устройств ввода. Программное обеспечение для ввода-вывода разрабатывается таким образом, что процессор непосредственно управляет операциями ввода-вывода, включая опознание состояния устройства, пересылку команд чтения-записи и передачу данных. Таким образом, в набор используемых команд входят команды ввода-вывода, принадлежащие следующим категориям.

- **Управление.** Команды этой категории используются для того, чтобы привести внешнее устройство в действие и сообщить ему, что нужно делать. Например, блоку с магнитной лентой можно отдать команду перемотки или перемещения вперед на одну запись.
- **Состояние.** Используется для проверки состояния модуля ввода-вывода и соответствующих периферийных устройств.
- **Передача.** Используется для чтения и/или записи данных в регистры процессора и внешние устройства и из регистров процессора и внешних устройств.

На рис. B.1, а приведен пример использования программируемого ввода-вывода для считывания в память блока данных, поступивших из внешнего устройства (например, записи с магнитной ленты). Данныечитываются по одному слову (16 бит) за раз. При считывании каждого слова процессор должен выполнять цикл проверки состояния, пока не обнаружит, что это слово уже доступно в регистре данных модуля ввода-вывода. На приведенной блок-схеме видны основные недостатки такого метода: процессор выполняет большое количество операций, которых можно было бы избежать; теряется много времени.

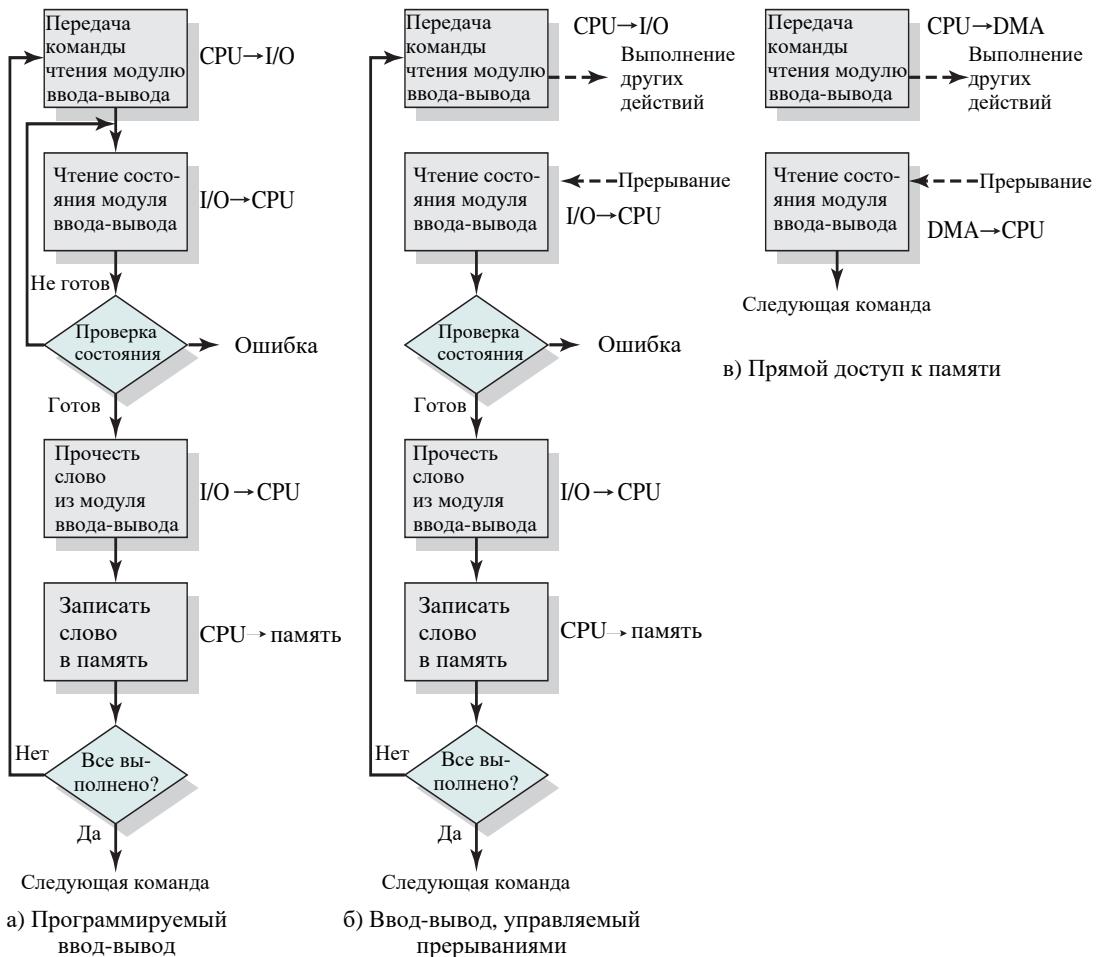


Рис. В.1. Три метода передачи блока входных данных

## ВВОД-ВЫВОД, УПРАВЛЯЕМЫЙ ПРЕРЫВАНИЯМИ

Проблема программируемого ввода-вывода состоит в том, что процессор должен долго ждать, пока модуль ввода-вывода будет готов читать или принимать новые данные. Во время ожидания процессор должен постоянно производить опрос, чтобы узнать состояние модуля ввода-вывода. В результате значительно падает производительность всей системы.

При альтернативном подходе процессор может передать модулю команду ввода-вывода, а затем перейти к выполнению другой полезной работы. Затем, когда модуль ввода-вывода снова будет готов обмениваться данными с процессором, он прервет процессор и потребует, чтобы его обслужили. Процессор передает ему новые данные, а затем возобновляет прерванную работу.

Рассмотрим для начала, как все это выглядит с точки зрения модуля ввода-вывода. Сначала он получает от процессора команду READ и переходит к считыванию данных из связанного с ним периферийного устройства. Как только эти данные поступят в регист-

ры данных модуля, он посыпает процессору по шине управления сигнал прерывания и ожидает, когда процессор запросит эти данные. При поступлении запроса модуль передает данные по информационнойшине и переходит в состояние готовности для новых операций ввода-вывода.

С точки зрения процессора передача входных данных выглядит следующим образом. Процессор генерирует команду READ, сохраняет содержимое программного счетчика и других регистров, соответствующих выполняемой программе, и переходит к выполнению других операций (например, в одно и то же время может выполнять несколько различных программ). В конце каждого цикла команды процессор проверяет наличие прерываний (см. рис. 1.7). При поступлении прерывания от модуля ввода-вывода процессор сохраняет контекст выполняющейся в данный момент задачи и выполняет программу, обрабатывающую прерывания. При этом он считывает слово данных из модуля ввода-вывода и сохраняет его в память. Затем он восстанавливает контекст программы, от которой поступила команда ввода-вывода, и продолжает работу.

Использование для чтения блока данных ввода-вывода, управляемого прерываниями, показано на рис. В.1, б. Ввод-вывод с прерываниями намного эффективнее, чем программируемый ввод-вывод, так как при нем исключается ненужное ожидание. Однако этот процесс все еще потребляет много процессорного времени, потому что каждое слово, которое передается из памяти в модуль ввода-вывода или в обратном направлении, должно пройти через процессор.

Почти в каждой компьютерной системе есть несколько модулей ввода-вывода, поэтому нужны механизмы, позволяющие процессору определить, какое из устройств вызвало прерывание, а если прерывание одно, то решить, какое из них будет обрабатываться в первую очередь. В некоторых системах имеется несколько шин прерываний, так что каждый модуль ввода-вывода посыпает сигнал по своей шине, причем у каждой шины — свой приоритет. Есть и другой вариант, когда прерывающая шина всего одна, но тогда используются дополнительные шины, по которым передаются адреса устройств. В этом случае каждому устройству также присваиваются разные приоритеты.

## Прямой доступ к памяти

Хотя ввод-вывод, управляемый прерываниями, более эффективен, чем простой программируемый ввод-вывод, он все еще занимает много процессорного времени для передачи данных между памятью и модулем ввода-вывода (при этом через процессор должны пройти все пересыпаемые данные). Таким образом, обе описанные формы ввода-вывода обладают двумя недостатками.

1. Скорость передачи данных при вводе-выводе ограничена скоростью, с которой процессор может проверять и обслуживать устройство.
2. Процессор занят организацией передачи данных; при вводе-выводе для каждой передачи данных должна быть выполнена определенная последовательность команд.

Для перемещения больших объемов данных может использоваться более эффективный метод — прямой доступ к памяти (direct memory access — DMA). Функции DMA выполняются отдельным модулем системной шины или могут быть встроены в модуль ввода-вывода. В любом случае метод работает следующим образом. Когда процессору нужно прочитать или записать блок данных, он генерирует команду для модуля DMA, отсылая ему следующую информацию:

- указание, требуется ли выполнить чтение или запись;
- адрес устройства ввода-вывода;
- начальный адрес блока памяти, использующегося для чтения или записи;
- количество слов, которые должны быть прочитаны или записаны.

Передав полномочия по выполнению этих операций модулю DMA, процессор продолжает работу. Модуль DMA слово за словом передает весь блок данных в память или из нее, не задействуя при этом процессор. После окончания передачи модуль DMA посылает процессору сигнал прерывания. Таким образом, процессор участвует только в начале и в конце передачи (см. рис. В.1,в).

Для передачи данных в память и из нее модулю DMA нужен контроль над шиной. Если в это время процессору также нужна шина, может возникнуть конфликтная ситуация, и процессор должен ждать окончания работы модуля DMA. Заметим, что в этом случае нельзя говорить о прерывании, так как процессор не сохраняет информацию о состоянии задачи и не переходит к выполнению других операций. Вместо этого он вынужден сделать паузу на время выполнения одного цикла шины. В результате это приведет к тому, что во время передачи данных с использованием прямого доступа к памяти замедляется выполнение процессором тех команд, для которых ему требуется шина. Тем не менее при передаче большого количества информации прямой доступ к памяти намного более эффективен, чем программируемый ввод-вывод или ввод-вывод, управляемый прерываниями.

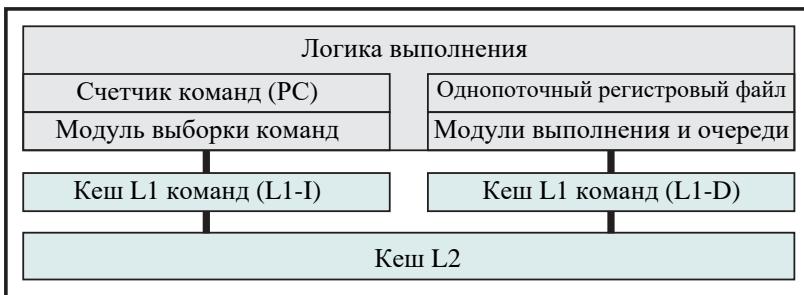
## **В.4. ВОПРОСЫ АППАРАТНОЙ ПРОИЗВОДИТЕЛЬНОСТИ В МНОГОЯДЕРНЫХ СИСТЕМАХ**

Микропроцессорные системы демонстрируют устойчивое экспоненциальное увеличение производительности на протяжении десятилетий. Это отчасти связано с улучшениями в организации процессора на чипе и отчасти — с увеличением тактовой частоты.

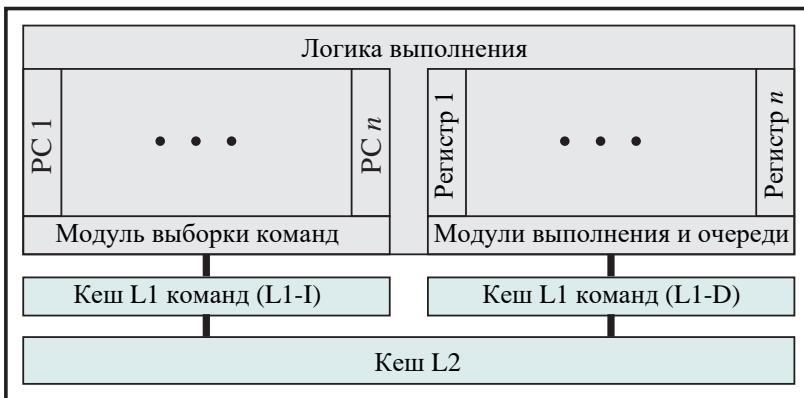
### **Увеличение степени параллелизма**

Организационные изменения в дизайне процессора были в основном сосредоточены на увеличении параллелизма на уровне инструкций, так, чтобы за каждый такт можно было выполнять большее количество работы. Эти изменения включают в хронологическом порядке следующие (рис. В.2).

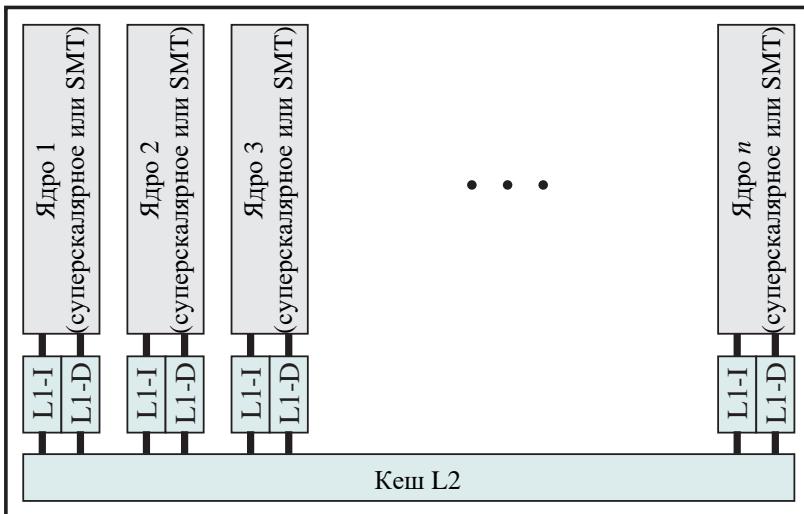
- **Конвейерная обработка.** Отдельные команды выполняются с использованием конвейера, так что в то время как одна команда выполняется на одной стадии конвейера, другая команда выполняется на другой стадии конвейера.
- **Суперскалярность.** Несколько конвейеров создаются путем репликации ресурсов выполнения. Тем самым обеспечивается параллельное выполнение команд в параллельных конвейерах.
- **Одновременная многопоточность** (simultaneous multithreading — SMT). Банки регистров реплицируются таким образом, что несколько потоков могут совместно использовать ресурсы конвейера.



а) Суперскалярность



б) Одновременная многопоточность (SMT)



в) Многоядерность

Рис. В.2. Альтернативные организации процессоров

Для каждого из этих нововведений проектировщики на протяжении многих лет пытались увеличить производительность системы путем добавления сложности. В случае конвейерной обработки простые трехстадийные конвейеры были заменены конвейерами с пятью стадиями обработки, а затем было добавлено еще много этапов, так что некоторые реализации имеют их более десятка. У этой тенденции есть практический предел, потому что с ростом числа стадий растет необходимость в большей логике, большем количестве соединений и большем количестве управляющих сигналов. В случае суперскалярной организации повышение производительности может быть достигнуто за счет увеличения количества параллельных конвейеров. Но и здесь с ростом количества конвейеров результирующий эффект снижается — требуется больше логики для управления рисками и обработки команд на каждой стадии. В конечном итоге единственный поток выполнения достигает точки, в которой риски и зависимости ресурсов препятствуют полному использованию нескольких доступных конвейеров. Такая же точка насыщения достигается и при использовании SMT, так как сложность управления несколькими потоками в наборе конвейеров ограничивает как количество потоков, так и количество эффективно используемых конвейеров.

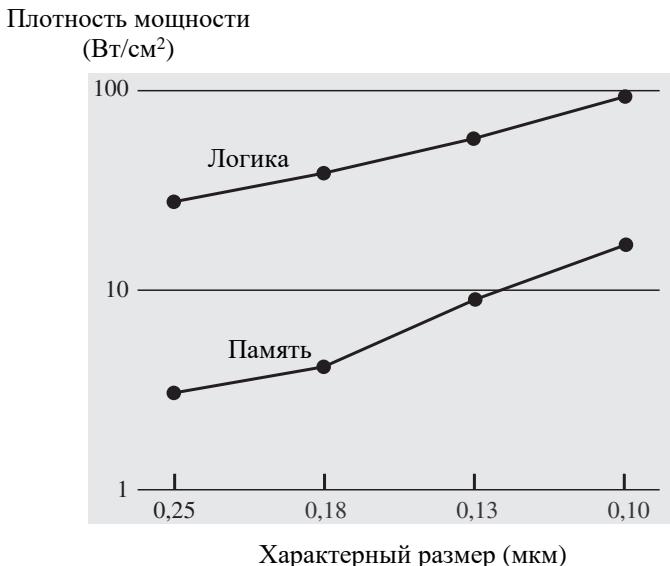
Существует также ряд проблем, связанных с разработкой и изготовлением компьютерных чипов. Увеличение сложности, необходимое для решения всех вопросов логики, связанных с очень длинными конвейерами, несколькими суперскалярными конвейерами и множественными банками регистров SMT означает, что все большее количество площади чипа занято не вычислениями, а координацией и логикой передачи сигналов. Это увеличивает сложность проектирования, изготовления и отладки чипов. Это одна из причин увеличения доли чипа процессора, отводимой более простой логике памяти. Другой причиной являются вопросы энергопотребления, обсуждаемые далее.

## Энергопотребление

Для поддержания тенденции повышения производительности с ростом количества транзисторов на чипе проектировщики прибегают ко все более и более сложным процессорам (конвейерной обработке, суперскалярности, SMT) и ко все более и более высоким тактовым частотам. К сожалению, требования к электропитанию растут с ростом плотности элементов на чипе и тактовой частоты в геометрической прогрессии.

Одним из способов контроля плотности мощности является использование большей площади чипа для кеш-памяти. Транзисторы памяти меньше по размеру и имеют плотность потребляемой мощности на порядок ниже, чем транзисторы логики (рис. В.3). Процент площади чипа, выделяемый для памяти, растет более чем на 50% быстрее роста плотности транзисторов на чипе.

Как использовать все эти транзисторы логики — ключевой вопрос проектирования. Как обсуждалось ранее, существуют ограничения эффективного использования таких методов, как суперскалярность и SMT. В общих чертах опыт последних десятилетий подытожен эмпирическим правилом, известным как правило Поллака (Pollack) [193], которое гласит, что повышение производительности примерно пропорционально квадратному корню от увеличения сложности. Другими словами, если вы удваиваете логику в ядре процессора, то это обеспечивает повышение производительности только на 40%. В принципе, использование множества ядер имеет потенциал обеспечения почти линейного улучшения производительности с увеличением количества ядер.

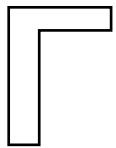


**Рис. В.3.** Энергопотребление процессоров

Соображения энергопотребления обеспечивают еще один мотив для перехода к многоядерной организации. Поскольку чип имеет такой огромный объем кеш-памяти, становится маловероятно, что какой-либо один поток выполнения сможет эффективно использовать всю эту память. Даже при использовании SMT многопоточность оказывается относительно ограниченной и потому не может полностью использовать гигантский кеш, в то время как ряд относительно независимых потоков или процессов имеют больше возможностей использовать кеш-память.

Tlgm: @it\_boooks

## ПРИЛОЖЕНИЕ



# ОБЪЕКТНО- ОРИЕНТИРОВАННОЕ ПРОЕКТИРОВАНИЕ

В ЭТОМ ПРИЛОЖЕНИИ...

### **Г.1. Мотивация**

### **Г.2. Объектно-ориентированные концепции**

Структура объектов

Классы объектов

Наследование

Полиморфизм

Интерфейсы

Включение

### **Г.3. Преимущества объектно-ориентированного подхода**

### **Г.4. CORBA**

### **Г.5. Дополнительные материалы**

Windows и некоторые другие современные операционные системы в значительной степени основаны на принципах объектно-ориентированного проектирования. В этом приложении представлен краткий обзор основных концепций этого подхода.

## Г.1. МОТИВАЦИЯ

Объектно-ориентированные концепции стали довольно популярными в области компьютерного программирования благодаря своему обещанию предоставления взаимозаменяемых, многоразовых, легко обновляемых и легко взаимосвязанных частей программного обеспечения. Совсем недавно к преимуществам объектной ориентированности обратились разработчики баз данных, в результате чего начинают появляться объектно-ориентированные системы управления базами данных (OODBMS). Разработчики операционных систем также осознали преимущества объектно-ориентированного подхода.

Объектно-ориентированное программирование и объектно-ориентированные системы управления базами данных — на самом деле разные вещи, но они разделяют одну ключевую концепцию: такое программное обеспечение или данные могут быть “упакованы в контейнеры”. Все складывается в один большой ящик, в котором могут быть коробки поменьше. В простейшей обычной программе один программный шаг соответствует одной инструкции; в объектно-ориентированном языке каждый шаг может представлять собой целый набор инструкций. Точно так же в объектно-ориентированной базе данных одна переменная может приравниваться к целому набору данных, а не к единственному элементу.

В табл. Г.1 перечислены некоторые ключевые термины объектно-ориентированного проектирования.

**Таблица Г.1. Ключевые термины объектно-ориентированного проектирования**

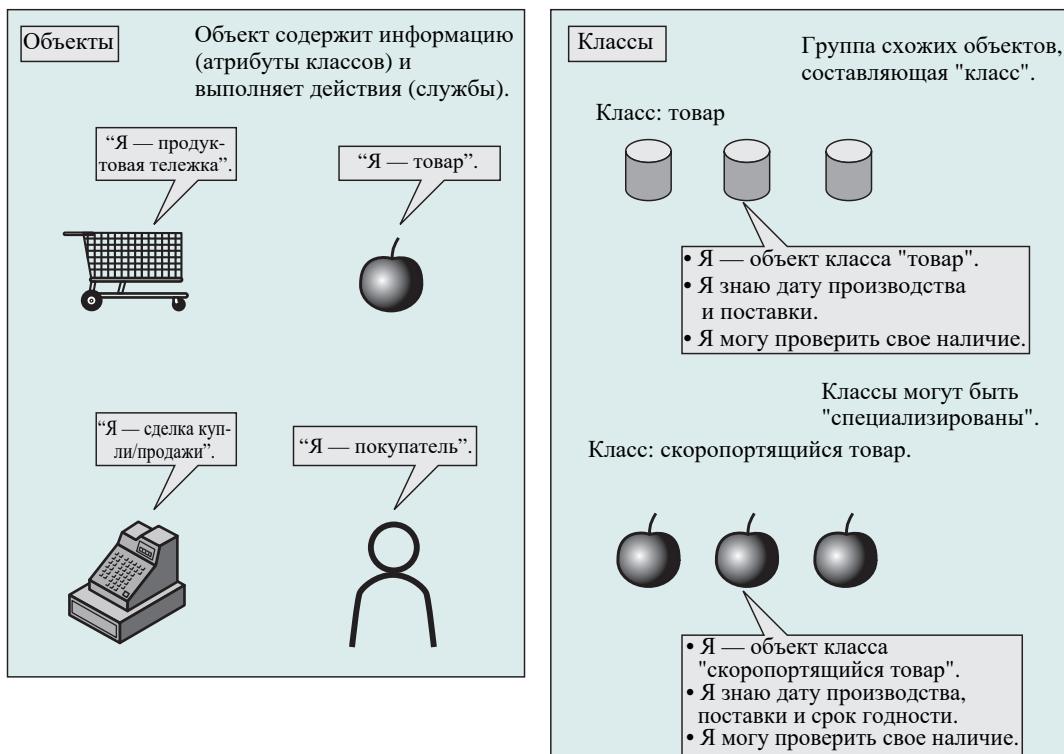
Термин	Определение
Атрибут	Переменные-данные, содержащиеся в объекте
Инкапсуляция	Отделение атрибутов и служб экземпляра объекта от внешней среды. Службы могут вызываться только по имени, а атрибуты — только через службы
Интерфейс	Описание, тесно связанное с классом объекта. Содержит определения методов (без реализаций) и константные значения. Не может быть инстанцирован как объект
Класс объекта	Именованное множество объектов с одинаковыми именами, наборами атрибутов и служб
Метод	Процедура, которая является частью объекта и может быть активирована извне объекта для выполнения определенных функций
Наследование	Связь между двумя классами объектов, в которой атрибуты и службы родительского класса приобретаются дочерним классом
Объект	Абстракция сущности реального мира
Полиморфизм	Означает существование нескольких объектов, которые используют одни и те же имена для служб и предоставляют один и тот же интерфейс для внешнего мира, но разные типы объектов
Служба	Функция, выполняющая операцию над объектом
Содержание	Отношения между двумя экземплярами объектов, в которых содержащий объект включает указатель на содержащийся объект
Сообщение	Средство взаимодействия объектов
Экземпляр объекта	Определенный член класса объекта с присвоенными атрибутам значениями

## Г.2. ОБЪЕКТНО-ОРИЕНТИРОВАННЫЕ КОНЦЕПЦИИ

Центральным понятием объектно-ориентированного дизайна является объект. Объект представляет собой отличный от других программный блок, содержащий набор связанных переменных (данных) и методов (процедур). В общем случае эти переменные и методы непосредственно за пределами объекта не видны. Вместо этого имеются четко определенные интерфейсы, которые позволяют другому программному обеспечению получать доступ к данным и процедурам.

Объект представляет некоторую сущность, будь то физическое лицо, концепция, модуль программного обеспечения или некоторый динамический объект, такой как соединение TCP. Значения переменных в объекте выражают информацию о том, что собой представляет объект. Методы включают в себя процедуры, выполнение которых влияет на значения в объект и, возможно, на представляемые им сущности.

На рис. Г.1 и Г.2 показаны ключевые объектно-ориентированные концепции.



**Рис. Г.1. Объекты**

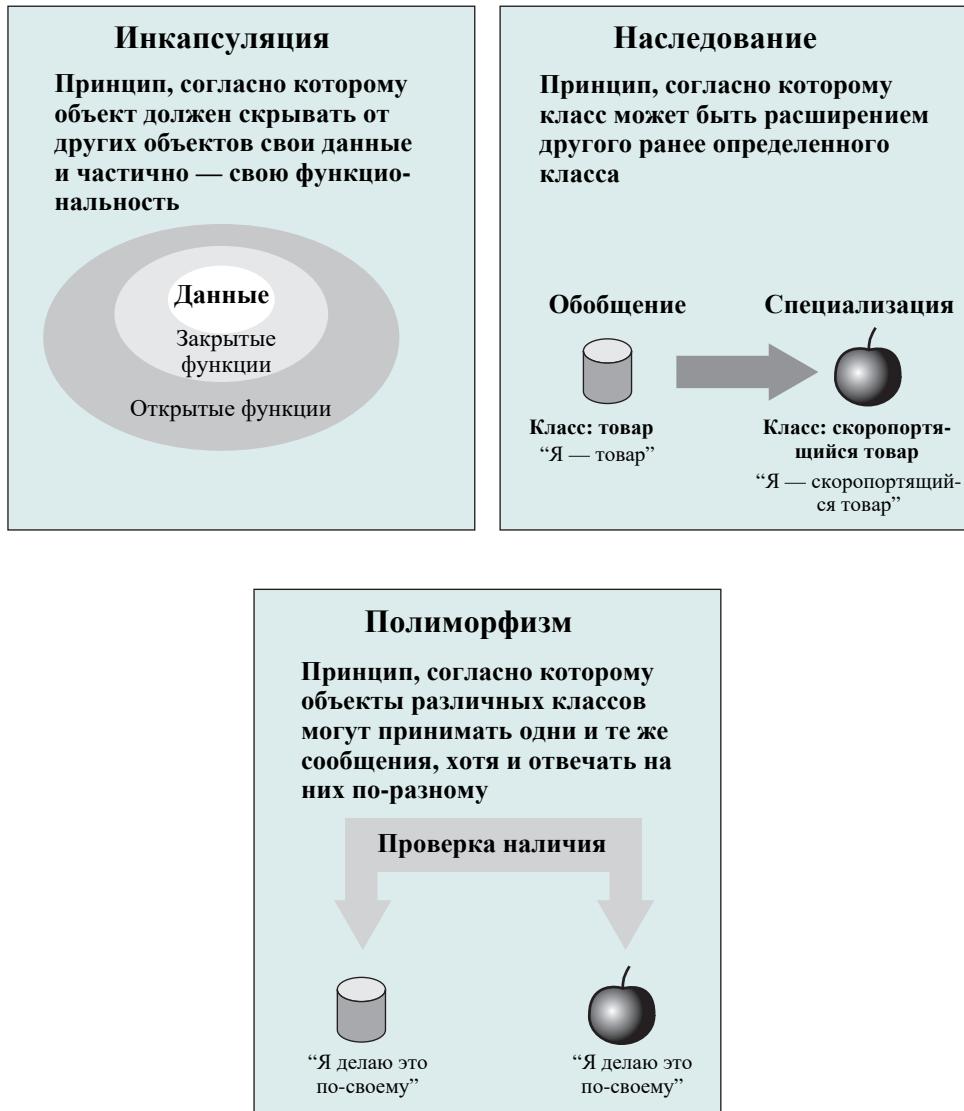


Рис. Г.2. Объектные концепции

## Структура объектов

Данные и процедуры, которые содержатся в объекте, обычно называют переменными и методами соответственно. Все, что известно объекту, можно выразить с помощью его переменных, а все, что он может выполнять, — с помощью его методов.

**Переменные** (variables) объекта, которые также называются **атрибутами** (attributes), обычно имеют вид простых скаляров или таблиц. Для каждой из переменных задан тип и, возможно, набор значений, которые может принимать эта переменная; переменная может быть определена как константа или как переменная (по соглашению термин *переменная* используется и для констант). Кроме того, могут быть наложены ограничения

на использование переменной определенными пользователями, классами пользователей или в определенных ситуациях.

**Методы** (methods) объекта — это процедуры, которые можно запускать извне для выполнения определенных функций. Метод может изменять состояние объекта, обновлять значения некоторых переменных или воздействовать на внешние ресурсы, к которым имеет доступ объект.

Объекты взаимодействуют между собой с помощью **сообщений** (messages). Сообщение содержит в себе имя объекта-отправителя, имя объекта-получателя, имя метода в объекте-получателе и прочие уточняющие параметры, необходимые для работы метода. Содержащийся в объекте метод можно вызвать только с помощью сообщения. Получить доступ к хранящимся в объекте данным можно только с помощью методов этого объекта. Таким образом, с помощью метода можно произвести нужное действие, получить доступ к переменным объекта или выполнить и то, и другое. Отправка сообщения локальному объекту равносильна вызову метода объекта. Для распределенных объектов пересылка сообщения означает именно то, что обычно под этим подразумевается.

Интерфейс объекта представляет собой набор открытых методов, поддерживаемых объектом. По интерфейсу нельзя судить о реализации; объекты разных классов могут иметь различные реализации одних и тех же интерфейсов.

Свойство объекта, состоящее в том, что его единственным связующим звеном с внешним миром являются сообщения, называется **инкапсуляцией** (encapsulation). Методы и переменные объекта являются инкапсулированными, и доступ к ним осуществляется только с помощью сообщений. Это свойство дает два преимущества.

1. Защищает переменные объекта от разрушения другими объектами. Эта защита может включать в себя защиту от несанкционированного доступа, а также защиту от таких проблем, возникающих при параллельном доступе, как взаимоблокировка и несогласованные значения.
2. Скрывает внутреннюю структуру объекта, поэтому взаимодействие с ним является сравнительно простым и стандартизованным. Более того, если внутренняя структура или процедуры объекта изменяются без изменения выполняемых ими внешних функций, это не влияет на другие объекты.

## Классы объектов

На практике обычно вещи одного типа представлены несколькими объектами. Например, если процесс представлен объектом, то в системе для каждого процесса будет присутствовать свой объект. Очевидно, что каждый такой объект нуждается в своем наборе переменных. Однако если методы объекта являются реентерабельными процедурами, то все похожие объекты могут совместно использовать одни и те же методы. Более того, может оказаться, что для каждого нового, но похожего на предыдущие, объекта неэффективно определять и методы, и переменные.

Чтобы избежать трудностей, нужно научиться различать класс объекта и экземпляр объекта. **Класс объекта** (object class) — это шаблон, по которому определяются методы и переменные, входящие в объект определенного вида. **Экземпляр объекта** (object instance) — это сам объект, включающий в себя характеристики того класса, в котором он определен. В экземпляре содержатся значения переменных, определенных в классе объектов. **Инстанцирование** (instantiation) — это процесс создания нового экземпляра объекта класса объекта.

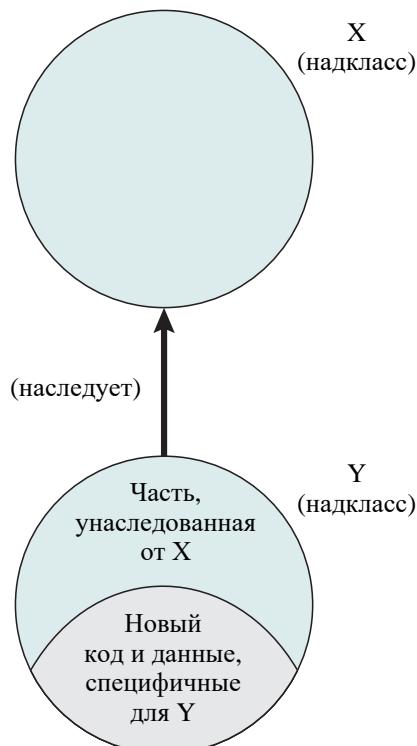
## Наследование

Благодаря существованию понятия класса объектов можно с минимальными усилиями создавать многие экземпляры объектов. Эта концепция стала еще мощнее благодаря механизму наследования [249].

Наследование позволяет определять новые классы объектов в терминах существующего класса. Новый класс (более низкого уровня), который называется **подклассом** (subclass) или **дочерним классом**, автоматически включает в себя определения методов и переменных исходного класса (более высокого уровня), который называется **надклассом**, **суперклассом** (superclass) или **родительским классом**. Подкласс может отличаться от своего надкласса по некоторым параметрам.

1. Подкласс может содержать в себе дополнительные методы и переменные, которых нет в надклассе этого подкласса.
2. В подклассе можно изменить определение любого метода или любой переменной, имеющейся в его надклассе; при этом новое определение используется с тем же именем.
3. Подклассы могут ограничивать методы или переменные, каким-либо образом унаследованные от соответствующего надкласса.

Эта концепция проиллюстрирована на рис. Г.3 из [138].



**Рис. Г.3.** Наследование

Механизм наследования является рекурсивным, что позволяет подклассу быть надклассом собственных подклассов. Таким образом, можно создать **иерархию наследования**. Концептуально иерархию наследования можно представлять себе как определение способа поиска методов и переменных. Получив сообщение с предписанием выполнить метод, который не задан в данном классе, объект осуществляет его поиск на других иерархических уровнях, пока не найдет нужный метод. Аналогично, если при выполнении метода происходит ссылка на переменную, которая не определена в данном классе, объект ищет переменную с таким именем на других иерархических уровнях.

## Полиморфизм

Полиморфизм — это мощная характеристика, позволяющая скрывать за общим интерфейсом различные реализации. В двух полиморфных по отношению друг к другу объектах используются одинаковые имена методов и представлены одинаковые интерфейсы по отношению к другим объектам. Можно определить несколько предназначенных для печати объектов, каждый для своего устройства вывода (например, `printDotmatrix` — для матричного принтера, `printLaser` — для лазерного принтера, `printScreen` — для вывода на экран и т.д.) или каждый для своего вида документа (например, `printText` — для текстовых документов, `printDrawing` — для рисунков, `printCompound` — для документов смешанного типа). Если в каждом таком объекте содержится метод под названием `print`, то любой документ можно распечатать, отправив сообщение `print` соответствующему объекту. При этом не имеет значения, как именно выполняется этот метод.

Интересно сравнить полиморфизм с методами обычного модульного программирования. Целью нисходящего модульного проектирования является разработка в рамках общей утилиты модулей более низких уровней с фиксированным по отношению к модулям более высоких уровней интерфейсом. Это позволяет различным модулям более высоких уровней вызывать один и тот же модуль низкого уровня. Если внутренняя структура модуля низкого уровня изменяется, не затрагивая при этом интерфейс, то это никак не влияет ни на один из модулей более высоких уровней, которые используют изменяемый модуль. Когда речь идет о полиморфизме, все происходит наоборот. Здесь имеется в виду способность объекта более высокого уровня вызывать с помощью сообщения в одном и том же формате различные объекты более низких уровней, выполняя, таким образом, подобные функции. При этом можно добавлять новые объекты низких уровней с минимальными изменениями в существующих объектах.

## Интерфейсы

Наследование интерфейсов позволяет объекту подкласса использовать функциональность суперкласса. Могут быть случаи, когда вы хотите определить подкласс, который имеет функциональность более чем одного суперкласса. Это может быть достигнуто путем наследования подкласса от нескольких суперклассов. Языком, который позволяет такое множественное наследование, является C++. Однако для простоты большинство современных объектно-ориентированных языков, включая Java, C# и Visual Basic .NET, ограничивают класс наследованием только от одного надкласса. Вместо этого, чтобы позволить классу заимствовать некоторые функциональные возможности из одного класса и другие функциональные возможности из совершенно другого класса, используются **интерфейсы**.

К сожалению, термин *интерфейс* используется в большей части литературы и как общий термин, и как имеющий определенное функциональное значение. Интерфейс, который мы обсуждаем здесь, представляет собой интерфейс прикладного программирования (API) для определенной функциональности. Он не определяет никакой реализации этого API. Синтаксис определения интерфейса обычно похож на определение класса, с тем отличием, что код методов не определен, имеются только имена методов, передаваемые аргументы и тип возвращаемого значения. Интерфейс может быть реализован классом. Этот подход работает во многом так же, как наследование. Если класс реализует интерфейс, он должен иметь свойства и методы интерфейса, определенные в классе. Реализуемые методы могут быть закодированы любым способом при условии, что имя, аргументы и тип возврата каждого метода из интерфейса идентичны определению в интерфейсе.

## Включение

Экземпляры объекта, которые содержат в себе другие объекты, называются **составными объектами** (composite objects). Включение может быть достигнуто путем использования в объекте указателя на другой объект. Преимущество составных объектов заключается в том, что они позволяют представлять сложные структуры. Например, объект, который входит в составной объект, сам может быть составным.

Обычно из составных объектов конструируются структуры, имеющие древовидную топологию, т.е. нельзя использовать никаких циклических ссылок, и каждый экземпляр дочернего объекта может иметь только один экземпляр родительского объекта.

Важно понять различие между иерархией наследования классов объектов и иерархией, возникающей в составных объектах. Они никак не связаны между собой. Наследование просто позволяет с минимальными усилиями определять различные типы объектов. Включение объектов позволяет создавать сложные структуры данных.

## Г.3. ПРЕИМУЩЕСТВА ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПОДХОДА

В [41] перечислены такие преимущества объектно-ориентированного подхода.

- **Лучшая организация сложных наследственных отношений.** Используя наследственность, можно эффективно определять взаимосвязанные понятия, ресурсы и другие объекты. С помощью включения можно конструировать произвольные структуры данных, отражающие поставленную задачу. Объектно-ориентированные языки программирования и структуры данных позволяют разработчикам описывать ресурсы и функции операционной системы таким способом, который отражает представление об этих ресурсах и функциях.
- **Снижение усилий, затрачиваемых на разработку, за счет повторного использования.** Повторное использование классов объектов, которые уже написаны, протестированы и опробованы другими, сокращает время, необходимое для разработки и тестирования.
- **Повышение степени расширяемости и облегчение поддержки систем.** Обычно на протяжении всего времени жизни продукта 65% его стоимости затрачивается

на поддержку этого продукта, включая его улучшение и устранение недостатков. Объектно-ориентированный подход сокращает этот показатель. Использование объектно-ориентированных программ помогает сократить число возможных взаимодействий между различными частями этих программ, обеспечивая тем самым минимальное воздействие, которое может оказаться изменение в реализации класса на остальную часть системы.

Эти преимущества способствуют тому, что разработка операционных систем ведется в направлении объектно-ориентированных систем. Объекты позволяют разработчикам изменять операционную систему так, чтобы она удовлетворяла новым требованиям, не нарушая ее целостности. Кроме того, объекты являются строительными блоками, использующимися при разработке распределенных систем. Благодаря тому что объекты взаимодействуют между собой с помощью сообщений, не имеет значения, находятся ли обменивающиеся сообщениями объекты в одной и той же системе или в двух разных системах, подключенных к одной сети. Данные, функции и потоки могут быть динамически назначены рабочим станциям и серверам. Благодаря всем перечисленным достоинствам объектно-ориентированного подхода растет его роль при разработке операционных систем, предназначенных для персональных компьютеров и рабочих станций.

## Г.4. CORBA

Как мы видели в этой книге, объектно-ориентированные концепции использовались для разработки и реализации ядер операционных систем, обеспечивая преимущества гибкости, управляемости и переносимости. Преимущества использования объектно-ориентированных методов расширяются с развитием распределенного программного обеспечения, включая распределенные операционные системы. Применение объектно-ориентированных методов проектирования и реализации для распределенного программного обеспечения называется распределенным объектным вычислением (*distributed object computing — DOC*).

Мотивацией для DOC служит возрастающая трудность в написании распределенного программного обеспечения: в то время как вычислительное и сетевое оборудование становится все меньше, быстрее и дешевле, программное обеспечение становится все больше, медленнее и дороже в разработке и обслуживании. Проблема распределенного программного обеспечения обусловлена двумя типами сложности.

- **Внутренне присущие сложности** возникают из-за фундаментальных проблем распределения. Главной среди них является обнаружение и восстановление сбоев сети и узлов, минимизация влияния задержки в линиях связи и определение оптимального разделения служебных компонентов и рабочей нагрузки среди компьютеров всей сети. Кроме того, достаточно сложным вопросом является параллельное программирование с его проблемами блокировки ресурсов и взаимоблокировок, а распределенные системы по самой сути являются параллельными.
- **Случайные сложности** возникают из-за ограничений, связанных с инструментами и методами, используемыми для создания распределенного программного обеспечения. Распространенным источником случайной сложности является широкое использование функционального дизайна, что приводит к нерасширяемым системам, которые невозможно использовать повторно.

DOC является многообещающим подходом к управлению сложностями обоих типов. Центральная часть подхода DOC — брокеры объектных запросов (ORB), которые выступают в качестве посредников для связи между локальными и удаленными объектами. Эти брокеры устраняют некоторые из утомительных, подверженных ошибкам и непереносимых аспектов проектирования и реализации распределенных приложений. В дополнение к брокерам требуется ряд соглашений и форматов для обмена сообщениями и определения интерфейса между приложениями и объектно-ориентированной инфраструктурой.

На рынке DOC существуют три основные конкурирующие технологии: архитектура группы управления объектами (object management group — OMG), именуемая Common Object Request Broker Architecture (CORBA — общая архитектура брокера объектных запросов); система удаленного вызова методов Java (remote method invocation — RMI); а также объектная модель распределенных компонентов Microsoft (distributed component object model — DCOM). CORBA является наиболее развитой и устоявшейся из всех трех технологий. Ряд лидеров отрасли, в том числе IBM, Sun, Netscape и Oracle, поддерживают CORBA, а Microsoft объявила, что будет связывать свою Windows-технологию DCOM только с CORBA. Остальная часть этого приложения дает краткий обзор CORBA.

В табл. Г.2 определены некоторые ключевые термины, используемые в CORBA. Основные особенности CORBA выглядят следующим образом (рис. Г.4).

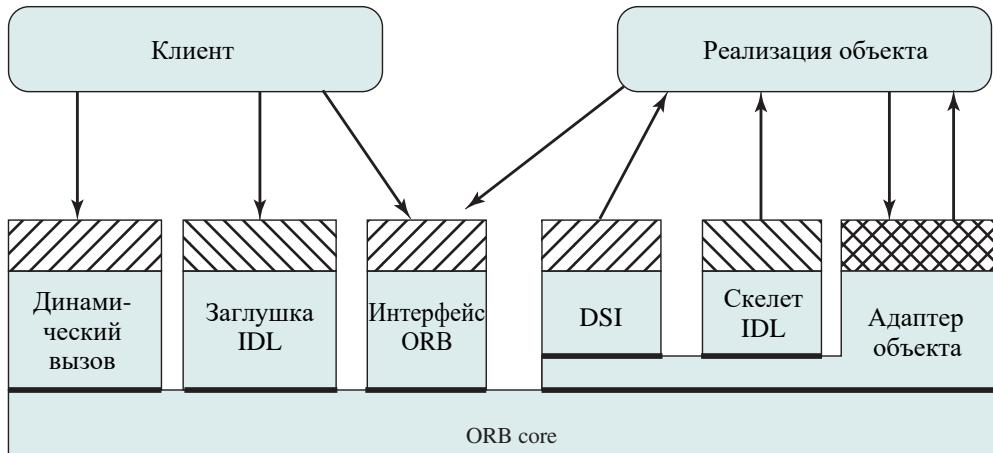
- **Клиенты.** Генерируют запросы и получают доступ к объектным службам с помощью различных механизмов, предоставляемых базовым ORB.
- **Реализации объектов.** Предоставляют службы, запрашиваемые различными клиентами в распределенной системе. Одно из преимуществ архитектуры CORBA заключается в том, что как клиентские, так и объектные реализации могут быть написаны на любом количестве языков программирования и при этом предоставлять полный спектр необходимых услуг.
- **Ядро ORB.** Отвечает за связь между объектами. ORB находит объект в сети, доставляет запросы к объекту, активирует объект (если он еще не активен) и возвращает любое сообщение обратно отправителю. Ядро ORB обеспечивает прозрачность доступа, поскольку при вызове локального или удаленного метода программисты используют один и тот же метод с одинаковыми параметрами. Ядро ORB также обеспечивает прозрачность местоположения — программистам не нужно указывать местоположение объекта.
- **Интерфейс объекта.** Определяет операции и типы, поддерживаемые объектом, и, таким образом, определяет запросы, которые могут быть сделаны к объекту. Интерфейсы CORBA аналогичны классам в C++ и интерфейсам в Java. В отличие от классов C++, интерфейс CORBA определяет методы, их параметры и возвращаемые значения, но ничего не говорит об их реализации. Два объекта одного и того же класса C++ имеют одинаковую реализацию своих методов.
- **Язык определения интерфейса (IDL).** Это язык, используемый для определения объектов. Вот пример определения интерфейса IDL:

```
//OMG IDL
interface Factory
{ Object create ();
};
```

Здесь определяется интерфейс с именем `Factory`, который поддерживает одну операцию, `create`. Операция `create` не принимает параметров и возвращает ссылку на объект типа `Object`. Для данной ссылки типа `Factory` клиент может вызвать ее для создания нового объекта CORBA. IDL представляет собой язык, не зависящий от языка программирования, и по этой причине клиент не вызывает непосредственно никакие операции над объектом. Для этого требуется отображение на язык программирования клиента. Возможно также, что сервер и клиент запрограммированы на разных языках. Использование языка определения является способом борьбы с гетерогенной обработкой с использованием нескольких языков и платформ. Таким образом, IDL обеспечивает независимость от платформы.

**Таблица Г.2. Ключевые концепции распределенной системы CORBA**

Концепция CORBA	Определение
Interface Definition Language (IDL)	Язык определения интерфейсов в CORBA
Вызов	Процесс, отправляющий запрос
Запрос	Сообщение, пересылаемое между клиентским приложением и приложением сервера
Интерфейс	Описывает, как ведут себя экземпляры объектов, в частности какие операции корректны для тех или иных объектов
Исключение	Содержит информацию, которая указывает, был ли запрос успешно выполнен
Клиентское приложение	Выполняет запросы к серверу для выполнения операций над объектами. Клиентское приложение использует одно или несколько определений интерфейса, которые описывают объекты и операции, которые клиент может запросить. Клиентское приложение при выполнении запросов использует не объекты, а ссылки на них
Метод	Код сервера, выполняющий работу, связанную с операцией. Методы содержатся в реализациях
Объект	Представляет человека, место, предмет или часть программного обеспечения. Объект может иметь выполняемые над ним операции, такие как, например, операция повышения сотрудника в должности
Операция	Действие над экземпляром объекта, которое клиент может запросить у сервера
Определение интерфейса	Описывает операции, доступные для определенного типа объектов
Реализация	Определяет и содержит один или несколько методов, выполняющих работу, связанную с операциями над объектом. Сервер может иметь одну или несколько реализаций
Серверное приложение	Содержит одну или несколько реализаций объектов и их операций
Ссылка на объект	Идентификатор экземпляра объекта
Экземпляр объекта	Появление одного конкретного вида объекта



- |  |                                       |
|--|---------------------------------------|
|  Однако для всех ORB                         | ORB Брокер запросов объектов          |
|  Заглушки и скелеты, зависящие от интерфейса | IDL Язык определения интерфейса       |
|  Может быть несколько адаптеров объектов     | DSPI Динамический скелетный интерфейс |
|  Закрытые интерфейсы ORB                     |                                       |

Рис. Г.4. Архитектура CORBA

- **Создание языковой привязки.** Компиляторы IDL отображают один файл IDL на другие языки программирования, которые могут как быть, так и не быть объектно-ориентированными, такие как Java, Smalltalk, Ada, C, C++ или COBOL. Это отображение включает в себя определение специфичных для языка типов данных и процедур интерфейсов доступа к объектам служб, интерфейса заглушки клиента IDL, скелета IDL, адаптеров объектов, динамического скелетного интерфейса и непосредственного интерфейса ORB. Обычно во время компиляции клиенты имеют доступ к информации об интерфейсе объекта и используют клиентские заглушки для выполнения статических вызовов; в некоторых случаях клиенты такой информацией не обладают и вынуждены прибегать к динамическим вызовам.
- **Заглушка IDL.** Выполняет вызовы ядра ORB от имени клиентского приложения. IDL-заглушки предоставляют набор механизмов, которые абстрагируют основные функции ORB в механизмы удаленного вызова процедур, которые могут использоваться приложениями конечных клиентов. Эти заглушки представляют комбинацию ORB и реализации удаленного объекта так, как будто они привязаны к одному и тому же процессу. В большинстве случаев компиляторы IDL генерируют библиотеки интерфейсов, которые завершают интерфейс между клиентом и реализацией объекта.

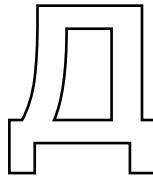
- **Скелет IDL.** Предоставляет код, который вызывает определенные методы сервера. Статические скелеты IDL представляют собой дополнения на стороне сервера к заглушкам IDL на стороне клиента. Они включают связывания между ядром ORB и реализациями объекта, которые завершают связь между реализациями клиента и объекта.
- **Динамический вызов.** Используя интерфейс динамического вызова (DII), клиентское приложение может выполнять запросы любого объекта, не имея информации об интерфейсах объекта времени компиляции. Детальная информация об интерфейсе заполняется путем консультации с хранилищем интерфейса и/или другими источниками времени выполнения. DII позволяет клиенту выдавать односторонние команды (для которых нет ответа).
- **Динамический каркасный интерфейс** (*dynamic skeleton interface — DSI*). Подобно взаимоотношениям между заглушками IDL и статическими скелетами IDL, DSI обеспечивает динамическую диспетчеризацию объектов. Эквивалент динамического вызова на стороне сервера.
- **Адаптер объекта.** Системный компонент CORBA, предоставляемый поставщиками CORBA для решения общих задач, связанных с ORB, таких как активация объектов и активирующие реализации. Адаптер принимает эти общие задачи и связывает их с конкретными реализациями и методами на сервере.

## Г.5. ДОПОЛНИТЕЛЬНЫЕ МАТЕРИАЛЫ

[138] представляет собой хороший обзор объектно-ориентированных концепций; [247] — описание объектно-ориентированного программирования. Интересная перспектива объектно-ориентированных концепций представлена в [239]. Обзор CORBA имеется в [262].



## ПРИЛОЖЕНИЕ



# ЗАКОН АМДАЛА

При рассмотрении производительности системы разработчики компьютерных систем ищут способы повышения производительности путем улучшения технологий или изменения проектов. Среди указанных способов — использование параллельных процессоров, иерархии кеш-памяти, а также ускорение доступа к памяти и повышение скорости передачи ввода-вывода с помощью технологических усовершенствований. Во всех этих случаях важно отметить, что ускорение в каком-то одном аспекте технологии или проектировании не приводит к соответствующему повышению общей производительности. Это ограничение кратко выражается законом Амдала.

Закон Амдала был впервые предложен Джином Амдалом в 1967 году [2] и касается потенциального ускорения многопроцессорной программы по сравнению с однопроцессорной. Рассмотрим программу, работающую на одном процессоре, такую, что  $(1 - f)$  часть времени выполнения представляет собой код, который по самой своей природе является последовательным, а часть  $f$  включает код, который можно до бесконечности распараллеливать без затрат на планирование. Пусть также  $T$  представляет собой общее время выполнения программы с использованием одного процессора. Тогда ускорение при максимальном распараллеливании для  $N$  процессоров выглядит следующим образом:

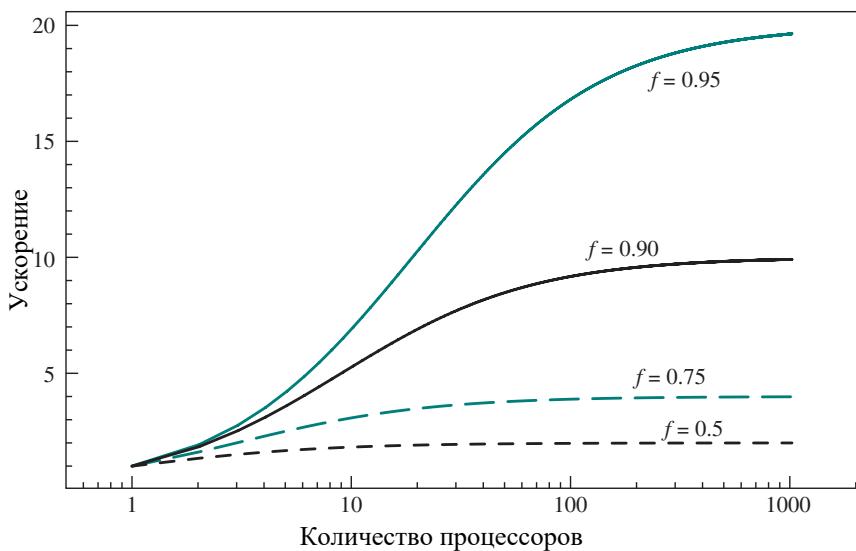
$$\text{Ускорение} = \frac{\text{Время выполнения программы на одном процессоре}}{\text{Время выполнения программы на } N \text{ параллельных процессорах}} = \\ = \frac{T(1-f) + Tf}{T(1-f) + \frac{Tf}{N}} = \frac{1}{(1-f) + \frac{f}{N}}$$

Это уравнение проиллюстрировано на рис. Д.1. Из уравнения и графиков можно сделать два важных вывода.

1. Когда  $f$  мало, использование параллельных процессоров малоэффективно.
2. Когда  $N$  приближается к бесконечности, ускорение ограничено значением  $1/(1-f)$ , что приводит к уменьшению эффективности в пересчете на один процессор при использовании большего количества процессоров.

Эти выводы слишком пессимистичны, как утверждается в [97]. Например, сервер может поддерживать несколько потоков или несколько заданий для обработки нескольких клиентов и выполнять потоки или задания параллельно, до предельной степени, соответствующей количеству процессоров. Многие приложения баз данных требуют вычислений над большими объемами данных, которые можно разделить на несколько параллельных заданий. Тем не менее закон Амдала иллюстрирует проблемы, стоящие перед промышленностью в области развития многоядерных машин с постоянно растущим числом ядер: программное обеспечение, работающее на таких машинах, должно быть

адаптировано к высокопараллельной среде выполнения, чтобы использовать всю мощь параллельной обработки.



**Рис. Д.1.** Закон Амдала для многопроцессорных систем

Закон Амдала может быть обобщен для оценки любого проектного или технического улучшения в компьютерной системе. Рассмотрим произвольное улучшение возможностей системы, которое приводит к ускорению. Ускорение может быть выражено следующим образом:

$$\begin{aligned} \text{Ускорение} &= \frac{\text{Производительность после усовершенствования}}{\text{Производительность до усовершенствования}} = \\ &= \frac{\text{Время работы программы до усовершенствования}}{\text{Время работы программы после усовершенствования}} \end{aligned}$$

Предположим, что некоторая возможность системы используется (до улучшения) долю времени работы, равную  $f$ , а ускорение этой возможности после усовершенствования равно  $SU_f$ . Тогда общее ускорение системы равно

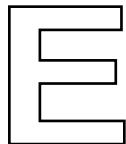
$$\text{Ускорение} = \frac{1}{(1-f) + \frac{f}{SU_f}}$$

Например, предположим, что задача широко использует операции с плавающей точкой, на которые затрачивается 40% времени. С новым оборудованием модуль с плавающей запятой ускоряется в  $K$  раз. Тогда общее ускорение составляет

$$\text{Ускорение} = \frac{1}{0,6 + \frac{0,4}{K}}$$

Итак, вне зависимости от  $K$ , максимальное ускорение составляет 1,67 раза.

## ПРИЛОЖЕНИЕ



# ХЕШ-ТАБЛИЦЫ

Рассмотрим следующую проблему. Множество из  $N$  элементов должно храниться в таблице. Каждый элемент состоит из метки и некоторой дополнительной информации, которую можно называть значением элемента. При этом хотелось бы иметь возможность выполнять ряд обычных операций над множествами, таких как вставка, удаление и поиск элемента по метке.

Если метки предметов являются числовыми в диапазоне от 0 до  $M-1$  включительно, то простейшим решением будет использование таблицы длиной  $M$ . Элемент с меткой  $i$  будет вставлен в таблицу в местоположении  $i$ . Если элементы имеют фиксированный размер, поиск в таблице тривиален и включает в себя индексирование на основе числового значения метки элемента. Более того, нет необходимости хранить метку элемента в таблице, потому что она определяется положением элемента. Такая таблица называется **таблицей прямого доступа**.

Если же метки не числовые, то все еще можно использовать подход прямого доступа. Обозначим элементы как  $A[1], \dots, A[N]$ . Каждый элемент  $A[i]$  состоит из метки, или ключа,  $k_i$  и значения  $v_i$ . Определим функцию отображения  $I(k)$  так, чтобы  $I(k)$  принимала значения от 1 до  $M$  для всех ключей и  $I(k_i) \neq I(k_j)$  — для любых  $i$  и  $j$ . В этом случае также может использоваться таблица прямого доступа, длина которой равна  $M$ .

С этими схемами возникает трудность, если  $M$  намного больше, чем  $N$ . В этом случае велика доля неиспользуемых записей в таблице, что приводит к неэффективному использованию памяти. Альтернативой может быть использование таблицы длиной  $N$  для хранения  $N$  элементов (метка плюс значение) в  $N$  записях таблицы. В такой схеме используемый объем памяти сведен к минимуму, но теперь поиск в таблице требует дополнительной работы. Есть несколько вариантов.

- **Последовательный поиск.** Этот метод грубой силы отнимает много времени для больших таблиц.
- **Ассоциативный поиск.** При наличии необходимого оборудования все элементы в таблице могут просматриваться одновременно. Этот подход не имеет общего назначения и не может быть применен ко всем интересующим нас таблицам.
- **Бинарный поиск.** Если метки (или их числовое отображение) расположены в таблице в порядке возрастания, то бинарный поиск выполняется намного быстрее, чем последовательный (табл. Е.1) и не требует специального оборудования.

**Таблица Е.1. Средняя продолжительность поиска одного из  $N$  элементов в таблице длиной  $M$**

Метод	Продолжительность поиска
Прямое обращение	1
Последовательный поиск	$(M + 1)/2$
Бинарный поиск	$\log_2 M$
Линейное хеширование	$\frac{2 - N/M}{2 - 2^N/M}$
Хеширование (переполнение с цепочками)	$1 + \frac{N - 1}{2M}$

Бинарный метод выглядит для поиска в таблице многообещающе. Основной его недостаток заключается в том, что добавление новых элементов обычно оказывается непростым процессом и требует переупорядочения записей. Поэтому бинарный поиск обычно используется только для статических таблиц, которые редко меняются.

Мы хотели бы избежать излишнего использования памяти при простом подходе с непосредственным обращением к элементам и расходов на обработку в перечисленных ранее альтернативных вариантах. Наиболее часто для достижения компромисса используется метод хеширования. Хеширование, которое было разработано в 1950-х годах, просто в реализации и имеет два преимущества. Во-первых, обеспечивается возможность поиска “в одно касание”, как при непосредственном доступе. Во-вторых, вставки и удаления могут обрабатываться без дополнительной сложной обработки.

Хеш-функция может быть определена следующим образом. Предположим, что в хеш-таблице длиной  $M$  должно храниться до  $N$  элементов, где  $M$  не меньше  $N$ , а ненамного его превышает. Чтобы вставить элемент в таблицу, выполните следующие действия.

11. Преобразуйте метку элемента в почти случайное число  $n$  между 0 и  $M-1$ . Например, если метка числовая, популярная функция отображения представляет собой остаток от деления ее значения на  $M$ .
12. Используйте этот остаток в качестве индекса в хеш-таблице.
  - а. Если соответствующая запись в таблице пуста, сохраните элемент (метку и значение) в этой записи.
  - б. Если запись уже занята, сохраните элемент в области переполнения (этот вопрос рассматривается ниже).

Чтобы выполнить поиск в таблице элемента, метка которого известна, выполните следующие действия.

11. Преобразуйте метку элемента в почти случайное число  $n$  между 0 и  $M-1$ , используя ту же функцию отображения, что и для вставки.

## L2. Используйте $n$ в качестве индекса в хеш-таблице.

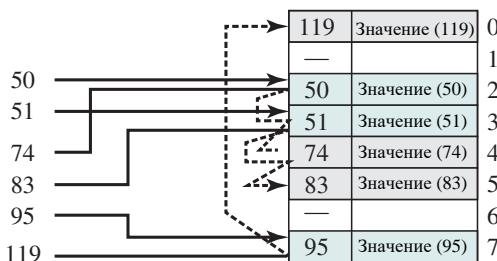
- Если соответствующая запись в таблице пуста, такой элемент ранее в таблице не сохранялся.
- Если запись уже занята и метка элемента соответствует искомой, элемент найден.
- Если запись уже занята, но метка элемента не соответствует искомой, следует продолжить поиск в области переполнения.

Схемы хеширования различаются способом обработки переполнения. Один распространенный метод известен как метод линейного хеширования и обычно используется в компиляторах. При этом подходе правило L2.b превращается в

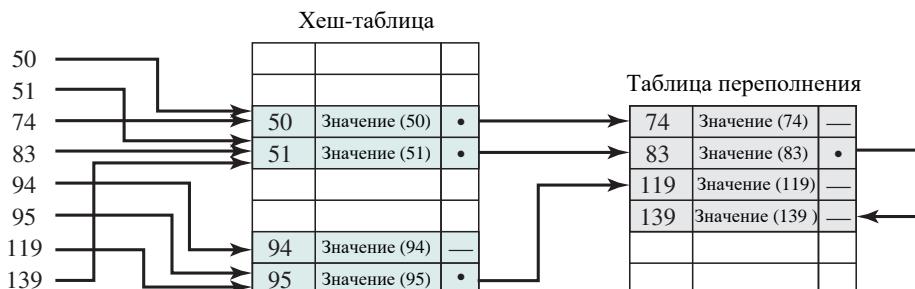
**I2.6.** Если запись уже занята, установите  $n = n + 1 \pmod M$  и вернитесь к шагу I2.a.

Правило L2.b изменяется соответствующим образом.

Пример этого подхода показан на рис. Е.1, а. В этом случае метки сохраняемых элементов представляют собой числовые значения, а хеш-таблица имеет восемь позиций ( $M=8$ ). Функция отображения заключается в получении остатка от деления на 8. На рисунке предполагается, что элементы были вставлены в возрастающем числовом порядке, хотя необходимости в этом нет.



а) Линейное хеширование



б) Переполнение с цепочками

**Рис. Е.1.** Хеширование

Таким образом, элементы 50 и 51 отображаются на позиции 2 и 3 соответственно, и поскольку последние пусты, элементы вставляются в указанные места. Элемент 74 также отображается на позицию 2, но поскольку она не пуста, исследуется позиция 3. Она также занята, поэтому в конечном итоге используется позиция 4.

Определить среднюю продолжительность поиска элемента в открытой хеш-таблице нелегко из-за эффекта кластеризации. Примерная формула была получена Шеем (Schay) и Спрутом (Spruth) [241]:

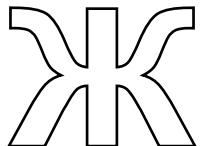
$$\text{Средняя длина поиска} = \frac{2 - \frac{N}{M}}{2 - 2 \frac{N}{M}}$$

Обратите внимание, что результат зависит не от размера таблицы, а только от того, насколько она заполнена. Удивительно, но даже если таблица заполнена на 80%, средняя длина поиска остается равной примерно 3.

Несмотря на это, поиск длиной 3 может считаться длинным; кроме того, таблица линейного хеширования имеет дополнительную проблему — в ней нелегко удалять элементы. Более привлекательный подход, который обеспечивает более короткую длину поиска (см. табл. Е.1) и позволяет выполнять удаление и дополнения, — это подход **переполнения с цепочками**. Этот метод показан на рис. Е.1, б. В этом случае имеется отдельная таблица, в которую вставляются записи, вызвавшие переполнение. Эта таблица включает цепочки записей, связанных с позициями в хеш-таблице. Средняя длина поиска в предположении случайного распределения данных при этом составляет

$$\text{Средняя длина поиска} = 1 + \frac{N-1}{2M}$$

Для больших значений  $N$  и  $M$  это значение приближается к величине 1,5 для  $N = M$ . Таким образом, этот метод обеспечивает компактное хранение при быстром поиске.



## ПРИЛОЖЕНИЕ

---

# ВРЕМЯ ОТКЛИКА

Время отклика — это время реакции системы на данный ввод. В случае интерактивных транзакций оно может быть определено как время между последним нажатием клавиши пользователем и началом отображения результата компьютером. Для иных типов приложений требуется немного другое определение. В общем случае это время, необходимое системе для ответа на запрос на выполнение конкретной задачи.

В идеале хотелось бы, чтобы время отклика для любого приложения было как можно более коротким. Тем не менее почти всегда более короткое время отклика требует больших затрат. Основные источники затрат следующие.

- **Мощность компьютера.** Чем быстрее процессор, тем короче время отклика. Конечно, увеличение вычислительной мощности означает увеличение стоимости системы.
- **Требования конкуренции.** Получение малого времени отклика на одни процессы может приводить к большому времени отклика у других.

Таким образом, значение каждого уровня времени отклика должно оцениваться с учетом стоимости достижения этого времени отклика.

В табл. Ж.1 из [163] перечислены шесть основных диапазонов времени отклика. Сложности проектирования возникают, когда требуется время отклика менее 1 с. Требование такого времени отклика генерируется системой, которая управляет или иным способом взаимодействует с некоторой текущей внешней деятельностью, такой как сборочная линия. Здесь требования просты. Когда мы рассматриваем взаимодействие человека с компьютером, например, в приложении для ввода данных, то находимся в области диалогового времени отклика. В этом случае все еще требуется короткое время отклика, но при этом достаточно сложно оценить, какое именно время является приемлемым.

То, что быстрое время отклика является ключом к производительности в интерактивных приложениях, было подтверждено в ряде исследований [98; 227; 256]. В них показано, что когда компьютер и пользователь взаимодействуют в темпе, который гарантирует отсутствие ожидания у каждой из сторон, то производительность значительно возрастает; соответственно, стоимость выполняемых на компьютере работ падает, а качество имеет тенденцию к улучшению. Общепризнанно, что относительно медленный (до 2 с) отклик был приемлем для большинства интерактивных приложений, потому что человек думал о следующей задаче. Однако в настоящее время выяснилось, что производительность с уменьшением времени отклика увеличивается.

**Таблица Ж.1. Диапазоны времени отклика****Более 15 с**

Это время исключает диалоговое взаимодействие. Для определенных типов приложений пользователи с определенными типами характеров могут позволить нахождение в терминале более 15 с в ожидании ответа на один простой запрос. Тем не менее для занятого человека ожидание более 15 с кажется невыносимым. Если такие задержки реальны, система должна быть спроектирована так, чтобы пользователь мог перейти к другим действиям и запросить ответ через некоторое время

**Более 4 с**

Как правило, такое время отклика слишком длинное для разговора, требующего от оператора сохранять информацию в краткосрочной памяти (памяти оператора, а не компьютера!). Такие задержки будут тормозить решение проблем и приводить к разочарованиям при вводе данных. Тем не менее после крупного события, такого как конец сделки, задержки от 4 до 15 с могут быть вполне допустимыми

**От 2 до 4 с**

Задержка, превышающая 2 с, может препятствовать работе с терминалом, требующей высокого уровня концентрации. Ожидание 2–4 с в терминале может показаться невероятно долгим, когда пользователь поглощен работой и эмоционально стремится завершить то, что делает. Задержка в этом диапазоне также может быть приемлемой после крупного события

**Менее 2 с**

Когда пользователь терминала должен помнить информацию на протяжении нескольких ответов, время ответа должно быть коротким. Чем более подробная информация запоминается, тем больше потребность в откликах менее чем через 2 с. Для сложных действий в терминале 2 с представляют собой важный предел времени отклика

**Менее секунды**

Определенные разновидности интенсивной работы, особенно с графическими приложениями, требуют очень короткого времени отклика, чтобы поддерживать интерес и внимание пользователя в течение длительного времени

**Десятые доли секунды**

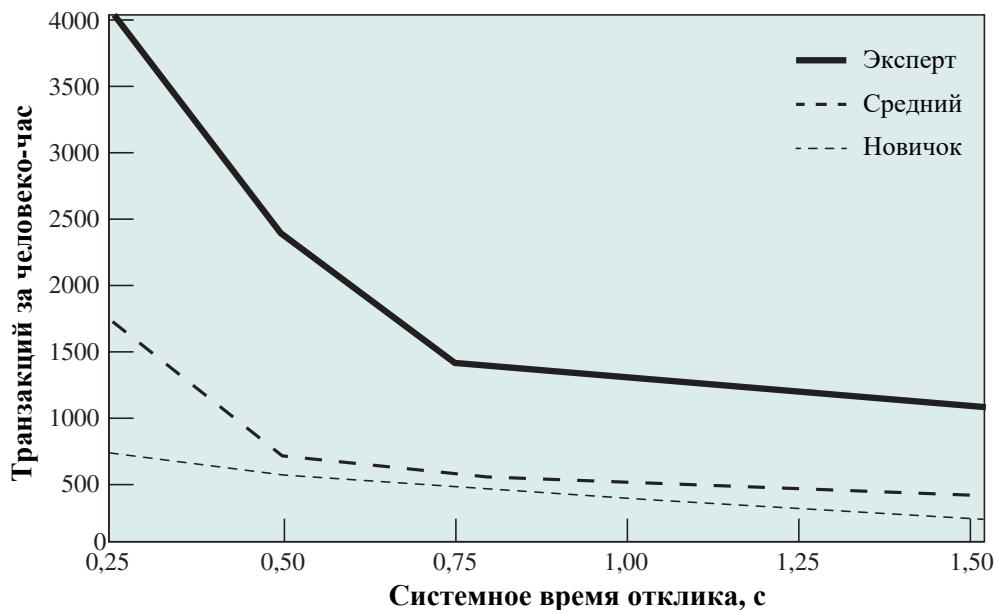
Отклик на нажатие клавиши и отображение символа на экране или щелчок мышью на экранном объекте должен быть почти мгновенным — менее 0,1 с после действия. Особенно быстрого отклика требует взаимодействие с мышью

Имеющиеся результаты по времени отклика получены из анализа онлайн-транзакций. Транзакция состоит из команды пользователя, переданной из терминала, и ответа системы. Это основная единица работы для пользователей онлайн-систем. Ее можно разделить на два временных промежутка.

- **Время ответа пользователя.** Промежуток времени между моментом, когда пользователь полностью получает ответ на одну команду и вводит следующую. Этот промежуток часто рассматривается как время размышлений.

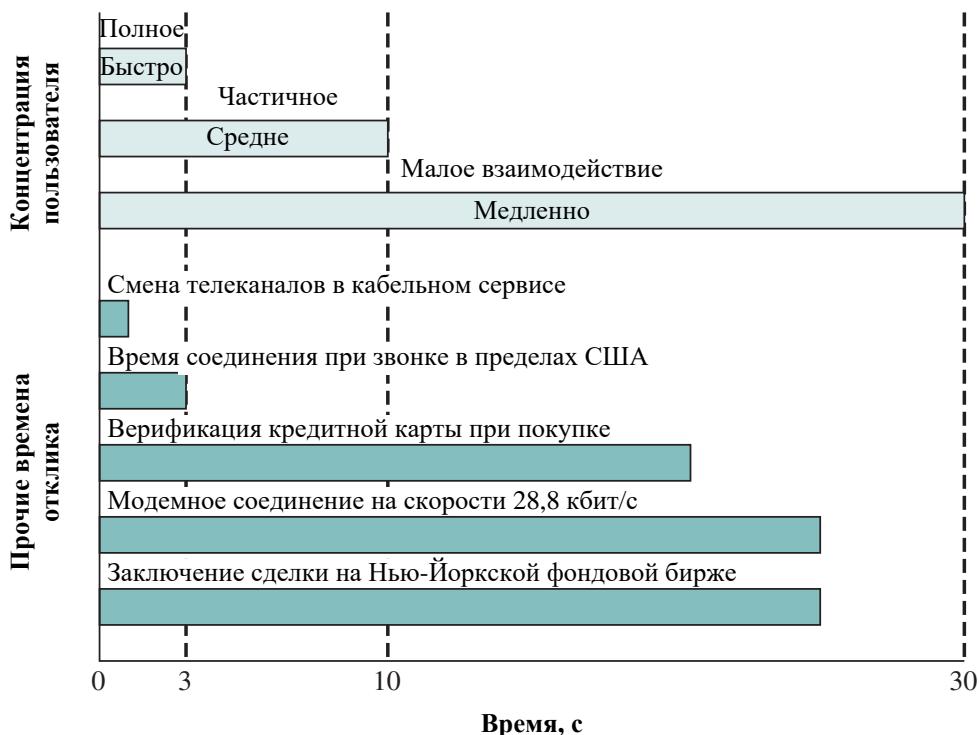
- **Время отклика системы.** Промежуток времени между моментом, когда пользователь вводит команду, и моментом полного отображения ответа на терминале.

В качестве примера эффекта уменьшения времени отклика системы на рис. Ж.1 показаны результаты исследования, проведенного с использованием графических программ проектирования микросхем и плат интегральных схем [236]. Каждая транзакция состоит из команды инженера-оператора, которая некоторым образом изменяет графическое изображение, отображаемое на экране. Результаты показывают, что скорость транзакций увеличивается с уменьшением времени отклика, причем наблюдается очень резкое возрастание, когда время отклика системы опускается ниже 1 с. При снижении времени отклика системы снижается и время отклика пользователя. Это связано с эффектами кратковременной памяти и человеческого внимания.



**Рис. Ж.1.** Изучение времени отклика в высокофункциональной графической системе

Другая область, в которой время отклика критично, — использование веба либо через Интернет, либо через корпоративную интрасеть. Время, необходимое для появления типичной веб-страницы на экране пользователя, сильно различается. Время отклика может быть измерено на основе уровня вовлеченности пользователей в сеанс; в частности, системы с малым временем отклика привлекают большее внимание пользователей. В исследованиях [217; 219; 220], результаты которых показаны на рис. Ж.2, веб-системы с временем отклика 3 с или менее поддерживают высокий уровень внимания пользователя. При времени отклика от 3 до 10 с концентрация пользователя несколько нарушается, а время отклика выше 10 с полностью обескураживает пользователя, который может просто прервать работу с сайтом. Другие исследования времени отклика в Интернете в целом подтверждают эти результаты [22].



**Рис. Ж.2.** Требования ко времени отклика

# ПРИЛОЖЕНИЕ 3

## Концепции теории массового обслуживания

В ЭТОМ ПРИЛОЖЕНИИ...

- 3.1. Зачем нужна теория массового обслуживания**
- 3.2. Очередь в случае одного сервера**
- 3.3. Многоканальная очередь**
- 3.4. Пуассонова скорость поступления**

В ряде глав этой книги используются результаты теории массового обслуживания. В главе 21, “Анализ очередей”, содержится подробный анализ очередей. Однако для понимания материала данной книги достаточно краткого обзора в этом приложении. Здесь мы даем краткое определение систем массового обслуживания и определения ключевых терминов.

### 3.1. ЗАЧЕМ НУЖНА ТЕОРИЯ МАССОВОГО ОБСЛУЖИВАНИЯ

Часто на основе информации о существующей нагрузке требуется делать прогнозы о расчетной производительности новой среды. При этом возможен ряд подходов.

1. Анализ *post factum* на основе реальных значений.
2. Простая экстраполяция, распространяющая имеющийся опыт на ожидаемую будущую среду.
3. Разработка аналитической модели, основанной на теории массового обслуживания.
4. Программирование и применение имитационного моделирования.

Вариант 1 вариантом не является, по сути, предлагая подождать и посмотреть, что получится. Вариант 2 выглядит более перспективным. Впрочем, аналитик может утверждать, что прогнозировать спрос с какой-либо степенью определенности невозможно, а потому бессмысленно пытаться выполнить какую-то точную процедуру моделирования. Скорее, приблизительный прогноз даст какие-то приблизительные оценки. Проблема в том, что при таком подходе поведение большинства систем при изменяющейся нагрузке оказывается не таким, которого можно было бы интуитивно ожидать. Если есть среда с совместно используемыми средствами (например, сеть, линия передачи или система с разделением времени), то производительность такой системы обычно реагирует на увеличение спроса экспоненциально.

Типичный пример приведен на рис. 3.1. Верхняя кривая показывает, что обычно происходит со временем отклика пользователя при совместно используемом объекте по мере увеличения нагрузки на этот объект. Нагрузка выражается в виде доли емкости. Таким образом, если мы имеем дело с маршрутизатором, который способен обрабатывать и пересыпать 1000 пакетов в секунду, то нагрузка 0,5 представляет собой скорость поступления 500 пакетов в секунду, а время отклика представляет собой количество времени, необходимое для повторной передачи любого входящего пакета. Нижняя линия представляет собой простую экстраполяцию, основанную на знании поведения системы до нагрузки 0,5. Обратите внимание, что хотя при простой экстраполяции все выглядит радужно, фактически система перестанет работать при нагрузке от 0,8 до 0,9.

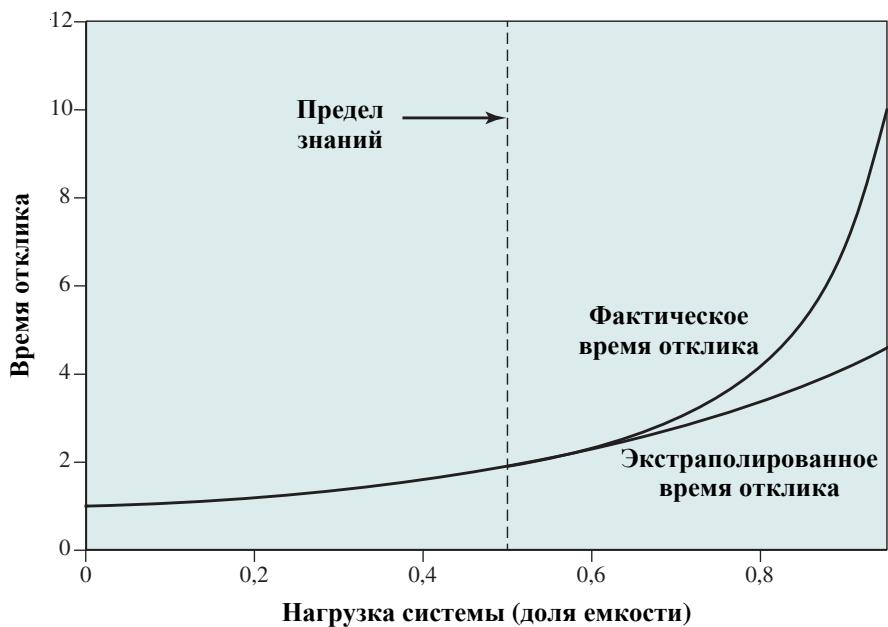


Рис. 3.1. Экстраполированное и фактическое время отклика

Таким образом, требуется более точный инструмент прогнозирования. Вариант 3 заключается в использовании аналитической модели, которая может быть выражена, например, в виде системы уравнений, которые могут при решении выдавать нужные параметры (время отклика, пропускную способность и т.д.). Для многих практических

проблем реального мира аналитические модели, основанные на теории массового обслуживания, обеспечивают достаточно хорошее соответствие реальности. Недостатком теории массового обслуживания является то, что требуется принятие ряда упрощающих предположений, необходимых для вывода уравнений для интересующих нас параметров.

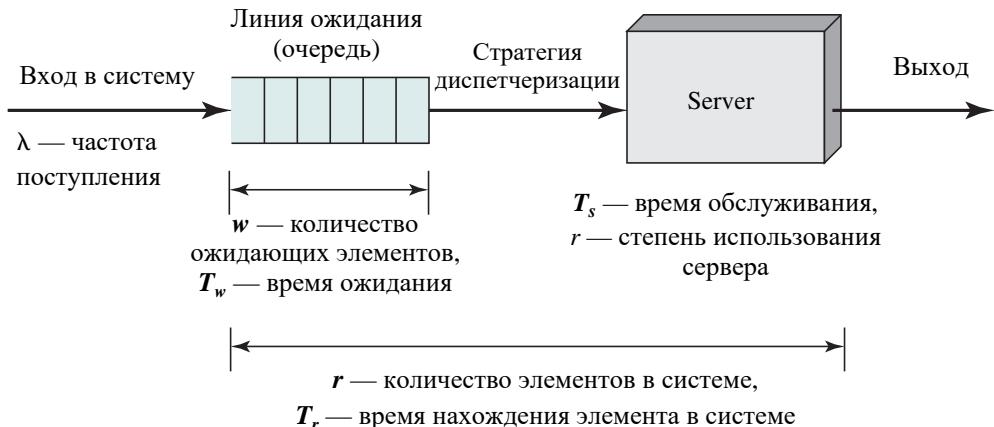
Последний подход представляет собой применение имитационного моделирования. При наличии достаточно мощного и гибкого языка программирования моделей аналитик может моделировать реальность с высокой степенью детализации и избегать применения многих предположений, необходимых для применения теории массового обслуживания. Тем не менее в большинстве случаев имитационная модель не нужна или по крайней мере рекомендуется в качестве первого шага анализа. С одной стороны, как существующие измерения, так и прогнозы будущей нагрузки отягощены определенным пределом погрешности. Таким образом, какой бы хорошей ни была имитационная модель, ценность результатов ограничена качеством входных данных. С другой стороны, несмотря на многие предположения, необходимые для теории массового обслуживания, получаемые результаты часто оказываются близкими к получаемым с помощью более тщательного имитационного моделирования. Кроме того, анализ очередей для четко определенной задачи может быть выполнен буквально в считанные минуты, в то время как программирование имитационных моделей и их выполнение могут длиться дни, недели, а то и дольше.

Таким образом, аналитику стоит овладеть основами теории массового обслуживания.

## 3.2. ОЧЕРЕДЬ В СЛУЧАЕ ОДНОГО СЕРВЕРА

Простейшая система обслуживания с очередью изображена на рис. 3.2. Центральным элементом системы является сервер, который предоставляет некоторые услуги элементам. Элементы из некоторой совокупности поступают в систему для обслуживания. Если сервер простирает, элемент обслуживается немедленно. В противном случае прибывающий элемент присоединяется к очереди ожидания. Когда сервер завершает обслуживание элемента, элемент покидает систему. Если в очереди ожидают другие элементы, один из них немедленно передается серверу. Сервер в этой модели может представлять что угодно, что выполняет некоторую функцию или службу для коллекции элементов. Примерами могут служить процессор, обслуживающий процессы; линия передачи, обеспечивающая услугу передачи пакетов или фреймов данных; устройство ввода-вывода, предоставляющее услуги чтения или записи в ответ на запросы ввода-вывода.

В табл. 3.1 приведены некоторые важные параметры, связанные с данной моделью. Элементы поступают в систему с некоторой средней скоростью (количество поступивших элементов в секунду)  $\lambda$ . В любой момент времени в очереди в состоянии ожидания находится некоторое определенное количество элементов (нуль или больше); среднее количество ожидающих элементов равно  $w$ , а среднее время ожидания одного элемента —  $T_w$ .  $T_w$  усредняется по всем входящим элементам, включая те из них, которые вообще не находились в состоянии ожидания. Сервер обрабатывает входящие элементы; среднее время обслуживания составляет  $T_s$ ; это время представляет собой интервал между передачей элемента на сервер и моментом, когда он покидает сервер. Степень использования сервера  $r$  представляет собой долю времени, когда сервер занят, измеряемую в течение некоторого интервала времени.



**Рис. 3.2.** Структура и параметры односерверной системы с очередью

Наконец, к системе в целом применимы два параметра. Это среднее количество элементов, находящихся в системе, включая обслуживаемые (если таковые имеются) и ожидающие элементы ( $r$ ); и среднее время, которое элемент затрачивает на нахождение в системе, т.е. на ожидание и обслуживание ( $T_r$ ); мы называем его средним временем пребывания в системе.

**Таблица 3.1. Понятия системы обслуживания**

---

$\lambda$	Частота поступления; среднее количество поступающих за секунду элементов
$T_s$	Среднее время обслуживания; время, затраченное на обслуживание, без учета времени ожидания в очереди
$\rho$	Степень использования сервера; доля времени, когда сервер занят
$w$	Среднее количество элементов, ожидающих обслуживания
$T_w$	Среднее время ожидания (усредняется по всем элементам, включая элементы с нулевым временем ожидания)
$r$	Среднее количество элементов, находящихся в системе (ожидающих и обслуживаемых)
$T_r$	Среднее время нахождения в системе (ожидания и обслуживания)

---

Если предположить, что емкость очереди бесконечна, то системой не будут потеряны никакие элементы; они просто задерживаются до тех пор, пока их не обслужат. При этом условии скорость выхода равна скорости поступления. По мере увеличения скорости поступления растет степень использования (а вместе с ней растут и “пробки”). Очередь становится длиннее, увеличивается время ожидания. При  $\rho=1$  сервер становится насыщенным, работая 100% времени. Таким образом, теоретическая максимальная скорость поступления, которая может быть обработана системой, составляет

$$\lambda_{\max} = \frac{1}{T_s}.$$

Однако вблизи точки насыщения системы очереди становятся очень большими, вырастая беспрепятственно при  $\rho=1$ . Практические соображения, такие как требования ко времени отклика или размеры буфера, как правило, ограничивают скорость поступления для одного сервера до 70–90% теоретического максимума.

Обычно делаются следующие предположения.

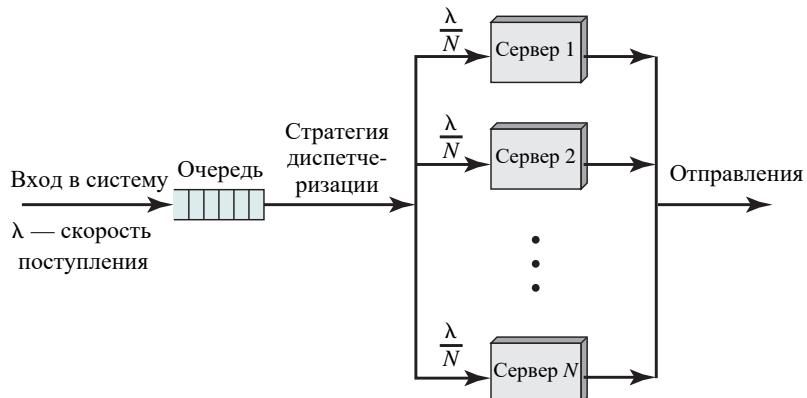
- **Количество элементов.** Обычно мы предполагаем бесконечное количество. Это означает, что скорость поступления не зависит от потерь элементов. Если количество элементов конечно, количество доступных для поступления элементов уменьшается на количество элементов, находящихся в настоящее время в системе; обычно это пропорционально снижает скорость поступления элементов.
- **Размер очереди.** Обычно мы предполагаем бесконечный размер очереди, так что линия ожидания может расти без ограничений. В случае конечной очереди можно потерять некоторые элементы. На практике любая очередь конечна, но во многих случаях это не приводит к существенной разнице при анализе.
- **Стратегия диспетчеризации.** Когда сервер становится свободным и имеется больше одного ожидающего элемента, необходимо принять решение о том, какой элемент обработать следующим. Самый простой подход — “первым пришел — первым вышел”; именно эта стратегия обычно подразумевается, когда используется термин *очередь*. Еще одна возможность — стратегия стека “последним пришел — первым вышел”. На практике можно встретить стратегию, основанную на времени обслуживания. Например, узел с коммутацией пакетов может выбирать пакеты на отправку, начиная с самого короткого или самого длинного. К сожалению, стратегию, основанную на времени обслуживания, очень сложно моделировать аналитически.

### 3.3. МНОГОКАНАЛЬНАЯ ОЧЕРЕДЬ

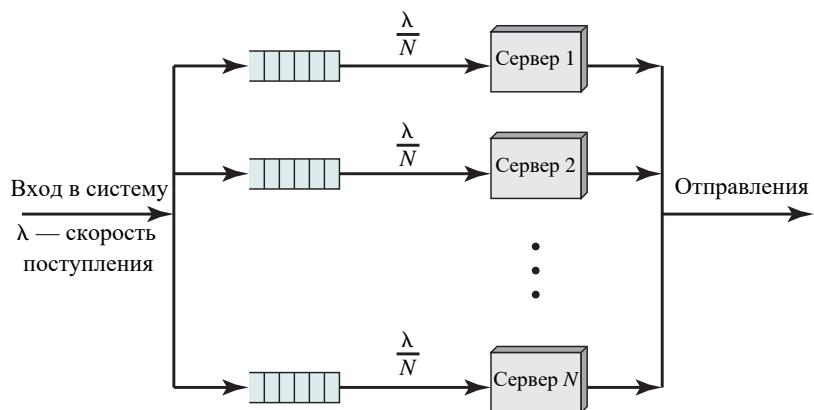
На рис. 3.3 показано обобщение простой модели на несколько серверов, совместно использующих общую очередь. Если на вход поступает элемент и хотя бы один сервер доступен, то элемент немедленно отправляется на этот сервер. Предполагается, что все серверы идентичны; таким образом, если доступно более одного сервера, то не имеет значения, какой сервер выбран для данного элемента. Если же все серверы заняты, начинает формироваться очередь. Как только становится свободным хотя бы один сервер, ему тут же передается элемент из очереди в соответствии с используемой стратегией диспетчеризации.

За исключением степени использования, все параметры, показанные на рис. 3.2, имеют ту же самую интерпретацию и в случае нескольких серверов. Если у нас есть  $N$  идентичных серверов, то  $\rho$  — это степень использования каждого сервера, и мы можем рассматривать  $N\rho$  как степень использования всей системы; этот термин часто называют интенсивностью трафика  $\mu$ . Таким образом, теоретическая максимальная степень использования равна  $N \cdot 100\%$ , а теоретическая максимальная входная частота составляет

$$\lambda_{\max} = \frac{N}{T_s}.$$



а) Многоканальная система массового обслуживания



б) Несколько одноканальных систем массового обслуживания

**Рис. 3.3.** Многоканальная система и система, состоящая из нескольких одноканальных систем

Ключевые характеристики, обычно используемые для многосерверной очереди, соответствуют характеристикам для очереди с одним сервером. То есть мы предполагаем бесконечное количество элементов и бесконечный размер очереди с единой бесконечной очередью, общей для всех серверов. Если не указано иное, стратегия диспетчеризации — FIFO. В случае с несколькими серверами, если предполагается, что все серверы идентичны, выбор конкретного сервера для элемента ожидания не влияет на время обслуживания.

В отличие от этого на рис. 3.3, б показана структура из нескольких односерверных очередей.

### 3.4. ПУАССОНОВА СКОРОСТЬ ПОСТУПЛЕНИЯ

Как правило, аналитические модели массового обслуживания предполагают, что скорость поступления подчиняется распределению Пуассона (именно это предполагается в результатах в табл. 9.6). Это распределение определяется следующим образом. Если элементы поступают в очередь в соответствии с распределением Пуассона, это можно выразить как

$$\Pr[\text{За время } T \text{ поступает } k \text{ элементов}] = \frac{(\lambda T)^k}{k!} e^{-\lambda T},$$

$$E[\text{Количество элементов, поступивших за время } T] = \lambda T,$$

$$\text{Средняя скорость поступления в элементах в с} = \lambda.$$

Поступления, происходящие в соответствии с распределением Пуассона, часто называют **случайными поступлениями**. Это связано с тем, что вероятность поступления элемента за небольшой промежуток времени пропорциональна длине промежутка и не зависит от количества времени, прошедшего с момента последнего поступления элемента. То есть элемент поступит в некоторый момент с той же вероятностью, что и в любой другой, независимо от того, в какие моменты поступают другие элементы.

Другим интересным свойством пуассоновского процесса является его связь с экспоненциальным распределением. Если мы рассмотрим время между поступлением элементов  $T_a$ , то обнаружим, что эта величина подчиняется экспоненциальному распределению:

$$\Pr[T_a < t] = 1 - e^{-\lambda T},$$

$$E[T_a] = \frac{1}{\lambda}.$$

Таким образом, средний интервал между поступлениями является обратной величиной к скорости поступления, как и следовало ожидать.





## ПРИЛОЖЕНИЕ

# Сложность алгоритмов

Центральным вопросом при оценке практичности алгоритма является относительное количество времени, необходимого для выполнения алгоритма. Как правило, нельзя быть уверенным, что для конкретной функции найден самый эффективный алгоритм. Самое большее, что можно сказать, — это определенный порядок величины времени работы конкретного алгоритма. Затем можно сравнить этот порядок величины со скоростью текущего или прогнозируемого процессора и определить степень практичности конкретного алгоритма.

Распространенной мерой эффективности алгоритма является его **временная сложность**. Мы определяем временную сложность алгоритма как  $f(n)$ , если для всех  $n$  и всех входных данных длиной  $n$  выполнение алгоритма занимает не более  $f(n)$  шагов. Таким образом, для данного размера входных данных и данной скорости процессора временная сложность является верхней границей времени выполнения.

Здесь есть несколько неясностей. Во-первых, определение шага не является точным. Шаг может быть одной операцией машины Тьюринга, командой однопроцессорной машины, единой языковой машинной командой высокого уровня и т.д. Тем не менее все эти различные определения шага должны быть связаны между собой простыми мультиплексивными константами. Для очень больших значений  $n$  эти константы не важны. Важно только, насколько быстро растет относительное время выполнения алгоритма.

Вторая проблема заключается в том, что, вообще говоря, мы не можем определить точную формулу для  $f(n)$ . Мы можем только приблизиться к этому. Но, опять же, нас, в первую очередь, интересует скорость изменения  $f(n)$  при очень больших  $n$ .

Для характеристики временной сложности алгоритмов существует стандартное математическое обозначение, известное как “большое  $O$ ”. Его определение таково:  $f(n) = O(g(n))$  тогда и только тогда, когда существуют два числа  $a$  и  $M$ , таких, что

$$|f(n)| \leq a \times |g(n)|, \quad n \geq M. \quad (3.1)$$

Для пояснения воспользуемся примером. Предположим, что мы хотим вычислить обобщенный полином вида

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0.$$

Рассмотрим следующий простой алгоритм из [224].

## 1116 ПРИЛОЖЕНИЕ И. Сложность алгоритмов

```

algorithm P1;
  n, i, j: integer; x, polyval: real;
  a, S: array [0..100] of real;
begin
  read(x, n);
  for i := 0 upto n do
  begin
    S[i] := 1; read(a[i]);
    for j := 1 upto i do S[i] := x * S[i];
    S[i] := a[i] * S[i]
  end;
  polyval := 0;
  for i := 0 upto n do polyval := polyval + S[i];
  write ('value at', x, 'is', polyval)
end.

```

В этом алгоритме каждое подвыражение вычисляется отдельно. Каждое  $S[i]$  требует  $(i+1)$  умножений:  $i$  умножений для вычисления  $x^i$  и еще одно — для умножения на  $a_i$ . Вычисление всех  $n$  членов требует

$$\sum_{i=0}^n (i+1) = \frac{(n+2)(n+1)}{2}$$

умножений. Требуется также  $(n+1)$  сложений, которыми мы пренебрегаем по сравнению с гораздо большим числом умножений. Таким образом, времененная сложность данного алгоритма представляет собой  $f(n) = (n+2)(n+1)/2$ . Покажем, что  $f(n) = O(n^2)$ . Исходя из определения (3.1), мы хотим показать, что для  $a = 1$  и  $M = 4$  указанное соотношение выполняется для  $g(n) = n^2$ . Будем доказывать это по индукции по  $n$ . Соотношение, очевидно, выполняется при  $n = 4$ , так как  $(4+2)(4+2)/2 = 15 < 4^2 = 16$ . Предположим теперь, что оно выполняется для всех значений  $n$  до  $k$ , т.е.  $(k+2)(k+1)/2 < k^2$ . Тогда для  $n = k + 1$ :

$$\begin{aligned}
\frac{(n+2)(n+1)}{2} &= \frac{(k+3)(k+2)}{2} = \\
&= \frac{(k+2)(k+1)}{2} + k + 2 \leq \\
&\leq k^2 + k + 2 \leq k^2 + 2k + 1 = \\
&= (k+1)^2 = n^2.
\end{aligned}$$

Таким образом, результат оказывается истинным для  $n = k + 1$ .

В общем случае запись с “большим  $O$ ” использует наиболее быстро растущий член, например

1.  $O[ax^7 + 3x^3 + \sin(x)] = O(ax^7) = O(x^7);$
2.  $O(e^n + an^{10}) = O(e^n);$
3.  $O(n! + n^{50}) = O(n!).$

Запись с “большим  $O$ ” на самом деле гораздо интереснее и сложнее, чем показано в этом коротком приложении. Заинтересованный читатель может обратиться за дополнительной информацией к работам [54; 94; 136].

Алгоритм со входом размера  $n$  имеет следующие названия.

1. **Логарифмический**, если время его работы —  $O(\log(n))$ .
2. **Линейный**, если время его работы —  $O(n)$ .
3. **Линарифмический** (линейно-логарифмический), если время его работы —  $O(n \log(n))$ .
4. **Полиномиальный**, если время его работы —  $O(n^t)$  для некоторой константы  $t$ .
5. **Экспоненциальный**, если время его работы —  $O(t^{h(n)})$  для некоторой константы  $t$  и полинома  $h(n)$ .

Как правило, задача, которая может быть решена за полиномиальное время, считается выполнимой, тогда как все, что требует больше полиномиального времени, и особенно экспоненциального, считается неразрешимым. Но вы должны быть осторожны с этими терминами. Во-первых, если размер входных данных достаточно мал, то даже очень сложные алгоритмы становятся выполнимыми. Предположим, например, что у вас есть система, которая может выполнять  $10^{12}$  операций в единицу времени. В табл. И.1 показан размер входных данных, который может быть обработан за единицу времени для алгоритмов различной сложности. Алгоритмы с экспоненциальным или факториальным временем работы могут работать только с очень небольшими входными данными.

**Таблица И.1. РАЗМЕР ВХОДНЫХ ДАННЫХ ДЛЯ РАЗНЫХ УРОВНЕЙ СЛОЖНОСТИ**

Сложность	Размер входных данных	Количество операций
$\log_2 n$	$2^{10^{12}} = 10^{3 \times 10^{11}}$	$10^{12}$
$n$	$10^{12}$	$10^{12}$
$n^2$	$10^6$	$10^{12}$
$n^6$	$10^2$	$10^{12}$
$2^n$	39	$10^{12}$
$n!$	15	$10^{12}$

Второе, на что следует обратить внимание, — это то, как характеризуются входные данные. Например, сложность криптоанализа алгоритма шифрования может одинаково хорошо характеризоваться с точки зрения как количества возможных ключей, так и длины ключа. Для стандарта шифрования AES, например, число возможных ключей —  $2^{128}$ , а длина ключа — 128 бит. Если рассматривать одно шифрование как “шаг”, а количество возможных ключей  $N=2^n$ , то временная сложность алгоритма является линейной с точки зрения количества ключей [ $O(N)$ ], но экспоненциальной по отношению к длине ключа [ $O(2^n)$ ].





# ДИСКОВЫЕ УСТРОЙСТВА ХРАНЕНИЯ

В ЭТОМ ПРИЛОЖЕНИИ...

## **K.1. Магнитные диски**

Организация данных и форматирование

Физические характеристики

## **K.2. Оптическая память**

CD-ROM

CD с возможностью записи

CD-R с возможностью перезаписи

DVD

Оптические диски высокой четкости

## **K.1. МАГНИТНЫЕ ДИСКИ**

Магнитный диск представляет собой круглый диск, изготовленный из металла или пластика, покрытого намагничиваемым материалом. Данные записываются, а затем извлекаются с диска с помощью проводящей катушки, которую называют **головкой** (head). Во время операции чтения или записи диск вращается под неподвижной головкой.

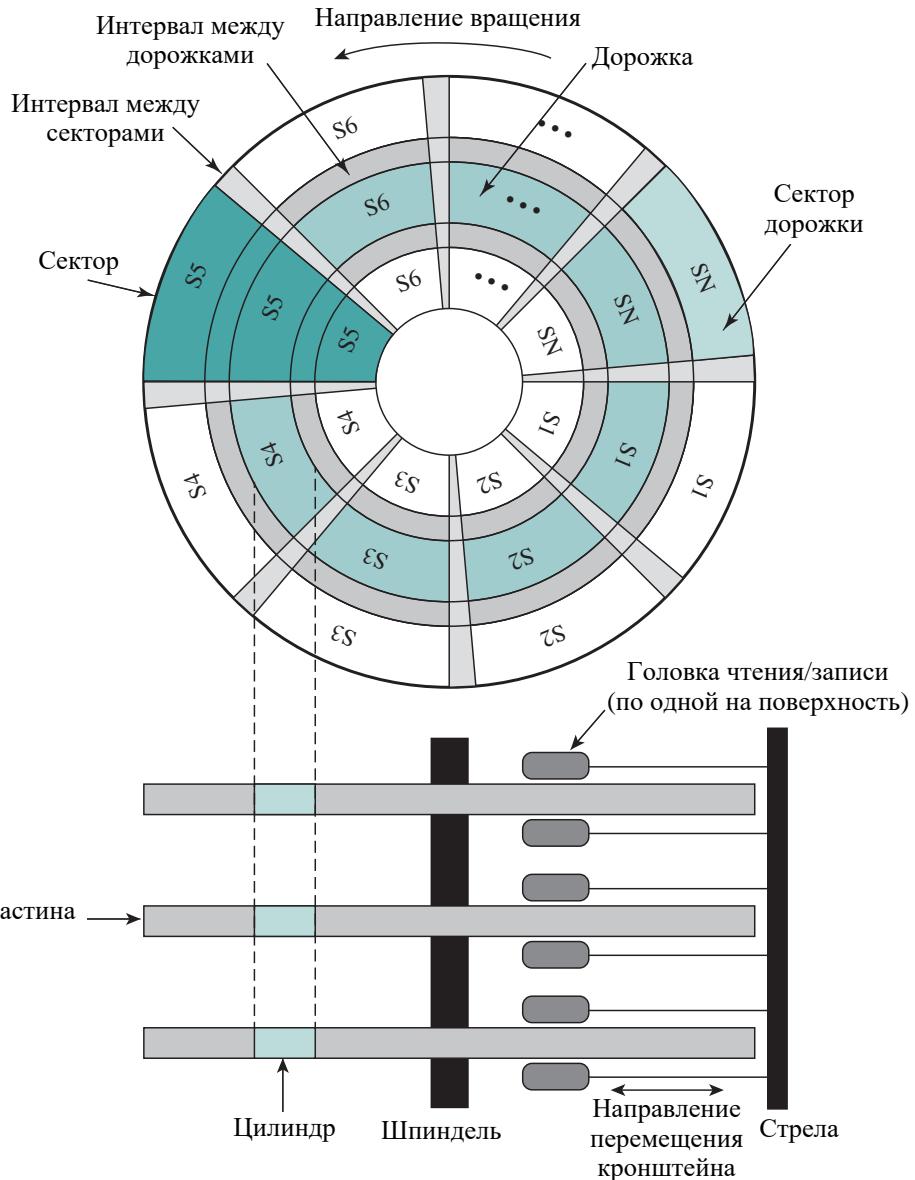
В механизме записи используется тот факт, что электричество, протекающее катушку, создает магнитное поле. В головку посылаются электрические импульсы, в результате чего поверхность под ней оказывается по-разному намагниченной, с различным намагничиванием для положительного и отрицательного токов. Механизм чтения основан на том факте, что движущееся относительно катушки магнитное поле генерирует в ней электрический ток. Когда под головкой проходит поверхность диска, она генерирует ток той же полярности, что и использовавшийся для записи.

### **Организация данных и форматирование**

Головка представляет собой относительно небольшое устройство, способное читать информацию с части вращающегося под ним диска или записывать ее на него. Это при-

водит к тому, что данные на диске организованы в виде концентрических колец, называемых **дорожками** или треками (track). Каждая дорожка имеет ту же ширину, что и головка чтения/записи. На поверхности диска располагаются тысячи дорожек.

На рис. К.1 представлена схема размещения данных. Соседние дорожки разделены **интервалами**, или **промежутками** (gaps). Это предотвращает (или по крайней мере минимизирует) возникновение ошибок, вызванных некорректным положением головки или простой интерференцией магнитных полей.

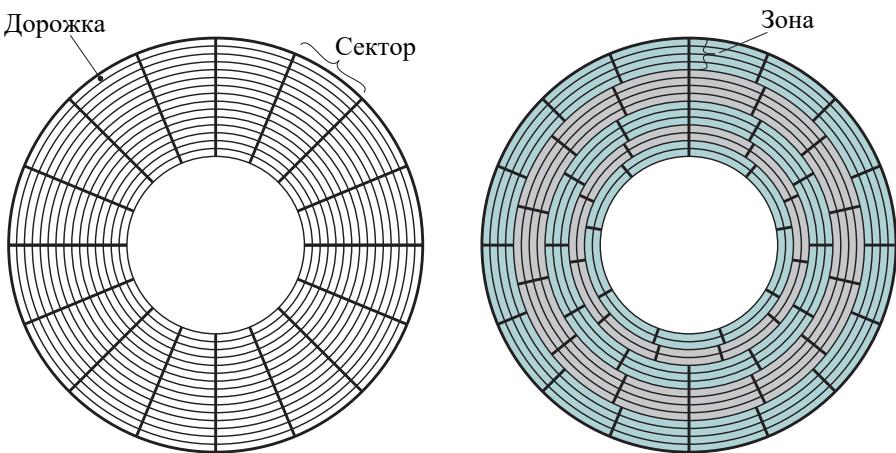


**Рис. К.1.** Схема размещения данных на диске

Данные пересыпаются на диск и из него **секторами** (sector). Обычно на одной дорожке располагается несколько сотен секторов, и они могут быть либо фиксированной, либо переменной длины. Для большинства дисков используется фиксированный размер сектора — 512 байт. Во избежание ошибок соседние секторы также разделены интервалами.

Вблизи центра вращающийся диск проходит мимо фиксированной точки (например, головки) медленнее, чем снаружи диска. Поэтому необходимо найти какой-то способ компенсировать изменение скорости, чтобы головка могла считывать все биты с одной и той же скоростью. Это можно сделать, увеличив интервал между записанными в сегментах диска битами информации. В этом случае информация может быть считана с той же скоростью при **постоянной угловой скорости** (constant angular velocity — CAV).

На рис. К.2, а показана схема диска с использованием CAV. Диск делится на ряд секторов в концентрических дорожках. Преимущество использования CAV заключается в том, что отдельные блоки данных могут быть адресованы непосредственно, с использованием дорожек и секторов. Чтобы переместить головку из ее текущего местоположения в некоторый конкретный адрес, требуется всего лишь короткое ее движение к определенной дорожке и короткое ожидание, пока в ходе вращения соответствующий сектор окажется прямо под головкой. Недостатком CAV является то, что объем данных, которые могут храниться на длинных внешних дорожках, тот же, что и на коротких внутренних.



а) Постоянная угловая скорость

б) Многозонная запись

**Рис. К.2.** Сравнение схем размещения данных на диске

Поскольку при перемещении от самой внешней дорожки к самой внутренней дорожке **плотность записи** в битах на линейный дюйм увеличивается, емкость дискового хранилища в простой системе CAV ограничивается максимальной плотностью записи, которая может быть достигнута на самой внутренней дорожке. Чтобы увеличить плотность, современные системы на жестких дисках используют технологию, известную как **многозонная запись**, в которой поверхность делится на несколько концентрических зон (обычно 16). В пределах зоны количество битов на дорожку является постоянным. Удаленные от центра зоны содержат больше битов (больше секторов), чем зоны ближе к

центру. Это позволяет увеличить общую емкость хранилища за счет несколько более сложных электронных схем. Когда головка диска перемещается из одной зоны в другую, длина отдельных битов (вдоль дорожки) изменяется, что приводит к изменению времени чтения и записи. На рис. К.2, б показана схема многозонной записи; на этом рисунке каждая зона имеет ширину, равную одной дорожке.

Для определения позиций секторов на дорожке требуются специальные средства. Ясно, что на дорожке должна существовать некая начальная точка и должен быть способ идентификации начала и конца каждого сектора. Эти требования удовлетворяются посредством контроля за записанными на диск данными. Так, диск форматируется с некоторыми дополнительными данными, используемыми только дисководом и недоступными пользователю.

## Физические характеристики

В табл. К.1 приведены основные характеристики, отличающие один тип дисков от другого. Головка диска может быть либо фиксированной, либо перемещаемой (в радиальном направлении пластины диска). В **дисках с фиксированной головкой** на одну дорожку приходится одна головка. Все головки смонтированы на жестком кронштейне, который располагается над всеми дорожками. В **диске с перемещающейся головкой** имеется только одна головка, закрепленная на кронштейне. Поскольку головка должна быть способна размещаться над любой дорожкой, кронштейн должен обеспечивать ее перемещение над диском.

**Таблица К.1. Физические характеристики дисковых систем**

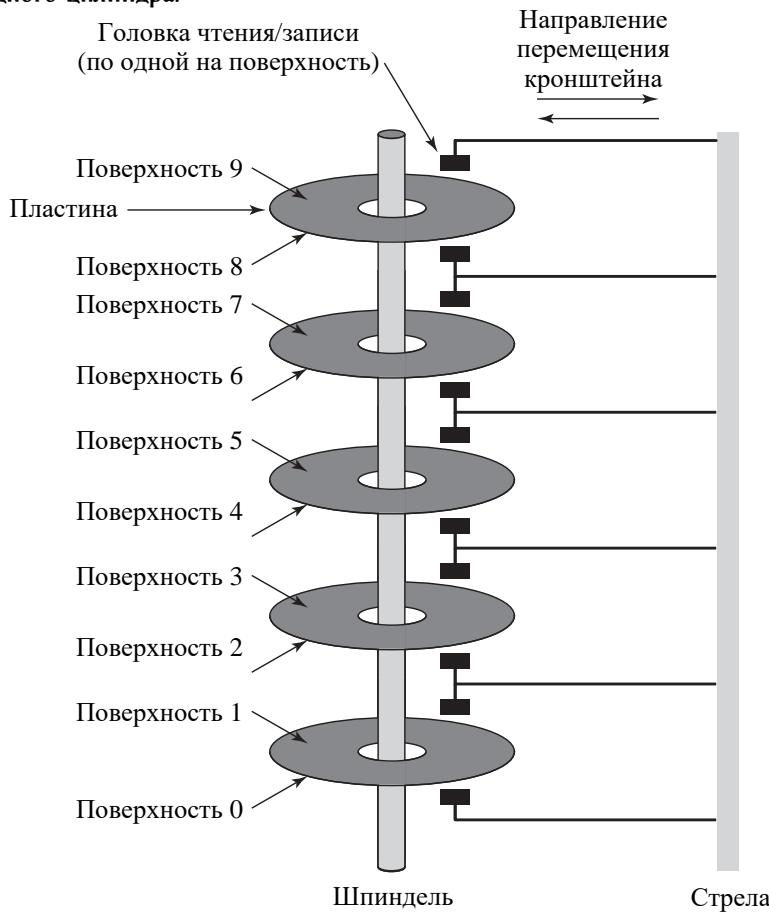
<b>Движение головки</b>	Фиксированная головка (одна на дорожку) Перемещаемая головка (одна на поверхность)
<b>Переносимость диска</b>	Стационарный диск Переносной (съемный) диск
<b>Стороны</b>	Односторонние Двухсторонние
<b>Дисковые пластины</b>	Одна пластина Множество пластин
<b>Механизм головки</b>	Контактный (флоппи-диск) С фиксированным зазором Аэродинамический зазор (винчестер)

Сам же диск смонтирован в дисководе, состоящем из кронштейна, шпинделя, врашающего диска, и электронной схемы, необходимой для осуществления ввода и вывода бинарных данных. **Стационарный диск** зафиксирован в дисководе, в то время как **переносной диск** может быть удален или заменен другим. Преимущество дисков последнего типа состоит в том, что при ограниченном количестве дисковых систем доступен

неограниченный объем данных. Кроме того, такой диск может быть перенесен из одной компьютерной системы в другую (при условии их совместимости).

В большинстве дисков магнитное покрытие наносится на обе стороны дисковой пластины (такой диск называется **двусторонним**). В некоторых менее дорогих системах используются **односторонние диски**.

В некоторых дисководах содержится **несколько дисковых пластин**, размещенных на вертикальной оси с интервалом в несколько дюймов. При этом предусматривается наличие нескольких кронштейнов. Набор дисковых пластин называется **пакетом дисков** (рис. К.3). Многопластинчатые диски снабжены перемещаемыми головками, по одной на одну поверхность пластины. Головки механически зафиксированы таким образом, что все они находятся на одинаковом расстоянии от центра диска и перемещаются одновременно. Таким образом, в любой момент времени все головки оказываются расположеными над дорожками, равноудаленными от центра диска. Множество дорожек, находящихся в одной и той же позиции по отношению к пластине, называется **цилиндром** (cylinder). Например, все закрашенные дорожки, показанные на рис. К.4, являются частью одного цилиндра.



**Рис. К.3. Компоненты магнитного диска**

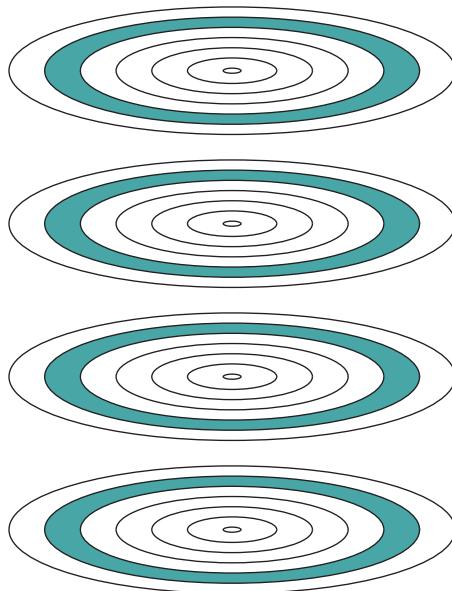


Рис. К.4. Дорожки и цилиндры

Наконец, механизм головки обеспечивает разделение дисков на три типа. Обычно головка чтения/записи располагается на фиксированном расстоянии над дисковой пластиной. В головках второго типа при считывании или записи осуществляется реальный физический контакт с поверхностью. Такой механизм используется в **гибких дисках**, имеющих небольшой размер, гибкую пластину и являющихся самыми дешевыми из всех типов дисков.

Чтобы объяснить принцип работы дисков третьего типа, необходимо прокомментировать соотношение между плотностью данных и размером воздушного промежутка. Головка должна генерировать или улавливать электромагнитное поле достаточной для нормального считывания или записи величины. Чем уже головка, тем ближе она должна быть расположена к поверхности пластины. Сужение головки предполагает сужение дорожки и, естественно, увеличение плотности записи, что очень желательно. Однако чем ближе головка расположена к поверхности пластины, тем больше вероятность возникновения ошибок, вызванных загрязнениями или физическими дефектами. Дальнейшее развитие технологии привело к появлению **винчестерных дисков**, головки которых используются в герметичных дисководах, защищенных от загрязнений. Они сконструированы для работы на расстоянии от поверхности пластины, меньшем, чем расстояние при работе с обычными жестко фиксированными головками, что приводит к более высокой плотности записи данных. Головка представляет собой своеобразную аэродинамическую фольгу, парящую над поверхностью пластины при вращении диска. Воздушное давление, создаваемое при вращении диска, достаточно для создания зазора между поверхностью и головкой. Такая бесконтактная схема обеспечивает работу очень узких, близко расположенных к поверхности головок.

В табл. К.2 приведены параметры типичных современных высокопроизводительных дисков с движущимися головками.

**Таблица К.2. Параметры типичных дисков**

<b>Характеристики</b>	<b>Seagate Barracuda ES.2</b>	<b>Seagate Barracuda 7200.10</b>	<b>Seagate Barracuda 7200.9</b>	<b>Seagate Microdrive</b>
Применение	Высокоемкостные серверы	Высокопроизводительные настольные системы	Простая настольная система	Лэптоп
Емкость	1 Тбайт	750 Гбайт	160 Гбайт	120 Гбайт
Минимальное время поиска от дорожки к дорожке, мс	0,8	0,3	1,0	—
Среднее время поиска, мс	8,5	3,6	9,5	12,5
Скорость вращения шпинделя, об/мин	7200	7200	7200	5400
Средняя задержка из-за вращения, мс	4,16	4,16	4,17	5,6
Максимальная скорость передачи данных	3 Гбайт/с	300 Мбайт/с	300 Мбайт/с	150 Мбайт/с
Количество байтов в секторе	512	512	512	512
Количество дорожек в цилиндре (количество поверхностей пластин)	8	8	2	8
				2

## К.2. ОПТИЧЕСКАЯ ПАМЯТЬ

В 1983 году был представлен один из самых удачных потребительских продуктов всех времен — компактная дисковая (CD) цифровая аудиосистема. CD является “нестираемым” диском, который может хранить более 60 мин аудиоинформации на одной стороне. Огромный коммерческий успех CD позволил развить недорогую технологию хранения данных на оптических дисках, что стало революционным шагом в компьютерном хранении данных. Разнообразные системы оптических дисков (которые вкратце будут рассмотрены далее в приложении) представлены в табл. К.3.

**Таблица К.3. Продукты на основе оптической технологии дисков**

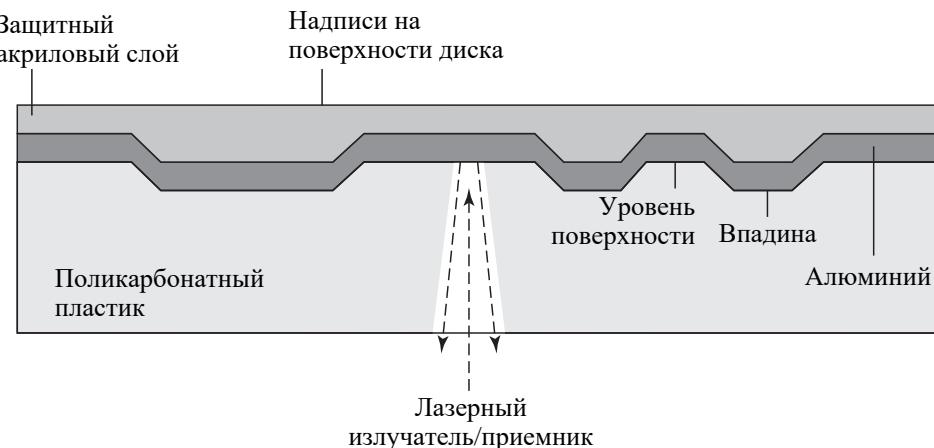
<b>CD</b>	Компакт-диск. Нестираемый диск, хранящий аудиоинформацию в цифровом виде. Стандартная система использует 12-сантиметровые диски и может записать более 60 мин непрерывного времени звучания
<b>CD-ROM</b>	Компакт-диск с постоянной памятью. Нестираемый диск, используемый для хранения компьютерных данных. Стандартная система использует 12-сантиметровые диски и способна содержать более 650 Мбайт информации
<b>CD-R</b>	CD с возможностью записи. Подобен CD-ROM. Пользователь может записать информацию на диск только один раз
<b>CD-RW</b>	CD с возможностью перезаписи. Подобен CD-ROM. Пользователь может неоднократно стирать и записывать информацию на диск
<b>DVD</b>	Универсальный цифровой диск. Технология для хранения как цифрового сжатого представления видеинформации, так и больших объемов других цифровых данных. Используются как 8-, так и 12-сантиметровые диски с двусторонней записью объемом до 15,9 Гбайт. Базовый DVD — с постоянной памятью (DVD-ROM)
<b>DVD-R</b>	DVD с возможностью записи. Подобен DVD-ROM. Пользователь может записать информацию на такой диск только один раз
<b>DVD-RW</b>	DVD с возможностью перезаписи. Подобен DVD-ROM. Пользователь может неоднократно стирать и записывать информацию на диск
<b>Blu-Ray DVD</b>	Видеодиск высокого разрешения. Обеспечивает значительно большую плотность хранения данных, чем DVD, используя лазер с длиной волны 405 нм (сине-фиолетовый). Один слой на одной стороне может хранить 25 Гбайт информации

### CD-ROM

В аудио-CD и CD-ROM (постоянное запоминающее устройство на компакт-диске) используется одна и та же технология. Главное различие заключается в том, что дисководы CD-ROM являются более грубыми и имеют возможности коррекции ошибок для гарантии корректности данных. Оба типа дисков изготавливаются одинаково. Диск выполняется на пластиковой основе, например на поликарбонате, и имеет поверхность с хорошим отражением (обычно алюминиевую). Информация, записанная в цифровом виде (музыкальные или компьютерные данные), отпечатывается в виде серий микроскопических впадин (питов) на отражающей поверхности. Это может быть выполнено

точно сфокусированным лазером высокой интенсивности для создания исходного диска, называемого мастер-диском. Мастер-диск, в свою очередь, используется для производства штампа, на основе которого изготавливаются копии. Поверхность дисков-копий защищена от пыли и царапин своеобразной "рубашкой" из чистого лака.

Извлечь информацию из CD или CD-ROM можно с помощью маломощного лазера, размещенного в дисководе. Луч лазера проходит сквозь чистое защищенное покрытие при вращении диска (рис. К.5). Интенсивность отраженного луча лазера изменяется при попадании на впадину. Это изменение фиксируется фотодатчиком и преобразовывается в цифровой сигнал.



**Рис. К.5. Работа CD**

Впадина вблизи центра вращения диска проходит мимо зафиксированной точки (такой, как лазерный луч) медленнее, чем впадина на внешнем краю диска, так что нужно найти способ компенсации изменения линейной скорости, необходимый для возможности сканирования информации с одинаковой частотой. Это может быть достигнуто, как и на магнитных дисках, посредством увеличения пространства между битами информации, записанной в сегментах диска. После этого информация может быть сканирована с одинаковой частотой при **постоянной угловой скорости**. Диск делится на концентрические дорожки с определенным количеством секторов. Преимущество использования постоянной угловой скорости заключается в возможности непосредственной адресации отдельных блоков путем указания их дорожки и сектора. Неудобство такой схемы заключается в том, что суммарное количество данных, которые хранятся на внешней дорожке, такое же, как и на внутренней.

Поскольку при размещении малого количества информации на участке внешней дорожки незэкономно расходуется пространство, метод постоянной угловой скорости не используется в CD и CD-ROM. Вместо этого информация равномерно распределяется по всем дисковым сегментам одинакового размера, которые впоследствии сканируются с одной и той же скоростью путем вращения диска с переменной скоростью. Впадины сканируются лазером при **постоянной линейной скорости**. При обращении к внешним дорожкам диск вращается медленнее, чем при обращении к внутренним, поэтому емкость дорожки и задержка из-за вращения увеличиваются для дорожек вблизи внешнего края диска. Емкость CD-ROM составляет около 680 Мбайт.

CD-ROM пригоден для распространения большого количества данных среди большого количества пользователей. Он обладает тремя главными преимуществами по сравнению с традиционными магнитными дисками.

- Объем хранимой информации намного больше у оптического диска.
- Оптический диск вместе с сохраненной на нем информацией может быть массово тиражирован при небольших денежных затратах, в отличие от магнитного диска. База данных на магнитном диске может быть воспроизведена путем копирования одного диска за один раз с использованием двух дисководов.
- Оптический диск является переносным, что позволяет использовать его для архивного хранения.

К недостаткам CD-ROM относятся следующее.

- Режим “только для чтения” не позволяет обновлять информацию.
- Время доступа к диску намного больше, чем у магнитного накопителя.

## CD с возможностью записи

Для тех случаев, когда требуется только одна копия диска или небольшое количество копий, можно использовать CD с возможностью записи (CD-R). CD-R подготавливается таким образом, чтобы он мог быть один раз последовательно записан лазерным лучом умеренной интенсивности. С помощью такого немного более дорогого, чем CD-ROM, дисковода покупатель может один раз записать и сколько угодно раз прочитать диск.

Физическая структура CD-R схожа со структурой CD или CD-ROM, но не идентична ей. В CD и CD-ROM информация записывается с помощью впадин на поверхности пластины, что приводит к изменению отражающей способности. В CD-R добавлен окрашенный слой. Краска используется для изменения отражающей способности и активизируется высокointенсивным лазером. Получившийся диск может быть считан приводом CD-R или CD-ROM.

Оптический диск CD-R удобен для архивного хранения документов и файлов. Он обеспечивает постоянную запись больших объемов пользовательских данных.

## CD-R с возможностью перезаписи

Оптический диск CD-RW может быть повторно записан и перезаписан, как и магнитный диск. Несмотря на то что было опробовано большое количество подходов, достоин внимания только чисто оптический подход, вызывающий фазовые изменения. Такой диск использует материал, имеющий отражательную способность, значительно отличающуюся в двух различных фазовых состояниях: аморфном состоянии, в котором молекулы ориентированы случайно и отражение луча света слабое, и кристаллическом состоянии, хорошо отражающем свет. Лазерный луч может изменить фазу материала с одной на другую. Основной недостаток оптических дисков с переменной фазой состоит в том, что материал в конечном счете теряет свои свойства. Материалы, используемые в настоящее время, могут выдержать от 500 тыс. до 1 млн циклов стирания информации.

CD-RW обладает очевидным преимуществом перед CD-ROM и CD-R, которое состоит в том, что информация на нем может быть перезаписана, и поэтому он может исполь-

зоваться как устройство для вспомогательного хранения данных. Поэтому CD-RW вполне может конкурировать с магнитными дисками. Основные преимущества стираемого оптического диска — большая емкость, портативность и надежность.

## DVD

В образе DVD электронная промышленность, наконец-то, нашла достойную замену аналоговым видеолентам VHS. Однако DVD заменит не только видеоленты, но и, что более важно для нас, CD-ROM в персональных компьютерах. С появлением DVD наступил век цифрового видео. Он позволяет получать фильмы с превосходным качеством, и при этом доступ к информации может осуществляться случайным образом — как у аудио CD, которые DVD-устройства также могут проигрывать. Сегодня объем DVD-диска приблизительно в семь раз больше объема обычного CD-ROM. Объем памяти DVD обеспечивает возможность создания и распространения более качественных и реалистичных игр для персональных компьютеров, а образовательное программное обеспечение может включать больше видео.

Большая емкость DVD обусловлена тремя отличиями от компакт-дисков.

1. Биты на DVD упакованы более плотно. Расстояние между окружностями спирали на компакт-диске составляет 1,6 мкм, а минимальное расстояние между углублениями вдоль спирали составляет 0,834 мкм. DVD использует лазер с более короткой длиной волны и обеспечивает расстояние между петлями 0,74 мкм и минимальное расстояние между точками 0,4 мкм. Результат этих двух улучшений — увеличение емкости примерно в семь раз, примерно до 4,7 Гбайт.
2. DVD может использовать второй слой впадин поверх первого слоя. Двухслойный DVD имеет полупрозрачный слой поверх отражающего слоя, так что, регулируя фокус, лазеры в приводах DVD могут считывать каждый слой отдельно. Этот метод почти удваивает емкость диска, примерно до 8,5 Гбайт. Более низкая отражательная способность второго слоя ограничивает его емкость, поэтому полное удвоение не достигается.
3. DVD-ROM может быть двухсторонним, тогда как на CD данные записываются только на одной стороне компакт-диска. Это дает возможность получения общей емкости диска около 17 Гбайт.

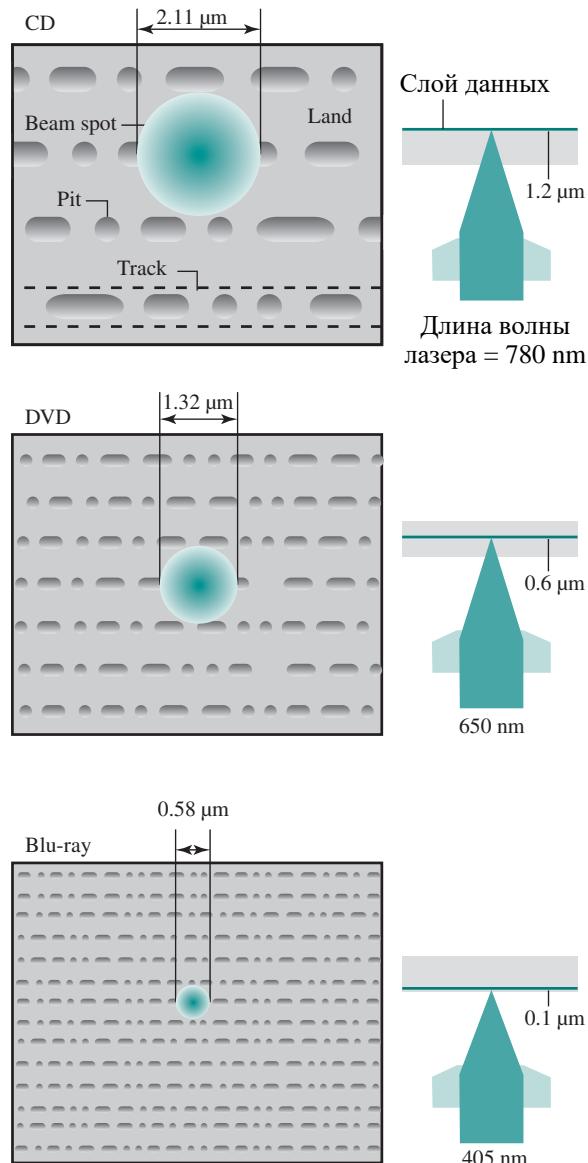
Как и в случае с компакт-дисками, DVD-диски выпускаются в версиях как только для чтения, так и для записи (см. табл. К.3).

## ОПТИЧЕСКИЕ ДИСКИ ВЫСОКОЙ ЧЕТКОСТИ

Оптические диски высокой четкости предназначены для хранения видео высокой четкости и обеспечивают значительно большую емкость по сравнению с DVD. Повышенная плотность битов достигается с помощью лазера с более короткой длиной волны, в сине-фиолетовом диапазоне. Питы данных, которые представляют собой цифровые 1 и 0, оказываются меньшего размера, чем в дисках DVD, из-за более короткой длины волны лазера.

Изначально за признание на рынке боролись две технологии и соответственно, два формата диска: HD DVD и Blu-Ray DVD. В конечном итоге доминирующее положение на рынке завоевала схема Blu-ray. Схема HD DVD может хранить 15 Гбайт на одном

слое на одной стороне; в Blu-ray слой данных на диске размещается ближе к лазеру (см. рис. К.6, справа). Это обеспечивает более четкую фокусировку и меньше искажений и, таким образом, меньшие питы и дорожки. Blu-ray может хранить до 25 Гбайт на одном слое. Доступны три версии дисков: только для чтения (BD-ROM), для однократной записи (BD-R) и перезаписываемая (BD-RE).



**Рис. К.6.** Характеристики оптических накопителей



# КРИПТОГРАФИЧЕСКИЕ АЛГОРИТМЫ

В ЭТОМ ПРИЛОЖЕНИИ...

## Л.1. Симметричное шифрование

DES

AES

## Л.2. Шифрование с открытым ключом

Алгоритм Ривеста–Шамира–Адлемана (RSA)

## Л.3. Аутентификация сообщений и хеш-функции

Аутентификация с использованием симметричного шифрования

Аутентификация без шифрования

Код аутентификации сообщения

Функция одностороннего хеширования

## Л.4. Безопасные хеш-функции

Основная технология, лежащая в основе практически всех систем безопасности сетей и компьютеров, — криптография. Используются два основных подхода: симметричное шифрование, также известное как обычное шифрование, и шифрование с открытым ключом, известное как асимметричное шифрование. В этом приложении представлены обзор обоих типов шифрования, а также краткое обсуждение некоторых важных алгоритмов шифрования.

## Л.1. СИММЕТРИЧНОЕ ШИФРОВАНИЕ

Симметричное шифрование было единственным типом шифрования, использовавшимся до введения шифрования с открытым ключом в конце 1970-х годов. Оно использовалось для тайного общения бесчисленного количества людей и групп, от Юлия Цезаря и немецких подводных лодок до современной дипломатической, военной и коммерческой переписки. Оно остается более широко используемым из двух указанных типов шифрования.

Симметричная схема шифрования содержит пять компонентов (рис. Л.1).

- **Открытый текст.** Исходное сообщение или данные, которые вводятся в алгоритм в качестве входных данных.
- **Алгоритм шифрования.** Выполняет различные замены и преобразования в открытом тексте.
- **Секретный ключ.** Также вводится в алгоритм шифрования в качестве входных данных. Замены и преобразования, выполняемые алгоритмом, зависят от этого ключа.
- **Зашифрованный текст.** Зашифрованное сообщение, созданное и представляющее собой выходные данные алгоритма. Зависит от открытого текста и секретного ключа. Для одного и того же сообщения два разных ключа будут давать два разных зашифрованных текста.
- **Алгоритм дешифрования.** По сути, это алгоритм шифрования, работающий в обратном направлении. Получает в качестве входных данных зашифрованный текст и секретный ключ и производит исходный простой текст.

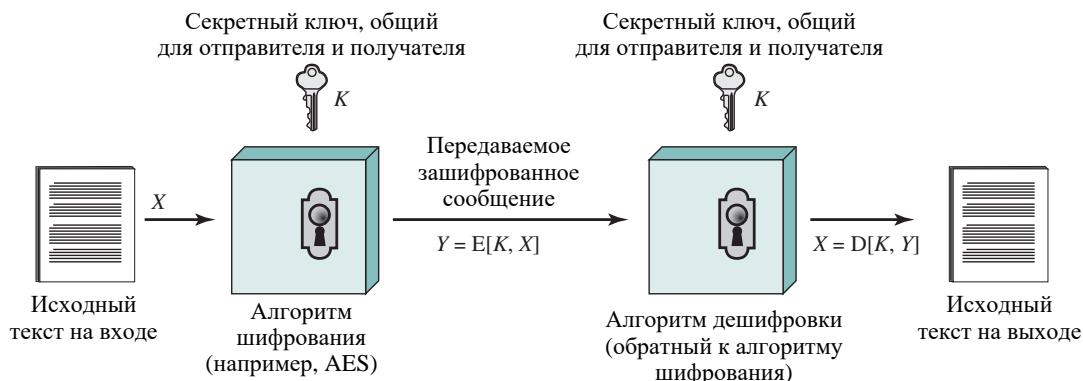


Рис. Л.1. Упрощенная модель симметричного шифрования

Для безопасного использования симметричного шифрования должны выполняться два требования.

1. Нужен хороший алгоритм шифрования. Как минимум хотелось бы, чтобы алгоритм был таким, чтобы противник, знающий алгоритм и имеющий доступ к одному или нескольким зашифрованным текстам, не мог расшифровать зашифрованный текст или получить ключ. Это требование обычно указывается в более сильной форме: противник не может расшифровать зашифрованный текст или найти ключ, даже обладая набором зашифрованных текстов вместе с открытыми текстами, для которых получаются эти зашифрованные тексты.
2. Отправитель и получатель должны безопасно получить копии секретного ключа и должны хранить его в секрете. Если кто-то сможет обнаружить ключ и узнать алгоритм, все сообщения, использующие этот ключ, могут быть прочитаны.

Существует два основных подхода к атаке на симметричную схему шифрования. Первая атака называется **криптоанализом**. Криптоаналитические атаки основаны на природе алгоритма и, возможно, на некоторых знаниях об общих характеристиках от-

крытого текста (или даже некоторых образцах пар открытого и зашифрованного текстов). Этот тип атаки использует характеристики алгоритма в попытках вывести конкретный открытый текст или используемый ключ. Если атаке удастся получить ключ, эффект будет катастрофическим: все будущие и прошлые сообщения, зашифрованные этим ключом, окажутся скомпрометированными.

Второй метод, известный как атака **грубой силы**, состоит в том, чтобы испытать все возможные ключи на части зашифрованного текста, пока не будет получен понятный открытый текст. В среднем для достижения успеха должна быть испытана половина всех возможных. В табл. Л.1 показано сколько времени уходит на работу с ключами разных размеров. В таблице приведены результаты для каждого размера ключа при условии, что для выполнения одного дешифрования требуется 1 мс — разумное допущение для современных компьютеров. С использованием микропроцессоров с высокой степенью параллельности может оказаться возможным достижение скорости обработки, большей на порядки. В последнем столбце таблицы рассматриваются результаты для системы, которая может обрабатывать 1 млн ключей в микросекунду. Как видите, при этом уровне производительности 56-битный ключ больше нельзя считать безопасным.

**Таблица Л.1. СРЕДНЕЕ ВРЕМЯ, НЕОБХОДИМОЕ ДЛЯ ПОДБОРА КЛЮЧА**

Размер ключа, битов	Количество разных ключей	Время при скорости 1 дешифровка/мкс	Время при скорости $10^6$ дешифровок/мкс
32	$2^{32} = 4,3 \times 10^9$	$2^{32}$ мкс = 35,8 мин	2,15 мс
56	$2^{56} = 7,2 \times 10^{16}$	$2^{55}$ мкс = 1142 года	10,01 ч
128	$2^{128} = 3,4 \times 10^{38}$	$2^{127}$ мкс = $5,4 \times 10^{24}$ лет	$5,4 \times 10^{18}$ лет
168	$2^{168} = 3,7 \times 10^{50}$	$2^{167}$ мкс = $5,9 \times 10^{36}$ лет	$5,9 \times 10^{30}$ лет
26 символов (перестановка)	$26! = 4 \times 10^{26}$	$2 \times 10^{26}$ мкс = $6,4 \times 10^{12}$ лет	$6,4 \times 10^6$ лет

Наиболее часто используемые алгоритмы симметричного шифрования представляют собой блочные шифры. Блочный шифр обрабатывает входной открытый текст блоками фиксированного размера и создает блок зашифрованного текста того же размера для каждого блока открытого текста. Два самых важных симметричных алгоритма, оба являющиеся блочными шифрами, — стандарт шифрования данных (Data Encryption Standard — DES) и расширенный стандарт шифрования (Advanced Encryption Standard — AES).

## DES

DES являлся доминирующим алгоритмом шифрования с момента его появления в 1977 году. Однако, поскольку DES использует только 56-битный ключ, вопрос его устаревания был не более чем вопросом времени — когда скорость компьютеров достигнет соответствующего значения. В 1998 году фонд Electronic Frontier Foundation (EFF) объявил, что проблема DES преодолена с помощью специализированной машины “DES cracker” стоимостью менее 250 тыс. долларов. Атака продолжалась менее трех дней. EFF опубликовал подробное описание машины, позволяя другим создавать собственный взломщик этого алгоритма шифрования. Поскольку цены на оборудование продолжают падать, а производительность — расти, DES становится все более бесполезным.

Срок службы DES был продлен за счет использования методики тройного DES (3DES), которая включает в себя повторение базового алгоритма DES три раза с двумя или тремя уникальными ключами, по сути, получая ключ размером 112 или 168 бит.

Основным недостатком 3DES является то, что алгоритм является относительно медленным. Вторым недостатком является то, что как DES, так и 3DES используют размер блока, равный 64 бит. Из соображений эффективности и безопасности желателен больший размер блока.

## AES

Из-за своих недостатков 3DES не является разумным кандидатом для долгосрочного использования. В качестве замены Национальный институт стандартов и технологий (National Institute of Standards and Technology — NIST) в 1997 году опубликовал предложения по новому расширенному стандарту шифрования (AES), который должен иметь степень безопасности не хуже, чем у 3DES, и значительно улучшенную эффективность. В дополнение к этим общим требованиям NIST указал, что AES должен быть симметричным блочным шифром с длиной блока 128 бит и поддержкой длин ключей 128, 192 и 256 бит. Критерии оценки метода шифрования включают безопасность, вычислительную эффективность, требования к памяти, совместимость аппаратного и программного обеспечения и гибкость. В 2001 году NIST выпустил AES в качестве федерального стандарта обработки информации (FIPS 197).

## Л.2. ШИФРОВАНИЕ С ОТКРЫТЫМ КЛЮЧОМ

Шифрование с открытым ключом, предложенное Диффи (Diffie) и Хеллманом (Hellman) в 1976 году, является первым по-настоящему революционным достижением в области шифрования буквально за тысячи лет. Во-первых, алгоритмы с открытым ключом основаны на математических функциях, а не на простых операциях с битовыми шаблонами. Во-вторых, что более важно, криптография с открытым ключом является асимметричной, включающей использование двух отдельных ключей, в отличие от симметричного шифрования, в котором используется только один ключ. Использование двух ключей имеет серьезные последствия в области конфиденциальности, распространения ключей и аутентификации.

Прежде чем продолжить, мы должны упомянуть несколько распространенных заблуждений относительно шифрования с открытым ключом. Первое — шифрование с открытым ключом более надежно защищено от криptoанализа, чем симметричное шифрование. На самом деле безопасность любой схемы шифрования зависит от длины ключа и вычислительной работы, связанной со взломом шифра. В принципе в симметричном шифровании или шифровании с открытым ключом нет ничего, что могло бы превзойти одно в другом с точки зрения сопротивления криptoанализу. Второе заблуждение состоит в том, что шифрование с открытым ключом является универсальной технологией, которая сделала симметричное шифрование устаревшим. Напротив, из-за вычислительных издержек современных схем шифрования с открытым ключом, по-видимому, симметричное шифрование будет продолжать активно использоваться. Наконец, существует мнение, что при использовании шифрования с открытым ключом передача ключей оказывается тривиальной задачей по сравнению с довольно громоздкими протоколами для симметричного шифрования. Но на самом деле необходима некоторая разновидность

протокола, часто с участием центрального агента, так что используемые при асимметричном шифровании процедуры не проще и не эффективнее, чем те, которые требуются симметричному шифрованию.

Схема шифрования с открытым ключом состоит из таких компонентов (рис. Л.2).

- **Открытый текст.** Читаемое человеком сообщение или данные, которые вводятся в алгоритм в качестве входных данных.
- **Алгоритм шифрования.** Выполняет различные преобразования над открытым текстом.
- **Открытый и закрытый (секретный) ключи.** Выбранная пара ключей, и если один ключ используется для шифрования, то другой используется для расшифровки. Преобразования, выполняемые алгоритмом шифрования, зависят от того, какой ключ поступает на его вход — закрытый или открытый.
- **Зашифрованный текст.** Зашифрованное сообщение, созданное в качестве выхода алгоритма. Зависит от открытого текста и ключа. Для данного сообщения два разных ключа будут создавать два разных зашифрованных текста.
- **Алгоритм дешифровки.** Принимает в качестве входных данных зашифрованный текст и соответствующий ключ, и выдает на выходе исходный открытый текст.

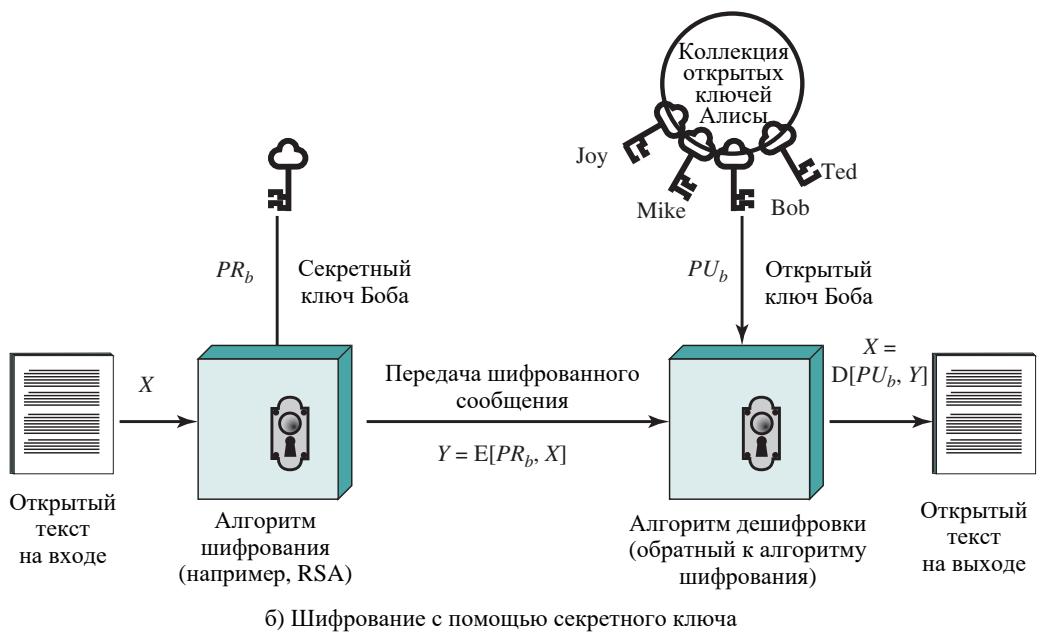
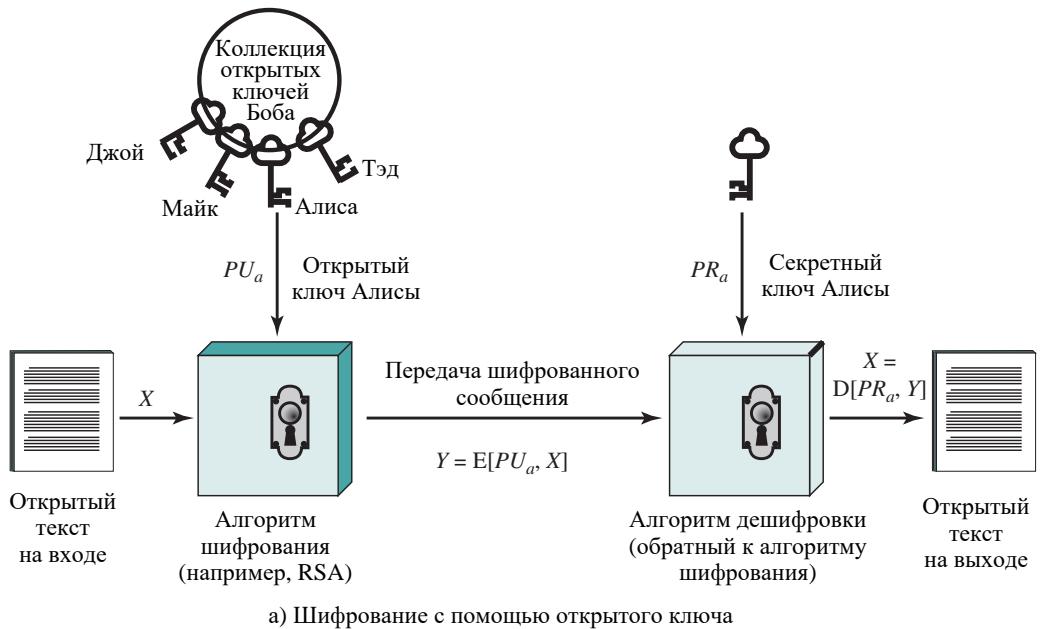
Процесс работает (выдает правильный текст на выходе) независимо от порядка, в котором используется пара ключей. Как следует из названия, открытый ключ пары становится общедоступным для использования другими лицами, а секретный ключ известен только его владельцу.

Теперь, скажем, Боб хочет отправить личное сообщение Алисе, и предположим, что он имеет открытый ключ Алисы, а Алиса имеет соответствующий секретный ключ (рис. Л.2, а). Боб с помощью открытого ключа Алисы шифрует сообщение для создания зашифрованного текста. Зашифрованный текст затем передается Алисе. Алиса, получив зашифрованный текст, расшифровывает его, используя свой секретный ключ. Поскольку закрытый ключ имеет только Алиса, никто другой не может прочитать сообщение.

Шифрование с открытым ключом можно использовать и другим способом, как показано на рис. Л.2, б. Предположим, что Боб хочет отослать сообщение Алисе, и хотя не важно, чтобы это сообщение было сохранено в тайне, важно, чтобы Алиса была уверена, что сообщение действительно от Боба. В этом случае Боб использует свой секретный ключ для шифрования сообщения. Получив зашифрованный текст, Алиса обнаруживает, что может расшифровать его с помощью открытого ключа Боба, а это доказывает, что сообщение должно было быть зашифровано Бобом: ведь ни у кого другого нет секретного ключа Боба, а потому никто другой не мог создать этот зашифрованный текст, который может быть расшифрован с помощью открытого ключа Боба.

Криптографический алгоритм общего назначения с открытым ключом опирается на два ключа: один — для шифрования и другой — связанный ключ для дешифрования. Кроме того, эти алгоритмы имеют следующие важные характеристики.

- Вычислительно невозможно определить ключ дешифрования только на основании знания криптографического алгоритма и ключа шифрования.
- Любой из двух связанных ключей может использоваться для шифрования, другой при этом используется для расшифровки.



**Рис. Л.2.** Шифрование с открытым ключом

Наиболее важными шагами являются следующие.

1. Каждый пользователь генерирует пару ключей, которые будут использоваться для шифрования и дешифрования сообщений.
2. Каждый пользователь помещает один из двух ключей в открытый реестр или другой доступный файл. Это открытый ключ. Второй ключ из пары хранится в секрете. Как показано на рис. Л.2, а, каждый пользователь поддерживает коллекцию открытых ключей, полученных от других.
3. Если Боб хочет отправить личное сообщение Алисе, он шифрует сообщение, используя открытый ключ Алисы.
4. Алиса, получив сообщение, расшифровывает его, используя свой секретный ключ. Никто другой не может расшифровать сообщение, потому что только Алиса знает свой секретный ключ.

При таком подходе все участники имеют доступ к открытым ключам, а секретные ключи генерируются локально каждым участником и никогда не должны быть доступны никому иному. Пока пользователь держит свой секретный ключ в секрете, входящая связь защищена. В любое время пользователь может изменить секретный ключ и опубликовать соответствующий ему открытый ключ для замены старого открытого ключа.

Ключ, используемый в симметричном шифровании, обычно называют **секретным ключом** (*secret key*). Два ключа, используемые для шифрования с открытым ключом, называются **открытым ключом** (*public key*) и **секретным (закрытым) ключом** (*private key*).

## **Алгоритм Ривеста–Шамира–Адлемана (RSA)**

Одна из первых схем с открытым ключом была разработана в 1977 году Роном Ривестом (Ron Rivest), Ади Шамиром (Adi Shamir) и Леном Адлеманом (Len Adleman) из Массачусетского технологического института. Схема RSA с тех пор царила как единственный широко принятый и реализованный подход к шифрованию с открытым ключом. RSA представляет собой шифр, в котором открытый текст и шифротекст являются целыми числами от 0 до  $n - 1$  для некоторого  $n$ . Шифрование включает в себя модульную арифметику. Надежность алгоритма основана на сложности факторизации (разделения на множители) больших чисел.

## **Л.3. АУТЕНТИФИКАЦИЯ СООБЩЕНИЙ И ХЕШ-ФУНКЦИИ**

Шифрование защищает от пассивной атаки (подслушивания). Еще одно требование заключается в защите от активной атаки (фальсификации данных и транзакций). Защита против таких атак известна как аутентификация сообщений.

Аутентификация сообщения представляет собой процедуру, позволяющую взаимодействующим сторонам убедиться, что полученные сообщения являются подлинными. Двумя важными аспектами являются проверка того, что содержание сообщения не было изменено и что источник сообщения также является подлинным. Возможна также проверка своевременности сообщения (что оно не было искусственно задержано) и его последовательности относительно других сообщений, передаваемых между двумя сторонами.

## Аутентификация с использованием симметричного шифрования

Можно выполнить аутентификацию с помощью симметричного шифрования. Если предположить, что совместно используют ключ только отправитель и получатель (как и должно быть), то только подлинный отправитель сможет успешно зашифровать сообщение для другого участника. Кроме того, если сообщение включает в себя код обнаружения ошибок и порядковый номер, получатель может быть уверен, что никаких изменений в сообщении сделано не было и последовательность сообщений получена в правильном порядке. Если сообщение, кроме того, содержит метку времени, то получатель может быть уверен, что сообщение не было задержано сверх ожидаемого времени доставки по сети.

## Аутентификация без шифрования

В этом разделе мы рассмотрим несколько подходов к аутентификации сообщений, которые не полагаются на шифрование. Во всех этих подходах генерируется дескриптор (тег) аутентификации, который добавляется к каждому передаваемому сообщению. Само сообщение не зашифровано и может быть прочитано в месте назначения независимо от результата аутентификации.

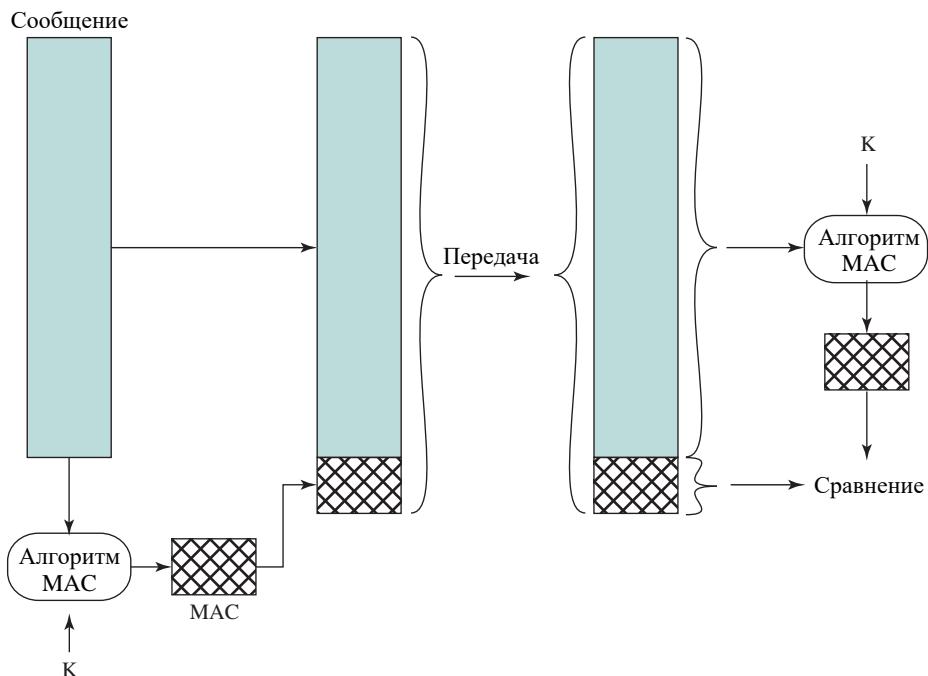
Поскольку подходы, обсуждаемые в этом разделе, не шифруют сообщение, конфиденциальность сообщения не обеспечивается. Но если симметричное шифрование легко доступно и обеспечивает аутентификацию, то почему бы не использовать рассмотренный выше подход, который обеспечивает и конфиденциальность, и аутентификацию? Вот три ситуации, в которых аутентификация сообщения без конфиденциальности предпочтительнее.

1. Существует ряд приложений, в которых одно и то же сообщение передается целому ряду получателей. Примером является уведомление пользователей о том, что сеть сейчас недоступна, или сигнал тревоги в центре управления. Дешевле и надежнее при этом иметь только один пункт назначения, отвечающий за проверку подлинности сообщений. Таким образом, сообщение должно быть передано в виде открытого текста со связанным с ним дескриптором аутентификации сообщения. Ответственная система выполняет аутентификацию и, если обнаруживается нарушение, то другие системы оповещаются с использованием общего сигнала тревоги.
2. Другой возможный сценарий — обмен, в котором одна сторона очень загружена и не может позволить себе расшифровывать все входящие сообщения. Аутентификация проводится на выборочной основе, сообщения для проверки выбираются случайным образом.
3. Аутентификация компьютерной программы, передаваемой без шифрования, оказывается весьма привлекательной: программа может быть выполнена без необходимости расшифровки при каждом запуске, что расточительно в отношении ресурсов процессора. Однако если к программе прикреплен дескриптор аутентификации, то его можно проверять всякий раз, когда требуется убедиться в целостности программы.

Таким образом, в сфере безопасности есть место как для аутентификации с шифрованием, так и без такового.

## Код аутентификации сообщения

Один из методов аутентификации включает использование секретного ключа для генерации небольшого блока данных, известного как код аутентификации сообщения (message authentication code — MAC), который добавляется к сообщению. Этот метод предполагает, что две взаимодействующие стороны, скажем, А и В, совместно используют общий секретный ключ  $K_{AB}$ . Когда у А имеется сообщение  $M$  для его отправки В, он вычисляет код аутентификации сообщения как функцию сообщения и ключа:  $\text{MAC}_M = F(K_{AB}, M)$ . Сообщение вместе с кодом передаются предполагаемому получателю. Получатель выполняет те же вычисления для полученного сообщения, используя тот же секретный ключ, и генерирует новый код аутентификации сообщения. Полученный код сравнивается с вычисленным (рис. Л.3).



**Рис. Л.3.** Аутентификация сообщения с использованием MAC

Если (в предположении, что только получатель и отправитель знают секретный ключ) полученный код соответствует вычисленному, то справедливо следующее.

1. Получатель может быть уверен, что сообщение не было изменено. Если злоумышленник изменит сообщение, но не изменит код, то вычисленный получателем код будет отличаться от полученного. Поскольку предполагается, что злоумышленник не знает секретный ключ, он не может изменить код так, чтобы тот соответствовал измененному сообщению.
2. Получатель может быть уверен, что сообщение получено от предполагаемого отправителя. Поскольку никто иной не знает секретный ключ, никто не может подготовить сообщение с правильным кодом.

3. Если сообщение содержит порядковый номер (такой, как используемый в протоколах X.25, HDLC или TCP), то получатель может быть уверен в правильной последовательности сообщений, потому что злоумышленник не может успешно изменить порядковый номер.

Для генерации кода может быть использован ряд алгоритмов. Национальное бюро стандартов в своей публикации *DES Modes of Operation* рекомендует DES. DES используется для генерации зашифрованной версии сообщения, а некоторое количество завершающих битов зашифрованного текста используется в качестве кода. Типичным является 16- или 32-битный код. Только что описанный процесс похож на шифрование. Единственное отличие состоит в том, что алгоритм аутентификации не должен быть обратимым, как это требуется для дешифрования. Оказывается, из-за математических свойств функций аутентификации они менее уязвимы для взлома, чем шифрование.

## ФУНКЦИЯ ОДНОСТОРОННЕГО ХЕШИРОВАНИЯ

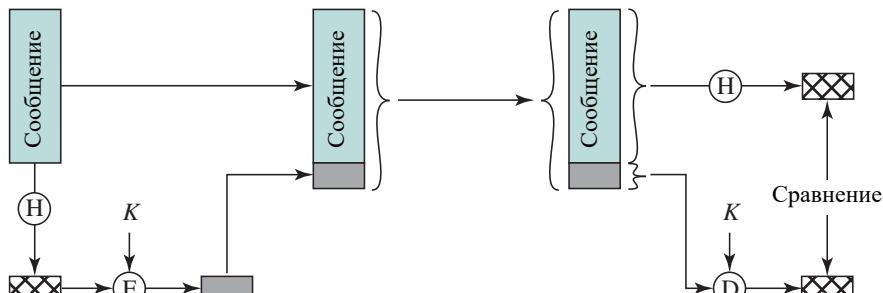
Функция одностороннего хеширования представляет собой вариант кода аутентификации сообщений, которому уделяется много внимания. Как и в коде аутентификации сообщения, в качестве входных данных хеш-функция принимает сообщение переменного размера  $M$  и создает дайджест сообщения фиксированного размера  $H(M)$  в качестве выходных данных. В отличие от MAC, хеш-функция не получает секретный ключ в качестве входных данных. Для проверки подлинности сообщения дайджест сообщения отправляется вместе с сообщением способом, благодаря которому обеспечивается подлинность дайджеста сообщения.

На рис. Л.4 показаны три способа аутентификации сообщения. Дайджест сообщения может быть зашифрован с использованием симметричного шифрования (часть  $a$ ); если это так, то предполагается, что ключ шифрования используется только отправителем и получателем, что и гарантирует аутентификацию. Дайджест сообщения может быть также зашифрован с использованием шифрования с открытым ключом (часть  $b$ ). Подход с открытым ключом имеет два преимущества — обеспечивает как цифровую подпись, так и аутентификацию сообщения и не требует распространения ключей для общения сторон.

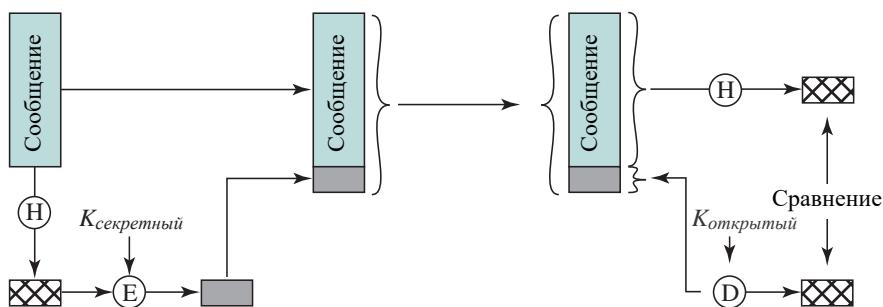
Эти два подхода имеют преимущество перед подходами, которые шифруют все сообщение, так как в этом случае требуется меньше вычислений. Тем не менее было бы интересно разработать методику, которая полностью исключает шифрование. Вот некоторые причины этого интереса.

- Программное обеспечение для шифрования достаточно медленное. Даже если объем шифруемых данных в сообщении мал, может быть большой поток сообщений в систему и из нее.
- Стоимостью аппаратного шифрования нельзя пренебречь. Доступны недорогие реализации чипов DES, но их стоимость умножается на количество узлов в сети, которые должны обладать этой возможностью.
- Аппаратное обеспечение для шифрования оптимизировано для больших объемов данных. Для небольших блоков данных большая часть времени затрачивается на накладные расходы инициализации и вызова.
- Алгоритмы шифрования могут быть защищены патентами и должны быть лицензированы, что вносит свою стоимость.
- Алгоритмы шифрования могут подлежать экспортному контролю.

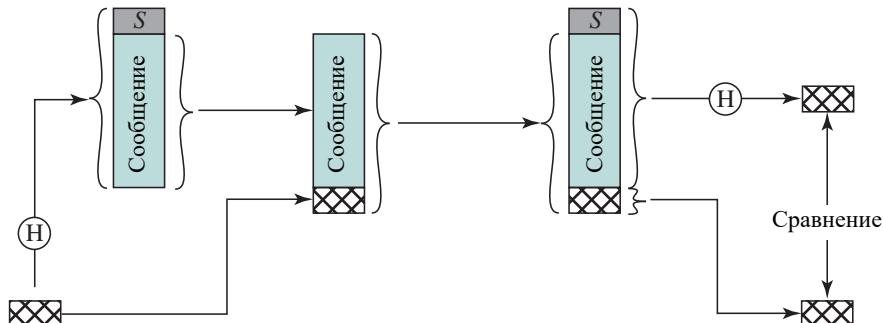
На рис. Л.4, в показан метод, который использует хеш-функцию, но не использует шифрование для аутентификации сообщений. Этот метод предполагает, что две общающиеся стороны, скажем, А и В, совместно используют общее секретное значение  $S_{AB}$ . Когда А хочет отправить сообщение В, он вычисляет хеш-функцию для объединения секретного значения и сообщения:  $MD_M = H(S_{AB} \parallel M)$ <sup>1</sup>. Затем он отправляет В сообщение  $[M \parallel MD_M]$ .



а) Использование симметричного шифрования



б) Использование шифрования с открытым ключом



в) Использование секретного значения

**Рис. Л.4.** Аутентификация сообщения с использованием функции одностороннего хеширования

<sup>1</sup> “||” означает конкатенацию.

Поскольку В знает  $S_{AB}$ , он может вычислить  $H(S_{AB} || M)$  и проверить значение  $MD_M$ . Поскольку само секретное значение не передается, злоумышленник не в состоянии изменить перехваченное сообщение. Пока секретное значение остается действительно секретным, злоумышленник не может сгенерировать ложное сообщение.

Именно этот третий метод, использующий общее секретное значение, принят для IP-безопасности; он также специфицирован и для SNMPv3.

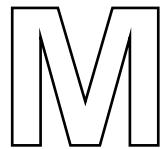
## Л.4. БЕЗОПАСНЫЕ ХЕШ-ФУНКЦИИ

Важным элементом многих служб и приложений безопасности являются безопасные хеш-функции. Хеш-функция принимает сообщение переменного размера  $M$  в качестве входных данных и производит в качестве выходных данных дескриптор фиксированного размера  $H(M)$ , иногда называемый дайджестом сообщения. Для цифровой подписи генерируется хеш-код зашифрованного с помощью секретного ключа отправителя сообщения и отправляется вместе с сообщением. Получатель вычисляет новый хеш-код для входящего сообщения, расшифровывает хеш-код с помощью открытого ключа отправителя и сравнивает их. Если сообщение было изменено в пути, будет обнаружено несоответствие.

Чтобы быть полезной для приложений безопасности, хеш-функция  $H$  должна обладать следующими свойствами.

1.  $H$  может применяться к блоку данных любого размера.
2.  $H$  производит вывод фиксированной длины.
3.  $H(x)$  относительно легко вычислить для любого данного  $x$ , что делает практическими как аппаратную, так программную реализации этой функции.
4. Для любого заданного значения  $h$  вычислительно невозможно найти  $x$ , такое, что  $H(x) = h$ . Это свойство иногда упоминается в литературе как **свойство односторонности** (one-way property).
5. Для любого данного блока  $x$  невозможно вычислить  $y \neq x$ , такое, что  $H(y) = H(x)$ . Это свойство иногда называют **стойкостью к коллизиям первого рода** (weak collision resistance).
6. В вычислительном отношении невозможно найти пару  $(x, y)$ , такую, что  $H(x) = H(y)$ . Это свойство иногда называют **стойкостью к коллизиям второго рода** (strong collision resistance).

В последние годы наиболее широко используемой хеш-функцией была Secure Hash Algorithm (SHA). SHA была разработана NIST и опубликована в качестве федерального стандарта обработки информации (FIPS 180) в 1993 году. Когда в SHA были обнаружены недостатки, была выпущена пересмотренная версия FIPS 180-1 в 1995 году (обычно упоминается как SHA-1). SHA-1 выдает хеш-значение размером 160 бит. В 2002 году NIST выпустил пересмотренную версию стандарта FIPS 180-2, в которой определены три новые версии SHA, с длинами хеш-значений 256, 384 и 512 бит, известные как SHA-256, SHA-384 и SHA-512. Эти новые версии имеют одинаковую базовую структуру и используют те же типы модульных арифметических и логических бинарных операций, что и SHA-1. В 2005 году NIST объявил о намерении отказаться от утверждения SHA-1 и полностью перейти на другие версии SHA к 2010 году, так как исследования продемонстрировали, что SHA-1 гораздо слабее, чем предполагается исходя из длины хеша в 160 бит.



## ПРИЛОЖЕНИЕ

# ВВЕДЕНИЕ В ПРОГРАММИРОВАНИЕ СОКЕТОВ

В ЭТОМ ПРИЛОЖЕНИИ...

### **М.1. Сокеты, дескрипторы, порты и соединения**

#### **М.2. Модель “клиент/сервер”**

Запуск программы с использованием сокетов на компьютере под управлением Windows, не подключенном к сети

Запуск программы с использованием сокетов на компьютере под управлением Windows, подключенном к сети, когда и сервер, и клиент находятся на одной машине

#### **М.3. Работа с сокетами**

Создание сокета

Адрес сокета

Привязка к локальному порту

Представление данных и порядок байтов

Подключение сокета

Функция `gethostbyname()`

Прослушивание входящих соединений

Прием соединения от клиента

Отправка и получение сообщения через сокет

Закрытие сокета

Сообщение об ошибках

Пример клиентской программы TCP/IP (инициация соединения)

Пример серверной программы TCP/IP (пассивное ожидание соединения)

#### **М.4. Сокеты потоков и дейтаграмм**

Пример клиентской программы UDP (инициация соединения)

Пример серверной программы UDP (пассивное ожидание соединения)

#### **М.5. Управление программой времени выполнения**

Неблокирующие вызовы сокетов

Асинхронный ввод-вывод (ввод-вывод, управляемый сигналом)

#### **М.6. Удаленное выполнение консольного приложения Windows**

Локальный код

Удаленный код

Концепция сокетов и их программирования была разработана в 1980-х годах в среде UNIX в виде интерфейса сокетов Беркли (Berkeley Sockets Interface). По сути, сокет обеспечивает связь между процессами клиента и сервера и может быть ориентирован на установленное соединение или без установления такового. Сокет можно считать конечным узлом при передаче сообщений. Клиентский сокет на одном компьютере использует для вызова серверного сокета на другом компьютере соответствующий адрес. Как только сокеты соединены, оба компьютера могут обмениваться данными.

Как правило, компьютеры с серверными сокетами поддерживают открытый порт TCP или UDP, готовый к незапланированным входящим вызовам. Клиент обычно идентифицирует сокет требуемого сервера, находя его в базе данных системы доменных имен (DNS). После установления соединения сервер переключает диалог на другой номер порта, чтобы освободить номер основного порта для дополнительных входящих вызовов.

Интернет-приложения, такие как TELNET или удаленный вход (rlogin), используют сокеты; при этом детали их применения скрыты от пользователя. Однако сокеты могут быть созданы и внутри программы (на языке программирования, таком как C или Java), что позволяет программисту легко поддерживать сетевые функции и приложения. Механизм программирования сокетов включает в себя достаточную семантику, обеспечивающую взаимодействие несвязанных процессов на разных хостах.

Интерфейс сокетов Berkeley де-факто представляет собой стандартный интерфейс прикладного программирования (API) для разработки сетевых приложений, охватывающий широкий спектр операционных систем. API сокетов предоставляет обобщенный доступ к службам межпроцессной коммуникации. Таким образом, функциональные возможности сокетов идеально подходят студентам для изучения принципов работы протоколов и распределенных приложений путем практической разработки программ.

API сокетов предоставляет библиотеку функций, которые программисты могут использовать для разработки приложений с поддержкой сети. API обладает функциями определения конечных точек соединения, установления связи, разрешения отправки сообщений, ожидания входящих сообщений, прекращения связи и обработки ошибок. Используемая операционная система и язык программирования определяют конкретный Sockets API.

Мы сконцентрируемся только на двух наиболее широко используемых интерфейсах — Berkley Software Distribution Sockets (BSD) для UNIX и его незначительной модификации Windows Sockets (WinSock) API от Microsoft.

Представленный здесь материал предназначен для программистов на языке C. (В нем также имеются ссылки на языки C++, Visual Basic и Pascal.) В центре нашего обсуждения находится операционная система Windows. В то же время рассматриваются темы из исходной спецификации BSD UNIX, чтобы указать (обычно незначительные) различия в спецификациях сокетов для двух операционных систем. Предполагается наличие у читателя базовых знаний сетевых протоколов TCP/IP и UDP. Большая часть кода будет компилироваться как в Windows, так и в UNIX-подобных системах.

Мы рассматриваем исключительно сокеты языка C, но большинство других языков программирования, таких как C++, Visual Basic и PASCAL, также могут использовать WinSock API. Единственным требованием является то, что язык должен уметь работать с динамически связываемыми библиотеками (DLL). Чтобы воспользоваться API-интерфейсом WinSock в 32-разрядной среде Windows, вам нужно будет импортировать библиотеку ws2\_32.dll. Эта библиотека должна быть скомпонована так, чтобы во время выполнения программы загружалась динамически связываемая библиотека ws2\_32.dll. Она работает через стек TCP/IP. Операционные системы Windows NT,

Windows 2000 и Windows 95 по умолчанию включают файл wsock32.dll. Когда вы создаете свои выполнимые файлы и связываете их с библиотекой wsock32.lib, вы неявно связываете программу с wsock32.dll во время выполнения, не добавляя для этого дополнительные строки кода в свой исходный файл.

На сайте книги содержатся ссылки на полезные сайты, посвященные сокетам.

## **М.1. Сокеты, дескрипторы, порты и соединения**

Сокеты представляют собой конечные точки соединений, обращение к которым происходит с помощью соответствующих дескрипторов сокетов, или, говоря проще, с помощью некоторых данных, описывающих связь сокета с конкретным компьютером или приложением (например, мы можем обращаться к сокету сервера как к `server_s`). Соединение (или пара сокетов) состоит из пары IP-адресов, которые обмениваются данными между собой, а также пары номеров портов, в которой номер порта представляет собой 16-разрядное положительное целое число, обычно записываемое в десятичном виде. Некоторые целевые номера портов хорошо известны и указывают тип подключаемой службы.

Для многих приложений среда TCP/IP предполагает, что приложения используют общизвестные порты для связи друг с другом. Это делается для того, чтобы клиентские приложения могли работать в предположении, что соответствующее серверное приложение прослушивает общизвестный порт, связанный с этим приложением. Например, номером порта для протокола HTTP, используемого для передачи HTML-страниц в World Wide Web, является TCP-порт 80. По умолчанию веб-браузер пытается открыть соединение с TCP-портом 80 целевого хоста, если только в URL не указан другой номер порта (например, 8000 или 8080).

*Порт* идентифицирует точку подключения в локальном стеке (например, порт 80 обычно используется веб-сервером). *Сокет* идентифицирует пару, состоящую из IP-адреса и номера порта (например, порт 192.168.1.20:80 является портом 80 веб-сервера на хосте с адресом 192.168.1.20; вместе адрес и порт считаются сокетом). *Пара сокетов* идентифицирует четыре компонента соединения (адрес и порт источника, а также целевые адрес и порт). Поскольку известные порты являются уникальными, они иногда используются для ссылки на конкретное приложение на любом хосте, на котором это приложение может выполняться. Однако использование слова “сокет” подразумевает конкретное приложение на некотором конкретном хосте. Соединение, или *пара сокетов*, обозначает соединение сокетов между двумя конкретными системами, которые обмениваются данными. TCP допускает несколько одновременных подключений, использующих один и тот же номер локального порта, при условии, что удаленные IP-адреса или номера портов различны для каждого подключения.

Номера портов делятся на три диапазона.

- Порты с 0 по 1023 — общизвестные, статически связанные со службами. Например, HTTP-серверы всегда принимают запросы через порт 80.
- Номера портов от 1024 до 49151 являются зарегистрированными. Они используются для нескольких целей.
- Динамические и частные порты — это порты с номерами от 49152 по 65535, и никакие службы не должны быть с ними связаны.

В действительности машины начинают назначать динамические порты начиная с 1024. Если вы разрабатываете протокол или приложение, которое потребует использования канала, сокета, порта, протокола и так далее, то свяжитесь с интернет-администрацией по назначению номеров (Internet Assigned Numbers Authority — IANA) для получения номера порта. IANA находится в Институте информационных наук Университета Южной Калифорнии и управляет им. Так называемый “запрос комментария” (RFC), опубликованный IANA, является официальной спецификацией, в которой перечислены все назначения портов. Вы можете получить к нему доступ по адресу <http://www.iana.org/assignments/port-numbers>.

Как в UNIX, так и в Windows для проверки состояния всех активных локальных сокетов может использоваться команда netstat. На рис. М.1 показан пример вывода netstat.

<b>Proto</b>	<b>Local Address</b>	<b>Foreign Address</b>	<b>State</b>
TCP	Mycomp:1025	Mycomp:0	LISTENING
TCP	Mycomp:1026	Mycomp:0	LISTENING
TCP	Mycomp:6666	Mycomp:0	LISTENING
TCP	Mycomp:6667	Mycomp:0	LISTENING
TCP	Mycomp:1234	mycomp:1234	TIME WAIT
TCP	Mycomp:1025	2hfc327.any.com:6667	ESTABLISHED
TCP	Mycomp:1026	46c311.any.com:6668	ESTABLISHED
UDP	Mycomp:6667	*.*	

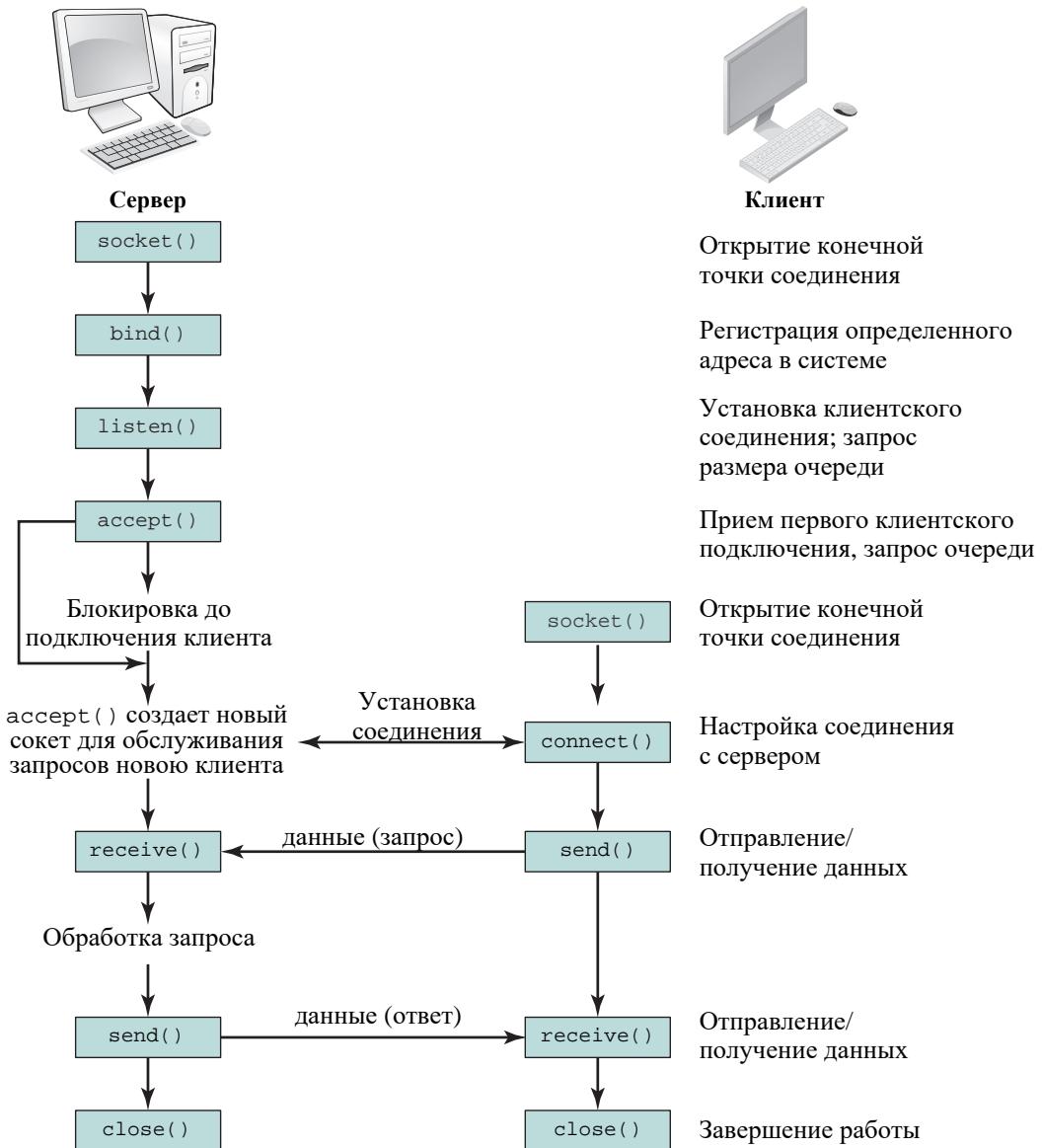
Рис. М.1. Пример вывода netstat

## М.2. Модель “клиент/сервер”

Приложение с использованием сокета состоит из кода, выполняемого на обоих концах соединения. Инициирующая передачу программа часто упоминается как клиент. Сервер же — это программа, которая пассивно ожидает входящих соединений от удаленных клиентов. Серверные приложения обычно загружаются во время запуска системы и активно прослушивают входящие соединения на их общеизвестном порту. Клиентские приложения будут пытаться подключиться к серверу, и тогда будет иметь место обмен TCP. По завершении сеанса обычно клиент должен разорвать соединение. На рис. М.2 изображена базовая модель потокового соединения (или соединения сокетов TCP/IP).

### Запуск программы с использованием сокетов на компьютере под управлением Windows, не подключенном к сети

Если поддержка TCP/IP установлена на одном компьютере, вы можете запускать на нем как сервер, так и код клиента. (Если у вас не установлен стек протоколов TCP/IP, следует ожидать, что операции с сокетами генерируют исключения, такие как `BindException`, `ConnectException`, `ProtocolException`, `SocketException` и т.п.) В качестве имени хоста следует использовать `localhost`, а в качестве IP-адреса — `127.0.0.1`.



**Рис. М.2.** Системные вызовы сокетов для ориентированного на соединения протокола

## Запуск программы с использованием сокетов на компьютере под управлением Windows, подключенном к сети, когда и сервер, и клиент находятся на одной машине

В таком случае вы будете общаться с самим собой. Важно знать, подключен ли ваш компьютер к сети Ethernet или обменивается данными с сетью через телефонный модем. В первом случае у вас будет IP-адрес, назначенный вашей машине, без усилий с вашей стороны. Когда вы общаетесь через modem, вам нужно соединиться по телефонной линии, получить IP-адрес, а затем “говорить с самим собой”. В обоих случаях вы можете узнать IP-адрес компьютера, который будете использовать, с помощью команд winipcfg для Win9X и ipconfig — для WinNT/2K и UNIX.

## M.3. РАБОТА С СОКЕТАМИ

### Создание сокета

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain, int type, int protocol)
```

- **domain.** Может принимать значения AF\_UNIX, AF\_INET, AF\_OSI и т.д. Значение AF\_INET предназначено для связи в Интернете с IP-адресом. Мы будем использовать только значение AF\_INET.
- **type.** Равен SOCK\_STREAM (TCP, ориентированный на соединение, надежный), SOCK\_DGRAM (UDP, дейтаграмма, ненадежный) либо SOCK\_RAW (уровень IP).
- **protocol.** Указывает используемый протокол. Обычно это значение 0, говорящее, что мы хотим использовать для выбранного домена и типа протокол по умолчанию. Мы всегда используем 0.

В случае успеха `socket()` возвращает дескриптор сокета, который является неотрицательным целым числом, и -1 — в случае неудачи. Вот пример вызова:

```
if ((sd = socket(AF_INET, SOCK_DGRAM, 0) < 0)
{
    printf(socket() failed.);
    exit(1);
}
```

### Адрес сокета

Структура для хранения адресов сокетов, используемых в домене AF\_INET:

```
struct in_addr {
    unsigned long s_addr;
};
```

`in_addr` просто предоставляет имя (`s_addr`) для типа языка C, связанного с IP-адресами.

```

struct sockaddr_in {
    unsigned short sin_family;      // Идентификатор AF_INET
    unsigned short sin_port;        // Номер порта, если 0 -
                                    // выбирается ядром
    struct in_addr sin_addr;       // IP-адрес
                                    // INADDR_ANY ссылается на
                                    // IP-адреса текущего хоста
    char sin_zero[8];              // Не используется, всегда 0
};

```

Локальные и удаленные адреса объявляются как структура `sockaddr_in`. В зависимости от этого объявления `sin_addr` будет представлять локальный или удаленный IP-адрес. (В UNIX-подобных системах для обеих структур нужно включить файл `<netinet/in.h>`.)

## Привязка к локальному порту

```

#define WIN                  // WIN для WinSock и BSD для сокетов BSD
#ifndef WIN

    ...

#include <windows.h>      // Для всех функций WinSock
    ...

#endif
#ifndef BSD

    ...

#include <sys/types.h>
#include <sys/socket.h> // Для структуры sockaddr
    ...

#endif
int bind(int local_s, const struct sockaddr * addr, int addrlen);

```

- `local_s`. Дескриптор локального сокета, созданного функцией `socket()`.
- `addr`. Указатель на (локальную) структуру адреса этого сокета.
- `addrlen`. Длина (в байтах) структуры, на которую ссылается `addr`.

`bind()` возвращает целое значение 0 при успехе и -1 при ошибке. После вызова `bind()` номер локального порта связан с сокетом, но еще не указан удаленный пункт назначения.

Вот пример вызова.

```

struct sockaddr_in name;
...
name.sin_family = AF_INET;    // Использование домена Интернета
name.sin_port = htons(0);     // Порт выбирает ядро
name.sin_addr.s_addr = htonl(INADDR_ANY); // Используются все
                                            // IP-адреса хоста
if (bind(local_socket, (struct sockaddr *)&name,
         sizeof(name)) != 0)
    // Вывод сообщения об ошибке и выход

```

Такая привязка является необязательной на стороне клиента, но необходима на стороне сервера. После вызова `bind()` для сокета мы можем получить его адресную структуру по дескриптору файла сокета, используя функцию `getsockname()`.

## Представление данных и порядок байтов

Некоторые компьютеры используют обратный порядок байтов (big endian). Порядок байтов определяет, как именно в памяти располагаются объекты, такие как целые числа в слове. Машина с обратным порядком байтов хранит их следующим образом: старший байт целого числа сохраняется в крайнем слева байте, а младший байт числа — в крайнем справа байте. Таким образом, число  $5 \times 2^{16} + 6 \times 2^8 + 4$  будет храниться следующим образом.

<b>Обратный порядок байтов</b>		5	6	4
<b>Прямой порядок байтов</b>	4	6	5	
<b>Адреса байтов</b>	0	1	2	3

Как видите, чтение значения с неправильным размером слова приведет к неверному значению в архитектуре с обратным порядком байтов; на машине с прямым порядком байтов иногда может вернуться правильный результат.

Sun Sparc является машиной с обратным порядком байтов. Когда она связывается с PC (где применяется прямой порядок байтов), получается следующая нестыковка: PC будет интерпретировать  $5 \times 2^{16} + 6 \times 2^8 + 4$  как  $4 \times 2^{16} + 6 \times 2^8 + 5$ . Чтобы избежать таких ситуаций, протокол TCP/IP определяет машинно-независимый стандарт для порядка байтов — порядок байтов сети. В пакете TCP/IP первым передается самый старший байт. Поскольку обратный порядок байтов представляет собой хранение старшего байта по самому низкому адресу памяти, который является адресом данных, TCP/IP определяет порядок байтов в сети как обратный порядок байтов.

WinSock использует сетевой порядок байтов для различных значений. Функции `htonl()`, `htons()`, `ntohl()`, `ntohs()` гарантируют, что в вызовах WinSock используется правильный порядок байтов независимо от того, какой порядок байтов использует компьютер: прямой или обратный.

Следующие функции используются для преобразования из порядка байтов хоста в порядок байтов сети перед передачей данных и из порядка байтов сети в порядок байтов хоста после приема.

- `unsigned long htonl (unsigned long n)`. Преобразование 32-битного значения из порядка байтов хоста в порядок байтов сети.
- `unsigned short htons (unsigned short n)`. Преобразование 16-битного значения из порядка байтов хоста в порядок байтов сети.
- `unsigned long ntohl (unsigned long n)`. Преобразование 32-битного значения из порядка байтов сети в порядок байтов хоста.
- `unsigned short ntohs (unsigned short n)`. Преобразование 16-битного значения из порядка байтов сети в порядок байтов хоста.

## ПОДКЛЮЧЕНИЕ СОКЕТА

Удаленный процесс идентифицируется по IP-адресу и номеру порта. Вызов `connect()` в локальной системе пытается установить соединение с удаленным адресатом. Это требуется в случае связи, ориентированной на соединения, такой как потоковые сокеты (TCP/IP). Иногда мы также вызываем `connect()` для сокетов дейтаграмм. Причина в том, что целевой адрес сохраняется локально, поэтому нам не нужно указывать его каждый раз, когда мы отправляем дейтаграммы, и таким образом, мы сможем использовать системные вызовы `send()` и `recv()` вместо `sendto()` и `recvfrom()`. Однако такие сокеты нельзя использовать для приема дейтаграмм с других адресов.

```
#define WIN                                // WIN для WinSock и BSD для BSD sockets
#ifndef WIN
#include <windows.h>          // Для всех функций WinSock
#endif
#ifndef BSD
#include <sys/types.h>    // Для системно определенных идентификаторов
#include <netinet/in.h> // Для структуры интернет-адреса
#include <sys/socket.h> // Для socket(), bind() и т.п...
#endif

int connect(int local_s, const struct sockaddr * remote_addr,
            int rmtaddr_len)
```

- `local_s`. Локальный дескриптор сокета.
- `remote_addr`. Указатель на адрес протокола другого сокета.
- `rmtaddr_len`. Длина структуры адреса в байтах.

Возвращаемое значение — целое число 0 (в случае успеха). Функция в Windows возвращает ненулевое значение, указывающее ошибку, в то время как в UNIX функция возвращает в таком случае отрицательное значение.

Вот пример вызова.

```
#define PORT_NUM 1050                      // Произвольный номер порта
struct sockaddr_in serv_addr;              // Интернет-адрес сервера
int rmt_s;                                // Дескриптор удаленного
                                           // сокета

// Заполнение информации удаленного адреса сокета
// и подключение к прослушивающему серверу.
server_addr.sin_family = AF_INET;          // Семейство адресов
server_addr.sin_port = htons(PORT_NUM);     // Номер порта
server_addr.sin_addr.s_addr
    = inet_addr(inet_ntoa(address)); // IP-адрес
if (connect(rmt_s,(struct sockaddr *)&serv_addr,
            sizeof(serv_addr)) != 0)
// Вывод сообщения об ошибке
```

## ФУНКЦИЯ `gethostbyname()`

Функция `gethostbyname()` получает в качестве аргумента имени хоста и возвращает NULL в случае сбоя или указатель на экземпляр `struct hostent` в случае успеха. В нем содержится информация об именах хостов, псевдонимах и IP-адресах. Эта информация получается из DNS или локальной базы данных конфигурации. `getservbyname()` определяет номер порта, связанный с именованным сервисом. Если вместо этого предоставляется числовое значение, оно преобразуется непосредственно в бинарное значение, используемое в качестве номера порта.

Другие функции, которые можно использовать для поиска хостов, служб, протоколов или сетей: `getpeername()`, `gethostbyaddr()`, `getprotobynumber()`, `getprotoent()`, `getservbyname()`, `getservbyport()`, `getservent()`, `getnetbyname()`, `getnetbynumber()`, `getnetent()`.

Вот пример вызова.

```
#ifdef BSD
. . .
#include <sys\types.h>           // Для типа caddr_t
. . .
#endif

#define SERV_NAME    somehost.somecompany.com
#define PORT_NUM     1050          // Произвольный номер порта
#define h_addr h_addr_list[0]      // Для хранения интернет-адреса
. . .
struct sockaddr_in myhost_addr;   // Интернет-адрес
struct hostent *hp;              // Буфер с информацией
                                // об удаленном хосте
int rmt_s;                      // Дескриптор удаленного сокета

bzero((char*)&myhost_addr, sizeof(myhost_addr));

    // Специфично для WinSock
memset( &myhost_addr, 0, sizeof(myhost_addr) );

    // Заполнение информации об адресе удаленного сокета
    // и подключение к прослушивающему серверу
myhost_addr.sin_family = AF_INET;        // Семейство адреса
myhost_addr.sin_port = htons(PORT_NUM);  // Номер порта

if (hp = gethostbyname(MY_NAME) == NULL)
    // Вывод сообщения об ошибке
```

```
// Часть, специфичная для UNIX
bcopy(hp->h_name, (char *)&myhost_addr.sin_addr,
      hp->h_length);

// Часть, специфичная для WinSock
memcpuy( &myhost_addr.sin_addr, hp->h_addr, hp->h_length );
if(connect(rmt_s,(struct sockaddr *)&myhost_addr,
           sizeof(myhost_addr))!=0)
    // Вывод сообщения об ошибке
```

**Функция UNIX bzero()** обнуляет буфер указанной длины. Это одна из группы функций для работы с массивами байтов. **bcopy()** копирует указанное число байтов из буфера-источника в целевой буфер. **bcmp()** сравнивает указанное количество байтов из двух байтовых буферов. Функции **UNIX bzero()** и **bcopy()** в **WinSock** недоступны, поэтому необходимо использовать **ANSI-функции** **memset()** и **memcpuy()**.

Пример программы с сокетами для получения IP-адреса хоста с данным именем.

```
#define WIN                                // WIN для WinSock, BSD для BSD sockets
#include <stdio.h>                          // Необходимо для printf()
#include <stdlib.h>                         // Необходимо для exit()
#include <string.h>                          // Необходимо для memcpuy() and strcpy()
#endif WIN
#include <windows.h>                         // Необходимо для WinSock
#endif
#ifndef BSD
#include <sys/types.h>                        // Для системных идентификаторов
#include <netinet/in.h>                       // Для структуры интернет-адреса
#include <arpa/inet.h>                         // Для inet_ntoa
#include <sys/socket.h>                        // Для socket(), bind() и др.
#include <fcntl.h>
#include <netdb.h>
#endif

void main(int argc, char* argv[])
{
#ifndef WIN
    WORD wVersionRequested = MAKEWORD(1, 1);
    // Для функций WSA
    WSADATA wsaData;
#endif
    struct hostent* host;          // Структура для gethostbyname()
    struct in_addr address;       // Структура для интернет-адреса
    char host_name[256];          // Стока для имени хоста

    if (argc != 2)
    {
        printf("**** ОШИБКА - неверное количество "
               "аргументов командной строки\n");
        printf("Использование: 'getaddr host_name' \n");
        exit(1);
    }
```

```

#ifndef WIN
    // Инициализация winsock
    WSAStartup(wVersionRequested, &wsaData);
#endif
    // Копирование имени хоста в host_name
    strcpy(host_name, argv[1]);
    // Вызов gethostbyname()
    printf("Поиск IP-адреса для '%s'... \n", host_name);
    host = gethostbyname(host_name);
    // Вывод найденного адреса
    if (host == NULL)
        printf("IP-адрес для '%s' не найден\n", host_name);
    else
    {
        memcpy(&address, host->h_addr, 4);
        printf("IP-адрес для '%s' - %s\n", host_name,
               inet_ntoa(address));
    }
#endif WIN
// Очистка winsock
WSACleanup();
#endif
}

```

## Прослушивание входящих соединений

Функция `listen()` используется на сервере в случае связи с установлением соединения для подготовки сокета для приема сообщений от клиентов. Вот ее прототип.

- ```
int listen (int sd, int qlen);
```
- `sd`. Дескриптор сокета после вызова `bind()`.
  - `qlen`. Указывает максимальное количество входящих запросов на соединение, которые могут дожидаться обработки сервером, пока сервер занят.

Вызов `listen()` возвращает целое число: 0 — при успехе и -1 — при ошибке, например:

```
if (listen(sd, 5) < 0) {
    // Вывод сообщения об ошибке
```

## Прием соединения от клиента

Функция `accept()` используется на сервере при использовании связи с установлением соединения (после вызова метода `listen()`), чтобы принять запрос на подключение от клиента.

```
#define WIN // WIN для WinSock, BSD для BSD sockets
#ifndef WIN
...
#endif
#include <windows.h> // Для функций WinSock
...
#endif
#ifndef BSD
...
#include <sys/types.h>
#include <sys/socket.h> // Для struct sockaddr
...
#endif
int accept(int server_s, struct sockaddr * client_addr,
           int * clntaddr_len)
```

- `server_s`. Дескриптор сокета, которым сервер прослушивает сеть.
- `client_addr`. Будет заполнен адресом клиента.
- `clntaddr_len`. Содержит длину структуры адреса клиента.

Функция `accept()` возвращает целое число, представляющее новый сокет (-1 в случае сбоя).

После выполнения приложением принимается первое входящее соединение в очередь, при этом создается и возвращается новый сокет — сокет, который сервер будет использовать для связи с данным клиентом. Несколько успешных вызовов `connect()` ведут к возврату нескольких новых сокетов.

Вот пример вызова.

```
struct sockaddr_in client_addr;
int server_s, client_s, clntaddr_len;
...
if ((client_s = accept(server_s, (struct sockaddr *)&
                       client_addr, &clntaddr_len) < 0)
    // Вывод сообщения об ошибке
```

Последовательные вызовы для одного и того же прослушивающего сокета возвращают разные подключенные сокеты, которые мультиплексируются на одном и том же порту сервера функциями стека TCP.

## Отправка и получение сообщения через сокет

В этом разделе мы представим только четыре вызова функций. Однако на самом деле имеется больше четырех способов отправки и получения данных через сокеты. Типичными функциями для сокетов TCP/IP являются `send()` и `recv()`.

```
int send(int socket, const void * msg,
         unsigned int msg_length, int flags);
int recv(int socket, void * rcv_buff,
         unsigned int buff_length, int flags);
```

- `socket`. Локальный сокет, используемый для отправки и получения.
- `msg`. Указатель на сообщение.

- `msg_length`. Длина сообщения.
- `recv_buff`. Указатель на приемный буфер.
- `buff_length`. Его длина.
- `flags`. Изменяет поведение вызова по умолчанию.

Например, определенное значение флагов может использоваться, чтобы указать, что сообщение отправляется без использования локальных таблиц маршрутизации (используемых по умолчанию).

Типичные функции для UDP-сокетов таковы.

```
int sendto(int socket, const void * msg, unsigned int msg_length,
           int flags, struct sockaddr * dest_addr,
           unsigned int addr_length);
int recvfrom(int socket, void * rcv_buff, unsigned int buff_length,
             int flags, struct sockaddr * src_addr,
             unsigned int addr_length);
```

Большинство параметров здесь такие же, как для `send()` и `recv()`, за исключением `dest_addr/src_addr` и `addr_length`. В отличие от потоковых сокетов, вызывающим `sendto()` функциям для дейтаграмм необходимо указывать адрес получателя для отправки сообщения, а вызывающим `recvfrom()` функциям нужно различать разные источники, отправляющие дейтаграммы вызывающему приложению. Мы предоставляем код для TCP/IP- и UDP-клиента и серверного приложения в следующих разделах, где вы сможете найти примеры вызовов всех четырех функций.

## ЗАКРЫТИЕ СОКЕТА

Вот прототип функции.

```
int closesocket(int sd);      // Windows
int close(int fd);          // BSD UNIX
```

`fd` и `sd` представляют собой дескриптор файла (такой же, как дескриптор сокета в UNIX) и дескриптор сокета.

Когда закрывается сокет на каком-либо надежном протоколе, таком как TCP/IP, ядро по-прежнему будет пытаться отправить все непереданные данные, так что соединение переходит в состояние TIME\_WAIT (см. рис. Л.1). Если приложение выбирает тот же номер порта для подключения, может возникнуть следующая ситуация. Когда это удаленное приложение вызывает `connect()`, локальное приложение предполагает, что существующее соединение все еще активно, и рассматривает входящее соединение как попытку продублировать существующее соединение. В результате возвращается ошибка [WSA]ECONNREFUSED. Операционная система поддерживает счетчик ссылок для каждого активного сокета. Вызов `close()`, по сути, уменьшает этот счетчик для переданного в качестве аргумента сокета. Это важно помнить, когда мы используем один и тот же сокет в нескольких процессах. Мы предоставим несколько примеров вызовов в фрагментах кода в следующих разделах.

## Сообщение об ошибках

Все предыдущие операции с сокетами при выполнении могут оказываться сбоями. Считается хорошей практикой программирования сообщать об этом с помощью кода возвращаемой ошибки. Большинство из этих ошибок предназначены для помощи разработчику при отладке процесса; некоторые из них могут быть выведены и для пользователя. В Windows все возвращаемые ошибки определены в заголовочном файле `winsock.h`. В UNIX-подобных системах эти определения можно найти в заголовочном файле `socket.h`. Коды в Windows вычисляются путем добавления 10000 к исходному номеру ошибки BSD и имеют префикс `WSA` перед именем ошибки BSD, например:

| Имя Windows   | Имя BSD    | Значение в Windows | Значение в BSD |
|---------------|------------|--------------------|----------------|
| WSAEPROTOTYPE | EPROTOTYPE | 10041              | 41             |

Имеются несколько специфичных для Windows ошибок, отсутствующих в UNIX.

|                    |       |                                                                                                                     |
|--------------------|-------|---------------------------------------------------------------------------------------------------------------------|
| WSASYSNOTREADY     | 10091 | Возвращается WSAStartup(), указывая, что сетевая подсистема непригодна для использования                            |
| WSAVERNOTSUPPORTED | 10092 | Возвращается WSAStartup(), указывая, что Windows Sockets DLL не в состоянии поддерживать данное приложение          |
| WSANOTINITIALISED  | 10093 | Возвращается любой функцией, кроме WSAStartup(), если ранее не было успешного инициализирующего вызова WSAStartup() |

Вот как выглядит пример исходного текста с перехватом ошибок, выводящего сообщение об ошибке и завершающего программу.

```

if (err == WSANO_RECOVERY)
    strcpy(err_descr, "WSANO_RECOVERY(11003) "
           "Невосстановимая ошибка.");

if (err == WSATRY AGAIN)
    . . .

fprintf(stderr, "%s: %s\n", program_msg, err_descr);
exit(1);
}

```

Можно расширить список ошибок, которые будут использоваться вашим приложением WinSock, обратившись по адресу <http://www.sockets.com>.

## Пример клиентской программы TCP/IP (инициация соединения)

Эта клиентская программа предназначена для получения одного сообщения от сервера; после получения сообщения она отправляет подтверждение на сервер, а затем завершается.

```

#define WIN                      // WIN для WinSock, BSD для BSD sockets
#include <stdio.h>            // Для printf()
#include <string.h>           // Для memcpuy() и strcpy()
#ifndef WIN
    #include <windows.h>       // Для вызовов WinSock
#endif
#ifndef BSD
    #include <sys/types.h>      // Для системных идентификаторов
    #include <netinet/in.h>     // Для структуры интернет-адреса
    #include <sys/socket.h>     // Для socket(), bind() и др.
    #include <arpa/inet.h>      // Для inet_ntoa()
    #include <fcntl.h>
    #include <netdb.h>
#endif
#define PORT_NUM 1050           // Номер порта на сервере
#define IP_ADDR "131.247.167.101" // "Прошитый" адрес сервера

int main()
{
#ifndef WIN
    WORD wVersionRequested = MAKEWORD(1, 1);    // Функции WSA
    WSADATA wsaData;                            // Функции WSA
#endif
    unsigned int server_s;                      // Дескриптор сокета сервера
    struct sockaddr_in server_addr;             // Интернет-адрес сервера
    char out_buf[100];                          // 100-байтный выходной буфер данных
    char in_buf[100];                           // 100-байтный входной буфер данных
#ifndef WIN
    WSASStartup(wVersionRequested, &wsaData);
#endif
    // Создание сокета
    server_s = socket(AF_INET, SOCK_STREAM, 0);

```

```

// Заполнение сокета и соединение с сервером.
// Вызов connect() блокирующий
server_addr.sin_family = AF_INET;           // Семейство адреса
server_addr.sin_port = htons(PORT_NUM);      // Номер порта
server_addr.sin_addr.s_addr = inet_addr(IP_ADDR); // IP-адрес
connect(server_s, (struct sockaddr*)&server_addr,
        sizeof(server_addr));

// Получение данных с сервера
recv(server_s, in_buf, sizeof(in_buf), 0);
printf("Получено с сервера... data = `%s`\n", in_buf);

// Отправка данных на сервер
strcpy(out_buf, "Сообщение от клиента серверу");
send(server_s, out_buf, (strlen(out_buf) + 1), 0);

// Закрытие открытого сокета
#endif WIN
    closesocket(server_s);
#endif
#ifndef BSD
    close(server_s);
#endif
#endif WIN // Очистка WinSock
    WSACleanup();
#endif
}

```

## Пример серверной программы TCP/IP (пассивное ожидание соединения)

Все, что делает следующая серверная программа, — это отправка сообщения клиенту, работающему на другом хосте. Программа создает один сокет в строке и прослушивает один входящий запрос от клиента через этот сокет. Когда запрос обработан, сервер завершает работу.

```

#define WIN                  // WIN для WinSock, BSD для BSD sockets
#include <stdio.h>          // Для printf()
#include <string.h>          // Для memcpy() и strcpy()
#ifndef WIN
    #include <windows.h>     // Для вызовов WinSock
#endif
#ifndef BSD
    #include <sys/types.h>    // Для системных идентификаторов
    #include <netinet/in.h>   // Для структуры интернет-адреса
    #include <sys/socket.h>   // Для socket(), bind() и др.
    #include <arpa/inet.h>    // Для inet_ntoa()
    #include <fcntl.h>
    #include <netdb.h>
#endif
#define PORT_NUM 1050          // Номер порта на сервере
#define MAX_LISTEN 3           // Максимальное количество прослушиваний

```

```

int main()
{
#ifdef WIN
    WORD wVersionRequested = MAKEWORD(1, 1);      // Функции WSA
    WSADATA wsaData;                            // Функции WSA
#endif
    unsigned int server_s;                      // Дескриптор сокета сервера
    struct sockaddr_in server_addr;             // Интернет-адрес сервера
    unsigned int client_s;                      // Дескриптор сокета клиента
    struct sockaddr_in client_addr;             // Интернет-адрес клиента
    struct in_addr client_ip_addr;              // IP-адрес клиента
    int addr_len;                             // Длина интернет-адреса
    char out_buf[100];                         // 100-байтный выходной буфер данных
    char in_buf[100];                          // 100-байтный входной буфер данных

#ifdef WIN // Инициализация WinSock
    WSASStartup(wVersionRequested, &wsaData);
#endif
    // Создание сокета
    // Семейство адресов AF_INET, сокеты SOCK_STREAM
    server_s = socket(AF_INET, SOCK_STREAM, 0);

    // Заполнение сокета и привязка к серверу
    // См. описание struct sockaddr_in в winsock.h
    server_addr.sin_family = AF_INET;           // Семейство адреса
    server_addr.sin_port = htons(PORT_NUM);     // Номер порта
    server_addr.sin_addr.s_addr = htonl(INADDR_ANY);

    // Прослушивание всех IP-адресов
    bind(server_s, (struct sockaddr*)&server_addr,
        sizeof(server_addr));

    // Прослушивание в ожидании соединения (очередь до MAX_LISTEN)
    listen(server_s, MAX_LISTEN);

    // Прием соединения. Вызов accept() блокирующий;
    // возвращает заполненную структуру client_addr
    addr_len = sizeof(client_addr);
    client_s = accept(server_s, (struct sockaddr*)&client_addr,
                      &addr_len);

    // Копирует 4-байтный IP-адрес клиента в структуру
    // См. описание struct in_addr в winsock.h
    memcpy(&client_ip_addr, &client_addr.sin_addr.s_addr, 4);

    // Вывод сообщения о соединении
    printf("accept() завершена!!! IP-адрес клиента %s, порт %d\n",
           inet_ntoa(client_ip_addr), ntohs(client_addr.sin_port));

    // Отправка информации клиенту
    strcpy(out_buf, "Сообщение от сервера клиенту");
    send(client_s, out_buf, (strlen(out_buf) + 1), 0);
}

```

```
// Получение информации от клиента
recv(client_s, in_buf, sizeof(in_buf), 0);
printf("Получено от клиента... data = `%s`\n", in_buf);

// Закрытие открытых сокетов
#endif WIN
closesocket(server_s);
closesocket(client_s);
#endif
#endif BSD
close(server_s);
close(client_s);
#endif
#endif WIN
// Очистка WinSock
WSACleanup();
#endif
}
}
```

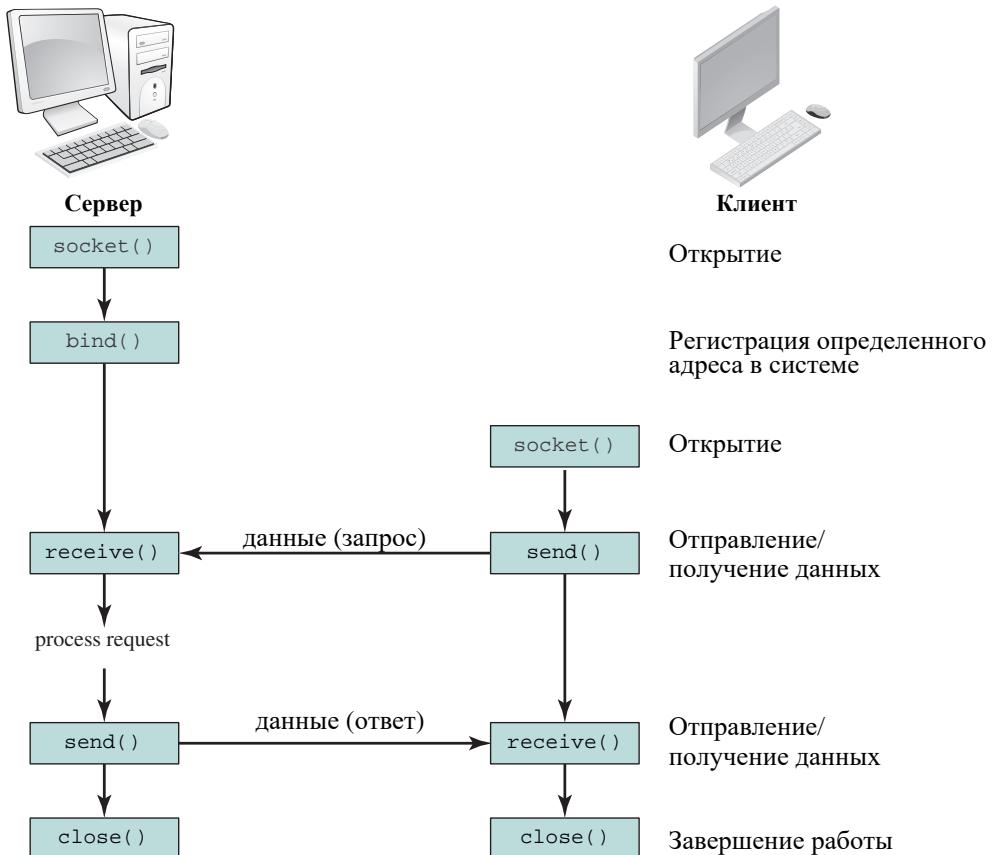
Это не очень реалистичная реализация. Чаще всего серверные приложения содержат некоторый бесконечный цикл и могут принимать несколько запросов. Представленный код может быть легко преобразован в такой более реалистичный сервер, если вставить его функциональность строки в цикл, в котором условие завершения никогда не выполняется (например, `while(1) { ... }`). Такие серверы будут создавать один постоянный сокет с помощью вызова `socket()`; временный сокет при этом отключается каждый раз, когда запрос принят. Таким образом, каждый временный сокет будет нести ответственность за обработку одного входящего соединения. Если работа сервера в конце концов будет завершена, постоянный сокет будет закрыт, как и каждый из активных временных сокетов. Реализация TCP определяет, когда номер порта будет доступным для повторного использования другими приложениями. Статус такого порта в течение некоторого предварительно определенного периода времени будет `TIME_WAIT`, как показано на рис. Л.1 для порта 1234.

## М.4. Сокеты потоков и дейтаграмм

Когда сокеты используются для отправки ориентированного на соединение, надежного потока байтов между машинами, они имеют тип `SOCK_STREAM`. Как мы уже говорили, в таких случаях перед их использованием требуется создание подключения. Данные передаются через двунаправленный поток байтов и гарантированно поступают в том порядке, в котором они были отправлены.

Сокеты типа `SOCK_DGRAM` (или сокеты дейтаграмм) поддерживают двунаправленный поток данных, но данные могут поступать не по порядку и, возможно, дублироваться (т.е. не гарантируется, что данные будут поступать последовательно или что они будут в единственном экземпляре). Дейтаграммы также не обеспечивают надежность, так как могут вообще не прийти. Важно, однако, отметить, что границы записи данных сохраняются, пока записи не длиннее, чем может обработать их получатель. В отличие от потоковых сокетов, сокеты дейтаграмм работают без соединения, поэтому они не должны быть связаны перед использованием. На рис. М.3 показана базовая блок-схема связи сокетов дейтаграмм. Принимая модель потоковой связи за основу, можно легко

заметить, что вызовы `listen()` и `accept()` удаляются, а вызовы `send()` и `recv()` заменяется вызовами `sendto()` и `recvfrom()`.



**Рис. М.3.** Системные вызовы сокетов для протокола без соединения

## Пример клиентской программы UDP (инициация соединения)

```

#define WIN          // WIN для WinSock, BSD для BSD sockets
#include <stdio.h>   // Для printf()
#include <string.h>  // Для memcpy() и strcpy()
#ifndef WIN
    #include <windows.h> // Для вызовов WinSock
#endif
#ifndef BSD
    #include <sys/types.h> // Для системных идентификаторов
    #include <netinet/in.h> // Для структуры интернет-адреса
    #include <sys/socket.h> // Для socket(), bind() и др.
    #include <arpa/inet.h> // Для inet_ntoa()

```

```

#include <fcntl.h>
#include <netdb.h>
#endif
#define PORT_NUM 1050           // Номер порта на сервере
#define IP_ADDR "131.247.167.101" // "Прошитый" адрес сервера

int main()
{
#ifdef WIN
    WORD wVersionRequested = MAKEWORD(1, 1);           // Функции WSA
    WSADATA wsaData;                                    // Функции WSA
#endif
    unsigned int server_s;                // Дескриптор сокета сервера
    struct sockaddr_in server_addr;        // Интернет-адрес сервера
    int addr_len;                         // Длина интернет-адреса
    char out_buf[100];                   // 100-байтный выходной буфер данных
    char in_buf[100];                    // 100-байтный входной буфер данных
#ifdef WIN                           // Инициализация WinSock
    WSASStartup(wVersionRequested, &wsaData);
#endif
    // Создание сокета
    server_s = socket(AF_INET, SOCK_DGRAM, 0);

    // Заполнение сокета и соединение с сервером.
    // Вызов connect() блокирующий
    server_addr.sin_family = AF_INET;           //Семейство адреса
    server_addr.sin_port = htons(PORT_NUM);     // Номер порта
    server_addr.sin_addr.s_addr = inet_addr(IP_ADDR); // IP-адрес

    // Отправка данных на сервер
    strcpy(out_buf, "Сообщение от клиента1 серверу1");
    // +1 обеспечивает пересылку завершающего нулевого символа
    sendto(server_s, out_buf, (strlen(out_buf) + 1), 0,
           (struct sockaddr *)&server_addr, sizeof(server_addr));

    addr_len = sizeof(server_addr);
    recvfrom(server_s, in_buf, sizeof(in_buf), 0,
             (struct sockaddr *)&server_addr, &addr_len);
    // Вывод полученного сообщения
    printf("Получено сообщение: `%s`\n", in_buf);

    // Закрытие открытого сокета
#ifdef WIN
    closesocket(server_s);
#endif
#ifdef BSD
    close(server_s);
#endif
#ifdef WIN // Очистка WinSock
    WSACleanup();
#endif
}

```

## Пример серверной программы UDP (пассивное ожидание соединения)

```

#define WIN                                //WIN для WinSock, BSD для BSD sockets
#include <stdio.h>                         // Для printf()
#include <string.h>                          // Для memcpuy() и strcpy()
#endif WIN                                  // Для вызовов WinSock
#ifndef BSD
    #include <windows.h>                      // Для системных идентификаторов
#endif BSD                                 // Для структуры интернет-адреса
#ifndef socket() bind() и др.
    #include <sys/types.h>                     // Для socket(), bind() и др.
    #include <netinet/in.h>                   // Для inet_ntoa()
    #include <sys/socket.h>
    #include <arpa/inet.h>
    #include <fcntl.h>
    #include <netdb.h>
#endif
#define PORT_NUM 1050                         // Номер порта на сервере
#define IP_ADDR "131.247.167.101"             // IP-адрес клиента

int main()
{
#endif WIN
    WORD wVersionRequested = MAKEWORD(1, 1);   // Функции WSA
    WSADATA wsaData;                           // Функции WSA
#endif
    unsigned int server_s;                     // Дескриптор сокета сервера
    struct sockaddr_in server_addr;            // Интернет-адрес сервера
    struct sockaddr_in client_addr;            // Интернет-адрес клиента
    int addr_len;                            // Длина интернет-адреса
    char out_buf[100];                        // 100-байтный выходной буфер данных
    char in_buf[100];                         // 100-байтный входной буфер данных
#endif WIN                                  // Инициализация WinSock
WSAStartup(wVersionRequested, &wsaData);
#endif
// Создание сокета
// Семейство адресов AF_INET, сокеты SOCK_DGRAM
server_s = socket(AF_INET, SOCK_DGRAM, 0);

// Заполнение сокета и привязка к серверу
// См. описание struct sockaddr_in в winsock.h
server_addr.sin_family = AF_INET;           // Семейство адреса
server_addr.sin_port = htons(PORT_NUM);      // Номер порта
server_addr.sin_addr.s_addr = htonl(INADDR_ANY);

// Прослушивание всех IP-адресов
bind(server_s, (struct sockaddr*)&server_addr,
     sizeof(server_addr));

client_addr.sin_family = AF_INET;             // Семейство адреса
client_addr.sin_port = htons(PORT_NUM);       // Номер порта
client_addr.sin_addr.s_addr = inet_addr(IP_ADDR);

```

```

// Ожидание получения сообщения от клиента
addr_len = sizeof(client_addr);

// Получение информации от клиента
recvfrom(server_s, in_buf, sizeof(in_buf), 0,
          (struct sockaddr *)&client_addr, &addr_len);
printf("Получено от клиента... data = `%s`\n", in_buf);

// Здесь нужен небольшой цикл для задержки,
// чтобы дать клиенту время на подготовку к приему

// Отправка информации клиенту
strcpy(out_buf, "Сообщение от сервера клиенту");
sendto(server_s, out_buf, (strlen(out_buf) + 1), 0,
       (struct sockaddr *)&client_addr, sizeof(client_addr));

// Закрытие открытых сокетов
#endif WIN
    closesocket(server_s);
#endif
#ifndef BSD
    close(server_s);
#endif
#ifndef WIN
    // Очистка WinSock
    WSACleanup();
#endif
#endif
}

```

## М.5. УПРАВЛЕНИЕ ПРОГРАММОЙ ВРЕМЕНИ ВЫПОЛНЕНИЯ

### Неблокирующие вызовы сокетов

По умолчанию сокет создается как блокирующий (т.е. блокирует дальнейшее выполнение до момента завершения вызова текущей функции). Например, если мы вызываем `accept()` для сокета, то процесс будет заблокирован, пока не будет получено входящее соединение от клиента. В UNIX в превращении блокирующего сокета в неблокирующий участвуют две функции: `ioctl()` и `select()`. Первый облегчает управление вводом-выводом для файлового дескриптора или сокета. Затем функция `select()` используется для определения состояния сокета — готов ли он к выполнению действия.

```

// Изменение блокирующего состояния сокета
unsigned long unblock = TRUE;
// TRUE для неблокирующего, FALSE для блокирующего
ioctl(s, FIONBIO, &unblock);

```

Затем мы периодически вызываем `accept()`.

```

while(client_s = accept(s, NULL, NULL) > 0)
{
    if (client_s == EWOULDBLOCK)
        // Ожидание получения соединения с клиентом,
        // выполняя тем временем полезные действия
    else
        // Процесс принял соединение
} // Вывод сообщения об ошибке и выход

```

Можно также использовать вызов функции `select()` для запроса состояния сокета, как показано в фрагменте программы с неблокирующим сокетом.

```

if (select(max_descr + 1, &sockSet, NULL, NULL,
           &sel_timeout) == 0)
    // Вывод сообщения пользователю
else
{
    . .
    client_s = accept(s, NULL, NULL);
    . .
}

```

Таким образом, когда некоторый дескриптор сокета готов к вводу-выводу, процесс должен постоянно опрашивать операционную систему с использованием вызова `select()`, пока сокет не станет готовым. Хотя процесс, выполняющий вызов `select()`, приостановит выполнение программы до тех пор, пока сокет станет готовым, или до истечения времени ожидания функции `select()`, это решение все равно оказывается неэффективным. Так же, как и вызов неблокирующего `accept()` внутри цикла, вызов `select()` внутри цикла приводит к напрасной потере тактов процессора.

## Асинхронный ввод-вывод (ввод-вывод, управляемый сигналом)

Лучшее решение состоит в том, чтобы использовать асинхронный ввод-вывод (т.е. когда у сокета обнаруживается активность в смысле ввода-вывода, операционная система немедленно информирует об этом процесс и, таким образом, освобождает его от бремени постоянного опроса состояния). В исходной BSD UNIX для этого предназначены вызовы `sigaction()` и `fcntl()`. Альтернатива опросу состояния сокета с использованием вызова `select()` позволяет ядру информировать приложение о событиях с помощью механизма сигналов, а именно — с помощью сигнала `SIGIO`. Для этого с помощью вызова `sigaction()` должен быть установлен обработчик сигнала `SIGIO`. Следующая программа не работает с сокетами; это просто пример того, как устанавливается обработчик сигнала. В данном примере путем установки с помощью `sigaction()` обработчика сигнала `SIGINT` (прерывание) выполняется перехват ввода символа прерывания (<Ctrl-C>).

```

#include <stdio.h>          // Для printf()
#include <sys/signal.h>      // Для sigaction()
#include <unistd.h>          // Для pause()

```

```
// Для обработки ошибок
void catch_error(char *errorMessage);

// Обработка сигнала
void InterruptSignalHandler(int signalType);

int main(int argc, char *argv[])
{
    // Обработчик сигнала
    struct sigaction handler;

    // Устанавливаем InterruptSignalHandler() в качестве обработчика
    handler.sa_handler = InterruptSignalHandler;

    // Создаем маску для всех сигналов
    if (sigfillset(&handler.sa_mask) < 0)
        catch_error(sigfillset() failed);

    // Флагов нет
    handler.sa_flags = 0;

    // Установка обработчика для перехватываемого сигнала
    if (sigaction(SIGINT, &handler, 0) < 0)
        catch_error(sigaction() failed);
    for(;;)
        pause(); // Приостановка до получения сигнала
    exit(0);
}

void InterruptSignalHandler(int signalType)
{
    printf("Перехват прерывания. Программа завершается.\n");
    exit(1);
}
```

В дескрипторе файла для сокета с помощью вызова `fcntl()` должен быть установлен флаг `FASYNC`. Говоря более подробно, сначала, используя `sigaction()`, мы уведомляем операционную систему о нашем желании установить новый перехватчик для `SIGIO`; затем, используя `fcntl()`, мы заставляем операционную систему подавать сигналы текущему процессу. Этот вызов необходим для того, чтобы гарантировать, что среди всех процессов, обращающихся к сокету, сигнал будет передан текущему процессу (или группе процессов). Далее мы вновь используем `fcntl()`, чтобы установить флаг состояния `FASYNC` для того же дескриптора сокета. Приведенный далее фрагмент программы следует этой схеме. Учтите, что все ненужные детали в коде для ясности опущены.

```
int main()
{
    . .
    // Создание сокета для отправки/получения дейтаграмм
```

```

// Настройка структуры адреса сервера
// Привязка к локальному адресу
// Установка обработчика сигнала SIGIO
// Создание маски, маскирующей все сигналы
if (sigfillset(&handler.sa_mask) < 0)
    // Вывод сообщения об ошибке и выход из программы

// Флагов нет
handler.sa_flags = 0;

if (sigaction(SIGIO, &handler, 0) < 0)
    // Вывод сообщения об ошибке и выход из программы

// Мы должны владеть сокетом для получения сообщения SIGIO
if (fcntl(s_socket, F_SETOWN, getpid()) < 0)
    // Вывод сообщения об ошибке и выход из программы

// Настройка для асинхронного ввода-вывода и получения SIGIO
if (fcntl(s_socket, F_SETFL, FASYNC | O_NONBLOCK) < 0)
    // Вывод сообщения об ошибке и выход из программы

for(;;)
    pause();
. . .
}

```

В Windows функция `select()` не реализована. `WSAAsyncSelect()` используется для запроса уведомления о сетевых событиях (например, запрос на отправку `Ws2_32.dll` сообщения в окно `hWnd`).

```
WSAAsyncSelect(SOCKET socket, HWND hWnd,
               unsigned int wMsg, long lEvent)
```

Тип `SOCKET` определен в `winsock.h`. В объявлении выше `socket` представляет собой дескриптор сокета, `hWnd` — дескриптор окна, `wMsg` — сообщение, а `lEvent` обычно является побитовым ИЛИ всех событий, о которых мы хотим получать уведомления, когда они будут завершены. Вот некоторые из значений событий: `FD_CONNECT` (соединение установлено), `FD_ACCEPT` (готов к приему), `FD_READ` (готов к чтению), `FD_WRITE` (готов к записи), `FD_CLOSE` (соединение закрыто). Вы можете легко включить следующий фрагмент кода в ранее представленные программы или в свое собственное приложение. (И вновь ненужные детали в коде для ясности опущены.)

```

// Сообщение для асинхронного уведомления
#define wMsg (WM_USER + 4)
. . .

// socket_s уже создан и привязан

// Прослушивание для соединения
if (listen(socket_s, 3) == SOCKET_ERROR)
    // Вывод сообщения об ошибке
    // Выход после очистки

```

```
// Получение окном уведомления о принятом соединении
if (WSAAsyncSelect(s, hWnd, wMsg, FD_ACCEPT)==SOCKET_ERROR)
    // Вывод сообщения об ошибке
    // Выход после очистки
else // Прием входящего соединения
```

Дополнительные сведения на тему об асинхронном вводе-выводе вы можете найти в Интернете или в книгах, таких как *The Pocket Guide to TCP/IP Sockets — C version* Донаху (Donahoo) и Калверта (Calvert) для UNIX и *Windows Sockets Network Programming* Боба Квинна (Bob Quinn) для Windows.

## М.6. УДАЛЕННОЕ ВЫПОЛНЕНИЕ КОНСОЛЬНОГО ПРИЛОЖЕНИЯ WINDOWS

Простые операции с сокетами могут использоваться для выполнения задач, которые трудно выполнить иными средствами. Например, используя сокеты, можно удаленно выполнить приложение. Далее представлен пример такого кода<sup>1</sup>. Имеются две программы с использованием сокетов, локальная и удаленная, которые переносят консольное приложение Windows (.exe-файл) с локального хоста на удаленный. Затем программа выполняется на удаленном хосте, а стандартный вывод возвращается на локальный хост.

### Локальный код

```
#include <stdio.h>           // Для printf()
#include <stdlib.h>          // Для exit()
#include <string.h>           // Для memcpuy() и strcpy()
#include <windows.h>          // Для Sleep() и WinSock

#include <fcntl.h>             // Для констант файлового
#include <sys\stat.h>          // ввода-вывода
#include <io.h>                // Для open(), close() и eof()
#define PORT_NUM 1050           // Произвольный номер порта для сервера
#define MAX_LISTEN 1            // Максимальное количество прослушивателей
#define SIZE 256                 // Размер буфера передачи в байтах

int main(int argc, char* argv[])
{
    WORD wVersionRequested = MAKEWORD(1, 1); // Функции WSA
    WSADATA wsaData;                  // Структуры данных Winsock API
    unsigned int remote_s;             // Дескриптор удаленного сокета

    struct sockaddr_in remote_addr;
    // Интернет-адрес удаленной системы

    struct sockaddr_in server_addr;
```

<sup>1</sup> Представленный здесь код принадлежит Кену Кристенсену (Ken Christensen) и Карлу Латаксе-су (Karl S. Lataxes) с факультета информатики Университета Южной Флориды (см. <http://www.csee.usf.edu/~christen/tools/>).

```

// Интернет-адрес сервера

unsigned char bin_buf[SIZE];           // Буфер для передачи файла
unsigned int fh;                      // Дескриптор файла
unsigned int length;                  // Длина передаваемого буфера
struct hostent* host;                // Структура для gethostbyname()
struct in_addr address;              // Структура для интернет-адреса
char host_name[256];                 // Стока для имени хоста
int addr_len;                       // Длина интернет-адреса
unsigned int local_s;                // Дескриптор локального сокета
struct sockaddr_in local_addr;        // Локальный интернет-адрес
struct in_addr remote_ip_addr;        // Удаленный IP-адрес

// Проверка корректности количества аргументов командной строки
if (argc != 4)
{
    printf("**** ОШИБКА - 'local (host) (exefile) (outfile)'\n");
    printf(" host - имя хоста или IP-адрес удаленного,\n");
    printf(" хоста, exefile - имя удаленно выполняемого\n");
    printf(" файла, а outfile - имя локального файла вывода.\n");
    exit(1);
}

// Инициализация winsock
WSAStartup(wVersionRequested, &wsaData);
// Копируем имя хоста в host_name
strcpy(host_name, argv[1]);
// Вызов gethostbyname()
host = gethostbyname(argv[1]);

if (host == NULL)
{
    printf("**** ОШИБКА - IP-адрес '%s' не найден\n",
           host_name);
    exit(1);
}

// Копируем IP-адрес клиента в структуру
memcpuy(&address, host->h_addr, 4);

// Создание сокета
remote_s = socket(AF_INET, SOCK_STREAM, 0);

// Заполнение информации об удаленном адресе и подключение к
// прослушивающему серверу.
server_addr.sin_family = AF_INET;          // Семейство адреса
server_addr.sin_port = htons(PORT_NUM);     // Используемый порт

// IP-адрес:
server_addr.sin_addr.s_addr = inet_addr(inet_ntoa(address));
connect(remote_s, (struct sockaddr*)&server_addr,
        sizeof(server_addr));

```

```

// Открытие и чтение файла
if ((fh = open(argv[2], O_RDONLY | O_BINARY, S_IREAD |
S_IWRITE)) == -1)
{
    printf("**** ОШИБКА открытия файла '%s'\n", argv[2]);
    exit(1);
}

// Сообщение, указывающее пересылку файла
printf("Отправка '%s' на удаленный сервер '%s'\n",
       argv[2], argv[1]);

// Отправка файла на удаленный сервер
while (!eof(fh))
{
    length = read(fh, bin_buf, SIZE);
    send(remote_s, bin_buf, length, 0);
}
close(fh);
// Закрытие сокета
closesocket(remote_s);
// Очистка Winsock
WSACleanup();
// Вывод сообщения об удаленном выполнении
printf("'%' выполняется на удаленном сервере\n", argv[2]);
// Задержка для выполнения всех очисток
Sleep(100);
// Инициализация winsock
WSAStartup(wVersionRequested, &wsaData);

// Создание нового сокета
local_s = socket(AF_INET, SOCK_STREAM, 0);

// Заполнение информации и привязка сокета
local_addr.sin_family = AF_INET;           // Семейство адреса
local_addr.sin_port = htons(PORT_NUM);      // Используемый порт
local_addr.sin_addr.s_addr = htonl(INADDR_ANY);

// Прослушивание всех адресов
bind(local_s, (struct sockaddr*)&local_addr,
     sizeof(local_addr));
listen(local_s, MAX_LISTEN);
// Принятие соединения. accept() блокирует выполнение и
// заполняет информацией remote_addr.
addr_len = sizeof(remote_addr);
remote_s = accept(local_s, (struct sockaddr*)
                  &remote_addr, &addr_len);
// Копирование клиентского IP-адреса
memcpy(&remote_ip_addr, &remote_addr.sin_addr.s_addr, 4);

```

```

// Создание и открытие файла для записи
if ((fh = open(argv[3], O_WRONLY | O_CREAT | O_TRUNC |
                O_BINARY,
                S_IREAD | S_IWRITE)) == -1)
{
    printf("**** ОШИБКА открытия '%s'\n", argv[3]);
    exit(1);
}

// Получение файла от сервера
length = SIZE;

while (length > 0)
{
    length = recv(remote_s, bin_buf, SIZE, 0);
    write(fh, bin_buf, length);
}
// Закрытие полученного файла
close(fh);
// Закрытие сокетов
closesocket(local_s);
closesocket(remote_s);
// Вывод сообщения о состоянии
printf("Выполнен '%s', вывод сохранен в '%s'.\n",
       argv[2], argv[3]);
// Очистка Winsock
WSACleanup();
}

```

## Удаленный код

```

#include <stdio.h>           // Для printf()
#include <stdlib.h>          // Для exit()
#include <string.h>           // Для memcpuy() и strcpy()
#include <windows.h>          // Для Sleep() и WinSock
#include <fcntl.h>             // Для констант файлового
#include <sys\stat.h>          // ввода-вывода
#include <io.h>                // Для open(), close() и eof()
#define PORT_NUM 1050           // Произвольный номер порта для сервера
#define MAX_LISTEN 1            // Максимальное количество прослушивателей
#define IN_FILE "run.exe"        // Имя передаваемого *.exe-файла
#define TEXT_FILE "output"       // Имя выходного файла
#define SIZE 256                 // Размер буфера передачи в байтах
void main(void)
{
    WORD wVersionRequested = MAKEWORD(1, 1); // Функции WSA
    WSADATA wsaData;           // Структуры данных Winsock API
    unsigned int remote_s;      // Дескриптор удаленного сокета
    struct sockaddr_in remote_addr;
    // Интернет-адрес удаленной системы
    struct sockaddr_in server_addr;
    // Интернет-адрес сервера

```

```

unsigned int local_s;           // Дескриптор локального сокета
struct sockaddr_in local_addr; // Локальный интернет-адрес
struct in_addr local_ip_addr; // Локальный IP-адрес
int addr_len;                 // Длина интернет-адреса
unsigned char bin_buf[SIZE];   // Буфер передачи файла
unsigned int fh;               // Дескриптор файла
unsigned int length;           // Длина буфера
// Бесконечный цикл
while (1)
{
    // Инициализация winsock
    WSAStartup(wVersionRequested, &wsaData);
    // Создание сокета
    remote_s = socket(AF_INET, SOCK_STREAM, 0);
    // Заполнение сокета информацией и привязка
    remote_addr.sin_family = AF_INET;           // Семейство адреса
    remote_addr.sin_port = htons(PORT_NUM);     // Номер порта
    remote_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    // Прослушивание на всех IP-адресах
    bind(remote_s, (struct sockaddr*)&remote_addr,
         sizeof(remote_addr));
    // Вывод сообщения об ожидании
    printf("Ожидание соединения...\n");
    // Прослушивание (до MAX_LISTEN подключений)
    listen(remote_s, MAX_LISTEN);
    // Принятие соединения. accept() блокирует выполнение и
    // заполняет информацией local_addr.
    addr_len = sizeof(local_addr);
    local_s = accept(remote_s, (struct sockaddr*)&local_addr, &addr_len);
    // Копирование клиентского IP-адреса
    memcpy(&local_ip_addr, &local_addr.sin_addr.s_addr, 4);
    // Вывод сообщения о подключении
    printf("Соединение установлено, прием удаленного файла\n");
    // Открытие IN_FILE для удаленного выполнимого файла
    if ((fh = open(IN_FILE, O_WRONLY | O_CREAT | O_TRUNC |
                   O_BINARY,
                   S_IREAD | S_IWRITE)) == -1)
    {
        printf("!!! ОШИБКА открытия выполнимого файла\n");
        exit(1);
    }
    // Получение выполнимого файла
    length = 256;
    while (length > 0)
    {
        length = recv(local_s, bin_buf, SIZE, 0);
        write(fh, bin_buf, length);
    }
    // Закрытие полученного файла IN_FILE
    close(fh);
}

```

```

// Закрытие сокетов
closesocket(remote_s);
closesocket(local_s);
// Очистка Winsock
WSACleanup();

// Вывод информационного сообщения
printf("Выполнение удаленного файла\n");
// Выполнение удаленного файла (в IN_FILE)
system(IN_FILE" > "TEXT_FILE");
// Инициализация Winsock для повторного открытия
// сокета и отправки вывода клиенту
WSAStartup(wVersionRequested, &wsaData);
// Создание сокета
local_s = socket(AF_INET, SOCK_STREAM, 0);
// Заполнение сокета информацией и подключение
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(PORT_NUM);
server_addr.sin_addr.s_addr = inet_addr(inet_ntoa(local_ip_addr));
connect(local_s, (struct sockaddr*)&server_addr,
        sizeof(server_addr));
// Вывод информационного сообщения
printf("Отправка вывода на локальный хост\n");
// Открытие файла вывода для передачи клиенту
if ((fh = open(TEXT_FILE, O_RDONLY |
               O_BINARY, S_IREAD | S_IWRITE)) == -1)
{
    printf("**** ОШИБКА открытия файла\n");
    exit(1);
}

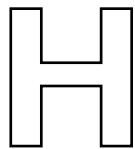
// Отправка файла вывода клиенту
while (!eof(fh))
{
    length = read(fh, bin_buf, SIZE);
    send(local_s, bin_buf, length, 0);
}

// Закрытие файла
close(fh);

// Закрытие сокетов
closesocket(remote_s);
closesocket(local_s);

// Очистка Winsock
WSACleanup();
// Задержка для выполнения всех очисток
Sleep(100);
}
}

```



## ПРИЛОЖЕНИЕ

# МЕЖДУНАРОДНЫЙ СПРАВОЧНЫЙ АЛФАВИТ

Знакомый всем пример данных — **текст**, или символьные строки. Хотя текстовые данные наиболее удобны для людей, они не могут быть легко сохранены или переданы в символьном виде в системах обработки данных и связи. Такие системы предназначены для бинарных данных. Поэтому был разработан ряд кодов, посредством которых символы представлены последовательностями битов. Возможно, наиболее ранний распространенный пример такого кода — азбука Морзе. Сегодня чаще всего используемым текстовым кодом является Международный справочный алфавит (International Reference Alphabet — IRA)<sup>1</sup>. Каждый символ в этом коде представлен уникальным 7-разрядным бинарным кодом. Таким образом, могут быть представлены 128 различных символов. В табл. Н.1 перечислены все значения данного кода. В таблице биты каждого символа обозначены как  $b_i$ , от  $b_7$ , который является старшим значащим битом, до младшего значащего бита  $b_1$ . Имеется два типа символов: печатные и управляющие (табл. Н.2). Печатные символы — это буквенные, цифровые и специальные символы, которые могут быть напечатаны на бумаге или отображены на экране. Например, битовое представление символа “К” имеет вид  $b_7b_6b_5b_4b_3b_2b_1 = 1001011$ . Некоторые из управляющих символов имеют отношение к управлению печатью или отображением символов; примером является возврат каретки. Другие управляющие символы относятся к процедурам связи.

IRA-кодированные символы почти всегда сохраняются и передаются с использованием 8 битов на символ. В этом случае восьмой бит является битом четности, используемым для обнаружения ошибок. Бит четности является самым старшим битом и поэтому обозначается как  $b_8$ . Этот бит установлен так, что общее число бинарных единиц в каждом октете всегда нечетно (нечетная четность) или всегда четно (четная четность). Таким образом, может быть обнаружена ошибка передачи, которая изменяет один бит (или любое нечетное количество битов).

<sup>1</sup> IRA определен в ITU-T Recommendation T.50 и ранее был известен как Международный алфавит номер 5 (International Alphabet Number 5 — IA5). Национальная версия IRA в США называется Американским стандартным кодом для обмена информацией (American Standard Code for Information Interchange — ASCII).

ТАБЛИЦА Н.1. МЕЖДУНАРОДНЫЙ СПРАВОЧНЫЙ АЛФАВИТ

| Битовые позиции |                |                |                |     |     |    |   |   |    |   |     |   |
|-----------------|----------------|----------------|----------------|-----|-----|----|---|---|----|---|-----|---|
|                 | b <sub>7</sub> |                |                | 0   | 0   | 0  | 0 | 1 | 1  | 1 | 1   | 1 |
|                 |                | b <sub>6</sub> |                | 0   | 0   | 1  | 1 | 0 | 0  | 1 | 1   | 1 |
|                 |                |                | b <sub>5</sub> | 0   | 1   | 0  | 1 | 0 | 1  | 0 | 1   | 1 |
| b <sub>4</sub>  | b <sub>3</sub> | b <sub>2</sub> | b <sub>1</sub> |     |     |    |   |   |    |   |     |   |
| 0               | 0              | 0              | 0              | NUL | DLE | SP | 0 | @ | P  | ‘ | p   |   |
| 0               | 0              | 0              | 1              | SOH | DC1 | !  | 1 | A | Q  | a | q   |   |
| 0               | 0              | 1              | 0              | STX | DC2 | ”  | 2 | B | R  | b | r   |   |
| 0               | 0              | 1              | 1              | ETX | DC3 | #  | 3 | C | S  | c | s   |   |
| 0               | 1              | 0              | 0              | EOT | DC4 | \$ | 4 | D | T  | d | t   |   |
| 0               | 1              | 0              | 1              | ENQ | NAK | %  | 5 | E | U  | e | u   |   |
| 0               | 1              | 1              | 0              | ACK | SYN | &  | 6 | F | V  | f | v   |   |
| 0               | 1              | 1              | 1              | BEL | ETB | ,  | 7 | G | W  | g | w   |   |
| 1               | 0              | 0              | 0              | BS  | CAN | (  | 8 | H | X  | h | x   |   |
| 1               | 0              | 0              | 1              | HT  | EM  | )  | 9 | I | Y  | i | y   |   |
| 1               | 0              | 1              | 0              | LF  | SUB | *  | : | J | Z  | j | z   |   |
| 1               | 0              | 1              | 1              | VT  | ESC | +  | ; | K | [  | k | {   |   |
| 1               | 1              | 0              | 0              | FF  | FS  | ,  | < | L | \  | l |     |   |
| 1               | 1              | 0              | 1              | CR  | GS  | -  | = | M | ]  | m | }   |   |
| 1               | 1              | 1              | 0              | SO  | RS  | .  | > | N | ^] | n | ~   |   |
| 1               | 1              | 1              | 1              | SI  | US  | /  | ? | O | _  | o | DEL |   |

## Таблица Н.2. Управляющие символы IRA

### Управление форматированием

**BS** (забой): перемещение печатного механизма или курсора вывода на дисплей на одну позицию назад

**HT** (горизонтальная табуляция): перемещение печатного механизма или курсора вывода на дисплей к следующей предопределенной позиции “табуляции”

**LF** (перевод строки): перемещение печатного механизма или курсора вывода на дисплей к началу следующей строки

**VT** (вертикальная табуляция): перемещение печатного механизма или курсора вывода на дисплей к следующей из ряда предопределенных строк вывода

**FF** (подача страницы): перемещение печатного механизма или курсора вывода на дисплей к началу следующей страницы, формы или экрана

**CR** (возврат каретки): перемещение печатного механизма или курсора вывода на дисплей в начальную позицию текущей строки

### Управление передачей

**SOH** (начало заголовка): используется для указания начала заголовка, который может содержать информацию об адресе или маршруте

**STX** (начало текста): используется для указания начала текста, а также указывает конец заголовка

**ETX** (конец текста): используется для указания конца текста, начало которого указано с помощью STX

**EOT** (конец передачи): указывает конец передачи, которая может включать один или несколько “текстов” со своими заголовками

**ENQ** (запрос идентификации): запрос ответа от удаленной станции. Может использоваться как запрос “Кто вы?” для самоидентификации станции

**ACK** (подтверждение приема): символ, передаваемый приемным устройством отправителю в качестве подтверждения успешного получения

**NAK** (отсутствие приема): символ, передаваемый приемным устройством отправителю в качестве отрицательного ответа

**SYN** (символ синхронизации): используется системами синхронной передачи в целях синхронизации. При отсутствии передаваемых данных система синхронной передачи может постоянно передавать символ SYN

**ETB** (конец передаваемого блока): указывает конец передаваемого блока для связи. Используется для указания блока данных, когда структура блока не обязательно связана с обрабатываемым форматом

### Неформальные разделители

**FS** (разделитель файлов)

**GS** (разделитель групп)

**RS** (разделитель записей)

**US** (разделитель единиц)

Информационные разделители для неформального использования; должна соблюдаться иерархия — от FS (наиболее охватывающий) до US (наименее охватывающий)

---

**Прочее**

**NUL** (нулевой символ): нет символа.

Используется как заполнитель при отсутствии данных

**BEL** (звуковой сигнал): используется, когда необходимо привлечь внимание

**SO** (смена регистра): указывает, что последующие кодовые комбинации должны рассматриваться как находящиеся вне стандартного набора символов, пока не будет получен символ SI

**SI** (возврат регистра): указывает, что последующие кодовые комбинации должны рассматриваться как находящиеся в стандартном наборе символов

**DEL** (удаление): используется для удаления нежелательных символов, например, перезаписью

**SP** (пробел): непечатаемый символ, используемый для разделения слов или для перемещения механизма печати или курсора отображения вперед на одну позицию

**DLE** (управляющий символ канала передачи данных): символ, изменяющий значение одного или нескольких последующих символов. Может сопровождаться дополнительными управляющими символами

**DC1, DC2, DC3, DC4** (управление устройством): символы для управления вспомогательными устройствами или специальными возможностями терминала

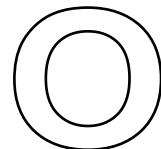
**CAN** (отмена): указывает, что предшествующие ему данные в сообщении или блоке следует игнорировать (обычно потому, что была обнаружена ошибка)

**EM** (конец среды): указывает физический конец ленты или другого носителя или конец необходимого, запрошенного или используемого носителя

**SUB** (замена): заменяет символ, который признан ошибочным или недействительным

**ESC** (управляющий символ): символ, изменяющий значение одного или нескольких непосредственно следующих за ним символов

---



## ПРИЛОЖЕНИЕ

---

# ПАРАЛЛЕЛЬНАЯ СИСТЕМА ПРОГРАММИРОВАНИЯ ВАСІ

В ЭТОМ ПРИЛОЖЕНИИ...

## О.1. Введение

## О.2. ВАСІ

Обзор системы

Параллельные конструкции ВАСІ

cobegin

Семафоры

Мониторы

Другие конструкции параллелизма

Как получить ВАСІ

## О.3. Примеры программ ВАСІ

## О.4. Проекты ВАСІ

Реализация примитивов синхронизации

Реализация машинных команд

Реализация честных семафоров (FIFO)

Семафоры, мониторы и их реализации

Семафоры А и В

Использование бинарных семафоров

Активное ожидание и семафоры

Семафоры и мониторы

Общие и бинарные семафоры

Проект монитора

Задача о популярном пекаре

## О.5. Усовершенствования системы ВАСІ

## О.1. ВВЕДЕНИЕ

В главе 5, “Параллельные вычисления: взаимоисключения и многозадачность”, были представлены основные концепции параллелизма (например, взаимное исключение и проблема критического участка), а также предложены методы синхронизации (такие, как семафоры, мониторы и передача сообщений). Проблемы взаимоблокировки и голодания в параллельных программах рассматриваются в главе 6, “Параллельные вычисления: взаимоблокировка и голодание”. В связи с растущим вниманием к параллельным и распределенным вычислениям понимание параллелизма и синхронизации разработчиками ныне необходимо как никогда. Чтобы получить полное понимание этих концепций, необходим практический опыт написания параллельных программ.

Имеется три основных варианта получения такого практического опыта. Во-первых, можно писать параллельные программы с использованием языка параллельного программирования, такого как Concurrent Pascal, Modula, Ada или язык программирования SR. Однако для экспериментов с использованием различных методов синхронизации требуется изучить синтаксис большого количества параллельных языков программирования. Во-вторых, можно писать параллельные программы на основе системных вызовов операционной системы, такой как UNIX. При этом легко отвлечься от главной цели — понимания параллельного программирования, — сосредоточившись на подробностях и особенностях конкретной операционной системы (например, деталях реализации семафоров в UNIX). Наконец, можно писать параллельные программы на языке, разработанном специально для получения опыта работы с параллельными концепциями, таком как Ben-Ari Concurrent Interpreter (BACI). Использование такого языка предлагает различные варианты методов синхронизации с привычным синтаксисом. Языки, разработанные специально для получения опыта работы с концепциями параллелизма, являются лучшим вариантом для получения желаемого практического опыта.

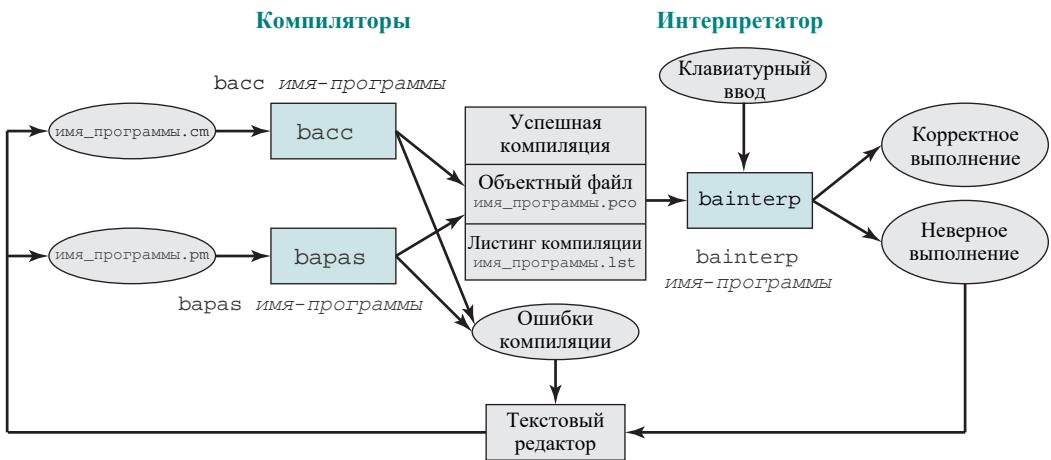
В разделе Н.2 содержится краткий обзор системы BACI и способов ее получения. В разделе Н.3 содержатся примеры программ BACI, а в разделе Н.4 — описание проектов для получения практического опыта параллелизма. Наконец, в разделе Н.5 содержится описание выполненных улучшений в системе BACI.

## О.2. BACI

### Обзор системы

BACI является прямым потомком модификации последовательного Pascal-S, выполненной М. Бен-Ари (M. Ben-Ari). Pascal-S представляет собой подмножество стандартного Pascal от Вирта (Wirth) без файлов (кроме INPUT и OUTPUT), множеств, переменных-указателей и операторов goto. Бен-Ари взял язык программирования Pascal-S и добавил в него параллельные программные конструкции, такие как `cobegin...coend` и тип переменной-семафора с операциями `wait` и `signal`. BACI — это модификация Бен-Ари языка Pascal-S с дополнительными возможностями синхронизации (например, мониторами), а также механизмами инкапсуляции, гарантирующими, что пользователь защищен от неправильного изменения переменной (например, переменная семафора должна изменяться только функциями семафора).

BACI моделирует параллельное выполнение процесса и поддерживает следующие методы синхронизации: общие семафоры, бинарные семафоры и мониторы. Система BACI состоит из двух подсистем, как показано на рис. Н.1. Первая подсистема, компилятор, компилирует пользовательскую программу в промежуточный объектный код, называемый РCODE. В системе BACI имеется два компилятора, соответствующих двум популярным языкам, которые преподаются на курсах начального программирования. Синтаксис одного компилятора похож на стандартный Pascal; программы на BACI, использующие синтаксис Pascal, записываются как имя\_программы.pm. Синтаксис другого компилятора подобен стандартному C++; эти программы записываются как имя\_программы.cl. Оба компилятора создают во время компиляции два файла: имя\_программы.clst и имя\_программы.pco.



**Рис. Н.1**

Вторая подсистема BACI, интерпретатор, выполняет объектный код, созданный компилятором. Другими словами, интерпретатор выполняет имя\_программы.pco. Ядром интерпретатора является вытесняющий планировщик; во время выполнения программы этот планировщик случайным образом обменивается параллельные процессы, имитируя их параллельное выполнение. Интерпретатор предлагает ряд параметров отладки, таких как пошаговое выполнение, дизассемблирование инструкций РCODE и отображение памяти программ.

## Параллельные конструкции BACI

В оставшейся части этого приложения мы сосредоточимся на компиляторе, аналогичном стандартному C++. Мы называем этот компилятор C—; хотя его синтаксис и похож на C++, он не включает наследование, инкапсуляцию или иные возможности объектно-ориентированного программирования. В этом разделе мы приведем обзор параллельных конструкций BACI; дополнительную информацию можно найти в руководстве пользователя на веб-сайте BACI.

**cobegin**

Список процессов, запускаемых одновременно, заключается в блок `cobegin`. Такие блоки не могут быть вложенными и должны находиться в основной программе.

```
cobegin { proc1(...); proc2(...); ... ; procN(...) ; }
```

Инструкции PCODE, созданные компилятором для указанного выше блока, чередуются интерпретатором в произвольном, “случайном” порядке; несколько выполнений одной и той же программы с блоком `cobegin` продемонстрируют ее недетерминированность.

**Семафоры**

Семафор в BACI представляет собой неотрицательную переменную типа `int`, которая может быть доступна только через вызовы семафора, которые будут определены впоследствии. Бинарный семафор в BACI, который принимает только значения 0 и 1, поддерживается подтипов `binarysem` типа `semaphore`. Во время компиляции и выполнения компилятор и интерпретатор обеспечивают выполнение ограничения, что переменная типа `binarysem` может иметь только значение 0 или 1 и что тип `semaphore` может быть только неотрицательным. Вызовы семафора BACI включают следующее.

- `initialsem(semaphore sem, int expression)`.
- `p(semaphore sem)`. Если значение `sem` больше нуля, то интерпретатор уменьшает значение `sem` на 1 и возвращает управление, что позволяет продолжить работу кода, вызвавшего `p`. Если значение `sem` равно нулю, интерпретатор переводит вызывающий код в режим сна. Команда `wait` трактуется как синоним `p`.
- `v(semaphore sem)`. Если значение `sem` равно нулю и один или несколько процессов находятся в состоянии ожидания `sem`, то один из этих процессов пробуждается. Если таких процессов, ожидающих `sem`, нет, то `sem` увеличивается на единицу. В любом случае код, вызвавший `v`, может продолжать выполнение. (BACI соответствует первоначальному предложению Дейкстры о случайном выборе процесса, просыпающегося при поступлении сигнала.) Команда `signal` рассматривается как синоним `v`.

**Мониторы**

BACI поддерживает концепцию мониторов с определенными ограничениями. Монитор представляет собой блок C++, подобный блоку, определяемому процедурой или функцией, с некоторыми дополнительными свойствами (например, условными переменными). В BACI монитор должен быть объявлен на внешнем, глобальном уровне и не может быть вложен в блок другого монитора. Процедурами и функциями монитора для управления параллелизмом используются три конструкции: условные переменные, `waitc` (ожидание условия) и `signalc` (сигнал об условии). Условие не имеет значения; это нечто, что можно ожидать или что может сигнализировать. Процесс монитора может ждать выполнения условия или сигнализировать о том, что условие выполнено, с использованием функций `waitc` и `signalc`. Вызовы `waitc` и `signalc` имеют следующий синтаксис и семантику.

- `waitc(condition cond, int prio)`. Процесс монитора (а следовательно, внешний процесс, вызвавший процесс монитора) блокируется условием `cond` и получает приоритет `prio`.

- `wait(condition cond)`. Имеет ту же семантику, что и `waitc`, но использует приоритет по умолчанию, равный 10.
- `signalc(condition cond)`. Активация некоторого процесса с наименьшим значением (т.е. наивысшим) приоритета, ожидающего условия `cond`; если никакие процессы не ожидают выполнения условия `cond`, ничего не происходит.

BACI отвечает требованию немедленного возобновления. Другими словами, процесс, ожидающий условия, имеет приоритет над процессом, пытающимся войти в монитор, если выполняется сигнал процессу, ожидающему условия.

### *Другие конструкции параллелизма*

Компилятор C— BACI предоставляет некоторые низкоуровневые конструкции параллелизма, которые можно использовать для создания новых примитивов управления параллелизмом. Если функция определена как атомарная, то такая функция невытесняема. Другими словами, интерпретатор атомарную функцию прерывать не будет. В BACI функция приостановки переводит вызывающий ее процесс в спящий режим, а функция восстановления активирует приостановленный процесс вновь.

### **Как получить BACI**

Система BACI с двумя руководствами пользователя (по одному для каждого из двух компиляторов) и подробным описанием проектов доступна на веб-сайте BACI<sup>1</sup>. Система BACI написана как на C, так и на Java. C-версия BACI может быть скомпилирована в Linux, RS/6000 AIX, Sun OS, DOS и CYGWIN в Windows с минимальными изменениями в файле Makefile. (Читайте детальную информацию по установке на той или иной платформе в файле README.)

## **О.3. ПРИМЕРЫ ПРОГРАММ BACI**

В главах 5, “Параллельные вычисления: взаимоисключения и многозадачность”, и 6, “Параллельные вычисления: взаимоблокировка и голодание”, обсуждался ряд классических задач синхронизации (например, задача читателей/писателей или задача об обдающих философах). В этом разделе мы проиллюстрируем систему BACI с помощью трех программ. Первый пример иллюстрирует недетерминированность при выполнении параллельных процессов в системе BACI. Рассмотрим следующий код.

```
const int m = 5;
int n;
void incr(char id)
{
    int i;
    for(i = 1; i <= m; i = i + 1)
    {
        n = n + 1;
        cout << id << " n =" << n << " i =" ;
        cout << i << " " << id << endl;
    }
}
```

---

<sup>1</sup> [http://inside.mines.edu/fs\\_home/tcamp/baci/baci\\_index.html](http://inside.mines.edu/fs_home/tcamp/baci/baci_index.html)

```

main()
{
    n = 0;
    cobegin {
        incr('A'); incr('B'); incr('C');
    }
    cout << "The sum is " << n << endl;
}

```

Обратите внимание, что если каждый из трех созданных процессов (A, B и C) выполняется последовательно, получаемая в результате сумма будет равна 15. Однако параллельное вычисление инструкции `n=n+1`; может привести к различным значениям суммы. После того как мы скомпилировали предыдущую программу с BACI, мы несколько раз выполнили PCODE-файл с помощью `bainterp`. Каждое выполнение приводило к суммам между 9 и 15. Вот один из примеров выполнения кода интерпретатором BACI.

```

Source file: incremen.cm Fri Aug 1 16:51:00 1997
CB n = 2 i =1 C n = 2
A n = 2 i = 1 i = 1 A
CB
    n = 3 i = 2 C
A n = 4 i = 2 C n = 5 i = 3 C
A
B n = 6C i = 2 B
    n = 7 i = 4 C
A n = 8 i = 3 A
BC n = 10 n = 10 i = 5 C
A n = i = 311 i = 4 A
B
A n = 12 i = B5 n = 13A
    i = 4 B
B n = 14 i = 5 B
The sum is 14

```

Для синхронизации доступа процессов к общей основной памяти необходимы специальные машинные команды. Затем поверх этих специальных инструкций строятся протоколы взаимного исключения или примитивы синхронизации. Интерпретатор BACI не прерывает функцию, определенную как атомарную, с переключением контекста. Эта возможность позволяет пользователям реализацию низкоуровневых специальных машинных команд. Например, следующая программа представляет собой реализацию функции `testset` в BACI. Команда `testset` проверяет значение аргумента функции `i`. Если значение `i` равно нулю, функция заменяет его значением `l` и возвращает `true`; в противном случае функция не меняет значение `i` и возвращает `false`. Как обсуждалось в разделе 5.2, специальные машинные команды (например, `testset`) позволяют выполнять несколько действий без прерывания. BACI для этой цели имеется ключевое слово `atomic`.

```

// Команда проверки и установки значения
//
atomic int testset(int& i)
{
    if (i == 0) {

```

```

    i = 1;
    return 1;
}
else
    return 0;
}

```

Мы можем использовать `testset` для реализации протоколов взаимного исключения, как показано в следующей программе. Эта программа является ВАСI-реализацией взаимного исключения и основана на функции `testset`. В ней предполагается наличие трех одновременных процессов; каждый процесс 10 раз запрашивает взаимное исключение.

```

int bolt = 0;
const int RepeatCount = 10;
void proc(int id)
{
    int i = 0;
    while(i < RepeatCount) {
        while (testset(bolt)); // Ожидание
        // Вход в критический участок
        cout << id;
        // Выход из критического участка
        bolt = 0;
        i++;
    }
}
main()
{
    cobegin {
        proc(0); proc(1); proc(2);
    }
}

```

Две следующие программы являются решением ВАСI для задачи производителя/потребителя с ограниченным буфером с семафорами (см. рис. 5.13). В этом примере имеются два производителя, три потребителя и буфер размером, равным пяти. Сначала приводится код программы для этой задачи, а затем — код включаемого в программу файла, который определяет реализацию ограниченного буфера.

```

// Решение задачи производителя/потребителя
// с ограниченным буфером
#include "boundedbuff.inc"
const int ValueRange = 20; // Производятся числа от 0 до 19

semaphore to; // Для исключительного доступа к выводу терминала
semaphore s; // Взаимное исключение для буфера
semaphore n; // Потребляемых элементов в буфере
semaphore e; // Пустых мест в буфере

```

```
int produce(char id)
{
    int tmp;
    tmp = random(ValueRange);
    wait(to);
    cout << "Producer " << id << " produces " << tmp
        << endl;
    signal(to);
    return tmp;
}
void consume(char id, int i)
{
    wait(to);
    cout << "Consumer " << id << " consumes " << i
        << endl;
    signal(to);
}
void producer(char id)
{
    int i;

    for (;;)
    {
        i = produce(id);
        wait(e);
        wait(s);
        append(i);
        signal(s);
        signal(n);
    }
}
void consumer(char id)
{
    int i;

    for (;;)
    {
        wait(n);
        wait(s);
        i = take();
        signal(s);
        signal(e);
        consume(id, i);
    }
}
main()
{
    initialsem(s, 1);
    initialsem(n, 0);
    initialsem(e, SizeOfBuffer);
    initialsem(to, 1);
```

```

cobegin
{
    producer('A');
    producer('B');
    consumer('x');
    consumer('y');
    consumer('z');
}
}

// boundedbuff.inc - реализация ограниченного буфера
const int SizeOfBuffer = 5;
int buffer[SizeOfBuffer];
int in = 0; // Индекс в буфере для очередного добавления
int out = 0; // Индекс в буфере для очередной выборки
void append(int v)
// Добавление v в буфер
// Предполагается, что о переполнении заботятся
// извне, через семафоры или условия
{
    buffer[in] = v;
    in = (in + 1) % SizeOfBuffer;
}
int take()
// Возврат элемента из буфера
// Предполагается, что об исчерпании заботятся
// извне, через семафоры или условия
{
    int tmp;
    tmp = buffer[out];
    out = (out + 1) % SizeOfBuffer;
    return tmp;
}

```

Вот один из примеров выполнения предыдущего решения с ограниченным буфером в BACI.

```

Source file: semprodcons.cm Fri Aug 1 12:36:55 1997
Producer B produces 4
Producer A produces 13
Producer B produces 12
Producer A produces 4
Producer B produces 17
Consumer x consumes 4
Consumer y consumes 13
Producer A produces 16
Producer B produces 11
Consumer z consumes 12
Consumer x consumes 4
Consumer y consumes 17
Producer B produces 6
...

```

## О.4. ПРОЕКТЫ ВАСІ

В этом разделе мы обсудим два основных типа проектов, которые можно реализовать в ВАСІ. Сначала мы рассмотрим проекты, связанные с реализацией операций низкого уровня (например, специальные машинные команды, которые используются для синхронизации доступа процессов к общей основной памяти). Затем будут рассмотрены проекты, которые построены на основе этих низкоуровневых операций (например, классические проблемы синхронизации). Дополнительную информацию по этим проектам можно найти в описаниях проектов, включенных в дистрибутив ВАСІ (чтобы получить решения некоторых из этих проектов, преподаватели должны связаться с авторами). В дополнение к проектам, рассматриваемым в этом разделе, в ВАСІ могут быть реализованы многие задачи, описанные в конце главы 5, “Параллельные вычисления: взаимоисключения и многозадачность”, и в приложении А, “Вопросы параллельности”.

### Реализация примитивов синхронизации

#### Реализация машинных команд

Имеется множество машинных команд, которые можно реализовать в ВАСІ. Например, можно реализовать команду сравнения и обмена, представленную на рис. 5.2. Реализация этих команд должна основываться на атомарной функции, которая возвращает значение типа `int`. Вы можете протестировать свою реализацию машинной команды, построив протокол взаимного исключения поверх своей низкоуровневой операции.

#### Реализация честных семафоров (FIFO)

Операции семафора в ВАСІ реализованы со случайным порядком пробуждения, именно так, как было первоначально определено для семафоров Дейкстры. Однако, как говорилось в разделе 5.3, самая честная стратегия — FIFO. В ВАСІ можно реализовать такие семафоры с указанным порядком пробуждения. В реализации должны быть определены по меньшей мере четыре следующие процедуры.

- `CreateSemaphores()` (для инициализации программного кода)
- `InitSemaphore(int sem_index)` (для инициализации семафора, представленного параметром `sem_index`)
- `FIFOP(int sem_index)`
- `FIFOV(int sem_index)`

Этот код должен быть написан как системная реализация и как таковой должен обрабатывать все возможные ошибки. Другими словами, проектировщик семафора отвечает за создание кода, надежного даже при невежественном, глупом или даже злонамеренном использовании пользователями.

### Семафоры, мониторы и их реализации

Есть много классических задач параллельного программирования: задача производителя/потребителя, задача об обедающих философах, задача читателя/писателя с различными приоритетами, задача спящего парикмахера и задача курильщика. Все эти задачи

могут быть реализованы в ВАСI. В этом разделе мы обсудим нестандартные проекты семафоров и мониторов, которые можно реализовать в ВАСI для дальнейшего содействия пониманию концепций параллелизма и синхронизации.

## **Семафоры A и B**

Рассмотрим следующий набросок программы ВАСI.

```
// Здесь объявлены глобальные семафоры
void A()
{
    Только p() и v()
}
void B()
{
    Только p() и v()
}

main()
{
    // Здесь выполняется инициализация семафоров
    cobegin {
        A(); A(); A(); B(); B();
    }
}
```

Завершите программу, используя наименьшее количество общих семафоров, причем так, чтобы процессы ВСЕГДА завершались в порядке А (любая копия), В (любая копия), А (любая копия), А, В. Используйте опцию командной строки `-t` интерпретатора для отображения завершения процесса. (Имеется множество вариаций этого проекта. Например, четыре параллельных процесса, которые должны завершаться в порядке АВАА, или восемь параллельных процессов, которые должны завершаться в порядке ААВАВАВВ.)

## **Использование бинарных семафоров**

Повторите предыдущий проект, используя бинарные семафоры. Поясните, почему в этом решении необходимы присваивания и инструкции `if-then-else`, хотя они и не были необходимы в решениях предыдущего проекта. Другими словами, объясните, почему в этом случае вы не можете ограничиться только применением операций P и V.

## **Активное ожидание и семафоры**

Сравните производительность решения со взаимным исключением, в котором используется активное ожидание (например, с помощью команды `testset`), с решением с использованием семафоров. Например, сравните решение рассмотренного выше проекта АВААВ с применением семафора и с использованием `testset`. В каждом случае используйте большое количество выполнений (скажем, 1000) для получения надежной статистики. Обсудите ваши результаты, объяснив, почему одна реализация предпочтительнее другой.

## **Семафоры и мониторы**

В духе задачи 5.17 реализуйте в BACI монитор, используя общие семафоры, а затем реализуйте общий семафор с помощью монитора.

### **Общие и бинарные семафоры**

Докажите, что общие и бинарные семафоры одинаково мощны благодаря реализации одного типа семафоров с помощью другого типа и наоборот.

### **Проект монитора**

Напишите программу, содержащую монитор AlarmClock (Будильник). Монитор должен иметь переменную int theClock (инициализируется нулем) и две функции.

- `Tick()`. При каждом вызове увеличивает значение `theClock`. Может выполнять и другие действия, такие как `signalc`, если это необходимо.
- `int Alarm(int id, int delta)`. Блокирует вызывающий ее код с идентификатором `id` как минимум на `delta` тактов `theClock`.

Основная программа также должна иметь две функции.

- `void Ticker()`. Вызывает `Tick()` в бесконечном цикле.
- `void Thread(int id, int myDelta)`. Вызывает `Alarm` в бесконечном цикле.

Можете снабдить монитор любыми другими переменными, которые ему нужны. Монитор должен быть в состоянии принять до пяти одновременных сигналов тревоги.

### **Задача о популярном пекаре**

Из-за популярности пекарни почти каждому клиенту приходится ждать обслуживания. Чтобы улучшить обслуживание, пекарь хочет установить систему билетов, которая гарантирует обслуживание клиентов по очереди. Постройте в BACI реализацию этой системы билетов.

## **О.5. УСОВЕРШЕНСТВОВАНИЯ СИСТЕМЫ ВАСИ**

Мы усовершенствовали систему BACI несколькими путями.

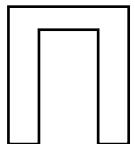
1. Мы реализовали систему BACI на Java (JavaBACI). Эта реализация вместе с нашей оригинальной реализацией BACI на C доступна по адресу [http://inside.mines.edu/fs\\_home/tcamp/baci/baci\\_index.html](http://inside.mines.edu/fs_home/tcamp/baci/baci_index.html). Классы JavaBACI и исходные файлы хранятся в самораспаковывающихся файлах .jar Java. JavaBACI включает в себя все приложения BACI: компиляторы C и Pascal, дизассемблер, архиватор, компоновщик, а также интерпретаторы PCODE в режимах командной строки и графического интерфейса пользователя. Ввод, поведение и вывод программ в JavaBACI идентичны вводу, поведению и выводу программ в нашей C-реализации BACI. Заметим, что в JavaBACI учащиеся продолжают писать параллельные программы на C – или Pascal (но не на Java). JavaBACI будет работать на любом компьютере, на котором установлена виртуальная машина Java.

2. Мы добавили графические пользовательские интерфейсы для JavaBACI и версии BACI для UNIX в С. Оконные среды позволяют пользователю контролировать все аспекты выполнения программы BACI; в частности, пользователь может устанавливать и удалять точки останова (либо по адресу PCODE, либо по строке исходного текста), просматривать значения переменных, стеки времени выполнения и таблицы процессов, а также исследовать чередование выполнения PCODE. Графические интерфейсы пользователя BACI доступны на веб-сайте BACI по адресу [http://inside.mines.edu/fs\\_home/tcamp/baci/index\\_gui.html](http://inside.mines.edu/fs_home/tcamp/baci/index_gui.html). Альтернативный графический интерфейс описан ниже.
3. Мы создали распределенную версию BACI. Как и в случае параллельных программ, правильность распределенных программ трудно доказать без реализации. Распределенный BACI позволяет легко реализовывать распределенные программы. В дополнение к доказательству корректности распределенной программы распределенный BACI можно использовать для тестирования производительности программы. Распределенная система BACI доступна на веб-сайте по адресу [http://inside.mines.edu/fs\\_home/tcamp/baci/dbaci.html](http://inside.mines.edu/fs_home/tcamp/baci/dbaci.html).
4. Наш дизассемблер PCODE предоставляет пользователю аннотированный список файлов PCODE, показывающий мнемонику каждой инструкции PCODE и, если возможно, соответствующий исходный текст, который генерировал данную инструкцию. Этот дизассемблер PCODE включен в систему BACI.
5. Мы добавили возможность раздельной компиляции и внешних переменных для обоих компиляторов (C и Pascal). Система BACI включает в себя архиватор и компоновщик, которые позволяют создавать и использовать библиотеки BACI PCODE. Более подробно этот вопрос освещается в руководстве пользователя BACI в разделе BACI Separate Compilation (раздельная компиляция в BACI).

Система BACI улучшена и другими разработчиками.

1. Студент Дэвид Страт (David Strite), в сотрудничестве с Линдой Налл (Linda Null) из Университета штата Пенсильвания, создал BACI Debugger — отладчик для системы BACI с графическим интерфейсом пользователя. Этот отладчик доступен по адресу <http://cs.hbg.psu.edu/~null/baci>.
2. Используя разработки Университета штата Пенсильвания, Моти Бен-Ари (Moti Ben-Ari) из Израильского института науки им. Вейцмана создал интегрированную среду разработки для обучения параллельному программированию путем имитации параллелизма, именуемую “jBACI”. jBACI доступен по адресу <https://code.google.com/archive/p/jbaci/>.





## ПРИЛОЖЕНИЕ

# УПРАВЛЕНИЕ ПРОЦЕДУРАМИ

В ЭТОМ ПРИЛОЖЕНИИ...

- П.1. Реализация стека**
- П.2. Вызов процедуры и возврат из нее**
- П.3. Реентерабельные процедуры**

Распространенный метод для управления выполнением вызовов процедур и возврата из них использует стеки. В этом приложении обобщены основные свойства стеков и рассмотрено их применение в управлении процедурами.

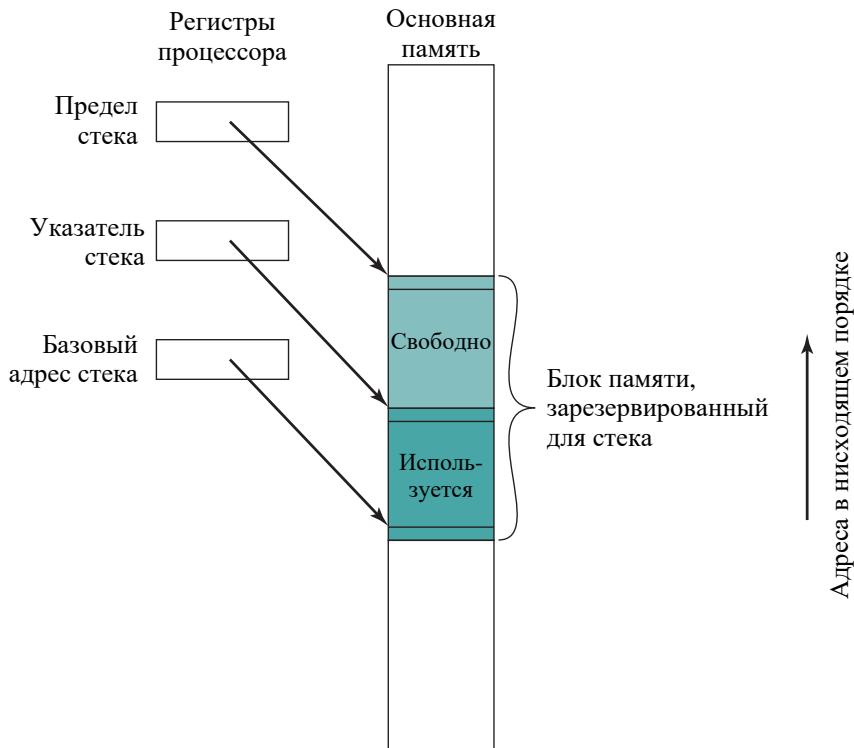
## П.1. РЕАЛИЗАЦИЯ СТЕКА

Стек представляет собой упорядоченный набор элементов, причем одновременно можно получить доступ только к одному из них (последнему добавленному). Точка доступа называется *вершиной* стека. Число элементов в стеке, или его длина, является переменной. Элементы могут добавляться или удаляться только из верхней части стека. По этой причине стек также известен как *список “последним пришел — первым вышел”* (last-in-first-out — LIFO).

Реализация стека требует наличия некоторого множества местоположений для хранения элементов стека. Типичный подход показан на рис. П.1. В основной (или виртуальной) памяти для стека зарезервирован непрерывный блок ячеек памяти. Большую часть времени блок частично заполнен элементами стека, а остальные доступны для роста стека. Для правильной работы стека необходимы три адреса, которые часто хранятся в регистрах процессора.

- **Указатель стека** (stack pointer). Содержит адрес текущей вершины стека. Если элемент добавляется в стек (push) или снимается со стека (pop), указатель уменьшается или увеличивается таким образом, чтобы содержать адрес новой вершины стека.

- **Базовый адрес стека (stack base).** Содержит адрес нижнего местоположения в зарезервированном блоке. Это первое местоположение, которое будет использовано при добавлении элемента в пустой стек. Попытка снять элемент с пустого стека приводит к ошибке времени выполнения.
- **Предел стека (stack limit).** Содержит адрес другого конца зарезервированного блока. Если предпринята попытка поместить элемент в заполненный стек, это приводит к ошибке времени выполнения.

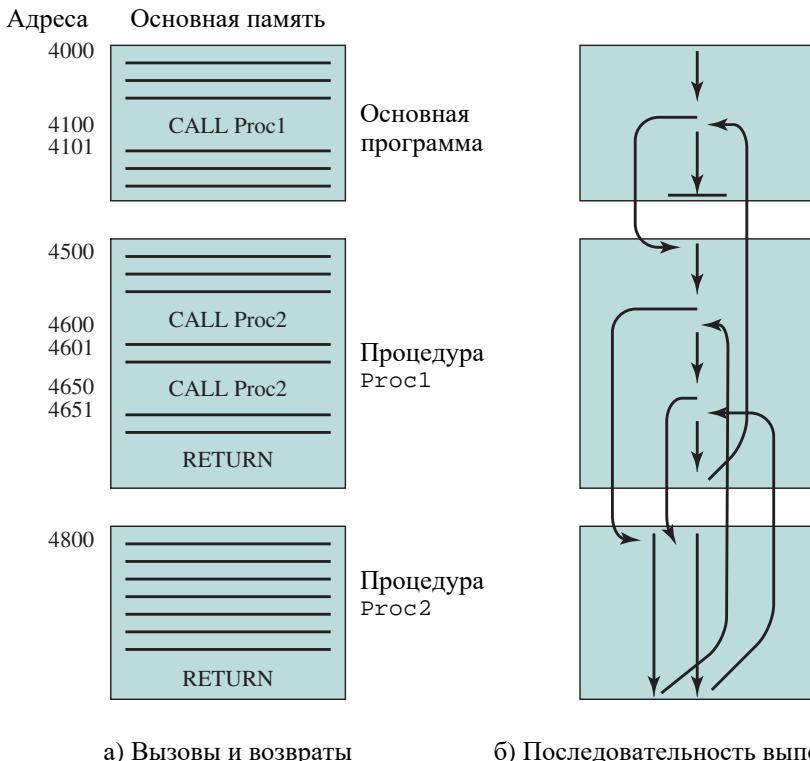


**Рис. П.1.** Типичная организация стека

Традиционно в большинстве современных процессоров базовый адрес стека соответствует наивысшему адресу зарезервированного блока стека, а предел стека соответствует наименьшему адресу блока. Таким образом, стек растет от более высоких адресов к более низким.

## П.2. ВЫЗОВ ПРОЦЕДУРЫ И ВОЗВРАТ ИЗ НЕЕ

Обычный метод управления вызовами процедур и возвратами из них использует стек. Когда процессор выполняет вызов, он помещает (push) в стек адрес возврата. Когда он выполняет возврат, он использует этот адрес из верхней части стека и удаляет (pop) этот адрес из стека. Для вложенных процедур, изображенных на рис. П.2, рис. П.3 иллюстрирует использование стека.



а) Вызовы и возвраты

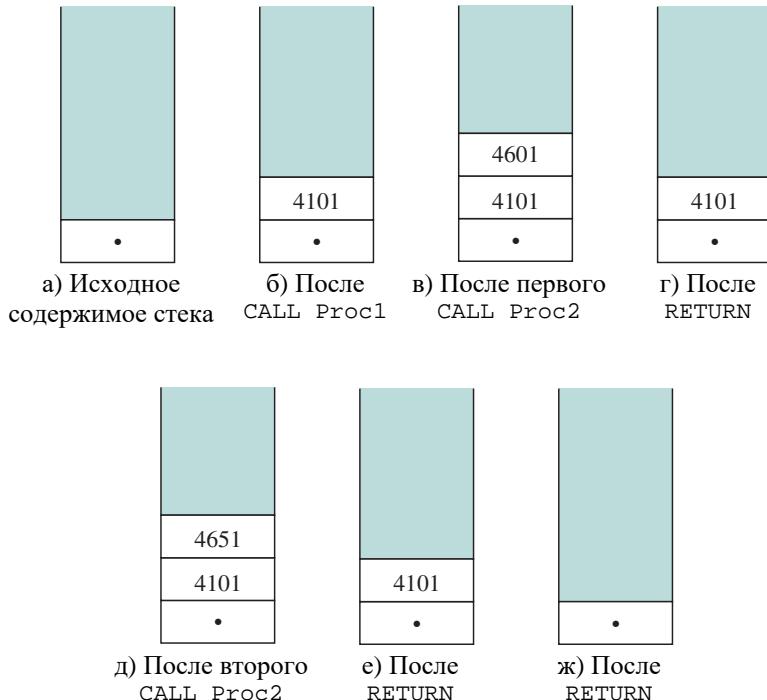
б) Последовательность выполнения

Рис. П.2. Вложенные вызовы процедур

Часто при вызове процедуры необходимо передавать параметры. Один из способов их передачи — в регистрах процессора. Другая возможность — сохранить параметры в памяти сразу после команды CALL. В этом случае возврат из процедуры должен выполняться в место, следующее за параметрами. Оба подхода имеют свои недостатки. Если используются регистры, вызываемая и вызывающая процедуры должны быть написаны согласованно, чтобы обеспечить правильное использование регистров. Хранение параметров в памяти затрудняет передачу переменного количества параметров.

Более гибким является подход к передаче параметров через стек. Когда процессор выполняет вызов, он помещает в стек не только адрес возврата, но и параметры, передаваемые в вызываемую процедуру. Вызываемая процедура может получить доступ к параметрам в стеке. При возврате из процедуры возвращаемые значения также могут быть помещены в стек под адресом возврата. Весь набор параметров, включая адрес возврата, сохраняемый при вызове процедуры, называется **кадром стека** (stack frame).

Соответствующий пример приведен на рис. П.4. Пример относится к процедуре P, в которой объявлены локальные переменные  $x_1$  и  $x_2$ , и к процедуре Q, которая вызывается процедурой P и в которой объявлены локальные переменные  $y_1$  и  $y_2$ . Первый элемент, хранящийся в каждом кадре стека, является указателем на начало предыдущего кадра. Это необходимо, если количество или длина параметров, которые должны быть помещены в стек, являются переменными. Далее сохраняется точка возврата для процедуры, которая соответствует этому кадру стека.



**Рис. П.3.** Использование стека при реализации вложенных процедур на рис. П.2

Наконец, в верхней части кадра стека выделяется место для локальных переменных. Эти локальные переменные могут быть использованы и для передачи параметров. Например, предположим, что когда Р вызывает Q, она передает одно значение параметра. Это значение может быть сохранено в переменной  $y_1$ . Таким образом, на языке высокого уровня в процедуре Р будет инструкция, которая выглядит следующим образом:

```
CALL Q(y1)
```

При выполнении вызова для Q создается новый кадр стека (см. рис. П.4,б), который включает в себя указатель на кадр стека для Р, адрес возврата для Р и две локальные переменные для Q, одна из которых инициализируется переданным значением параметра из Р. Другая локальная переменная,  $y_2$ , является просто локальной переменной, используемой Q в своих вычислениях. Необходимость включения таких локальных переменных в кадр стека обсуждается в следующем разделе.

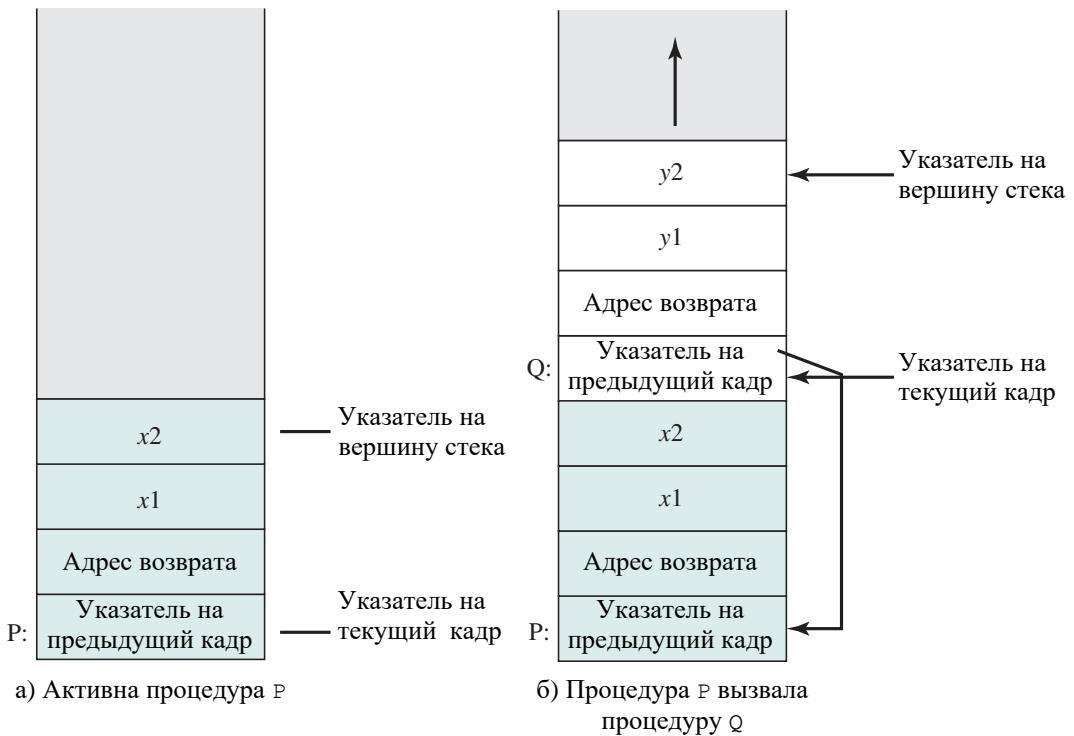
## П.3. РЕЕНТЕРАБЕЛЬНЫЕ ПРОЦЕДУРЫ

Полезной концепцией, в особенности в многопользовательской системе, является реентерабельная процедура. Реентерабельная процедура — это процедура, единая копия кода которой одновременно используется несколькими пользователями. Реентерабельность имеет два ключевых аспекта: программный код не может изменять сам себя, а локальные данные для каждого пользователя должны храниться отдельно. Реентерабельная процедура может быть прервана, вызвана прервавшей ее программой и все равно выполниться правильно после возврата из процедуры.

В многопользовательской системе реентерабельность позволяет более эффективно использовать основную память: в основной памяти хранится одна копия программного кода, но вызвана процедура может быть более чем одним приложением.

Таким образом, реентерабельная процедура должна иметь постоянную часть (инструкции, которые составляют процедуру) и временную часть (указатель на вызывающую программу, а также память для локальных переменных, используемых программой). Каждый выполняемый экземпляр выполняет код из постоянной части, но должен иметь собственную копию локальных переменных и параметров. Временная часть связана с определенным выполняемым экземпляром (именуемым активацией) и называется *записью активации*.

Наиболее удобный способ поддержки реентерабельных процедур — стек. Когда вызывается реентерабельная процедура, ее запись активации может храниться в стеке. Таким образом, запись активации становится частью кадра стека, который создается при вызове процедуры.



**Рис. П.4.** Рост стека при вызове процедур P и Q



# ПРИЛОЖЕНИЕ

P

## eCos

В ЭТОМ ПРИЛОЖЕНИИ...

### P.1. Настраиваемость

### P.2. Компоненты eCos

- Уровень аппаратных абстракций
- Ядро eCos
- Система ввода-вывода
- Стандартные библиотеки C

### P.3. Планировщик eCos

- Планировщик битовой карты
- Планировщик многоуровневой очереди

### P.4. Синхронизация потоков eCos

- Мьютексы
- Семафоры
- Условные переменные
- Флаги событий
- Почтовые ящики
- Циклические блокировки

Встраиваемая настраиваемая (конфигурируемая) операционная система (Embedded Configurable Operating System — eCos) является операционной системой реального времени с открытым исходным кодом, без лицензионных платежей, предназначеннной для встраиваемых приложений. Система ориентирована на небольшие высокопроизводительные встраиваемые системы. Для таких систем встроенная форма Linux или другой коммерческой операционной системы не предоставляет необходимое программное обеспечение. Программное обеспечение eCos было реализовано для различных процессорных платформ, включая Intel IA32, PowerPC, SPARC, ARM, CalmRISC, MIPS и NEC V8xx. Это одна из наиболее широко используемых встроенных операционных систем. Она реализована на языках программирования C/C++.

## P.1. НАСТРАИВАЕМОСТЬ

Встроенная операционная система, достаточно гибкая для использования в широком спектре встроенных приложений и на разнообразных встроенных платформах, должна обеспечивать большую функциональность, чем потребуется для любого конкретного приложения и платформы. Например, многие операционные системы реального времени поддерживают переключение задач, управление параллелизмом и множество механизмов планирования с использованием приоритетов. Для относительно простой встроенной системы вся эта функциональность излишня.

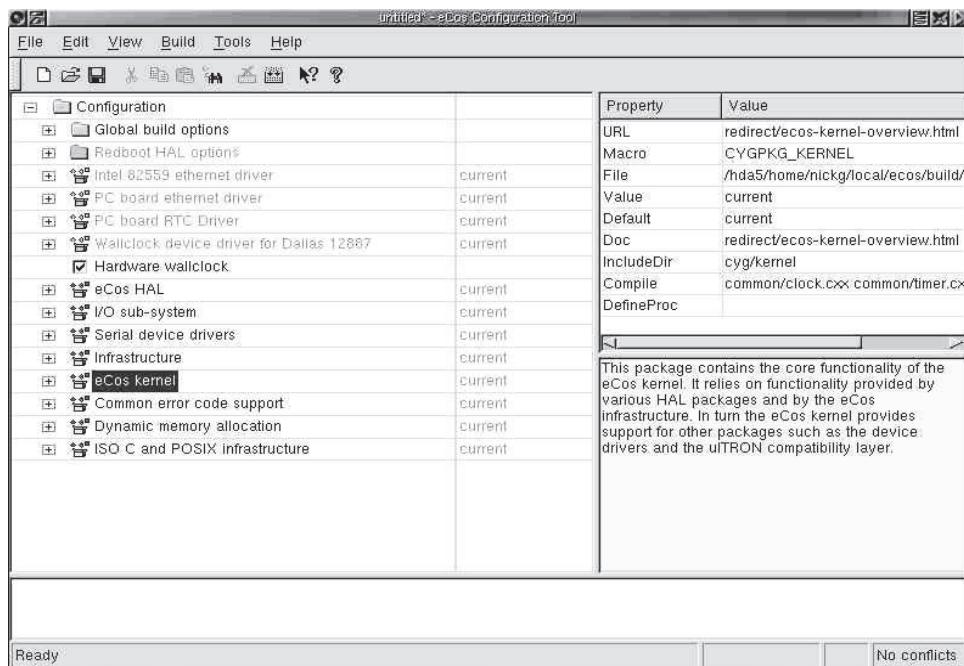
Задача состоит в том, чтобы обеспечить эффективный, удобный для пользователя механизм настройки выбранных компонентов и для включения/отключения определенных функциональных возможностей компонентов. Инструмент настройки eCos, работающий в Windows или Linux, используется для настройки пакета eCos для запуска на целевой встроенной системе.

Полный пакет eCos структурирован иерархически, что делает легкой (используя инструмент конфигурирования) сборку целевой конфигурации. На верхнем уровне eCos состоит из ряда компонентов, и пользователь может выбрать для конфигурации только те компоненты, которые необходимы для целевого приложения.

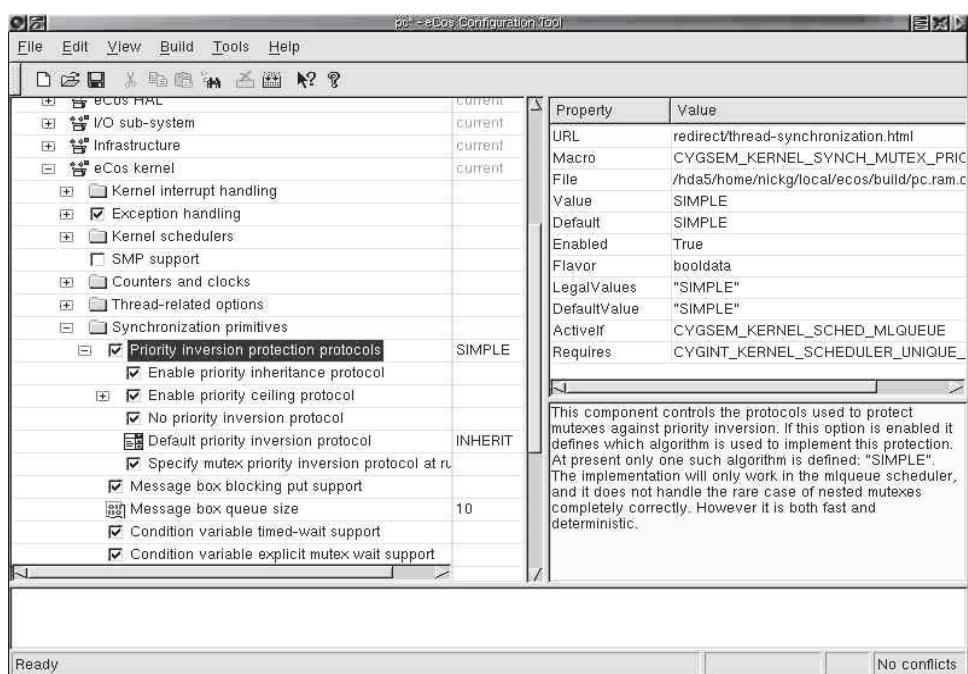
Например, система может иметь определенное устройство последовательного ввода-вывода. Пользователь для своей конфигурации выбирает последовательный ввод-вывод, затем выбирает поддержку одного или нескольких конкретных устройств ввода-вывода. Инструмент конфигурации включает для запрошенной поддержки минимально необходимое программное обеспечение. Пользователь может также выбрать конкретные параметры, такие как используемые скорость передачи данных по умолчанию и размер буферов ввода-вывода.

Этот процесс конфигурации может быть расширен до более мелких уровней детализации, вплоть до уровня отдельных строк кода. Например, инструмент конфигурации предоставляет возможность включения или исключения протокола наследования приоритетов.

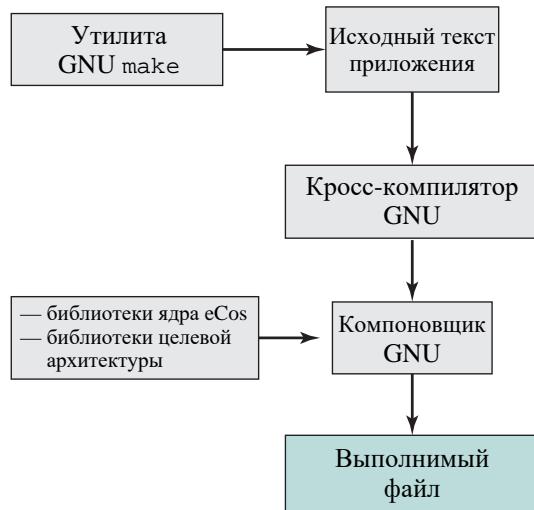
На рис. P.1 показан верхний уровень инструмента настройки eCos с точки зрения пользователя. Каждый из элементов в списке в левом окне можно выбрать (или отменить выбор). Когда элемент выделен, в нижнем правом окне можно прочесть его описание, а верхнее правое окно содержит ссылку на дополнительную документацию и дополнительную информацию о выделенном элементе. Элементы в списке могут быть раскрыты, чтобы предоставить более детальное меню вариантов выбора. На рис. P.2 показаны расширенные настройки ядра eCos. На этом рисунке выбрана для включения обработка исключений, но поддержка SMP (симметричной многопроцессорной обработки) опущена. В общем случае могут быть выбраны или опущены как компоненты, так и отдельные параметры. В некоторых случаях могут быть установлены индивидуальные значения; например, минимально допустимый размер стека является целочисленным значением, которое можно установить или для которого можно оставить заданным значение по умолчанию.



**Рис. Р.1. Инструмент конфигурирования eCos: верхний уровень**



**Рис. Р.2. Инструмент конфигурирования eCos: детали ядра**

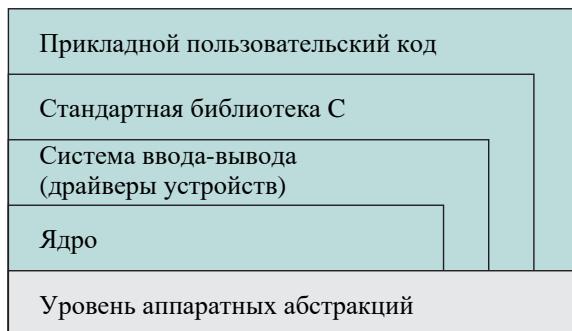


**Рис. Р.3.** Загрузка конфигурации eCos

На рис. Р.3 показан типичный пример общего процесса создания бинарного образа для выполнения во встроенной системе. Этот процесс выполняется в исходной системе, такой как платформа Windows или Linux, а выполнимый образ предназначен для работы в целевой встроенной системе, такой как датчик в промышленной среде. На самом высоком уровне программного обеспечения находится исходный код приложения для конкретного встроенного приложения. Этот код не зависит от eCos, но использует интерфейсы прикладного программирования (API), чтобы находиться поверх программного обеспечения eCos. Может быть как только одна версия исходного текста приложения, так и варианты для разных версий целевой встроенной платформы. В данном примере для выборочного определения того, какие именно части программы должны быть скомпилированы или перекомпилированы (в случае изменения версии исходного текста) используется утилита GNU make, которая и выдает команды для перекомпиляции. Затем кросс-компилятор GNU, работающий на исходной платформе, генерирует бинарный выполнимый код для целевой встроенной платформы. Компоновщик GNU связывает объектный код приложения с кодом, сгенерированным инструментом конфигурации eCos. Этот последний набор программного обеспечения включает в себя отдельные части ядра eCos плюс выбранное программное обеспечение для целевой встроенной системы. Затем результат может быть загружен в целевую систему.

## P.2. Компоненты eCos

Ключевым требованием к дизайну eCos является переносимость на различные архитектуры и платформы с минимальными усилиями. Чтобы удовлетворить это требование, eCos состоит из многоуровневого набора компонентов (рис. Р.4).



**Рис. Р.4.** Многоуровневая структура eCos

## Уровень аппаратных абстракций

Нижним уровнем является уровень аппаратной абстракции (hardware abstraction layer — HAL). Это программное обеспечение, которое представляет согласованный API для верхних уровней и отображает операции верхнего уровня на конкретную аппаратную платформу. Таким образом, HAL для каждой аппаратной платформы различается. На рис. Р.5 приведен пример, демонстрирующий, как HAL абстрагирует аппаратные реализации для одного и того же вызова API на двух разных платформах. Как показано в этом примере, вызов верхнего уровня для разрешения прерываний одинаков на обеих платформах, но реализация функции на языке C для каждой платформы своя.

```

1 #define HAL_ENABLE_INTERRUPTS() \
2     asm volatile ( \
3         "mrs r3, cpsr;" \
4         "bic r3, r3, #0xC0;" \
5         "mrs cpsr, r3;" \
6         : \
7         : \
8         : "r3" \
9     );

```

a) Архитектура ARM

```

1 #define HAL_ENABLE_INTERRUPTS() \
2     CYG_MACRO_START \
3     cyg_uint32 tmp1, tmp2 \
4     asm volatile ( \
5         "mfmsr %0;" \
6         "ori %1,%1,0x800;" \
7         "rlwimi %0,%1,0,16,16;" \
8         "mtmsr %0;" \
9         : "=r" (tmp1), "=r" (tmp2)) \
10    CYG_MACRO_END

```

б) Архитектура PowerPC

**Рис. Р.5.** Две реализации макроса HAL\_ENABLE\_INTERRUPTS()

HAL реализован в виде трех отдельных модулей.

- **Архитектура.** Определяет тип семейства процессоров. Этот модуль содержит код, необходимый для запуска процессора, предоставления прерываний, переключения контекста и другой функциональности, специфичной для архитектуры набора команд данного семейства процессоров.
- **Вариант.** Поддерживает функциональные возможности конкретного процессора семейства. Примером такой поддерживаемой функциональной возможности является встроенный модуль, такой как модуль управления памятью (memory management unit — MMU).
- **Платформа.** Расширяет поддержку HAL для тесно связанных периферийных устройств, таких как контроллеры прерываний и таймеры. Этот модуль определяет платформу или плату, включающую выбранную архитектуру и вариант процессора. Сюда входит код запуска, конфигурации устройств выбора микросхемы, контроллеров прерываний и таймера.

Обратите внимание, что интерфейс HAL может использоваться любым из верхних уровней непосредственно, способствуя написанию эффективного кода.

## Ядро eCos

Ядро eCos было разработано для достижения четырех основных целей.

- **Низкая задержка прерывания.** Время, необходимое для ответа на прерывание и начала выполнения ISR.
- **Низкая задержка переключения задач.** Время, которое проходит с момента, когда поток становится доступным, и до начала фактического выполнения кода.
- **Малый объем памяти.** Ресурсы памяти для программы и данных поддерживаются минимальными, путем разрешения всем компонентам настраивать память по мере необходимости.
- **Детерминированное поведение.** При всех аспектах выполнения производительность ядра должна быть предсказуемой и удовлетворять требованиям приложений реального времени.

Ядро eCos предоставляет основные функциональные возможности, необходимые для разработки многопоточных приложений.

1. Возможность создавать в системе новые потоки либо во время запуска, либо когда система уже работает.
2. Контроль над различными потоками в системе, например управление их приоритетами.
3. Выбор планировщиков, определяющих, какой поток должен в данный момент выполняться.
4. Набор примитивов синхронизации, позволяющих потокам безопасно взаимодействовать и обмениваться данными.
5. Интеграция с системной поддержкой прерываний и исключений.

Некоторая функциональность, которая обычно включена в ядро операционной системы, не включена в ядро eCos. Например, выделение памяти обрабатывается отдельным пакетом. Точно так же каждый драйвер устройства представляет собой отдельный пакет. Различные пакеты объединяются и конфигурируются с использованием технологии конфигурации eCos для удовлетворения требованиям приложения. Это позволяет уменьшить размер ядра. Кроме того, минимализм ядра означает, что для некоторых встроенных платформ ядро eCos вообще не используется. Простые однопоточные приложения могут быть запущены непосредственно над HAL. Такие конфигурации могут включать в себя необходимые функции библиотеки C и драйверы устройств, но позволяют избежать накладных расходов затрат памяти и процессорного времени ядра.

Есть два разных метода использования функций ядра в eCos. Один из них — использование C API ядра. Примерами таких функций являются `cyg_thread_create` и `cyg_mutex_lock`. Они могут быть вызваны непосредственно из кода приложения. С другой стороны, функции ядра могут быть вызваны с использованием пакетов совместимости для существующих API, например для потоков POSIX или µTRON. Пакеты совместимости позволяют коду приложения вызывать стандартные функции, такие как `pthread_create`, а эти функции реализованы с использованием базовых функций, предоставляемых ядром eCos. С помощью пакетов совместимости легко достигается совместное использование кода и возможность повторного использования уже разработанного кода.

## Система ввода-вывода

Система ввода-вывода eCos является каркасом для поддержки драйверов устройств. Множество драйверов для различных платформ предоставляется в конфигурационном пакете eCos. Сюда входят драйверы для последовательных устройств, Ethernet, интерфейсы флеш-памяти и различного межкомпонентного ввода-вывода, такого как PCI и USB. Кроме того, пользователи могут разрабатывать собственные драйверы устройств.

Основное назначение системы ввода-вывода — эффективность без дополнительных программных уровней или посторонних функций. Драйверы устройств обеспечивают необходимые функции для ввода, вывода, буферизации и управления устройством.

Как уже упоминалось, драйверы устройств и другое программное обеспечение более высокого уровня могут быть реализованы непосредственно над HAL, если это уместно. Если необходимы специализированные функции ядра, драйвер устройства реализуется с помощью API ядра. Ядро обеспечивает трехуровневую модель прерывания.

- **Подпрограммы обработки прерываний** (interrupt service routines — ISR). Вызываются в ответ на аппаратное прерывание. Аппаратные прерывания с минимальным вмешательством доставляются в ISR. HAL декодирует аппаратный источник прерывания и вызывает ISR подключенного к прерыванию объекта. Этот ISR может управлять оборудованием, но ему разрешено делать только ограниченный набор вызовов API драйвера. По возвращении управления ISR может запросить, чтобы была запланирована для выполнения его отложенная подпрограмма обслуживания (DSR).
- **Отложенная подпрограмма обслуживания** (deferred service routine — DSR). Вызывается в ответ на запрос ISR. DSR будет запущена, когда это можно будет сделать безопасно, без взаимодействия с планировщиком. Большую часть времени

DSR будет работать сразу после ISR, но если в планировщике находится текущий поток, DSR будет отложена до завершения выполнения потока. DSR разрешено делать больший набор вызовов API драйвера, включая, в частности, возможность вызывать `cyg_drv_cond_signal()` для пробуждения ожидающих потоков.

- **Потоки.** Клиенты драйвера. Могут делать все вызовы API, в частности им разрешено ожидать мьютексы и условные переменные.

В табл. Р.1 и Р.2 показан интерфейс драйвера устройства к ядру. Эти таблицы дают представление о типе функциональности, доступной в ядре для поддержки драйверов устройств. Обратите внимание, что интерфейс драйвера устройства можно настроить для одного или нескольких из следующих механизмов параллелизма: циклических блокировок, условных переменных и мьютексов. Они описаны в следующем разделе этого приложения.

**Таблица Р.1. ИНТЕРФЕЙС ДРАЙВЕРА УСТРОЙСТВА К ЯДРУ ECOS: ПАРАЛЛЕЛЬНОСТЬ**

|                                            |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
|--------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>cyg_drv_spinlock_init</code>         | Инициализировать спин-блокировку в заблокированном или разблокированном состоянии                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| <code>cyg_drv_spinlock_destroy</code>      | Уничтожить спин-блокировку, которая больше не используется                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| <code>cyg_drv_spinlock_spin</code>         | Запросить спин-блокировку, ожидая ее освобождения в бесконечном цикле                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <code>cyg_drv_spinlock_clear</code>        | Очистить спин-блокировку. Это позволяет другому процессору затребовать ее. Если в <code>cyg_drv_spinlock_spin</code> состоянии ожидания находится более одного процессора, то продолжить работу будет разрешено только одному из них                                                                                                                                                                                                                                                                                                                                                                                                           |
| <code>cyg_drv_spinlock_test</code>         | Проверка состояния спин-блокировки. Если она не заблокирована, то результат — TRUE. Если заблокирована, то результат — FALSE                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| <code>cyg_drv_spinlock_spin_intsave</code> | Эта функция ведет себя, как <code>cyg_drv_spinlock_spin</code> , но отключает прерывания, прежде чем пытаться затребовать блокировку. Текущее состояние разрешения прерывания сохраняется в <code>*istate</code> . Прерывания остаются отключенными и по получении спин-блокировки должны быть восстановлены с помощью вызова <code>cyg_drv_spinlock_clear_intsave</code> . Чтобы обеспечить правильное взаимоисключение с кодом, работающим как на других процессорах, так и на текущем процессоре, драйверы устройств должны использовать для запроса и освобождения именно этот вариант функции, а не функции без <code>_intsave()</code> . |
| <code>cyg_drv_mutex_init</code>            | Инициализировать мьютекс                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| <code>cyg_drv_mutex_destroy</code>         | Уничтожить мьютекс. Мьютекс должен быть разблокирован. Во время вызова не должно быть никаких потоков, ожидающих этот мьютекс                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |

## Окончание табл. Р.1

|                                     |                                                                                                                                                                                                                                                                                                                            |
|-------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>cyg_drv_mutex_lock</code>     | Если мьютекс уже заблокирован другим потоком, текущий поток будет ждать, пока этот поток закончится. Если результатом этой функции является FALSE, то ожидание потока было прервано другим потоком. В этом случае мьютекс не будет заблокирован                                                                            |
| <code>cyg_drv_mutex_trylock</code>  | Попытка заблокировать мьютекс, на который указывает аргумент, без ожидания. Если мьютекс уже заблокирован другим потоком, то функция возвращает FALSE. Если функция может заблокировать мьютекс без ожидания, функция возвращает TRUE                                                                                      |
| <code>cyg_drv_mutex_unlock</code>   | Разблокировать мьютекс, на который указывает аргумент. Если есть какие-то потоки, ожидающие блокировки, один из них активируется, чтобы попытаться захватить мьютекс                                                                                                                                                       |
| <code>cyg_drv_mutex_release</code>  | Освободить все потоки, ожидающие мьютекс                                                                                                                                                                                                                                                                                   |
| <code>cyg_drv_cond_init</code>      | Инициализирует условную переменную, связанную с мьютексом. Поток может ожидать эту условную переменную только тогда, когда уже заблокировал связанный с ней мьютекс. Ожидание приводит к тому, что мьютекс будет разблокирован, а когда поток будет пробужден, он автоматически запросит мьютекс перед продолжением работы |
| <code>cyg_drv_cond_destroy</code>   | Уничтожить условную переменную                                                                                                                                                                                                                                                                                             |
| <code>cyg_drv_cond_wait</code>      | Ожидание сигнала условной переменной                                                                                                                                                                                                                                                                                       |
| <code>cyg_drv_cond_signal</code>    | Сигнал условной переменной. Если есть какие-либо потоки, ожидающие эту переменную, по меньшей мере один из них будет пробужден                                                                                                                                                                                             |
| <code>cyg_drv_cond_broadcast</code> | Сигнал переменной состояния. Если есть какие-либо потоки, ожидающие эту переменную, все они будут активированы                                                                                                                                                                                                             |

Таблица Р.2. Интерфейс драйвера устройства к ядру eCos: прерывания

|                                 |                                                                                                                                                                                         |
|---------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>cyg_drv_isr_lock</code>   | Отключение доставки прерываний, предотвращая запуск всех ISR. Эта функция поддерживает счетчик количества вызовов                                                                       |
| <code>cyg_drv_isr_unlock</code> | Включение доставки прерываний, позволяющее запускать ISR. Эта функция уменьшает счетчик, поддерживаемый <code>cyg_drv_isr_lock</code> , и разрешает прерывания только при его обнулении |
| <code>cyg_ISR_t</code>          | Определение прототипа ISR                                                                                                                                                               |
| <code>cyg_drv_dsr_lock</code>   | Отключение планирования DSR. Эта функция поддерживает счетчик количества вызовов                                                                                                        |
| <code>cyg_drv_dsr_unlock</code> | Включение планирования DSR. Эта функция уменьшает счетчик, поддерживаемый <code>cyg_drv_dsr_lock</code> , и разрешает планирование DSR только при его обнулении                         |
| <code>cyg_DSR_t</code>          | Определение прототипа DSR                                                                                                                                                               |

|                                                                                                            |                                                                                                                                                                                                                                                                                                                                                                                             |
|------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>cyg_drv_interrupt_create</code>                                                                      | Создает объект прерывания и возвращает его дескриптор                                                                                                                                                                                                                                                                                                                                       |
| <code>cyg_drv_interrupt_delete</code>                                                                      | Отсоединяет прерывание от вектора и освобождает память для повторного использования                                                                                                                                                                                                                                                                                                         |
| <code>cyg_drv_interrupt_attach</code>                                                                      | Присоединяет прерывание к вектору, чтобы происходящие прерывания доставлялись ISR                                                                                                                                                                                                                                                                                                           |
| <code>cyg_drv_interrupt_detach</code>                                                                      | Отсоединяет прерывание от вектора, чтобы происходящие прерывания не доставлялись ISR                                                                                                                                                                                                                                                                                                        |
| <code>cyg_drv_interrupt_mask</code>                                                                        | Программирует контроллер прерываний таким образом, чтобы остановить доставку прерываний для заданного вектора                                                                                                                                                                                                                                                                               |
| <code>cyg_drv_interrupt_mask_intunsafe</code>                                                              | Программирует контроллер прерываний таким образом, чтобы остановить доставку прерываний для заданного вектора. Эта версия отличается от <code>cyg_drv_interrupt_mask</code> тем, что не является безопасной с точки зрения прерываний. Так что в ситуациях, в которых, например, прерывания уже известны как отключенные, эта функция может быть вызвана, чтобы избежать накладных расходов |
| <code>cyg_drv_interrupt_unmask,</code><br><code>cyg_drv_interrupt_</code><br><code>unmask_intunsafe</code> | Программирует контроллер прерываний таким образом, чтобы разрешить пересылку прерываний по заданному вектору                                                                                                                                                                                                                                                                                |
| <code>cyg_drv_interrupt_acknowledge</code>                                                                 | Выполняет любую обработку, необходимую контроллеру прерываний и процессору, чтобы отменить текущий запрос прерывания                                                                                                                                                                                                                                                                        |
| <code>cyg_drv_interrupt_configure</code>                                                                   | Программирует контроллер прерываний с использованием характеристик источника прерывания                                                                                                                                                                                                                                                                                                     |
| <code>cyg_drv_interrupt_level</code>                                                                       | Программирует контроллер прерываний для доставки данного прерывания на переданном уровне приоритета                                                                                                                                                                                                                                                                                         |
| <code>cyg_drv_interrupt_set_cpu</code>                                                                     | В многопроцессорных системах приводит к тому, что все прерывания по данному вектору направляются указанному процессору. Впоследствии все такие прерывания будут обрабатываться этим процессором                                                                                                                                                                                             |
| <code>cyg_drv_interrupt_get_cpu</code>                                                                     | В многопроцессорных системах эта функция возвращает идентификатор процессора, которому в настоящее время доставляются прерывания по данному вектору                                                                                                                                                                                                                                         |

## Стандартные библиотеки С

Предоставляется полная стандартная библиотека времени выполнения языка программирования С. В операционную систему включена также полная математическая библиотека времени выполнения математических функций высокого уровня, включая полную библиотеку IEEE-754 для работы с плавающей точкой для платформ без аппаратной поддержки чисел с плавающей точкой.

## P.3. Планировщик eCos

Ядро eCos может быть настроено для предоставления одного из двух вариантов планировщика: планировщик образов и многоуровневый планировщик очереди. Пользователь конфигурации выбирает подходящий для его среды и приложений планировщик. Планировщик образов обеспечивает эффективное планирование для системы с небольшим количеством потоков, которые могут быть активными в любой момент времени. Планировщик с несколькими очередями подходит, если число потоков является динамическим или если желательно иметь несколько потоков с одинаковым уровнем приоритетов. Многоуровневый планировщик также необходим, если требуется разделение времени.

### Планировщик битовой карты

Планировщик битовой карты (*bitmap scheduler*) поддерживает несколько уровней приоритетов, но на каждом уровне приоритета в любой момент времени может существовать только один поток. Решения по планированию при таком планировании довольно просты (см. рис. P.6, а). Когда заблокированный поток готов к запуску, он может выгрузить поток с более низким приоритетом. Когда работающий поток приостанавливается, диспетчеризуется готовый поток с наивысшим приоритетом. Поток может быть приостановлен, потому что он заблокирован примитивом синхронизации, прерван или сам отдает управление. Поскольку для каждого уровня приоритета имеется не более одного потока, планировщик не должен принимать решение о том, какой поток с заданным уровнем приоритета должен быть отправлен на выполнение.

Планировщик битовой карты настраивается для работы с 8, 16 или 32 уровнями приоритета. Поддерживается простая битовая карта потоков, которые готовы к выполнению. Для принятия решения о планировании планировщик должен всего лишь определить положение старшего значащего бита в битовой карте.

### Планировщик многоуровневой очереди

Как и планировщик битовой карты, планировщик многоуровневой очереди поддерживает до 32 уровней приоритетов. Данный планировщик позволяет создавать несколько активных потоков на каждом уровне приоритета, будучи ограниченным только системными ресурсами.

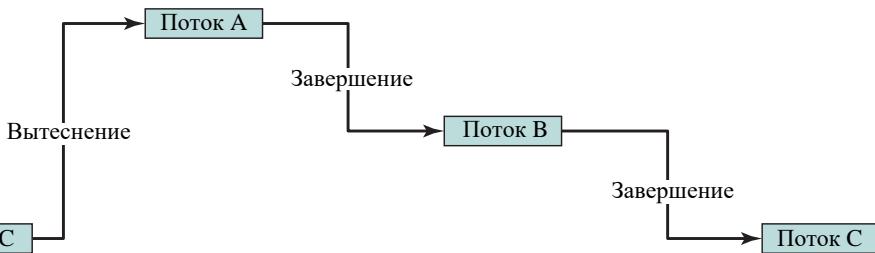
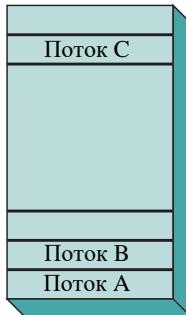
На рис. P.6, б иллюстрируется природа планировщика многоуровневой очереди. Структура данных представляет количество готовых потоков на каждом уровне приоритета. Когда заблокированный поток становится готовым к запуску, он может вытеснить поток с более низким приоритетом. Как и в планировщике битовой карты, работающий поток может быть заблокирован примитивом синхронизации, из-за прерывания или из-за отказа от управления. Когда поток заблокирован, планировщик должен сначала определить, готов ли к выполнению один или несколько потоков с тем же уровнем приоритета, что и заблокированный поток. Если это так, планировщик выбирает тот, который находится в начале очереди. В противном случае планировщик ищет следующий уровень приоритета с одним или несколькими готовыми потоками и диспетчеризует один из этих потоков.

Очередь планировщика  
битовой карты

Максимальный приоритет: 31

•  
•  
•

Минимальный приоритет: 0



a) Операции планировщика битовой карты

Многоуровневая очередь планирования

Максимальный  
приоритет: 31

•  
•  
•

Минимальный  
приоритет: 0

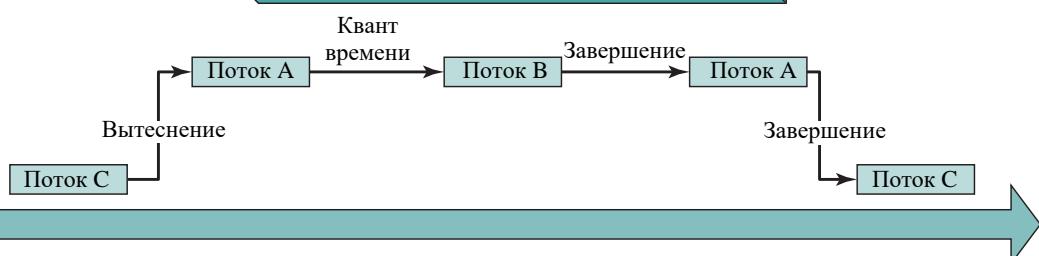
Поток С

Поток В Поток А

Квант  
времени

Завершение

Завершение



б) Операции планировщика многоуровневой очереди

**Рис. Р.6.** Варианты планировщика eCos

Кроме того, планировщик многоуровневой очереди может быть настроен на квантование времени. Таким образом, если поток работает и имеется один или несколько готовых потоков с тем же уровнем приоритета, планировщик приостановит работающий поток после одного кванта времени и выберет следующий поток в очереди на этом же уровне приоритета (круговая стратегия в рамках одного уровня приоритета). Не все приложения требуют выделения времени. Например, приложение может содержать только потоки, которые регулярно блокируются по некоторой иной причине. Для таких приложений пользователь может отключить квантование времени, что уменьшает накладные расходы, связанные с прерываниями таймера.

## P.4. Синхронизация потоков eCos

Ядро eCos может быть настроено на включение одного или нескольких из шести различных механизмов синхронизации потоков. К ним относятся классические механизмы синхронизации, такие как мьютексы, семафоры и условные переменные. Кроме того, eCos поддерживает два механизма синхронизации/коммуникации, которые распространены в системах реального времени, а именно — флаги событий и почтовые ящики. Наконец, ядро eCos поддерживает циклические блокировки, которые полезны в SMP (симметричных многопроцессорных) системах.

### Мьютексы

Мьютексы были введены в главе 6, “Параллельные вычисления: взаимоблокировка и голодание”. Напомним, что мьютекс используется для обеспечения взаимоисключающего доступа к ресурсу, позволяя получить доступ к ресурсу одновременно только одному потоку. У мьютекса есть только два состояния: заблокирован и разблокирован. Этим он похож на бинарный семафор: когда мьютекс заблокирован одним потоком, любой другой поток, пытающийся заблокировать мьютекс, блокируется в ожидании; когда мьютекс разблокирован, то один из потоков, заблокированных этим мьютексом, разблокируется и может заблокировать мьютекс и получить доступ к ресурсу.

Мьютекс отличается от бинарного семафора в двух отношениях. Во-первых, поток, который блокирует мьютекс, должен быть тем же, что и разблокирующий его. В случае бинарных семафоров вполне возможно, что один поток блокирует бинарный семафор, а другой его разблокирует. Другим отличием является то, что мьютекс обеспечивает защиту от инверсии приоритета, тогда как семафор этого не делает.

Ядро eCos можно настроить как для поддержки протокола наследования приоритетов, так и для протокола потолка приоритетов (оба они описаны в главе 10, “Многопроцессорное планирование и планирование реального времени”).

### Семафоры

Ядро eCos обеспечивает поддержку семафоров со счетчиком. Вспомните из главы 5, “Параллельные вычисления: взаимоисключения и многозадачность”, что семафор со счетчиком является целочисленным значением, используемым для передачи сигналов между потоками. Функция `cug_semaphore_init` используется для инициализации семафора. Функция отправки сообщения семафору `cug_semaphore_post` увеличивает счетчик семафора. Если новое значение счетчика меньше или равно нулю, то поток,

ожидающий этот семафор, пробуждается. Функция `sys_semaphore_wait` проверяет значение счетчика семафоров. Если оно равно нулю, поток, вызывающий эту функцию, будет ожидать семафора. Если счетчик не равен нулю, то счетчик уменьшается и поток продолжает выполнение.

Семафоры со счетчиками подходят для того, чтобы позволить потокам ждать, пока произойдет событие. Событие может быть сгенерировано производителем потока или DSR в ответ на аппаратное прерывание. С каждым семафором связан целочисленный счетчик, который отслеживает количество еще не обработанных событий. Если этот счетчик равен нулю, попытка потока потребителя ожидать семафора будет блокировать его до тех пор, пока какой-либо другой поток или DSR не увеличит значение счетчика семафора. Если счетчик больше нуля, то попытка ожидания семафора приведет к уменьшению его счетчика и немедленному возврату из функции. Увеличение значения семафора вызовет пробуждение первого из ожидающих потоков; он продолжит выполнение операции ожидания семафора и тут же уменьшит значение его счетчика.

Еще одно применение семафоров — для определенных форм управления ресурсами. Счетчик соответствует количеству ресурсов определенного типа, доступных в данный момент, причем потоки используют ожидание семафора, чтобы запросить ресурс, и отправляют ему сообщения при освобождении ресурса. На практике для подобных операций обычно гораздо лучше подходят условные переменные.

## Условные переменные

Условная переменная используется для блокировки потока до тех пор, пока некоторое условие не станет истинным. Условные переменные используются вместе с мьютексами, чтобы позволить нескольким потокам получать доступ к общим данным. Их можно использовать для реализации мониторов описанного в главе 6, “Параллельные вычисления: взаимоблокировка и голодание”, типа (см., например, рис. 6.14). Основные команды для работы с условными переменными следующие.

- `sys_cond_wait`. Заставляет текущий поток ожидать указанную условную переменную и одновременно разблокирует мьютекс, присоединенный к условной переменной.
- `sys_cond_signal`. Пробуждает один из потоков, ожидающих эту условную переменную, в результате чего поток становится владельцем мьютекса.
- `sys_cond_broadcast`. Пробуждает все потоки, ожидающие эту условную переменную. Каждый поток, который ожидал условную переменную, при его выполнении становится владельцем мьютекса.

В eCos условные переменные обычно используются в сочетании с мьютексами для реализации долгосрочных ожиданий выполнения некоторых условий. Рассмотрим следующий пример. На рис. Р.7 определен набор функций для управления доступом к пулу ресурсов с использованием мьютексов. Мьютекс используется для того, чтобы сделать распределения и освобождения ресурсов из пула атомарными. Функция `res_allocate` проверяет, доступна ли одна или несколько единиц ресурса, и, если это так, занимает одну единицу. Эта операция защищена мьютексом, поэтому никакой другой поток не может проверить или изменить пул ресурсов, пока мьютекс контролируется данным потоком. Функция `res_free` позволяет потоку освободить одну единицу ресурса, которую он получил ранее. Эта операция также делается атомарной с помощью мьютекса.

```

cyg_mutex_t res_lock;
res_t res_pool[RES_MAX];
int res_count = RES_MAX;

void res_init(void)
{
    cyg_mutex_init(&res_lock);
    <Заполнение пула ресурсов>
}

res_t res_allocate(void)
{
    res_t res;
    cyg_mutex_lock(&res_lock);           // Блокировка мьютекса
    if( res_count == 0 )                // Проверка доступного ресурса
        res =RES_NONE;                 // return RES_NONE, если ресурса нет
    else {
        res_count--;                  // Выделение ресурса
        res =res_pool[res_count];
    }
    cyg_mutex_unlock(&res_lock); // Разблокирование мьютекса
    return res;
}
void res_free(res_t res)
{
    cyg_mutex_lock(&res_lock);           // Блокировка мьютекса
    res_pool[res_count] =res;           // Освобождение ресурса
    res_count++;
    cyg_mutex_unlock(&res_lock); // Разблокирование мьютекса
}

```

**Рис. Р.7.** Управление доступом к пулу ресурсов с использованием мьютексов

В этом примере, если поток пытается получить доступ к ресурсу, но ни один ресурс не доступен, функция возвращает RES\_NONE. Предположим, однако, что мы хотим, чтобы поток не возвращал RES\_NONE, а был заблокирован в ожидании, пока ресурс станет доступным. На рис. Р.8 это достигается с помощью условной переменной, связанной с мьютексом. Когда функция res\_allocate обнаруживает, что ресурсов нет, она вызывает cyg\_cond\_wait. Эта последняя функция разблокирует мьютекс и переводит вызывающий поток в состояние ожидания условной переменной. Когда в итоге вызывается res\_free, она помещает ресурс обратно в пул и вызывает cyg\_cond\_signal, чтобы разбудить любой поток, ожидающий условную переменную. Когда этот ожидающий поток снова активируется, перед возвратом из cyg\_cond\_wait он заблокирует мьютекс.

```

cyg_mutex_t res_lock;
cyg_cond_t res_wait;
res_t res_pool[RES_MAX];
int res_count =RES_MAX;
void res_init(void)
{
    cyg_mutex_init(&res_lock);
    cyg_cond_init(&res_wait, &res_lock);
    <fill pool with resources>
}
res_t res_allocate(void)
{
    res_t res;
    cyg_mutex_lock(&res_lock);           // Блокировка мьютекса
    while( res_count == 0 )             // Ожидание ресурса
        cyg_cond_wait(&res_wait);
    res_count--;                      // Выделение ресурса
    res =res_pool[res_count];
    cyg_mutex_unlock(&res_lock);       // Разблокирование мьютекса
    return res;
}
void res_free(res_t res)
{
    cyg_mutex_lock(&res_lock);           // Блокировка мьютекса
    res_pool[res_count] =res;           // Освобождение ресурса
    res_count++;
    cyg_cond_signal(&res_wait);         // Активация любого
    // ожидавшего потока
    cyg_mutex_unlock(&res_lock);       // Разблокирование мьютекса
}

```

**Рис. Р.8.** Управление доступом к пулу ресурсов с использованием мьютексов и условных переменных

В этом примере (и в использовании условных переменных в целом) есть две важные особенности. Во-первых, разблокировка мьютекса и ожидание в `cyg_cond_wait` являются атомарными: никакой другой поток не может выполняться между разблокировкой и ожиданием. Если бы это было не так, то вызов `res_free` каким-либо другим потоком мог бы освободить ресурс, но вызов `cyg_cond_signal` был бы потерян, и первый поток оказался бы в состоянии ожидания при наличии доступных ресурсов.

Вторая особенность заключается в том, что вызов `cyg_cond_wait` находится в цикле `while`, а не просто в инструкции `if`. Это связано с необходимостью повторной блокировки мьютекса в `cyg_cond_wait`, когда сигнализируемый поток повторно пробуждается. Если в очереди уже есть другие потоки, то данный поток должен ждать, чтобы выполнить блокировку. В зависимости от планировщика и порядка очереди многие другие потоки, возможно, вошли в критический участок раньше данного потока. Таким образом, ожидаемое условие может оказаться ложным. Использование цикла вокруг операции ожидания условной переменной — единственный способ гарантировать, что ожидаемое условие остается в силе после ожидания.

## Флаги событий

Флаг события представляет собой 32-битовое слово, используемое в качестве механизма синхронизации. Код приложения может ассоциировать различные события с каждым битом в флаге. Поток может ожидать либо отдельного события, либо комбинации событий, проверяя один или несколько битов в соответствующем флаге. Поток может быть заблокирован до тех пор, пока будут установлены все требуемые биты (И) или пока будет установлен хотя бы один из битов (ИЛИ). Сигнализирующий поток может устанавливать или сбрасывать биты в зависимости от конкретных условий или событий, так что соответствующий поток разблокируется. Например, бит 0 может представлять завершение конкретной операции ввода-вывода, делая данные доступными, а бит 1 может указывать, что пользователь нажал кнопку запуска. Поток-производитель или DSR может установить эти два бита, и поток-потребитель, ожидающий этих двух событий, будет разбужен.

Поток может ожидать одно или несколько событий, используя функцию `sug_flag_wait`, которая принимает три аргумента: конкретный флаг события, комбинацию битовых позиций в флаге и параметр режима. Параметр режима указывает, будет ли поток блокироваться до тех пор, пока будут установлены все биты (И) или пока будет установлен хотя бы один из битов (ИЛИ). Этот же параметр также может указывать, что при успешном ожидании флаг события должен быть полностью сброшен (все биты становятся нулевыми).

## Почтовые ящики

Почтовые ящики (mailbox), также называемые ящиками сообщений, представляют собой механизм синхронизации eCos, который обеспечивает средства для обмена информацией двумя потоками. В разделе 5.5 представлено общее обсуждение синхронизации с использованием передачи сообщений; здесь же мы рассмотрим специфику версии eCos.

Механизм почтового ящика eCos может быть настроен для блокирующих или неблокирующих действий как на отправляющей, так и на принимающей стороне. Максимальный размер очереди сообщений, связанной с заданным почтовым ящиком, также можно настроить.

Примитив отправки сообщения включает два аргумента: дескриптор почтового ящика и указатель на само сообщение. Есть три варианта этого примитива.

- `sug_mbox_put`. Если в почтовом ящике есть свободный слот, то новое сообщение размещается в нем; если есть ожидающий сообщения поток, он будет разбужен, чтобы получить сообщение. Если почтовый ящик в данный момент заполнен, `sug_mbox_put` блокируется до тех пор, пока станут доступными соответствующая операция получения и слот.
- `sug_mbox_timed_put`. То же, что и `sug_mbox_put`, если есть свободный слот. Иначе говоря, функция будет ждать заданное количество времени и поместит сообщение, если за это время будет в наличии доступный слот. Если срок истекает без отправки сообщения, функция возвращает `false`. Таким образом, `sug_mbox_timed_put` блокируется только на время, не большее указанного аргументом вызова.
- `sug_mbox_tryput`. Это неблокирующая версия, которая возвращает `true`, если сообщение отправлено успешно, и `false`, если почтовый ящик заполнен.

Точно так же есть три варианта примитива получения сообщения.

- `cug_mbox_get`. Если в указанном почтовом ящике есть ожидающее сообщение, `cug_mbox_get` возвращаетется с сообщением, помещенным в почтовый ящик. В противном случае эта функция блокируется, пока не будет выполнена соответствующая операция отправки сообщения.
- `cug_mbox_timed_get`. Немедленно возвращает сообщение, если оно доступно. В противном случае функция будет ждать, пока появится сообщение или пока пройдет указанное аргументом время. Если срок истекает без получения сообщения, функция возвращает нулевой указатель. Таким образом, `cug_mbox_timed_get` блокируется только на время, не большее указанного аргументом вызова.
- `cug_mbox_tryget`. Это неблокирующая версия, которая возвращает сообщение, если такое доступно, и нулевой указатель, если почтовый ящик пуст.

## Циклические блокировки

Циклическая блокировка представляет собой флаг, который поток может проверить перед выполнением определенного фрагмента код. Вспомните из нашего обсуждения циклической блокировки (спин-блокировки) Linux в главе 6, “Параллельные вычисления: взаимоблокировка и голодание”, основные ее свойства. Только один поток одновременно может получить спин-блокировку. Попытки любого другого потока получить ту же блокировку будут продолжаться (циклически) до тех пор, пока он сможет ее получить. По сути, спин-блокировка строится на целочисленной переменной в памяти, которая проверяется каждым потоком, прежде чем он входит в свой критический участок. Если значение переменной равно 0, поток устанавливает ее значение равным 1 и входит в критический участок. Если значение переменной отлично от нуля, поток постоянно проверяет переменную, пока ее значение станет равным нулю.

Циклическая блокировка не должна использоваться в однопроцессорной системе. В качестве примера ее опасности рассмотрим однопроцессорную систему с вытесняющим планированием, при котором поток с более высоким приоритетом пытается получить спин-блокировку, уже удерживаемую потоком с более низким приоритетом. Поток с более низким приоритетом не может выполняться, чтобы завершить свою работу и освободить спин-блокировку, потому что поток с более высоким приоритетом его вытесняет. Поток с более высоким приоритетом может выполнятся, но “застрять” на проверке значения блокировки. В результате поток с более высоким приоритетом будет просто вечно зациклен, в то время как поток с меньшим приоритетом не получит шанса запуститься и освободить спин-блокировку. В системе SMP текущий владелец спин-блокировки может продолжить работу на другом процессоре.

# Глоссарий

Tlqm: @it\_boooks

## Beowulf

Класс кластерных вычислений, который фокусируется на минимизации соотношения цены к производительности всей системы без ущерба для ее возможности выполнять вычислительную работу, для которой она предназначена. Большинство систем Beowulf реализованы на компьютерах под управлением Linux

## В-дерево (B-tree)

Методика организации индексов. Чтобы время доступа было минимальным, ключи данных хранятся в сбалансированной иерархии, которая постоянно перестраивается по мере вставки и удаления элементов. Таким образом, все узлы всегда имеют примерно одинаковое количество ключей

## Адресное пространство (address space)

Диапазон адресов, доступных компьютерной программе

## Архитектура связи (communications architecture)

Аппаратная и программная структура, которая реализует функции связи

## Асинхронная операция (asynchronous operation)

Операция, которая выполняется без регулярной или предсказуемой связи по времени с определенным событием, например вызов диагностической процедуры ошибки, которая может получить управление в любое время во время выполнения компьютерной программы

## База данных (database)

Набор взаимосвязанных данных, часто с контролируемой избыточностью, организованный так, чтобы обслуживать одно или несколько приложений; данные хранятся таким образом, чтобы они могли использоваться различными программами, не заботясь о структуре данных или их организации. Распространенный подход, используемый для добавления новых данных и изменения и выборки существующих данных

## Базовый адрес (base address)

Адрес, который используется в качестве исходного при расчете адресов в ходе выполнения компьютерной программы

|                                                                             |                                                                                                                                                                                                                                                                                                                                                                                 |
|-----------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Бинарный семафор<br/>(binary semaphore)</b>                              | Семафор, который принимает только значения 0 и 1. Бинарный семафор позволяет одновременный доступ к общему критическому ресурсу только одному процессу или потоку                                                                                                                                                                                                               |
| <b>Блок (block)</b>                                                         | 1. Коллекция последовательных записей, записанных как единый модуль; модули разделены межблочными промежутками.<br>2. Группа битов, которые передаются как единое целое                                                                                                                                                                                                         |
| <b>Бригадное планирование<br/>(gang scheduling)</b>                         | Планирование набора связанных потоков для одновременного запуска на множестве процессоров в отношении “один к одному”                                                                                                                                                                                                                                                           |
| <b>Брокер объектных запросов<br/>(object request broker)</b>                | Сущность в объектно-ориентированной системе, действующая как посредник для запросов, отправленных с клиента на сервер                                                                                                                                                                                                                                                           |
| <b>Буфер быстрой переадресации<br/>(translation lookaside buffer — TLB)</b> | Высокоскоростной кеш, используемый для хранения недавно использованных ссылок на записи таблицы страниц как части схемы виртуальной памяти со страницей загрузкой. TLB уменьшает частоту обращений к основной памяти для получения записей таблицы страниц                                                                                                                      |
| <b>Буферизация (spooling)</b>                                               | Использование вторичной памяти в качестве буферного хранилища для уменьшения обработки задержки при передаче данных между периферийным оборудованием и процессорами компьютера                                                                                                                                                                                                  |
| <b>Взаимоблокировка<br/>(deadlock)</b>                                      | 1. Тупиковая ситуация, возникающая, когда несколько процессов ожидают доступности ресурса, который не может стать доступным, потому что удерживается другим процессом, который находится в аналогичном состоянии ожидания. 2. Тупиковая ситуация, когда несколько процессов ожидают действия или ответа от другого процесса, который находится в аналогичном состоянии ожидания |
| <b>Взаимоисключение<br/>(mutual exclusion)</b>                              | Состояние, при котором имеется множество процессов, но в любой момент времени только один из них может получить доступ к данному ресурсу или выполнить данную функцию. См. <i>критический участок</i>                                                                                                                                                                           |
| <b>Виртуальная машина<br/>(virtual machine)</b>                             | Экземпляр операционной системы вместе с одним или несколькими приложениями, работающий в изолированном разделе на компьютере. Позволяет одновременно запускать на одном компьютере разные операционные системы, а также предотвращать взаимодействие приложений одно с другим                                                                                                   |

**Виртуальная память  
(virtual memory)**

Место хранения, которое пользователь компьютерной системы может рассматривать как адресуемую основную память, в которой виртуальные адреса отображаются в реальные адреса. Размер виртуальной памяти ограничен схемой адресации компьютерной системы и объемом доступной вторичной памяти, а не фактическим количеством основной памяти

**Виртуальный адрес  
(virtual address)**

Адрес места хранения в виртуальной памяти

**Вирус (virus)**

Программное обеспечение, которое при запуске пытается скопировать себя в другой выполняемый файл; при успешном выполнении файл называется зараженным. Когда выполняется зараженный код, выполняется и вирус

**Включенное прерывание  
(enabled interrupt)**

Условие, обычно создаваемое операционной системой, во время которого процессор будет отвечать на сигналы запроса прерывания указанного класса

**Внешняя фрагментация  
(external fragmentation)**

Происходит, когда память разделена на разделы переменного размера, соответствующие блокам данных, назначенным памяти (например, сегментам в основной памяти). При перемещении сегментов в память и из нее между занятymi частями памяти будут возникать промежутки

**Внутренняя фрагментация  
(internal fragmentation)**

Происходит, когда память разделена на разделы фиксированного размера (например, кадры страниц в основной памяти, физические блоки на диске). Если блок данных назначается одному или нескольким разделам, то в последнем разделе может остаться неиспользованная память. Это происходит, когда последняя часть данных будет меньше последнего раздела

**Вредоносное программное обеспечение  
(malicious software)**

Любое программное обеспечение, предназначенное для нанесения ущерба или использования ресурсов целевого компьютера. Вредоносное программное обеспечение часто скрывается внутри легального программного обеспечения (или маскируется под него). В некоторых случаях распространяется на другие компьютеры через электронную почту или зараженные диски. К типам вредоносного программного обеспечения относятся вирусы, троянские кони, черви и скрытое программное обеспечение для запуска атаки отказа в обслуживании

|                                                                   |                                                                                                                                                                                                                                                                                                                                      |
|-------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Время отклика<br/>(response time)</b>                          | Время между окончанием передачи сообщения с запросом и началом получения ответного сообщения; изменяется на терминале                                                                                                                                                                                                                |
| <b>Время цикла памяти<br/>(memory cycle time)</b>                 | Время, которое требуется, чтобы прочитать одно машинное слово из памяти или записать одно слово в память. Эта величина обратна скорости, с которой слова могут быть прочитаны из памяти или записаны в нее                                                                                                                           |
| <b>Вторичная память<br/>(secondary memory)</b>                    | Память, расположенная вне самой компьютерной системы, которая не может быть обработана процессором непосредственно. Сначала она должна быть скопирована в основную память. Примеры включают диск и ленточный накопитель                                                                                                              |
| <b>Вызов удаленной процедуры<br/>(remote procedure call, RPC)</b> | Метод, с помощью которого две программы на разных машинах взаимодействуют с использованием синтаксиса и семантики вызова/возврата процедур. Вызываемая и вызывающая программы ведут себя так, как если бы обе программы работали на одном и том же компьютере                                                                        |
| <b>Вытеснение (preemption)</b>                                    | Возврат ресурса от процесса, который еще не завершил им пользоваться                                                                                                                                                                                                                                                                 |
| <b>Голодание (starvation)</b>                                     | Состояние, при котором процесс откладывается на неопределенное время, потому что предпочтение постоянно отдается другим процессам                                                                                                                                                                                                    |
| <b>Дескриптор процесса<br/>(process descriptor)</b>               | То же, что и <i>управляющий блок процесса</i>                                                                                                                                                                                                                                                                                        |
| <b>Динамическая взаимоблокировка (livelock)</b>                   | Состояние, при котором два или более процессов постоянно изменяют свое состояние в ответ на изменения в другом процессе (или процессах) без какой-либо полезной работы. Это похоже на взаимоблокировку тем, что при этом отсутствует какой-либо прогресс, но отличается тем, что ни один процесс не заблокирован и не ждет чего-либо |
| <b>Динамическое перемещение<br/>(dynamic relocation)</b>          | Процесс, который назначает новые абсолютные адреса компьютерной программе во время выполнения, так что программа может быть выполнена из другой области основной памяти                                                                                                                                                              |
| <b>Дисковый кеш (disk cache)</b>                                  | Буфер, обычно в основной памяти, который функционирует как кеш дисковых блоков, между дисковой памятью и остальной частью основной памяти                                                                                                                                                                                            |
| <b>Диспетчеризация (dispatch)</b>                                 | Распределение времени процессора для заданий или задач, готовых к выполнению                                                                                                                                                                                                                                                         |

|                                                                                |                                                                                                                                                                                                                                                                    |
|--------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Доверенная система<br/>(trusted system)</b>                                 | Компьютер и операционная система, которые могут быть проверены в рамках данной стратегии безопасности                                                                                                                                                              |
| <b>Драйвер устройства<br/>(device driver)</b>                                  | Модуль операционной системы (обычно в ядре), который работает непосредственно с устройством или модулем ввода-вывода                                                                                                                                               |
| <b>Задание (job)</b>                                                           | Набор вычислительных шагов, упакованных для запуска как единое целое                                                                                                                                                                                               |
| <b>Задача (task)</b>                                                           | То же, что и <i>процесс</i>                                                                                                                                                                                                                                        |
| <b>Задача реального времени<br/>(real-time task)</b>                           | Задача, которая выполняется в связи с некоторым процессом, функцией или набором событий, внешних по отношению к компьютерной системе, и должна соответствовать одному или нескольким крайним срокам для эффективного и правильного взаимодействия с внешней средой |
| <b>Замещение страниц по требованию (demand paging)</b>                         | Перенос при необходимости страницы из вторичной памяти в основную память                                                                                                                                                                                           |
| <b>Запись (record)</b>                                                         | Группа элементов данных, рассматриваемых как единое целое                                                                                                                                                                                                          |
| <b>Запуск процесса<br/>(process spawning)</b>                                  | Создание нового процесса другим процессом                                                                                                                                                                                                                          |
| <b>Инверсия приоритетов<br/>(priority inversion)</b>                           | Обстоятельства, при которых операционная система вынуждает задачу с более высоким приоритетом ожидать выполнения задачи с более низким приоритетом                                                                                                                 |
| <b>Индексированный доступ<br/>(indexed access)</b>                             | Организация и доступ к записям в хранилище через отдельный индекс, указывающий расположение хранимых записей                                                                                                                                                       |
| <b>Индексированный последовательный доступ<br/>(indexed sequential access)</b> | Организация и доступ к записям в хранилище через индекс ключей, которые хранятся в произвольно разделенных последовательных файлах                                                                                                                                 |
| <b>Индексированный последовательный файл<br/>(indexed sequential file)</b>     | Файл, записи в котором упорядочены в соответствии со значениями ключевого поля. Основной файл дополняется индексным файлом, который содержит частичный список значений ключа; индекс предоставляет возможность быстрого поиска окрестности требуемой записи        |
| <b>Индексированный файл<br/>(indexed file)</b>                                 | Файл, доступ к записям которого выполняется в соответствии со значением ключевого поля. Для доступа требуется индекс, который указывает местоположение каждой записи на основе ключевого значения                                                                  |

**Интерфейс прикладного программирования  
(application programming interface — API)**

Стандартизированная библиотека программных инструментов, используемых разработчиками программного обеспечения для написания приложений, совместимых с определенной операционной системой или графическим интерфейсом пользователя

**Кадр (frame)**

В страничном виртуальном хранилище — блок основной памяти фиксированной длины, который используется для хранения одной страницы виртуальной памяти

**Кадр страницы (page frame)**

Непрерывный блок основной памяти фиксированного размера, используемый для хранения страницы

**Канал (pipe)**

Циклический буфер, позволяющий двум процессам взаимодействовать согласно модели “производитель–потребитель”. Таким образом, это очередь “первым пришел — первым вышел”, в которую пишет один процесс, а читает другой. В некоторых системах используются обобщенные каналы, позволяющие выбрать для потребления любой элемент в очереди

**Карусельное (круговое) планирование (round robin)**

Алгоритм планирования, в котором процессы активируются в фиксированном циклическом порядке (все процессы находятся в кольцевой очереди). Процесс, который не может продолжаться, потому что ожидает некоторое событие (например, завершение дочернего процесса или операцию ввода-вывода), возвращает управление планировщику

**Квант времени (time slice)**

Максимальное количество времени, которое процесс может выполнять до прерывания

**Квантование времени (time slicing)**

Режим работы, при котором двум или более процессам назначаются кванты времени на одном процессоре

**Кеш-память (cache memory)**

Память, которая меньше и быстрее основной памяти и находится между процессором и основной памятью. Кеш действует в качестве буфера для недавно использованных областей памяти

**Кластер (cluster)**

Группа взаимосвязанных целых компьютеров, работающих вместе как единый вычислительный ресурс, что может создать иллюзию единства машины. Термин *целый компьютер* означает систему, которая может работать самостоятельно, вне кластера

**Клиент (client)**

Процесс, который запрашивает обслуживание у сервера, отправляя сообщения его процессам

|                                                       |                                                                                                                                                                                                                                                         |
|-------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Командный такт<br/>(instruction cycle)</b>         | Период времени, в течение которого извлекается из памяти и выполняется одна команда компьютера на машинном языке                                                                                                                                        |
| <b>Контекст выполнения<br/>(execution context)</b>    | См. <i>состояние процесса</i>                                                                                                                                                                                                                           |
| <b>Коэффициент попадания<br/>(hit ratio)</b>          | В двухуровневой памяти доля нахождений информации в более быстрой памяти (например, в кеше) по отношению ко всем обращениям к памяти                                                                                                                    |
| <b>Критический участок<br/>(critical section)</b>     | В асинхронной процедуре компьютерной программы часть кода, которая не может выполняться одновременно с ее выполнением другой асинхронной процедурой                                                                                                     |
| <b>Ловушка (trap)</b>                                 | Незапрограммированный условный переход по указанному адресу, автоматически активируемый аппаратно; место, из которого был выполнен переход, записывается                                                                                                |
| <b>Логическая запись<br/>(logical record)</b>         | Запись, не зависящая от ее физической среды; части одной логической записи могут находиться в разных физических записях, как и в нескольких логических записях, или части логических записей могут быть расположены в одной физической записи           |
| <b>Логический адрес<br/>(logical address)</b>         | Ссылка на ячейку памяти, независимую от текущего назначения памяти для данных. До обращения к памяти требуется выполнить перевод логического адреса в физический                                                                                        |
| <b>Локальность ссылок<br/>(locality of reference)</b> | Тенденция процессора обращаться в течение короткого периода времени к одному и тому же множеству мест в памяти                                                                                                                                          |
| <b>Макроядро (macrokernel)</b>                        | Большое ядро операционной системы, предоставляющее широкий спектр служб                                                                                                                                                                                 |
| <b>Метод доступа<br/>(access method)</b>              | Метод, используемый для поиска файла, записи или множества записей                                                                                                                                                                                      |
| <b>Миграция процесса<br/>(process migration)</b>      | Передача достаточной информации о состоянии процесса от одной машины к другой для выполнения процесса на целевой машине                                                                                                                                 |
| <b>Микроядро<br/>(microkernel)</b>                    | Небольшое привилегированное ядро операционной системы, обеспечивающее планирование процессов, управление памятью и службы связи и опирающееся на другие процессы для выполнения некоторых функций, традиционно связываемых с ядром операционной системы |

|                                                                                                              |                                                                                                                                                                                                                                                                                                                                                                                                                         |
|--------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Многозадачность<br/>(multitasking)</b>                                                                    | Режим работы, который обеспечивает одновременную работу или чередование выполнения двух и более компьютерных задач. То же, что и мультипрограммирование, с использованием другой терминологии                                                                                                                                                                                                                           |
| <b>Многопоточность<br/>(multithreading)</b>                                                                  | Многозадачность в рамках одной программы. Разрешает нескольким потокам одновременно выполняться в одной и той же программе; каждый поток обрабатывает свою транзакцию или сообщение. Каждый поток является “подпроцессом”, и операционная система обычно взаимодействует с приложением для обработки потоков                                                                                                            |
| <b>Многоуровневая безопасность<br/>(multilevel security)</b>                                                 | Функциональная возможность, обеспечивающая контроль доступа на нескольких уровнях классификации данных                                                                                                                                                                                                                                                                                                                  |
| <b>Монитор (monitor)</b>                                                                                     | Конструкция языка программирования, которая инкапсулирует переменные, процедуры доступа и код инициализации в абстрактном типе данных. Получить доступ к переменной монитора можно только через процедуры доступа, и одновременно может иметь активный доступ к монитору только один процесс. Процедуры доступа являются <i>критическими участками</i> . Монитор может иметь очередь процессов, ожидающих доступ к нему |
| <b>Монолитное ядро<br/>(monolithic kernel)</b>                                                               | Большое ядро, содержащее практически всю операционную систему, включая планирование, файловую систему, драйверы устройств и управление памятью. Все функциональные компоненты ядра имеют доступ ко всем его внутренним структурам данных и процедурам. Как правило, монолитное ядро реализовано как единый процесс со всеми элементами, разделяющими одно и то же адресное пространство                                 |
| <b>Мультипрограммирование<br/>(multiprogramming)</b>                                                         | Режим работы, который обеспечивает чередующееся выполнение двух или более компьютерных программ одним процессором. То же, что и многозадачность, но с использованием другой терминологии                                                                                                                                                                                                                                |
| <b>Мультипроцессность<br/>(multiprocessing)</b>                                                              | Режим работы, который предусматривает параллельную обработку двумя или более процессорами или мультипроцессорами                                                                                                                                                                                                                                                                                                        |
| <b>Мультипроцессор<br/>(multiprocessor)</b>                                                                  | Компьютер с двумя или более процессорами, имеющими общий доступ к основной памяти                                                                                                                                                                                                                                                                                                                                       |
| <b>Мультипроцессор с неоднородным доступом к памяти<br/>(nonuniform memory access (NUMA) multiprocessor)</b> | Мультипроцессор с совместно используемой памятью, в котором время доступа данного процессора к слову памяти зависит от местоположения слова памяти                                                                                                                                                                                                                                                                      |

**Мьютекс (mutex)**

Программный флаг, используемый для захвата и освобождения объекта. Когда получены данные, которые не могут использоваться совместно, или запущена обработка, которая не может выполняться одновременно в другом месте системы, мьютекс устанавливается в заблокированное состояние, что препятствует другим попыткам его использовать. Когда данные больше не нужны или процедура завершена, мьютекс разблокируется. Мьютекс похож на *бинарный семафор*. Ключевое различие между ними заключается в том, что процесс, который блокирует мьютекс (устанавливает его значение равным 0), должен и разблокировать его (установить его значение равным 1). В отличие от мьютекса, заблокировать бинарный семафор может один процесс, а разблокировать — другой

**Непrivилегированное состояние (nonprivileged state)**

Контекст выполнения, который не позволяет выполнять определенные аппаратные команды, такие как команда останова и команды ввода-вывода

**Облегченный процесс (lightweight process)**

Поток выполнения

**Обмен (swapping)**

Процесс обмена содержимого области основной памяти с содержимым области во вторичной памяти.

**Обнаружение взаимоблокировки (deadlock detection)**

Технология, в которой запрашиваемые ресурсы всегда предоставляются по мере доступности. Периодически операционная система выполняет проверку на наличие взаимоблокировки

**Оболочка (shell)**

Часть операционной системы, которая интерпретирует интерактивные пользовательские команды и команды языка управления заданиями. Функционирует как интерфейс между пользователем и операционной системой

**Обработчик прерывания (interrupt handler)**

Процедура (как правило, часть операционной системы). Когда происходит прерывание, управление передается соответствующему обработчику прерываний, который предпринимает определенные действия в ответ на условие, вызвавшее прерывание

**Образ процесса (process image)**

Все компоненты процесса, включая программу, данные, стек и управляющий блок процесса

**Операционная система (operating system)**

Программное обеспечение, которое контролирует выполнение программ и обеспечивает такие службы, как распределение ресурсов, планирование, управление вводом-выводом и данными

|                                                                         |                                                                                                                                                                                                                  |
|-------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Организация файла<br/>(file organization)</b>                        | Физический порядок записей в файле, определяемый методом доступа и используемый для их хранения и извлечения                                                                                                     |
| <b>Основная память<br/>(main memory)</b>                                | Внутренняя для компьютерной системы память, являющаяся программно адресуемой и могущей быть загруженной в регистры для последующего выполнения или обработки                                                     |
| <b>Отключенное прерывание<br/>(disabled interrupt)</b>                  | Условие, обычно создаваемое операционной системой, во время которого процессор будет игнорировать сигналы запроса прерывания указанного класса                                                                   |
| <b>Относительный адрес<br/>(relative address)</b>                       | Адрес, рассчитанный как смещение от базового адреса                                                                                                                                                              |
| <b>Ошибка страницы<br/>(page fault)</b>                                 | Происходит, когда страница, содержащая слово, к которому выполняется обращение, не находится в основной памяти. При этом вызывается прерывание и выполняется загрузка соответствующей страницы в основную память |
| <b>Пакетная обработка<br/>(batch processing)</b>                        | Методика выполнения набора компьютерных программ таким образом, что каждая завершается до запуска следующей программы из набора                                                                                  |
| <b>Параллельный (concurrent)</b>                                        | Относящийся к процессам или потокам, которые выполняются в одном интервале времени, в течение которого они могут поочередно использовать общие ресурсы                                                           |
| <b>Первым пришел — первым вышел (first-in-first-out — FIFO)</b>         | Технология очереди, когда следующим будет выбран элемент, который находился в очереди в течение самого длительного времени                                                                                       |
| <b>Первым пришел — первым обслужен (first-come-first-served — FCFS)</b> | То же, что и <i>Первым пришел — первым вышел</i>                                                                                                                                                                 |
| <b>Пережидание занятости<br/>(busy waiting)</b>                         | Повторное выполнение цикла кода в ожидании события                                                                                                                                                               |
| <b>Переключение потоков<br/>(thread switch)</b>                         | Акт переключения управления процессором из одного потока в другой в том же процессе                                                                                                                              |
| <b>Переключение процессов<br/>(process switch)</b>                      | Операция, которая переключает процессор с одного процесса на другой, сохраняя весь управляющий блок процесса, регистры и другую информацию для первого процесса и заменяя их информацией второго процесса        |

**Переключение режимов  
(mode switch)**

Аппаратная операция, которая заставляет процессор работать в другом режиме (режиме ядра или режиме процесса). Когда режим переключается с режима процесса в режим ядра, сохраняются счетчик программы, слово состояния процессора и другие регистры. Когда режим переключается вновь в режим процесса, эта информация восстанавливается

**Планирование (scheduling)**

Выбор заданий или задач, которые необходимо диспетчеризовать. В некоторых операционных системах могут также планироваться другие единицы работы, такие как операции ввода-вывода

**Повторно используемый ресурс (reusable resource)**

Ресурс, который может безопасно использоваться одновременно только одним процессом, и этим использованием не исчерпывается. Процессы получают повторно используемые единицы ресурса, которые позже освобождаются для повторного использования другими процессами. Примеры таких многократно используемых ресурсов включают процессоры, каналы ввода-вывода, основную и дополнительную память, устройства и структуры данных, такие как файлы, базы данных и семафоры

**Подкачка страниц (paging)**

Перенос страниц между основной и вторичной памятью

**Поле (field)**

1. Определенные логические данные, которые являются частью записи. 2. Элементарная единица записи, которая может содержать элемент данных, совокупность данных, указатель или ссылку

**Пользовательский режим  
(user mode)**

Наименее привилегированный режим выполнения. Отдельные области основной памяти и некоторые машинные команды в этом режиме не могут быть использованы

**Последним вошел — первым вышел (last-in-first-out — LIFO)**

Технология очередей, в которой очередным выбираемым элементом является элемент, последним помещенный в очередь

**Последовательный доступ  
(sequential access)**

Возможность ввода данных в запоминающее устройство или носитель данных в той же последовательности, в которой данные упорядочены, или получения данных в том же порядке, в котором они были записаны

**Последовательный файл  
(sequential file)**

Файл, в котором записи упорядочены в соответствии со значениями одного или нескольких ключевых полей и обрабатываются в одной и той же последовательности от начала файла

|                                                              |                                                                                                                                                                                                                                                                                                                                                         |
|--------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Поток (thread)</b>                                        | Диспетчеризуемая единица работы. Включает в себя контекст процессора (в который входят программный счетчик и указатель стека) и собственную область данных для стека (для возможности ветвления). Поток выполняется последовательно и может быть прерван, так что процессор может заняться другим потоком. Процесс может состоять из нескольких потоков |
| <b>Почтовый ящик (mailbox)</b>                               | Структура данных, совместно используемая несколькими процессами как очередь для сообщений. Вместо передачи непосредственно от отправителя к получателю сообщения отправляются в почтовый ящик и извлекаются из него                                                                                                                                     |
| <b>Предварительный поиск (prepaging)</b>                     | Поиск страниц, отличных от тех, которые требуются из-за ошибки страницы, в надежде, что эти дополнительные страницы будут необходимы в ближайшем будущем, что сэкономит дисковый ввод-вывод. Сравните с <i>замещением страниц по требованию</i>                                                                                                         |
| <b>Предотвращение взаимоблокировки (deadlock avoidance)</b>  | Динамическая технология, которая исследует каждый новый запрос ресурса на возможность возникновения взаимоблокировки. Если новый запрос может привести к взаимоблокировке, он отклоняется                                                                                                                                                               |
| <b>Предупреждение взаимоблокировки (deadlock prevention)</b> | Технология, которая гарантирует, что взаимоблокировка не возникнет. Предупреждение достигается путем обеспечения отсутствия одного из необходимых условий взаимоблокировки                                                                                                                                                                              |
| <b>Прерывание (interrupt)</b>                                | Приостановка процесса, такого как выполнение компьютерной программы, вызванная событием, внешним по отношению к этому процессу, и выполняемая таким образом, что процесс может быть возобновлен                                                                                                                                                         |
| <b>Привилегированная команда (privileged instruction)</b>    | Команда, которая может быть выполнена только в определенном режиме, обычно программой-супервизором                                                                                                                                                                                                                                                      |
| <b>Привилегированный режим (privileged mode)</b>             | То же, что и <i>режим ядра</i>                                                                                                                                                                                                                                                                                                                          |
| <b>Программный ввод-вывод (programmed I/O)</b>               | Форма ввода-вывода, когда процессор выдает команду ввода-вывода модулю ввода-вывода, а затем должен дождаться завершения операции, прежде чем продолжить работу                                                                                                                                                                                         |
| <b>Процесс (process)</b>                                     | Выполняемая программа. Процесс управляется и планируется операционной системой. То же, что и <i>задача</i>                                                                                                                                                                                                                                              |

**Процессор (central processing unit — CPU)**

Часть компьютера, которая выбирает и выполняет машинные команды. Состоит из арифметико-логического устройства, управляющего модуля и регистров

**Процессор (processor)**

В компьютере — функциональный блок, который интерпретирует и выполняет команды. Процессор состоит по меньшей мере из блока управления командами и арифметического блока

**Прямой доступ (direct access)**

Возможность получать данные с устройства хранения или вводить данные в запоминающее устройство в последовательности, не зависящей от их относительного положения, с помощью адреса, указывающего физическое местоположение данных

**Прямой доступ к памяти (direct memory access — DMA)**

Форма ввода-вывода, в которой обменом данными между основной памятью и устройством ввода-вывода управляет специальный модуль DMA. Процессор отправляет запрос на передачу блока данных модулю DMA и прерывается только после того, как весь блок был передан

**Рабочее множество (working set)**

Рабочее множество с параметром  $\Delta$  для процесса в виртуальный момент времени  $t$ ,  $W(t, \Delta)$ , представляет собой множество страниц этого процесса, на которые были ссылки за последние  $\Delta$  единиц времени. Сравните с **резидентным множеством**

**Разделение времени (time sharing)**

Одновременное использование устройства несколькими пользователями

**Разделение памяти (memory partitioning)**

Разделение хранилища на самостоятельные разделы

**Рандеву (rendezvous)**

При передаче сообщения — условие, когда отправитель и получатель сообщения блокируются до тех пор, пока сообщение будет доставлено

**Распределенная операционная система (distributed operating system)**

Общая операционная система, совместно используемая сетью компьютеров. Распределенная операционная система обеспечивает поддержку межпроцессного взаимодействия, миграцию процессов, взаимные исключения и предотвращение или обнаружение взаимоблокировок

**Расходуемый ресурс (consumable resource)**

Ресурс, который может быть создан (произведен) и уничтожен (потреблен). Когда ресурс приобретается процессом, он перестает существовать. Примерами расходуемых ресурсов являются прерывания, сигналы, сообщения и информация в буферах ввода-вывода

**Реальный адрес (real address)** Физический адрес в основной памяти

**Регистры (registers)**

Высокоскоростная память внутри процессора. Одни регистры видны пользователю (доступны программисту через набор машинных команд). Другие регистры используются только центральным процессором для целей управления

**Реентерабельная процедура (reentrant procedure)**

Процедура, повторный вход в которую может быть выполнен до завершения предыдущего выполнения этой же процедуры, при этом процедура выполняется корректно

**Режим ядра (kernel mode)**

Привилегированный режим выполнения, зарезервированный для ядра операционной системы. Как правило, режим ядра обеспечивает доступ к областям основной памяти, недоступным процессам, выполняющимся в менее привилегированном режиме, а также позволяет выполнять определенные машинные команды, которые могут выполняться только в режиме ядра

**Резидентное множество (resident set)**

Часть процесса, которая в данный момент времени находится в основной памяти. Сравните с *рабочим множеством*

**Связный список (chained list)**

Список, в котором элементы данных могут быть распределены в памяти, но в котором каждый элемент содержит идентификатор, указывающий местоположение следующего элемента

**Сеанс (session)**

Набор из одного или нескольких процессов, представляющих одно интерактивное пользовательское приложение или функцию операционной системы. Весь ввод с клавиатуры и мыши направляется сеансу переднего плана, а все выходные данные сеанса переднего плана направляются на экран дисплея

**Сегмент (segment)**

В виртуальной памяти блок, который имеет виртуальный адрес. Блоки программ могут иметь разную (и даже динамически изменяющуюся) длину

**Сегментация (segmentation)**

Разделение программы или приложения на сегменты как часть схемы виртуальной памяти

**Семафор (semaphore)**

Целочисленное значение, используемое для передачи сигналов между процессами. Всего доступны три операции над семафором (все они атомарны): инициализация, уменьшение и увеличение. В зависимости от точного определения семафора, операция уменьшения может привести к блокировке процесса, а операция увеличения — к разблокировке. Также известен как семафор со счетчиком или общий семафор

|                                                                                        |                                                                                                                                                                                                                                                          |
|----------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Сервер (server)</b>                                                                 | 1. Процесс, который отвечает на запросы от клиентов с помощью сообщений. 2. В сети — станция данных, которая предоставляет службы другим станциям, например файловый сервер, сервер печати или почтовый сервер                                           |
| <b>Сильный семафор<br/>(strong semaphore)</b>                                          | Семафор, в котором все процессы, ожидающие один и тот же семафор, поставлены в очередь и в конечном итоге продолжают работу в том же порядке, в котором вызвали операцию ожидания ( $P$ ) (порядок FIFO)                                                 |
| <b>Симметричная много-процессорная обработка<br/>(symmetric multiprocessing — SMP)</b> | Форма многопроцессорной обработки, которая позволяет операционной системе выполняться на любом доступном процессоре или на нескольких доступных процессорах одновременно                                                                                 |
| <b>Синхронизация<br/>(synchronization)</b>                                             | Ситуация, когда два или более процессов координируют свою деятельность на основании условия                                                                                                                                                              |
| <b>Синхронная операция<br/>(synchronous operation)</b>                                 | Операция, которая выполняется регулярно или предсказуемо по отношению к указанному событию в другом процессе, например к вызову подпрограммы ввода-вывода, которая получает управление в предопределенном месте компьютерной программы                   |
| <b>Система реального времени<br/>(real-time system)</b>                                | Операционная система, которая должна планировать задачи и управлять ими в реальном времени                                                                                                                                                               |
| <b>Система управления файлами<br/>(file management system)</b>                         | Набор системного программного обеспечения, которое предоставляет пользователям и приложениям услуги по использованию файлов, включая доступ к файлам, обслуживание каталогов и контроль доступа                                                          |
| <b>Системная шина<br/>(system bus)</b>                                                 | Шина, используемая для соединения основных компонентов компьютера (процессора, памяти, системы ввода-вывода)                                                                                                                                             |
| <b>Системный режим<br/>(system mode)</b>                                               | То же, что и <i>режим ядра</i>                                                                                                                                                                                                                           |
| <b>Слабый семафор<br/>(weak semaphore)</b>                                             | Семафор, в котором порядок активации процессов, ожидающих семафор, неизвестен или не определен                                                                                                                                                           |
| <b>След (trace)</b>                                                                    | Последовательность команд, которые выполняются при запуске процесса                                                                                                                                                                                      |
| <b>Слово (word)</b>                                                                    | Упорядоченный набор байтов или битов, который является естественной единицей хранения, передачи или использования информации в данном компьютере. Как правило, если процессор имеет набор команд фиксированной длины, то длина команды равна длине слова |

|                                                                   |                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|-------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Слово состояния программы<br/>(program status word — PSW)</b>  | Регистр или набор регистров, которые содержат коды условий, режим выполнения и другую информацию, которая отражает состояние процесса                                                                                                                                                                                                                                                                                                                              |
| <b>Снижение пропускной способности (thrashing)</b>                | Явление в схемах виртуальной памяти, когда процессор большую часть своего времени занимается не выполнением процессов, а выгрузкой и загрузкой в основную память                                                                                                                                                                                                                                                                                                   |
| <b>Сообщение (message)</b>                                        | Блок информации, которым процессы могут обмениваться в качестве средства коммуникации                                                                                                                                                                                                                                                                                                                                                                              |
| <b>Состояние гонки<br/>(race condition)</b>                       | Ситуация, когда несколько процессов получают доступ к совместно используемым данным и управляют ими, а результат зависит от относительной синхронизации процессов                                                                                                                                                                                                                                                                                                  |
| <b>Состояние процесса<br/>(process state)</b>                     | Вся информация, необходимая операционной системе для управления процессом, а процессору — для правильного его выполнения. Состояние процесса включает в себя содержимое различных регистров процессора, таких как счетчик программ и регистры данных; также включает информацию, используемую операционной системой, такую как приоритет процесса и ожидает ли процесс завершения определенного события ввода-вывода. Другое название — <i>контекст выполнения</i> |
| <b>Стек (stack)</b>                                               | Упорядоченный список, в который элементы добавляются и из которого удаляются из одного и того же конца списка, известного как вершина. Таким образом, очередной добавленный в список элемент становится верхним, а следующий элемент, который будет удален из списка, — это элемент, который был в списке наименьшее время. Этот метод характеризуется фразой “последним вошел — первым вышел”                                                                     |
| <b>Счетчик команд<br/>(program counter)</b>                       | Регистр адресов команд                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <b>Таблица размещения диска<br/>(disk allocation table)</b>       | Таблица, которая указывает, какие блоки во вторичной памяти свободны и доступны для размещения файлов                                                                                                                                                                                                                                                                                                                                                              |
| <b>Таблица размещения файла<br/>(file allocation table — FAT)</b> | Таблица, которая указывает физическое местоположение пространства, выделенного для файла, во вторичной памяти. Имеется по одной таблице размещения файлов для каждого файла                                                                                                                                                                                                                                                                                        |

|                                                                |                                                                                                                                                                                                                                                                           |
|----------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Транслятор адресов<br/>(address translator)</b>             | Функциональный модуль, преобразующий виртуальные адреса в реальные                                                                                                                                                                                                        |
| <b>Троянский конь<br/>(Trojan horse)</b>                       | Компьютерная программа, которая выглядит как имеющая полезную функцию, но при этом имеет скрытую и потенциально вредоносную функцию. Уклоняется от механизмов безопасности, иногда используя законные полномочия системного объекта, вызвавшего программу троянского коня |
| <b>Уплотнение (compaction)</b>                                 | Техника, используемая при разделении памяти на разделы переменного размера. Время от времени операционная система сдвигает разделы, чтобы они (и следовательно, вся свободная память) представляли собой один непрерывный блок                                            |
| <b>Управляющий блок процесса<br/>(process control block)</b>   | Олицетворение процесса в операционной системе. Структура данных, содержащая информацию о характеристиках и состоянии процесса                                                                                                                                             |
| <b>Уровень мультипрограммирования (multiprogramming level)</b> | Количество процессов, которые являются частично или полностью резидентными в основной памяти                                                                                                                                                                              |
| <b>Файл (file)</b>                                             | Набор связанных записей, рассматриваемых как единое целое                                                                                                                                                                                                                 |
| <b>Физический адрес<br/>(physical address)</b>                 | Абсолютное местоположение единицы данных в памяти (например, слово или байт в основной памяти, блок во вторичной памяти)                                                                                                                                                  |
| <b>Хеширование (hashing)</b>                                   | Выбор места хранения элемента данных путем расчета адреса как функции содержимого данных. Эта техника усложняет функцию распределения памяти, но приводит к быстрой случайной выборке                                                                                     |
| <b>Хеш-файл (hash file)</b>                                    | Файл, в котором доступ к записям осуществляется в соответствии со значениями ключевого поля. Для поиска записи на основе значения ключа используется хеширование                                                                                                          |
| <b>Циклическая блокировка<br/>(spin lock)</b>                  | Механизм взаимного исключения, в котором процесс выполняется в бесконечном цикле ожидания значения переменной блокировки, указывающей на доступность                                                                                                                      |

**Червь (worm)**

Деструктивная программа, которая копирует себя на одном компьютере или по сети, как проводной, так и беспроводной. Может нанести ущерб путем воспроизведения, использования внутренних дисков и ресурсов памяти на одном компьютере или исчерпанием пропускной способности сети. Может также размещать троянских коней, превращая компьютер в “зомби” для рассылки спама или иных вредоносных целей. Очень часто термины “червь” и “вирус” используются как синонимы, однако червь подразумевает автоматический способ воспроизведения себя на других компьютерах сети

**Шифрование (encryption)**

Преобразование простого текста или данных в непонятную форму с помощью обратимого математического вычисления

**Ядро (kernel)**

Часть операционной системы, которая включает чаще всего используемые части программного обеспечения. Как правило, ядро постоянно находится в основной памяти. Ядро работает в привилегированном режиме и отвечает на вызовы процессов и прерываний от устройств

**Язык управления заданиями (job control language — JCL)**

Проблемно-ориентированный язык, предназначенный для выражения утверждений относительно заданий, которые используются для определения задания или для описания его требований к операционной системе

# Сокращения

|       |                                                       |                                                           |
|-------|-------------------------------------------------------|-----------------------------------------------------------|
| AES   | Advanced Encryption Standard                          | Усовершенствованный стандарт шифрования                   |
| API   | Application Programming Interface                     | Интерфейс прикладного программирования                    |
| CD    | Compact Disk                                          | Компакт-диск                                              |
| CORBA | Common Object Request Broker Architecture             | Общая архитектура брокера запросов к объектам             |
| CPU   | Central Processing Unit                               | Центральный процессор                                     |
| CTSS  | Compatible Time-Sharing System                        | Совместимые системы с разделением времени                 |
| DES   | Date Encryption Standard                              | Стандарт шифрования данных                                |
| DMA   | Direct Memory Access                                  | Прямой доступ к памяти                                    |
| DVD   | Digital Versatile Disk                                | Цифровой многофункциональный диск                         |
| FAT   | File Allocation Table                                 | Таблица размещения файла                                  |
| FCFS  | First Come First Served                               | Первым вошел — первым обслужен                            |
| FIFO  | First-In, First-Out                                   | Первым вошел — первым вышел                               |
| GUI   | Graphical User Interface                              | Графический интерфейс пользователя                        |
| IBM   | International Business Machines Corporation           | Корпорация IBM                                            |
| I/O   | Input/Output                                          | Ввод-вывод                                                |
| IP    | Internet Protocol                                     | Протокол Интернета                                        |
| IPC   | Interprocess Communication                            | Межпроцессное взаимодействие                              |
| JCL   | Job Control Language                                  | Язык управления заданиями                                 |
| LAN   | Local Area Network                                    | Локальная сеть                                            |
| LIFO  | Last-In, First-Out                                    | Последним вошел — первым вышел                            |
| LRU   | Least Recently Used                                   | Последний использовавшийся                                |
| MVS   | Multiple Virtual Storage                              | Многосегментная виртуальная память                        |
| NTFS  | NT File System                                        | Файловая система NTFS                                     |
| NUMA  | Nonuniform Memory Access                              | Мультипроцессор с неоднородным доступом к памяти          |
| ORB   | Object Request Broker                                 | Брокер объектных запросов                                 |
| OSI   | Open Systems Interconnection                          | Взаимодействие открытых систем                            |
| PC    | Program Counter                                       | Счетчик команд                                            |
| PCB   | Process Control Block                                 | Управляющий блок процесса                                 |
| PSW   | Processor Status Word                                 | Слово состояния процессора                                |
| RAID  | Redundant Array of Independent Disks                  | Массив независимых дисков с избыточностью                 |
| RISC  | Reduced Instruction Set Computer                      | Архитектура компьютера с сокращенным набором команд       |
| RPC   | Remote Procedure Call                                 | Вызов удаленной процедуры                                 |
| SMP   | Symmetric Multiprocessing or Symmetric Multiprocessor | Симметричная многопроцессорная обработка                  |
| SPOOL | Simultaneous Peripheral Operation Online              | Параллельная онлайновая работа периферийного оборудования |
| SVR4  | System V Release 4                                    | Операционная система UNIX System V, издание 4             |
| TCP   | Transmission Control Protocol                         | Протокол управления передачей                             |
| TLB   | Translation Lookaside Buffer                          | Буфер быстрой переадресации                               |
| UDP   | User Datagram Protocol                                | Протокол пользовательских дейтаграмм                      |

# Список литературы

## Сокращения

- ACM Association for Computing Machinery (Ассоциация вычислительной техники)  
IBM International Business Machines Corporation (Корпорация IBM)  
IEEE Institute of Electrical and Electronics Engineers (Институт инженеров по электротехнике и электронике)

1. Agarwal, A. *Analysis of Cache Performance for Operating Systems and Multiprogramming*. Norwell, MA: Kluwer Academic Publishers, 1989.
2. Amdahl, G. "Validity of the Single-Processor Approach to Achieving Large-Scale Computing Capability." *Proceedings, of the AFIPS Conference*, 1967.
3. Ananda, A.; Tay, B.; and Koh, E. "A Survey of Asynchronous Remote Procedure Calls." *Operating Systems Review*, April 1992.
4. Anderson, E. *μClbc*. Slide Presentation, Codepoet Consulting, January 26, 2005. <http://www.codepoet-consulting.com/>.
5. Anderson, J. *Computer Security Threat Monitoring and Surveillance*. Fort Washington, PA: James P. Anderson Co., April 1980.
6. Anderson, T.; Bershad, B.; Lazowska, E.; and Levy, H. "Thread Management for Shared-Memory Multiprocessors." In [259].
7. Anderson, T.; Lazowska, E.; and Levy, H. "The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors." *IEEE Transactions on Computers*, December 1989.
8. Andrianoff, S. "A Module on Distributed Systems for the Operating System Course." *Proceedings, Twenty-First SIGCSE Technical Symposium on Computer Science Education, SIGCSE Bulletin*, February 1990.
9. Arden, B., ed. *What Can Be Automated?* The Computer Science and Engineering Research Study, National Science Foundation, 1980.
10. Artsy, Y. "Designing a Process Migration Facility: The Charlotte Experience." *Computer*, September 1989.
11. Artsy, Y., ed. "Special Issue on Process Migration." *Newsletter of the IEEE Computer Society Technical Committee on Operating Systems*, Winter 1989.
12. Atlas, A., and Blundon, B. "Time to Reach for It All." *UNIX Review*, January 1989.
13. Baccelli, E.; Hahm, O.; Wahlsch, M.; Gunes, M.; and Schmidt, T. "RIOT OS: Towards an OS for the Internet of Things." *Proceedings of IEEE INFOCOM, Demo/Poster for the 32nd IEEE International Conference on Computer Communications*, Turin, Italy, April 2013.

14. Bach, M. *The Design of the UNIX Operating System*. Englewood Cliffs, NJ: Prentice Hall, 1986.
15. Bacon, J., and Harris, T. *Operating Systems: Concurrent and Distributed Software Design*. Reading, MA: Addison-Wesley, 2003.
16. Baer, J. *Computer Systems Architecture*. Rockville, MD: Computer Science Press, 1980.
17. Barbosa, V. "Strategies for the Prevention of Communication Deadlocks in Distributed Parallel Programs." *IEEE Transactions on Software Engineering*, November 1990.
18. Barkley, R., and Lee, T. "A Lazy Buddy System Bounded by Two Coalescing Delays per Class." *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, December 1989.
19. Bays, C. "A Comparison of Next-Fit, First-Fit, and Best-Fit." *Communications of the ACM*, March 1977.
20. Belady, L. "A Study of Replacement Algorithms for a Virtual Storage Computer." *IBM Systems Journal*, No. 2, 1966.
21. Ben-Ari, M. *Principles of Concurrent and Distributed Programming*. Harlow, England: Addison-Wesley, 2006.
22. Bhatti, N.; Bouch, A.; and Kuchinsky, A. "Integrated User-Perceived Quality into Web Server Design." *Proceedings, 9<sup>th</sup> International World Wide Web Conference*, May 2000.
23. Black, D. "Scheduling Support for Concurrency and Parallelism in the Mach Operating System." *Computer*, May 1990.
24. Bolosky, W.; Fitzgerald, R.; and Scott, M. "Simple but Effective Techniques for NUMA Memory Management." *Proceedings, Twelfth ACM Symposium on Operating Systems Principles*, December 1989.
25. Bonwick, J. "The Slab Allocator: An Object-Caching Kernel Memory Allocator." *Proceedings, USENIX Summer Technical Conference*, 1994.
26. Borg, A.; Kessler, R.; and Wall, D. "Generation and Analysis of Very Long Address Traces." *Proceedings of the 17th Annual International Symposium on Computer Architecture*, May 1990.
27. Bormann, C.; Ersue, M.; and Keranen, A. *Terminology for Constrained-Node Networks*. RFC 7228, May 2014.
28. Brent, R. "Efficient Implementation of the First-Fit Strategy for Dynamic Storage Allocation." *ACM Transactions on Programming Languages and Systems*, July 1989.
29. Brewer, E. "Clustering: Multiply and Conquer." *Data Communications*, July 1997.
30. Briand, L., and Roy, D. *Meeting Deadlines in Hard Real-Time Systems: The Rate Monotonic Approach*. Los Alamitos, CA: IEEE Computer Society Press, 1999.
31. Brinch Hansen, P., ed. *Classic Operating Systems: From Batch Processing to Distributed Systems*. New York, NY: Springer-Verlag, 2001.
32. Buonadonna, P.; Hill, J.; and Culler, D. "Active Message Communication for Tiny Networked Sensors." *Proceedings, IEEE INFOCOM 2001*, April 2001.

33. Buttazzo, G., Sensini, F. "Optimal Deadline Assignment for Scheduling Soft Aperiodic Tasks in Hard Real-Time Environments." *IEEE Transactions on Computers*, October 1999.
34. Cabreir, L. "The Influence of Workload on Load Balancing Strategies." *USENIX Conference Proceedings*, Summer 1986.
35. Callaway, B., and Esker, R. "OpenStack Deployment and Operations Guide." *NetApp White Paper*, May 2015.
36. Carr, R. *Virtual Memory Management*. Ann Arbor, MI: UMI Research Press, 1984.
37. Carr, S.; Mayo, J.; and Shene, C. "Race Conditions: A Case Study." *Journal of Computing in Colleges*, October 2001.
38. Carrier, B. *File System Forensic Analysis*. Upper Saddle River, NJ: Addison- Wesley, 2005.
39. Carriero, N., and Gelernter, D. "How to Write Parallel Programs: A Guide for the Perplexed." *ACM Computing Surveys*, September 1989.
40. Casavant, T., and Singhal, M. *Distributed Computing Systems*. Los Alamitos, CA: IEEE Computer Society Press, 1994.
41. Castillo C., Flanagan E., Wilkinson N. Object-Oriented Design and Programming. — *AT&T Technical Journal*, November/December 1992.
42. Chandras, R. "Distributed Message Passing Operating Systems." *Operating Systems Review*, January 1990.
43. Chen, J.; Borg, A.; and Jouppi, N. "A Simulation-Based Study of TLB Performance." *Proceedings, 19th Annual International Symposium on Computer Architecture*, May 1992.
44. Choi, H., and Yun, H. "Context Switching and IPC Performance Comparison between  $\mu$ Clinux and Linux on the ARM9 Based Processor." *Proceedings, Samsung Conference*, 2005.
45. Chu, W., and Opderbeck, H. "The Page Fault Frequency Replacement Algorithm." *Proceedings, Fall Joint Computer Conference*, 1972.
46. Clark, D., and Emer, J. "Performance of the VAX-11/780 Translation Buffer: Simulation and Measurement." *ACM Transactions on Computer Systems*, February 1985.
47. Clark, L. "Intro to Embedded Linux Part 1: Defining Android vs. Embedded Linux." *Libby Clark Blog*, Linux.com, March 6, 2013.
48. Coffman, E.; Elphick, M.; and Shoshani, A. "System Deadlocks." *Computing Surveys*, June 1971.
49. Comer, D. "The Ubiquitous B-Tree." *Computing Surveys*, June 1979.
50. Conway, M. "Design of a Separable Transition-Diagram Compiler." *Communications of the ACM*, July 1963.
51. Corbato, F. "A Paging Experiment with the Multics System." *MIT Project MAC Report MAC-M-384*, May 1968.

52. Corbato, F.; Merwin-Daggett, M.; and Daley, R. "An Experimental Time-Sharing System." *Proceedings of the 1962 Spring Joint Computer Conference*, 1962. Reprinted in [31].
53. Corbet, J. "The SLUB Allocator." April 2007. <http://lwn.net/Articles/229984/>.
54. Cormen, T., et al. *Introduction to Algorithms*. Cambridge, MA: MIT Press, 2009. Имеется русский перевод: Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн. *Алгоритмы: построение и анализ*. 3-е изд. — М.: ООО "И.Д. Вильямс", 2013.
55. Cox, A., and Fowler, R. "The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with PLATINUM." *Proceedings, Twelfth ACM Symposium on Operating Systems Principles*, December 1989.
56. Daley, R., and Dennis, R. "Virtual Memory, Processes, and Sharing in MULTICS." *Communications of the ACM*, May 1968.
57. Dasgupta, P., et al. "The Clouds Distributed Operating System." *IEEE Computer*, November 1991.
58. Datta, A., and Ghosh, S. "Deadlock Detection in Distributed Systems." *Proceedings, Phoenix Conference on Computers and Communications*, March 1990.
59. Datta, A.; Javagal, R.; and Ghosh, S. "An Algorithm for Resource Deadlock Detection in Distributed Systems," *Computer Systems Science and Engineering*, October 1992.
60. Denning, P. "The Working Set Model for Program Behavior." *Communications of the ACM*, May 1968.
61. Denning, P. "Third Generation Computer Systems." *ACM Computing Surveys*, December 1971.
62. Denning, P. "Virtual Memory." *Computing Surveys*, September 1970.
63. Denning, P. "Working Sets Past and Present." *IEEE Transactions on Software Engineering*, January 1980.
64. Denning, P.; Buzen, J.; Dennis, J.; Gaines, R.; Hansen, P.; Lynch, W.; and Organick, E. "Operating Systems." In [9].
65. Dijkstra, E. "Hierarchical Ordering of Sequential Processes." *Acta informatica*, Volume 1, Number 2, 1971. Reprinted in [31].
66. Dijkstra, E. *Cooperating Sequential Processes*. Technological University, Eindhoven, The Netherlands, 1965. Reprinted [144] and in [31].
67. Dong, W., et al. "Providing OS Support for Wireless Sensor Networks: Challenges and Approaches." *IEEE Communications Surveys & Tutorials*, Fourth Quarter, 2010.
68. Douglas, F., and Ousterhout, J. "Process Migration in Sprite: A Status Report." *Newsletter of the IEEE Computer Society Technical Committee on Operating Systems*, Winter 1989.
69. Douglas, F., and Ousterhout, J. "Transparent Process Migration: Design Alternatives and the Sprite Implementation." *Software Practice and Experience*, August 1991.

70. Downey, A. *The Little Book of Semaphores Version 2.2.1.* 2016. [www.greenteapress.com/sema.../](http://www.greenteapress.com/sema.../).
71. Dube, R. *A Comparison of the Memory Management Sub-Systems in FreeBSD and Linux.* Technical Report CS-TR-3929, University of Maryland, September 25, 1998.
72. Eager, D.; Lazowska, E.; and Zahnorjan, J. "Adaptive Load Sharing in Homogeneous Distributed Systems." *IEEE Transactions on Software Engineering*, May 1986.
73. Eischen, C. "RAID 6 Covers More Bases." *Network World*, April 9, 2007.
74. EmCraft Systems. "What Is the Minimal Footprint of  $\mu$ Clinux?" *EmCraft Documentation*, May 19, 2015. <http://www.emcraft.com/stm32f429discovery/what-is-minimal-footprint>.
75. Eskicioglu, M. "Design Issues of Process Migration Facilities in Distributed Systems." *Newsletter of the IEEE Computer Society Technical Committee on Operating Systems and Application Environments*, Summer 1990.
76. eTutorials.org. *Embedded Linux Systems.* 2016. <http://etutorials.org/Linux+systems/embedded+linux+systems/>.
77. Feitelson, D., and Rudolph, L. "Distributed Hierarchical Control for Parallel Processing." *Computer*, May 1990.
78. Feitelson, D., and Rudolph, L. "Mapping and Scheduling in a Shared Parallel Environment Using Distributed Hierarchical Control." *Proceedings, 1990 International Conference on Parallel Processing*, August 1990.
79. Ferrari, D., and Yih, Y. "VWS: The Variable-Interval Sampled Working Set Policy." *IEEE Transactions on Software Engineering*, May 1983.
80. Finkel, R. "The Process Migration Mechanism of Charlotte." *Newsletter of the IEEE Computer Society Technical Committee on Operating Systems*, Winter 1989.
81. Finkel, R. *An Operating Systems Vade Mecum*, Second edition. Englewood Cliffs, NJ: Prentice Hall, 1988.
82. Foster, I. "Automatic Generation of Self-Scheduling Programs." *IEEE Transactions on Parallel and Distributed Systems*, January 1991.
83. Franz, M. "Dynamic Linking of Software Components." *Computer*, March 1997.
84. Frenzel, L. "12 Wireless Options for IoT/M2M: Diversity or Dilemma?" *Electronic Design*, June 2016.
85. Ganapathy, N., and Schimmel, C. "General Purpose Operating System Support for Multiple Page Sizes." *Proceedings, USENIX Symposium*, 1998.
86. Gay, D., et al. "The nesC Language: A Holistic Approach to Networked Embedded Systems." *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, 2003.
87. Gehringer, E.; Siewiorek, D.; and Segall, Z. *Parallel Processing: The Cm\* Experience.* Bedford, MA: Digital Press, 1987.
88. Gibbons, P. "A Stub Generator for Multilanguage RPC in Heterogeneous Environments." *IEEE Transactions on Software Engineering*, January 1987.

89. Gingras, A. "Dining Philosophers Revisited." *ACM SIGCSE Bulletin*, September 1990.
90. Goldman, P. "Mac VM Revealed." *Byte*, November 1989.
91. Gopal, I. "Prevention of Store-and-Forward Deadlock in Computer Networks." *IEEE Transactions on Communications*, December 1985.
92. Goyeneche, J., and Souse, E. "Loadable Kernel Modules." *IEEE Software*, January/February 1999.
93. Graham, G., and Denning, P. "Protection — Principles and Practice." *Proceedings, AFIPS Spring Joint Computer Conference*, 1972.
94. Graham, R.; Knuth, D.; and Patashnik, O. *Concrete Mathematics: A Foundation for Computer Science*. Reading, MA: Addison-Wesley, 1994. Имеется русский перевод: Рональд Л. Грэхем, Дональд Э. Кнут, Орен Паташник. *Конкремтная математика. Математические основы информатики*, 2-е изд. — М.: ООО “И.Д. Вильямс”, 2010.
95. Grosshans, D. *File Systems: Design and Implementation*. Englewood Cliffs, NJ: Prentice Hall, 1986.
96. Gupta, R., and Franklin, M. "Working Set and Page Fault Frequency Replacement Algorithms: A Performance Comparison." *IEEE Transactions on Computers*, August 1978.
97. Gustafson, J. "Reevaluating Amdahl's Law." *Communications of the ACM*, May 1988.
98. Guynes, J. "Impact of System Response Time on State Anxiety." *Communications of the ACM*, March 1988
99. Hahm, O.; Baccelli, E.; Petersen, H.; and Tsiftes, N. "Operating Systems for Low-End Devices in the Internet of Things: A Survey." *IEEE Internet of Things Journal*, December 2015.
100. Haldar, S., and Subramanian, D. "Fairness in Processor Scheduling in Time Sharing Systems." *Operating Systems Review*, January 1991.
101. Handy, J. *The Cache Memory Book, Second edition*. San Diego, CA: Academic Press, 1998.
102. Harris, W. "Multi-Core in the Source Engine." bit-tech.net technical paper, November 2, 2006. [bit-tech.net/gaming/2006/11/02/Multi\\_core\\_in\\_the\\_Source\\_Engine/1](http://bit-tech.net/gaming/2006/11/02/Multi_core_in_the_Source_Engine/).
103. Henry, G. "The UNIX System: The Fair Share Scheduler." *AT&T Bell Laboratories Technical Journal*, October 1984.
104. Herlihy, M. "A Methodology for Implementing Highly Concurrent Data Structures." *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, March 1990.
105. Hewlett Packard. *White Paper on Clustering*. June 1996.
106. Hill, J., et al. "System Architecture Directions for Networked Sensors." *Proceedings, Architectural Support for Programming Languages and Operating Systems*, 2000.
107. Hoare, C. "Monitors: An Operating System Structuring Concept." *Communications of the ACM*, October 1974.

108. Holland, D.; Lim, A.; and Seltzer, M. "A New Instructional Operating System." *Proceedings of SIGCSE 2002*, 2002.
109. Holt, R. "Some Deadlock Properties of Computer Systems." *Computing Surveys*, September 1972.
110. Howard, J. "Mixed Solutions for the Deadlock Problem." *Communications of the ACM*, July 1973.
111. Huck, J., and Hays, J. "Architectural Support for Translation Table Management in Large Address Space Machines." *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993.
112. Huck, T. *Comparative Analysis of Computer Architectures*. Stanford University Technical Report Number 83-243, May 1983.
113. Hwang, K., et al. "Designing SSI Clusters with Hierarchical Checkpointing and Single I/O Space." *IEEE Concurrency*, January–March 1999.
114. Hyman, H. "Comments on a Problem in Concurrent Programming Control." *Communications of the ACM*, January 1966.
115. Islor, S., and Marsland, T. "The Deadlock Problem: An Overview." *Computer*, September 1980.
116. Iyer, S., and Druschel, P. "Anticipatory Scheduling: A Disk Scheduling Framework to Overcome Deceptive Idleness in Synchronous I/O." *Proceedings, 18th ACM Symposium on Operating Systems Principles*, October 2001.
117. Jackson, J. "Multicore Requires OS Rework, Windows Architect Advises." *Network World*, March 19 2010.
118. Johnson, T., and Davis, T. "Space Efficient Parallel Buddy Memory Management." *Proceedings, Fourth International Conference on Computers and Information*, May 1992.
119. Johnston, B.; Javagal, R.; Datta, A.; and Ghosh, S. "A Distributed Algorithm for Resource Deadlock Detection." *Proceedings, Tenth Annual Phoenix Conference on Computers and Communications*, March 1991.
120. Jones, A., and Schwarz, P. "Experience Using Multiprocessor Systems—A Status Report." *Computing Surveys*, June 1980.
121. Jones, M. "What Really Happened on Mars?" [http://research.microsoft.com/~mbj/Mars\\_Pathfinder/Mars\\_Pathfinder.html](http://research.microsoft.com/~mbj/Mars_Pathfinder/Mars_Pathfinder.html), 1997.
122. Jul, E. "Migration of Light-Weight Processes in Emerald." *Newsletter of the IEEE Computer Society Technical Committee on Operating Systems*, Winter 1989.
123. Jul, E.; Levy, H.; Hutchinson, N.; and Black, A. "Fine-Grained Mobility in the Emerald System." *ACM Transactions on Computer Systems*, February 1988.
124. Kapp, C. "Managing Cluster Computers." *Dr. Dobb's Journal*, July 2000.
125. Katz, R.; Gibson, G.; and Patterson, D. "Disk System Architecture for High Performance Computing." *Proceedings of the IEEE*, December 1989.
126. Kay, J., and Lauder, P. "A Fair Share Scheduler." *Communications of the ACM*, January 1988.

127. Kerner, S. "Inside the Box: Can Containers Simplify Networking?" *Network Evolution*, February 2016.
128. Kessler, R., and Hill, M. "Page Placement Algorithms for Large Real-Indexed Caches." *ACM Transactions on Computer Systems*, November 1992.
129. Khalidi, Y.; Talluri, M.; Williams, D.; and Nelson, M. "Virtual Memory Support for Multiple Page Sizes." *Proceedings, Fourth Workshop on Workstation Operating Systems*, October 1993.
130. Khusainov, V. "Practical Advice on Running  $\mu$ Clinux on Cortex-M3/M4." *Electronic Design*, September 17, 2012.
131. Kilburn, T.; Edwards, D.; Lanigan, M.; and Sumner, F. "One-Level Storage System." *IRE Transactions*, April 1962.
132. Kleiman, S., Eykholt, J. "Interrupts as Threads." *Operating System Review*, April 1995.
133. Kleiman, S.; Shah, D.; and Smallders, B. *Programming with Threads*. Upper Saddle River, NJ: Prentice Hall, 1996.
134. Kleinrock, L. *Queueing Systems, Volume II: Computer Applications*. New York: Wiley, 1976.
135. Knuth, D. "An Experimental Study of FORTRAN Programs." *Software Practice and Experience*, Vol. 1, 1971.
136. Knuth, D. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Reading, MA: Addison-Wesley, 1997. Имеется русский перевод: Кнут Дональд Эрвин. *Искусство программирования, том 1. Основные алгоритмы*, 3-е изд. — М.: Издательский дом “Вильямс”, 2000.
137. Knuth, D. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Reading, MA: Addison-Wesley, 1998. Имеется русский перевод: Кнут Дональд Эрвин. *Искусство программирования, том 3. Сортировка и поиск*, 2-е изд. — М.: Издательский дом “Вильямс”, 2000.
138. Korson, T., and McGregor, J. "Understanding Object-Oriented: A Unifying Paradigm." *Communications of the ACM*, September 1990.
139. Lamport, L. "A New Solution to Dijkstra's Concurrent Programming Problem." *Communications of the ACM*, August 1974.
140. Lamport, L. "The Mutual Exclusion Problem Has Been Solved." *Communications of the ACM*, January 1991.
141. Lamport, L. "Time, Clocks, and the Ordering of Events in a Distributed System." *Communications of the ACM*, July 1978.
142. Lampson, B. "Protection." *Proceedings, Fifth Princeton Symposium on Information Sciences and Systems*, March 1971; Reprinted in *Operating Systems Review*, January 1974.
143. Lampson, B., and Redell D. "Experience with Processes and Monitors in Mesa." *Communications of the ACM*, February 1980.
144. Laplante, P., ed. *Great Papers in Computer Science*. New York, NY: IEEE Press, 1996.

145. LaRowe, R.; Holliday, M.; and Ellis, C. "An Analysis of Dynamic Page Placement in a NUMA Multiprocessor." *Proceedings, 1992 ACM SIGMETRICS and Performance '92*, June 1992.
146. LeBlanc, T., and Mellor-Crummey, J. "Debugging Parallel Programs with Instant Replay." *IEEE Transactions on Computers*, April 1987.
147. Leland, W., and Ott, T. "Load-Balancing Heuristics and Process Behavior." *Proceedings, ACM SigMetrics Performance 1986 Conference*, 1986.
148. Leonard, T. "Dragged Kicking and Screaming: Source Multicore." *Proceedings, Game Developers Conference 2007*, March 2007.
149. Leroudier, J., and Potier, D. "Principles of Optimality for Multiprogramming." *Proceedings, International Symposium on Computer Performance Modeling, Measurement, and Evaluation*, March 1976.
150. Letwin, G. *Inside OS/2*. Redmond, WA: Microsoft Press, 1988.
151. Leutenegger, S., and Vernon, M. "The Performance of Multiprogrammed Multiprocessor Scheduling Policies." *Proceedings, Conference on Measurement and Modeling of Computer Systems*, May 1990.
152. Levin, J. "GCD Internals." *Mac OS X and iOS Internals: To the Apple's Core*. newosxbook.com, 2016.
153. Levis, P. "Experiences from a Decade of TinyOS Development." *10th USENIX Symposium on Operating Systems Design and Implementation*, 2012.
154. Lewis, B., and Berg, D. *Threads Primer*. Upper Saddle River, NJ: Prentice Hall, 1996.
155. Lhee, K., and Chapin, S., "Buffer Overflow and Format String Overflow Vulnerabilities." *Software: Practice and Experience*, Volume 33, 2003.
156. Ligneris, B. "Virtualization of Linux Based Computers : The Linux-VServer Project." *Proceedings of the 19th International Symposium on High Performance Computing Systems and Applications*, 2005.
157. Liu, C., and Layland, J. "Scheduling Algorithms for Multiprogramming in a Hard Real-time Environment." *Journal of the ACM*, January 1973.
158. Love, R. "I/O Schedulers." *Linux Journal*, February 2004.
159. Lynch, N. *Distributed Algorithms*. San Francisco, CA: Morgan Kaufmann, 1996.
160. Mackall, M. "Slob: Introduce the SLOB Allocator." November 2005.  
<http://lwn.net/Articles/157944/>.
161. Maekawa, M.; Oldehoeft, A.; and Oldehoeft, R. *Operating Systems: Advanced Concepts*. Menlo Park, CA: Benjamin Cummings, 1987.
162. Majumdar, S.; Eager, D.; and Bunt, R. "Scheduling in Multiprogrammed Parallel Systems." *Proceedings, Conference on Measurement and Modeling of Computer Systems*, May 1988.
163. Martin, J. *Principles of Data Communication*. Englewood Cliffs, NJ: Prentice Hall, 1988.
164. Marwedel, P. *Embedded System Design*. Dordrecht, The Netherlands: Springer, 2006.

165. McCullough, D. “*µLinux for Linux Programmers.*” *Linux Journal*, July 2004.
166. McDougall, R., and Laudon, J. “Multi-Core Microprocessors Are Here.” *;login:*, October 2006.
167. McDougall, R., and Mauro, J. *Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture*. Palo Alto, CA: Sun Microsystems Press, 2007.
168. McKusick, M.; Neville-Neil, J.; and Watson, R. *The Design and Implementation of the FreeBSD Operating System*. Upper Saddle River, NJ: Addison-Wesley, 2015.
169. Menage, P. “Adding Generic Process Containers to the Linux Kernel.” *Linux Symposium*, June 2007.
170. Mesnier, M.; Ganger, G.; and Riedel, E. “Object-Based Storage.” *IEEE Communications Magazine*. August 2003.
171. Microsoft Corp. *Microsoft Windows NT Workstation Resource Kit*. Redmond, WA: Microsoft Press, 1996.
172. Milojicic, D.; Dougulis, F.; Paindaveine, Y.; Wheeler, R.; and Zhou, S. “Process Migration.” *ACM Computing Surveys*, September 2000.
173. Min, R., et al. “Energy-Centric Enabling Technologies for Wireless Wensor Networks.” *IEEE wireless communications*, vol. 9, no. 4, 2002.
174. Morgan, K. “The RTOS Difference.” *Byte*, August 1992.
175. Morra, J. “Google Rolls Out New Version of Android Operating System.” *Electronic Design*, August 24, 2016.
176. Mosberger, D., and Eranian, S. *IA-64 Linux Kernel: Design and Implementation*. Upper Saddle River, NJ: Prentice Hall, 2002.
177. National Institute of Standards and Technology. *Guide to General Server Security*. Special Publication 800-124, July 2008.
178. Nelson, G. *Systems Programming with Modula-3*. Englewood Cliffs, NJ: Prentice Hall, 1991.
179. Nelson, M.; Welch, B.; and Ousterhout, J. “Caching in the Sprite Network File System.” *ACM Transactions on Computer Systems*, February 1988.
180. Ousterhout, J., et al. “A Trace-Drive Analysis of the UNIX 4.2 BSD File System.” *Proceedings, Tenth ACM Symposium on Operating System Principles*, 1985.
181. Ousterhout, J., et al. “The Sprite Network Operating System.” *Computer*, February 1988.
182. Pabla, C. “Completely Fair Scheduler.” *Linux Journal*, August 2009.
183. Parker-Johnson, P. “Getting to Know OpenStack Neutron: Open Networking in Cloud Services.” *TechTarget article*, December 13, 2013. <http://searchtelecom.techtarget.com/tip/Getting-to-know-OpenStack-Neutron-Open-networking-in-cloud-services>.
184. Patterson, D. “Reduced Instruction Set Computers.” *Communications of the ACM*, January 1985.
185. Patterson, D., and Sequin, C. “A VLSI RISC.” *Computer*, September 1982.

186. Patterson, D.; Gibson, G.; and Katz, R. "A Case for Redundant Arrays of Inexpensive Disks (RAID)." *Proceedings, ACM SIGMOD Conference of Management of Data*, June 1988.
187. Pazzini, M., and Navaux, P. "TRIX, A Multiprocessor Transputer-Based Operating System." *Parallel Computing and Transputer Applications*, edited by M. Valero et al., Barcelona, Spain: IOS Press/CIMNE, 1992.
188. Peir, J.; Hsu, W.; and Smith, A. "Functional Implementation Techniques for CPU Cache Memories." *IEEE Transactions on Computers*, February 1999.
189. Petersen, H., et al. "Old Wine in New Skins? Revisiting the Software Architecture for IP Network Stacks on Constrained IoT Devices." *ACM MobiSys Workshop on IoT Challenges in Mobile and Industrial Systems (IoTSys)*, May 2015.
190. Peterson, G. "Myths about the Mutual Exclusion Problem." *Information Processing Letters*, June 1981.
191. Peterson, J., and Norman, T. "Buddy Systems." *Communications of the ACM*, June 1977.
192. Pizzarelli, A. "Memory Management for a Large Operating System." *Proceedings, International Conference on Measurement and Modeling of Computer Systems*, May 1989.
193. Pollack, F. "New Microarchitecture Challenges in the Coming Generations of CMOS Process Technologies (keynote address)." *Proceedings of the 32nd annual ACM/IEEE International Symposium on Microarchitecture*, 1999.
194. Popek, G., and Walker, B. *The LOCUS Distributed System Architecture*, Cambridge, MA: MIT Press, 1985.
195. Przybylski, S.; Horowitz, M.; and Hennessy, J. "Performance Trade-offs in Cache Design." *Proceedings, Fifteenth Annual International Symposium on Computer Architecture*, June 1988.
196. Ramamirtham, K., and Stankovic, J. "Scheduling Algorithms and Operating Systems Support for Real-Time Systems." *Proceedings of the IEEE*, January 1994.
197. Rashid, R., et al. "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures." *IEEE Transactions on Computers*, August 1988.
198. Raynal, M. *Algorithms for Mutual Exclusion*. Cambridge, MA: MIT Press, 1986.
199. Raynal, M. *Distributed Algorithms and Protocols*. New York: Wiley, 1988.
200. Reimer, J. "Valve Goes Multicore." *Ars Technica*, November 5, 2006. [arstechnica.com/articles/paedya/cpu/valve-multicore.ars](http://arstechnica.com/articles/paedya/cpu/valve-multicore.ars).
201. Ricart, G., and Agrawala, A. "An Optimal Algorithm for Mutual Exclusion in Computer Networks." *Communications of the ACM*, January 1981 (*Corrigendum in Communications of the ACM*, September 1981).
202. Ricart, G., and Agrawala, A. "Author's Response to 'On Mutual Exclusion in Computer Networks' by Carvalho and Roucairol." *Communications of the ACM*, February 1983.

203. Ridge, D., et al. "Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs." *Proceedings, IEEE Aerospace Conference*, 1997.
204. Ritchie, D. "The Evolution of the UNIX Time-Sharing System." *AT&T Bell Labs Technical Journal*, October 1984.
205. Ritchie, D. "UNIX Time-Sharing System: A Retrospective." *The Bell System Technical Journal*, July–August 1978.
206. Ritchie, D., and Thompson, K. "The UNIX Time-Sharing System." *Communications of the ACM*, July 1974.
207. Roberson, J. "ULE: A Modern Scheduler for FreeBSD." *Proceedings of BSDCon '03*, September 2003.
208. Robinson, J., and Devarakonda, M. "Data Cache Management Using Frequency-Based Replacement." *Proceedings, Conference on Measurement and Modeling of Computer Systems*, May 1990.
209. Romer, K., and Mattern, F. "The Design Space of Wireless Sensor Networks." *IEEE Wireless Communications*, December 2004.
210. Rosado, T., and Bernardino, J. "An Overview of OpenStack Architecture." *ACM IDEAS '14*, July 2014.
211. Rosenkrantz, D.; Stearns, R.; and Lewis, P. "System Level Concurrency Control in Distributed Database Systems." *ACM Transactions on Database Systems*, June 1978.
212. Russinovich, M.; Solomon, D.; and Ionescu, A. *Windows Internals: Covering Windows 7 and Windows Server 2008 R2*. Redmond, WA: Microsoft Press, 2011.
213. Saraswat, L., and Yadav, P. "A Comparative Analysis of Wireless Sensor Network Operating Systems." *The 5th National Conference; INDIACom*, 2011.
214. Satyanarayanan, M. and Bhandarkar, D. "Design Trade-Offs in VAX-11 Translation Buffer Organization." *Computer*, December 1981.
215. Sauer, C., and Chandy, K. *Computer Systems Performance Modeling*. Englewood Cliffs, NJ: Prentice Hall, 1981.
216. Seghal, A., et al. "Management of Resource Constrained Devices in the Internet of Things." *IEEE Communications Magazine*, December 2012.
217. Selvidge, P. "How Long Is Too Long to Wait for a Webpage to Load." *Usability News*, Wichita State University, July 1999.
218. Serfaoui, O.; Aissaoui, M.; and Eleuldj, M. "OpenStack: Toward an Open- Source Solution for Cloud Computing." *International Journal of Computer Applications*, October 2012.
219. Sevcik, P. "Designing a High-Performance Web Site." *Business Communications Review*, March 1996.
220. Sevcik, P. "Understanding How Users View Application Performance." *Business Communications Review*, July 2002.
221. Sha, L.; Klein, M.; and Goodenough, J. "Rate Monotonic Analysis for Real-Time Systems." in [257].

222. Sha, L.; Rajkumar, R.; and Sathaye, S. "Generalized Rate-Monotonic Scheduling Theory: A Framework for Developing Real-Time Systems." *Proceedings of the IEEE*, January 1994.
223. Shah, A. "Smart Devices Could Get a Big Battery Boost from ARM's New Chip Design." *PC World*, June 1, 2015.
224. Shaw, A. *The Nature of Computation: An Introduction to Computer Science*. Rockville, MD: Computer Science Press, 1981.
225. Shene, C. "Multithreaded Programming Can Strengthen an Operating Systems Course." *Computer Science Education Journal*, December 2002.
226. Shivaratri, N.; Krueger, P.; and Singhal, M. "Load Distributing for Locally Distributed Systems." *Computer*, December 1992.
227. Shneiderman, B. "Response Time and Display Rate in Human Performance with Computers." *ACM Computing Surveys*, September 1984.
228. Shore, J. "On the External Storage Fragmentation Produced by First-Fit and Best-Fit Allocation Strategies." *Communications of the ACM*, August, 1975.
229. Shub, C. "A Unified Treatment of Deadlock." *Journal of Computing in Small Colleges*, October 2003. Available through the ACM Digital Library.
230. Shub, C. "ACM Forum: Comment on a Self-Assessment Procedure on Operating Systems." *Communications of the ACM*, September 1990.
231. Silberschatz, A.; Galvin, P.; and Gagne, G. *Operating System Concepts with Java*. Reading, MA: Addison-Wesley, 2004.
232. Singhal, M. "Deadlock Detection in Distributed Systems." In [40].
233. Siracusa, J. "Grand Central Dispatch." *Ars Technica Review*, 2009. <http://arstechnica.com/apple/reviews/2009/08/mac-os-x-10-6.ars/12>.
234. Smith, A. "Cache Memories." *ACM Computing Surveys*, September 1982.
235. Smith, A. "Disk Cache—Miss Ratio Analysis and Design Considerations." *ACM Transactions on Computer Systems*, August 1985.
236. Smith, D. "Faster Is Better: A Business Case for Subsecond Response Time." *Computerworld*, April 18, 1983.
237. Smith, J. "A Survey of Process Migration Mechanisms." *Operating Systems Review*, July 1988.
238. Smith, J. "Implementing Remote *fork()* with Checkpoint/restart." *Newsletter of the IEEE Computer Society Technical Committee on Operating Systems*, Winter 1989.
239. Snyder, A. "The Essence of Objects: Concepts and Terms." *IEEE Software*, January 1993.
240. Soltesz, S., et al. "Container-Based Operating System Virtualization: A Scalable High-Performance Alternative to Hypervisors." *Proceedings of the EuroSys 2007 2nd EuroSys Conference, Operating Systems Review*, June 2007.
241. Schay, G., and Spruth, W. "Analysis of a File Addressing Method." *Communications of the ACM*, August 1962.

242. Stallings, W. *Computer Organization and Architecture*, 10th ed. Upper Saddle River, NJ: Pearson, 2016.
243. Stallings, W. *Foundations of Modern Networking: SDN, NFV, QoE, IoT and Cloud*. Upper Saddle River, NJ: Pearson, 2016.
244. Stankovic, J. "Research Directions for the Internet of Things." *Internet of Things Journal*, Volume 1, Number 1, 2014.
245. Steensgaard, B., and Jul, E. "Object and Native Code Mobility among Heterogeneous Computers." *Proceedings, 15th ACM Symposium on Operating Systems Principles*, December 1995.
246. Strecker, W. "Transient Behavior of Cache Memories." *ACM Transactions on Computer Systems*, November 1983.
247. Stroustrup, B. "What is Object-Oriented Programming?" *IEEE Software*, May 1988.
248. Suzuki, I., and Kasami, T. "An Optimality Theory for Mutual Exclusion Algorithms in Computer Networks." *Proceedings of the Third International Conference on Distributed Computing Systems*, October 1982.
249. Taivalsaari A. On the Nature of Inheritance. — *ACM Computing Surveys*, September 1996.
250. Takada, H. "Real-Time Operating System for Embedded Systems." In Imai, M. and Yoshida, N. eds. *Asia South-Pacific Design Automation Conference*, 2001.
251. Talluri, M.; Kong, S.; Hill, M.; and Patterson, D. "Tradeoffs in Supporting Two Page Sizes." *Proceedings of the 19th Annual International Symposium on Computer Architecture*, May 1992.
252. Tamir, Y., and Sequin, C. "Strategies for Managing the Register File in RISC." *IEEE Transactions on Computers*, November 1983.
253. Tanenbaum, A. "Implications of Structured Programming for Machine Architecture." *Communications of the ACM*, March 1978.
254. Tauro, C.; Ganesan, N.; and Kumar, A. "A Study of Benefits in Object Based Storage Systems." *International Journal of Computer Applications*, March 2012.
255. Tevanian, A., et al. "Mach Threads and the UNIX Kernel: The Battle for Control." *Proceedings, Summer 1987 USENIX Conference*, June 1987.
256. Thadhani, A. "Interactive User Productivity." *IBM Systems Journal*, No. 1, 1981.
257. Tilborg, A., and Koob, G. eds. *Foundations of Real-Time Computing: Scheduling and Resource Management*. Boston: Kluwer Academic Publishers, 1991.
258. TimeSys Corp. "Priority Inversion: Why You Care and What to Do about It." *TimeSys White Paper*, 2002. [https://linuxlink.timesys.com/docs/priority\\_inversion](https://linuxlink.timesys.com/docs/priority_inversion).
259. Tucker, A. ed. *Computer Science Handbook*, Second Edition. Boca Raton, FL: CRC Press, 2004.
260. Tucker, A., and Gupta, A. "Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors." *Proceedings, Twelfth ACM Symposium on Operating Systems Principles*, December 1989.

261. Vahalia, U. *UNIX Internals: The New Frontiers*. Upper Saddle River, NJ: Prentice Hall, 1996.
262. Vinoski, S. "CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments." *IEEE Communications Magazine*, February 1997.
263. Walker, B., and Mathews, R. "Process Migration in AIX's Transparent Computing Facility." *Newsletter of the IEEE Computer Society Technical Committee on Operating Systems*, Winter 1989.
264. Ward, S. "TRIX: A Network-Oriented Operating System." *Proceedings, COMPCON '80*, 1980.
265. Warren, C. "Rate Monotonic Scheduling." *IEEE Micro*, June 1991.
266. Weizer, N. "A History of Operating Systems." *Datamation*, January 1981.
267. Wendorf, J.; Wendorf, R.; and Tokuda, H. "Scheduling Operating System Processing on Small-Scale Microprocessors." *Proceedings, 22nd Annual Hawaii International Conference on System Science*, January 1989.
268. Wiederhold, G. *File Organization for Database Design*. New York, NY: McGraw-Hill, 1987.
269. Woodbury, P. et al. "Shared Memory Multiprocessors: The Right Approach to Parallel Processing." *Proceedings, COMPCON Spring '89*, March 1989.
270. Woodside, C. "Controllability of Computer Performance Tradeoffs Obtained Using Controlled-Share Queue Schedulers." *IEEE Transactions on Software Engineering*, October 1986.
271. Zahorjan, J., and McCann, C. "Processor Scheduling in Shared Memory Multiprocessors." *Proceedings, Conference on Measurement and Modeling of Computer Systems*, May 1990.
272. Zajcew, R., et al. "An OSF/1 UNIX for Massively Parallel Multicomputers." *Proceedings, Winter USENIX Conference*, January 1993.
273. Zhuravlev, S., et al. "Survey of Scheduling Techniques for Addressing Shared Resources in Multicore Processors." *ACM Computing Surveys*, November 2012.

# Предметный указатель

## A

ABI, 88  
 ALPC, 124  
 Android, 28; 141  
     HAL, 149  
     RPC, 386  
     блокировка сна, 149  
     будильник, 149  
     каркас приложений, 143  
     операция, 149; 252; 253  
     приложение, 142; 252  
     провайдер контента, 252  
     процесс, 255  
     системная архитектура, 147  
     системные службы, 148  
     служба, 252  
     управление памятью, 489  
     файловая система, 693  
     ядро, 144  
 AOSP, 141  
 AOT, 146  
 APC, 639  
 API, 88  
 APK, 146  
 ART, 145  
 ASCII, 891; 1175

## B

BACI, 1180  
 Beowulf, 928  
 BSD, 134  
 В-дерево, 659

## C

CFS, 575  
 CoAP, 857  
 COM, 919  
 CORBA, 919; 1092  
 CSC, 823

## D

Dalvik, 145; 146  
 DAS, 837  
 DCOM, 1092  
 DDP, 868  
 DMA, 65; 66; 597; 599; 1077  
 DNS, 880  
 Docker, 752  
 DVM, 145

## F

FAT, 670; 684  
 FIFO, 293  
 FSF, 135  
 FTP, 879

## G

GCD, 120  
 GNU, 135  
 GPL, 135

## H

HAL, 121; 149  
 HTTP, 840

## I

IAB, 872  
 IANA, 1146  
 IDS, 775  
 IDT, 129  
 IETF, 875  
 I/OAT, 759  
 IoT, 121; 821; 851  
 IPC, 111  
 IRA, 1175  
 ISA, 87  
 ISR, 557

## J

JCL, 93  
 JIT, 146  
 JVM, 764

**K**

**Kubernetes**, 750

**L**

**Linux**, 29; 134

  μClinix, 718

  атомарная операция, 371

  барьер памяти, 377

  ввод-вывод, 633

  виртуальная память, 483

  встроенная система, 714

  загружаемый модуль, 136

  кеш страничный, 637

  планирование, 574

    групповое, 575

    дисковое, 633

      реального времени, 572

  поток, 248

  пространство имен, 249

  процесс, 246

    клонирование, 248

  семафор, 375

  сеть, 884

  сигнал, 139

    реального времени, 370

  системные вызовы, 139

  управление памятью, 481–485

  файловая система

    cramfs, 717

    ext2, 684

    jffs2, 717

    squashfs, 717

    ubifs, 717

    yaffs2, 717

  виртуальная. См. VFS

  ядро, 136

    компоненты, 138

**LXC**, 752

**M**

**MAC**, 1139

**MANO**, 841

**MFT**, 691; 692

**MTTF**, 114

**MTTR**, 114

**μ**

μClinix, 718

μITRON, 1205

**N**

**NAS**, 838

**NFS**, 840

**NIS**, 250

**NIST**, 750; 778; 794; 821; 823; 828; 841

**NTFS**, 688

  безопасность, 689

  восстановление данных, 692

  главная файловая таблица, 691

  журналирование, 689

  кластер, 689

  сектор, 689

  том, 689; 690

**NUMA**, 458

**O**

**OLE**, 919

**OpenGL**, 144

**OpenStack**, 842

**P**

**POSIX**, 240; 248

**PSW**, 183; 1073

**Q**

**QoS**, 838

**R**

**RAID**, 617

  0, 618

  1, 622

  2, 623

  3, 623

  4, 624

  5, 625

  6, 625

  в Windows, 640

**RCU**, 378

**RFID**, 853

**RIOT**, 862

  ядро, 863

**RMS**, 566

**RPC**, 124; 218; 915

  асинхронный, 918

  передача параметров, 917

  привязка, 917

  синхронный, 918

**RPS**, 609

**S**

SAN, 838  
SHA, 1142  
SHA-1, 1142  
SHA-256, 1142  
SHA-384, 1142  
SHA-512, 1142  
SMB, 840  
SMP, 67; 112; 117; 579  
SMTP, 879  
SNA, 886  
SNMP, 874  
Solaris, 134  
  мьютекс, 380  
  процесс, 241  
  семафор, 381  
  условная переменная, 382  
SQL, 903  
SQLite, 144; 695  
SRM, 152  
SSH, 879  
SSTF, 615  
SVA, 745

**T**

TCP, 880  
TCP/IP, 872  
TFTP, 891–895  
TinyOS, 721  
  арбитр, 730  
  задание, 726  
  команда, 726  
  компоненты, 724  
  планировщик, 727  
  событие, 726  
TLB, 445

**U**

UART, 861  
UDP, 874; 881  
UMS, 234  
UNIX, 28; 129  
  BSD, 134  
FreeBSD, 134  
  SMP, 579  
  классы приоритетов, 578  
  планирование, 578

**NetBSD, 134**

OpenBSD, 134  
Solaris, 134  
SVR4  
  ввод-вывод, 630  
  дисковый кеш, 630  
  классы приоритетов, 577  
  планирование, 576  
  процессы, 198  
  управление памятью, 477  
  устройства ввода-вывода, 632  
архитектура, 130  
виртуальная память, 455; 476  
канал, 367  
каталоги, 683  
планирование, 528  
размещение файлов, 682  
семафор, 368  
сигнал, 369  
совместно используемая память, 368  
сообщение, 368  
суперпользователь, 798  
управление доступом, 797  
файловая система, 679

**V**

VFS, 684  
  запись каталога, 686; 687  
  индексный узел, 686; 687  
  кеш, 688  
  суперблок, 686  
  файл, 686; 687  
VM. См. Виртуальная машина  
VMM, 737  
VMX, 744

**W**

Windows, 28; 121  
BitLocker, 640  
RAID, 640  
SMP, 126  
архитектура, 122  
безопасность, 806  
ввод-вывод, 638  
  асинхронный, 639  
  дисковый кеш, 638  
исполнительная система, 123  
кластеризация, 926  
критический участок, 384  
мьютекс, 383

объект синхронизации, 383  
 планирование, 582  
   многопроцессорное, 584  
 поток, 233  
   атрибуты, 238  
 приложение, 233  
 приоритет, 582  
 процесс, 233; 236  
   атрибуты, 237; 238  
 свопинг, 489  
 семафор, 383  
 событие, 384  
 таймер ожидания, 383  
 теневая копия тома, 640  
 управление доступом, 807  
 управление памятью, 486–488  
 условная переменная, 385  
 файловая система, 638  
   NTFS, См. NTFS  
 функция ожидания, 382  
 шифрование тома, 640  
 ядро, 123  
**WinSock**, 880  
**Wireshark**, 890  
**WNS**, 235  
**WSN**, 724

**А**

**Автоковариация**, 989  
**Адрес**  
   базовый, 1217  
   виртуальный, 106  
   реальный, 106  
**Алгоритм**  
   Айзенберга–Мак-Гуайра, 332  
   банкира, 355  
   булочкой, 330  
   взаимного исключения, 949; 953  
     распределенный, 950  
     централизованный, 949  
   Деккера, 269; 326  
   лифта, 615  
   Лэмпорта, 953  
     оптимизация, 956  
   обнаружения взаимоблокировки, 360  
   обнаружения распределенной взаимобло-  
     ковки, 964  
   передачи эстафеты, 958  
   Петерсона, 275

распределенных моментальных снимков, 945  
 Ривеста-Шамира-Адлемана, 1137  
 система двойников, 412  
   ленивая, 480  
 сложность, 1115  
**Анализ**  
   очередей, 1002; 1107–1111  
     С-функция Эрланга, 1023  
     коэффициент Пуассона, 1023  
   многоканальная система массового обслу-  
     живания, 1015; 1022; 1111  
   обозначение Кендалла, 1018  
   одноканальная система массового обслужи-  
     вания, 1011; 1018  
   с приоритетами, 1029  
   формула Литтла, 1017  
   систем массового обслуживания. См. Ана-  
     лиз очередей

**Ансамбль**, 996

**Атомарность**, 268  
**Аутентификация**, 776; 1137  
   без шифрования, 1138  
   код аутентификации, 1139

**Б**

**База данных**, 649  
   реляционная, 902  
**Байткод**, 145  
**Барьер памяти**, 377  
**Безопасность**, 773–813  
   Windows, 806  
   автентификация, 776  
   брандмауэр, 777; 804  
   защита  
     времени выполнения, 785  
     времени компиляции, 782  
   зловредная программа, 774  
    злоумышленник, 774  
   канарейка, 785  
   обнаружение вторжений, 775  
   обновление, 805  
   переполнение буфера, 778  
   переполнение стека, 779  
   протоколирование, 805  
   резервное копирование, 805; 806  
**Блокировка**  
   гибкая, 385  
   циклическая, 373  
**Брандмауэр**, 777; 804

**Буфер**

быстрой переадресации. См. TLB  
циклический, 607

**В****Ввод-вывод**

буферизация, 604–607  
двойная, 607  
канал, 599  
логический, 602; 652  
планирование, 602  
программируемый, 597; 1075  
проектирование, 601  
универсальность, 602  
управляемый прерываниями, 597; 1076  
эффективность, 601

**Вероятность**, 976; 979  
условная, 979

**Взаимоблокировка**, 268; 282; 343  
алгоритм обнаружения, 360  
динамическая, 273  
обнаружение, 360  
предотвращение, 352; 960  
при обмене сообщениями, 968  
распределенная, 958–971  
условия возникновения, 351  
устранение, 354  
фантомная, 960

**Взаимоисключение**, 269; 282  
алгоритм Деккера, 269  
программный подход, 269  
распределенное, 947–958  
через сообщения, 316

**Виртуализация**, 736; 835  
аппаратная поддержка, 744  
контейнерная, 745  
на уровне процессора, 753  
паравиртуализация, 743

**Виртуальная машина**, 737

Java VM, 764  
Linux VServer, 765  
ввод-вывод, 757  
гипервизор, 737; 740–745  
гостевая операционная система, 738  
изоляция файловой системы, 765  
монитор, 737  
самоанализ, 745  
степень консолидации, 738  
управление памятью, 755

**Вирус**, 1219

**Внешняя фрагментация**, 410  
**Внутренняя фрагментация**, 407  
**Время оборота**, 509  
**Время отклика**, 506; 1103  
**Вызов удаленных процедур**. См. RPC  
**Вытеснение**, 168  
**Вычисления**  
  клиент/сервер, 899  
  облачные, 821–830  
  распределенные, 901; 934  
  туманные, 853

**Г**

**Гипервизор**, 737; 740–745  
ESXi, 760  
Microsoft Hyper-V , 764  
Xen, 762

**Голодание**, 269; 282

**Д**

**Дайджест**, 1140; 1142  
**Двойная буферизация**, 607  
**Диаграмма Венна**, 977  
**Дисковый накопитель**  
  Blu-ray, 1129  
  CD-R, 1128  
  CD-ROM, 1126  
  CD-RW, 1128  
  CD с возможностью записи, 1128  
  DVD, 1129  
  винчестер, 1124  
время доступа, 609  
время поиска, 609  
дорожка, 1120  
многозонная запись, 1121  
сектор, 1121  
цилиндр, 1123

**Диспетчер**, 159

**Доступность**, 115

**З**

**Задание**, 91; 112  
**Задача**  
  о парикмахерской, 1052  
об обедающих философах, 363  
производителя/потребителя, 296  
читателей/писателей, 318  
**Закон Амдала**, 228; 1097

**И**

Имитационное моделирование, 1010  
 Инверсия приоритета, 570  
 Индексный узел, 680  
 Инкапсуляция, 1087  
 Инстанцирование, 1087  
 Интернет вещей. См. IoT  
 Интерфейс, 1089; 1090

**К**

Кадр, 400; 416  
 стека, 1195  
 Канал, 367; 944  
 Каталог  
 рабочий, 666  
 текущий, 666  
 Квантование времени, 99; 513  
 Кеш, 60; 554; 706  
 дисковый, 626  
 согласованность, 69; 908  
 Кластер, 919; 922  
 Beowulf, 928  
 архитектура, 924  
 распараллеливание вычислений, 923  
 распределение нагрузки, 923

Клиент  
 толстый, 906  
 тонкий, 907  
 Ковариация, 986  
 Код Хэмминга, 623  
 Кольцо защиты, 755  
 Команда  
 выборка, 43  
 категории, 43  
 цикл, 43

Компиляция  
 AOT, 146  
 JIT, 146  
 Контейнер, 746  
 Docker, 752  
 микрослужба, 750  
 файловая система, 750  
 Контекст выполнения, 103  
 Коэффициент корреляции, 986; 989  
 Криптография, 1131  
 AES, 1134  
 DES, 1133

**RSA, 1137**

блочный шифр, 1133  
 криптоанализ, 1132  
 симметричное шифрование, 1131  
 шифрование с открытым ключом, 1134  
 Критический участок, 268; 384; 948  
 Кросскомпилятор, 712

**Л**

Ловушка, 192  
 Логический адрес, 416  
 Локальность, 75; 626  
 временная, 77  
 пространственная, 77

**М**

Маршаллинг, 387  
 Маршрутизатор, 873  
 Метод, 1087  
 Микроконтроллер, 707  
 Микропроцессор, 705  
 Микрослужба, 750  
 Многозадачность, 95; 267  
 Многопоточность, 111; 214; 238  
 Многопроцессорность, 267  
 симметричная. См. SMP  
 Модуляризация, 90  
 Моментальный снимок, 944  
 распределенный, 944  
 Монитор, 91; 290; 304  
 виртуальных машин, 737  
 Лэмпсона–Ределла, 309  
 Хоара, 308  
 Мьютекс, 289; 293

**Н**

Надежность, 114; 557  
 восстановлением после отказа, 923  
 время  
 безотказной работы, 115  
 восстановления среднее, 114  
 наработки на отказ среднее, 114  
 доступность, 115  
 отказ, 116  
 преодоление отказа, 923  
 простой, 115  
 реального времени, 557  
 Накачка, 757

**Наследование, 1088**  
**иерархия, 1089**  
**Наследование приоритетов, 572**

**О**

**Облако**  
 аудитор, 828; 830  
 брокер, 828; 830  
 гибридное, 827  
 закрытое, 826  
 коллективное, 826  
 оператор, 828; 830  
 открытое, 825

**Облачные вычисления, 821–830**  
 гибридное облако, 827  
 закрытое облако, 826  
 коллективное облако, 826  
 модель обслуживания, 823  
 открытое облако, 825  
 характеристики, 822  
 эталонная архитектура, 828

**Обработка данных, 906**

**Объектно-ориентированное проектирование, 1083–1090**

**Оверлей, 404; 406**

**Операционная система, 85**

    AIX, 938

    Android, См. Android

    CTSS, 98

    eCos, 1199

    GCOS 8, 473

    Linux, См. Linux

    LOCUS, 938

    Mac OS X, 256

    MS-DOS, 121

    OpenStack, 842

    OS/2, 260

    OS/390, 152; 261

    OSF/1 AD, 938

    OS/MFT, 408

    OS/MVT, 408

    RIOT, 862

    Solaris, 241

    Sprite, 941

    TinyOS, См. TinyOS

    TRIX, 226

    UNIX, См. UNIX

    VAX/VMS, 207

    Windows, См. Windows

**Xenix, 133**  
**безопасность, 773–813**  
**встроенная, 703–714**  
**гостевая, 738**  
**детерминированная, 556**  
**для Интернета вещей, 856**  
**облачная, 831**  
     архитектура, 835  
     виртуальная сеть, 838  
     виртуальное хранилище, 837  
     виртуальные вычисления, 837  
**обновления, 801; 805**  
**пакетная, 91**  
**предоставляемые услуги, 86**  
**распределенная, 113; 869; 934**  
**сетевая, 869**  
**установка, 801**  
**эволюция, 90**  
**ядро, 88; 188**  
     микроядро, 111  
     монолитное, 111

**Основная память, 40**

**Отказ, 116**

    восстановление после отказа, 923  
     преодоление, 923

**Отказоустойчивость, 114**

**Очередь, 316; 1014; 1111**  
     распределенная, 953  
     с приоритетами, 1029

**П**

**Память**

**адрес**

    логический, 415; 416  
     относительный, 415  
     физический, 415; 416

**виртуальная, 60; 106; 404; 435–441**

**внешняя фрагментация, 410**

**внутренняя фрагментация, 407**

**вторичная, 60**

**двухуровневая, 78**

**защита, 94; 401**

**иерархия, 57**

**кадр, 416**

**кеш, 447; 465; 554**

**логическая организация, 403**

**оперативная, 40**

**основная, 40**

**первичная, 40**

- перегрузка, 757  
перемещение, 401  
размер страницы, 448  
распределение, 404  
  динамическое, 408  
  фиксированное, 404  
реальная, 437  
резидентное множество, 436; 465  
  стратегия PFF, 471  
  стратегия VSWS, 472  
  стратегия рабочего множества, 469  
сегмент, 419  
сегментация, 419; 451  
система двойников, 412  
  ленивая, 480  
система управления, 98  
словарь перемещения, 429  
совместное использование, 403  
страница, 416  
  буферизация, 464  
  замещение, 458  
страничная организация, 416–419  
таблица страниц, 416  
таблицы, 178  
уплотнение, 410  
управление, 105  
управление загрузкой, 473  
физическая организация, 403  
Паравиртуализация, 743  
Параллельные вычисления, 267; 1072  
Пережидание занятости, 268; 269; 288  
Переполнение  
  буфера, 778  
  стека, 779  
Планирование, 500  
  бригадное, 548; 550  
  в Linux, 572  
  ввода-вывода, 602  
  групповое, 550  
  динамическое, 548; 553  
  дисковое, 608  
    C-SCAN, 616  
    FIFO, 614  
    LIFO, 615  
    SCAN, 615  
    SSTF, 615  
  приоритеты, 614  
  случайное, 611  
  долгосрочное, 502  
  краткосрочное, 504  
  многопроцессорное, 541; 544  
    FCFS, 546  
    ведущий/ведомый, 544  
  динамическое, 553  
  назначение процессоров, 551  
  планирование потоков, 546  
  равноправные процессы, 545  
  разделение загрузки, 548  
  многоядерное, 554  
  реального времени, 555; 560  
    LLF, 587  
    MUIF, 588  
  динамическое наилучшего результата, 561  
  динамическое на основе расписания, 561  
  жесткие задания, 556  
  мягкие задания, 556  
  статическое на основе приоритетов, 561  
  статическое с использованием таблиц, 561  
  частотно-монотонное, 566  
  сопланирование, 550  
  справедливое, 526  
  среднесрочное, 503  
  стратегия  
    FCFS, 512  
    круговое, 513  
    круговое виртуальное, 514  
    наивысшее отношение отклика, 519  
    наименьшее оставшееся время, 517  
    самый короткий процесс, 515  
    снижение приоритета, 519  
    справедливый планировщик, 527  
    энергосберегающее, 863  
Поле, 648  
  ключевое, 656  
Полиморфизм, 1089  
Порт, 314; 877; 880; 1145  
Порядок байтов, 1150  
  сети, 1150  
Последовательная обработка, 91  
Поток, 105; 111; 214  
  миграция, 581  
  пользовательский, 220  
  синхронизация, 219  
  уровня ядра, 224  
Почтовый ящик, 314  
Правило Поллака, 1080  
Прерывание, 46; 94; 102; 191  
  запрет, 55  
  обработчик, 49

Принцип локальности, 439  
локальности обращений, 59

Приоритет, 562  
инверсия, 570  
наследование, 572  
потолок, 572

Простой, 115

Пространство имен, 249

Протокол, 870  
FTP, 879  
HTTP, 840  
IP, 873  
MLP, 1041  
NFS, 840  
SMB, 840  
SMTP, 879  
SNMP, 874  
SSH, 879  
TCP, 873; 880  
TCP/IP, 872  
TFTP, 870  
UDP, 874; 881  
архитектура, 869  
квитирования, 1045  
сетевой, 141

Процедура  
вызов и возврат, 1194  
запись активации, 1197  
передача параметров, 1195  
реентерабельная, 1196

Процесс, 101; 112; 156–158  
атрибуты, 181  
броневского движения, 992

Винера-Леви, 992  
выселение, 941  
вытеснение, 168  
дочерний, 164  
завершение, 164  
идентификатор, 158; 181  
изоляция, 105; 766  
контейнер, 746  
контекст, 158  
модель с двумя состояниями, 162  
модель с пятью состояниями, 166  
назначение процессору, 544  
образ, 180  
переключение, 190  
перенос, 934

порождение, 164  
приоритет, 158; 506  
Пуассоновский, 994  
родительский, 164  
с долговременной памятью, 990  
с кратковременной памятью, 990  
след, 159  
случайный, 987  
создание, 163; 189  
состояние, 103; 104; 158; 942; 944  
глобальное, 944  
стохастический, 987  
стационарный, 989  
управляющий блок, 159; 180; 186

Процессор, 40; 1072  
выполнение команд, 43  
графический, 42  
прикладной, 705  
регистры, 1072  
специализированный, 705  
ядро, 42

Прямой доступ к памяти. См. DMA

Пылинка, 857

**P**

Разделение времени, 98  
Разделение загрузки, 548  
Рандеву, 313  
Распределенная обработка данных, 868  
Расщепление данных, 618  
Режим  
переключение, 192  
пользовательский, 94; 188  
разделения времени, 98  
системный, 188  
ядра, 94; 188

Резервное копирование, 806  
Резидентное множество, 436

Ресурс  
граф распределения, 349  
повторно используемый, 347  
расходуемый, 349

**C**

Свопинг, 171  
Сегмент, 400  
Семафор, 289; 290; 1044  
бинарный, 291

- реализация, 302  
 сильный, 293  
 слабый, 293  
**Сеть**, 870  
 Симметричная многопроцессорность. *См.* SMP  
**Синхронизация**, 118; 312  
 зернистость, 542  
**Системная шина**, 41  
**Слово**, 1072  
 Слово состояния программы, 52; 183; 1073  
**Случайная переменная**, 981  
 независимые переменные, 986  
**Событие**, 951  
 в теории вероятности, 976  
 упорядочение, 951  
**Сокет**, 879; 880; 1144; 1145  
 Linux, 885  
 WinSock, 880  
 адрес, 1148  
 асинхронный ввод-вывод, 1166  
 дейтаграмм, 880; 1161  
 закрытие, 1156  
 отправка и получение сообщения, 1155  
 подключение, 1151  
 потоковый, 880  
 привязка к порту, 1149  
 применение, 881  
 прослушивание, 1154  
 создание, 1148  
**Сообщение**, 312; 368  
 адресация, 314  
**Сопланирование**, 550  
**Сопрограмма**, 270; 327  
**Состояние гонки**, 269; 279; 1044  
**Спектральная плотность**, 990  
**Спектр мощности**, 990  
**Стек**, 1073; 1193  
 кадр, 1195  
**Степень консолидации**, 738  
**Страница**, 400; 416  
 буферизация, 464  
 замещение, 458  
 защитная, 787
- T**
- Таблица  
 битовая, 676  
 ввода-вывода, 178
- дискового размещения, 676  
 памяти, 178  
 процессов, 179  
 прямого доступа, 1099  
 размещения файлов, 670  
 страниц, 416; 441  
 инвертированная, 443  
 файлов, 179  
 хеш-таблица, 1100
- Таймер**, 94
- Теорема**  
 Байеса, 980  
 Джексона, 1032  
 центральная предельная, 985
- Теория вероятности**, 976–981  
 вероятность, 976; 979  
 условная, 979  
 второй момент, 982  
 дисперсия, 982  
 испытание, 978  
 ковариация, 986  
 коэффициент корреляции, 986; 989  
 математическое ожидание, 982  
 первый момент, 982  
 распределение вероятности  
 нормальное, 985  
 Пуассона, 984; 1113  
 функция, 982  
 экспоненциальное, 983  
 случайная переменная, 981  
 случайный процесс, 987  
 событие, 976  
 дополнение, 977  
 достоверное, 976  
 независимые события, 980  
 среднее значение, 982  
 стандартное отклонение, 982  
 стохастический процесс, 987  
 функция  
 автокорреляции, 989
- Теория массового обслуживания. *См.***  
**Анализ очередей**
- Том, 678
- y**
- Упорядочение событий, 951  
**Управление**  
 памятью, 105; 400  
 ресурсами, 108

**Условная переменная**, 305  
**Устройство**  
 ввода-вывода, 41; 87  
 виртуальное, 744  
 с ограниченными ресурсами, 857

**Ф**

**Файл**, 106; 647; 649  
 атрибуты, 691  
 журнальный, 656  
 запись, 648  
 именование, 665  
 индексированный, 654; 658  
 индексно-последовательный, 654; 657  
 каталог, 662  
 организация, 654  
 поле, 648  
 полное имя, 665  
 последовательный, 654; 656  
 права доступа, 667  
 прямого доступа, 654; 658  
 путь, 665  
 смешанный, 654  
 совместное использование, 667  
 функции управления, 652  
**Файловая система**, 604; 647  
 архитектура, 651  
 виртуальная. См. VFS  
 каталог, 662  
 контейнера, 750  
 корневая, 712  
**Физический адрес**, 416  
**Формула**  
 Литтла, 1017; 1034  
 Шея–Спрута, 1102  
**Фрейм**. См. Кадр памяти  
**Функция**  
 С-функция Эрланга, 1023  
 автокорреляции, 989  
 нормализованная, 989  
 плотности вероятности, 982  
 распределения вероятности, 982  
 хеш-функция, 1100  
 безопасная, 1142

**Х**

**Хеширование**, 1099–1102  
 линейное, 1101

**Ц**

**Циклическая блокировка**, 373

**Э**

**Эргодичность**, 996

**Я**

**Ядро**, 705  
 операционной системы, 88  
 процессора, 42; 69  
**Язык**  
 Ada, 1094  
 BACI, 1180  
 C, 76; 127; 130; 144; 256; 685; 781; 783;  
 1094; 1144  
 C#, 1089  
 C++, 144; 256; 685; 1089; 1094; 1144  
 COBOL, 1094  
 Concurrent Pascal, 308; 1180  
 IDL, 1092; 1093  
 Java, 685; 1089; 1094  
 Mesa, 309  
 Modula3, 309  
 nesC, 726  
 Pascal, 76; 1144  
 Pascal-S, 1180  
 Smalltalk, 1094  
 SQL, 695; 903  
 Visual Basic, 1144  
 Visual C++, 785  
 определения интерфейса, 1092  
 управления заданиями, 93

# ОПЕРАЦИОННЫЕ СИСТЕМЫ

*Внутренняя структура и принципы  
проектирования*

Новое издание многократно издававшейся ранее книги В. Столлингса позволит читателю узнать, что такое операционные системы и как они функционируют, проектируются и реализуются на практике.

Новое, девятое, издание книги существенно обновлено и дополнено. Здесь вы найдете информацию как об общих принципах построения операционных систем и функционировании таких вещей, как сети, файловые системы, многозадачность, распределенные системы или облачные вычисления, так и о конкретных реализациях тех или иных подсистем в новейших версиях операционных систем Windows, Linux, Android, а также во многих других.

Книга пригодна в качестве учебника для преподавателей и студентов соответствующих специальностей, снабжена огромным количеством задач и дополнительных материалов, но будет интересна всем, кого интересует этот увлекательный раздел информатики.

## ОБ АВТОРЕ

**Доктор Вильям Столлингс** является автором 18 книг, а включая переиздания, — более 40 книг по компьютерной безопасности, компьютерным сетям и архитектуре компьютеров. Его перу принадлежат многочисленные публикации в журналах, включая такие издания, как *Proceedings of the IEEE*, *ACM Computing Reviews* и *Cryptologia*.

От Ассоциации академических авторов Вильям Столлингс 13 раз получал награду за лучший учебник года в области компьютерных наук.

Столлингс имеет ученую степень доктора философии в области компьютерных наук в МТИ и степень бакалавра электротехники в Нотр-Дам.

ISBN 978-5-907203-08-2



19178

 **ДИАЛЕКТИКА**  
[www.dialektika.com](http://www.dialektika.com)

 Pearson

9 785907 203082