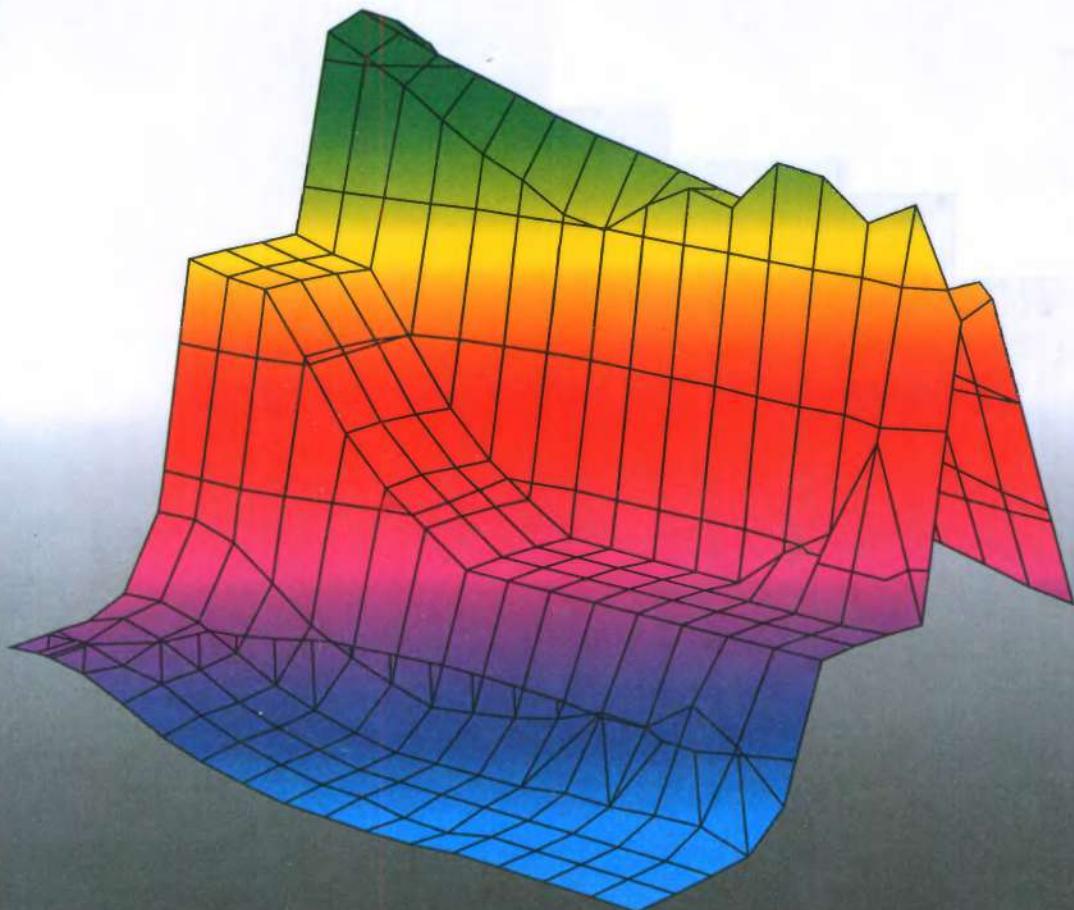


КОМПЬЮТЕРНЫЕ СИСТЕМЫ

АРХИТЕКТУРА И ПРОГРАММИРОВАНИЕ

ВЗГЛЯД ПРОГРАММИСТА



Рэндал Э. Брайант, Дэвид Р. О'Халларон

bhv®

Рэндал Э. Брайант
Дэвид Р. О'Халларон

Компьютерные системы Архитектура и программирование взгляд программиста

Санкт-Петербург
«БХВ-Петербург»
2005

Computer Systems

A Programmer's Perspective

Randal E. Bryant

David R. O'Hallaron



Pearson Education, Inc.
Upper Saddle River, New Jersey 07458

УДК 681.3.06
ББК 32.973.26
Б87

Брайант Р., О'Халларон Д.

Б87 Компьютерные системы: архитектура и программирование.
Пер. с англ. — СПб.: БХВ-Петербург, 2005. — 1104 с.: ил.

ISBN 5-94157-433-9

В основу книги положен разработанный авторами учебный курс "Введение в компьютерные системы", преподаваемый более чем в 90 университетах по всему миру. Описывается компьютерная система, под которой понимаются не только "стандартные элементы архитектуры", такие как центральный процессор, память, порты ввода-вывода и др., но также операционная система, компилятор и сетевое окружение. Рассмотрено представление данных и программ на машинном уровне, архитектура процессора, оптимизация программ, связывание и управление потоками, виртуальная память и управление памятью, ввод-вывод на системном уровне, сетевое и параллельное программирование. Описано, каким образом перечисленные выше аспекты необходимо учитывать программисту при разработке собственных приложений и систем. Приведенные в книге примеры для процессоров, совместимых с Intel (IA32), написаны на языке C и выполняются в операционной системе Unix или сходных, например Linux.

Для преподавателей, студентов и программистов

УДК 681.3.06
ББК 32.973.26

Группа подготовки издания:

Главный редактор	Екатерина Кондукова	Редактор	Елена Кашакова
Зам. главного редактора	Игорь Шиншенин	Компьютерная верстка	Ольги Сергиенко
Зав. редакцией	Григорий Добкин	Корректор	Элинна Дмитриева
Перевод с английского	Дмитрий Ежков, Станислава Шестакова	Дизайн обложки	Игоря Цыбульникова
		Зав. производством	Николай Тверских

Authorized translation from the English language edition, entitled COMPUTER SYSTEMS: A PROGRAMMER'S PERSPECTIVE, 1st Edition, ISBN 0-13-034074-X, by BRYANT, RANDAL E. and O'HALLARON, DAVID R., published by Pearson Education, Inc., publishing as Prentice Hall, Copyright © 2003. All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc. RUSSIAN language edition published by BHV—St. Petersburg, Copyright © 2005.

Авторизованный перевод английской редакции, выпущенной Prentice Hall, Pearson Education, Inc., © 2003. Все права защищены. Никакая часть настоящей книги не может быть воспроизведена или передана в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на то нет разрешения Pearson Education, Inc. Перевод на русский язык "БХВ-Петербург", © 2005.

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 31.05.05.

Формат 70×100 $\frac{1}{4}$. Печать офсетная. Усл. печ. л. 89.

Тираж 3000 экз. Заказ № 4071

"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 55.

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

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

ISBN 0-13-034074-X (англ.)
ISBN 5-94157-433-9 (рус.)

© 2003, Pearson Education, Inc., Pearson Prentice Hall
© Перевод на русский язык "БХВ-Петербург", 2005

Оглавление

Предисловие	1
Что нужно знать перед прочтением	1
Как читать книгу	2
Происхождение книги	3
Обзор книги	4
Благодарности	7
Информация об авторах	9
ЧАСТЬ ВВОДНАЯ. ОБЗОР КОМПЬЮТЕРНЫХ СИСТЕМ	11
Глава 1. Экскурс в компьютерные системы	13
1.1. Информация — это биты + контекст	14
1.2. Программы, которые переводятся другими программами в различные формы	16
1.3. Как работает система компиляции	18
1.4. Процессоры читают и интерпретируют инструкции, сохраняемые в памяти	19
1.4.1. Организация аппаратных средств системы	20
Шины	20
Устройства ввода-вывода	21
Оперативная память	21
Процессор	22
1.4.2. Выполнение программы <i>hello</i>	23
1.5. Различные виды кэш-памяти	25
1.6. Устройства памяти образуют иерархию	26
1.7. Операционная система управляет работой аппаратных средств	27
1.7.1. Процессы	29
1.7.2. Потоки	30
1.7.3. Виртуальная память	31
1.7.4. Файлы	32
1.8. Обмен данных в сетях	33
1.9. Следующие шаги	35
1.10. Резюме	35
Библиографические заметки	36

ЧАСТЬ I. СТРУКТУРА И ВЫПОЛНЕНИЕ ПРОГРАММЫ	37
Глава 2. Представление информации и работа с ней	39
2.1. Хранение информации	41
2.1.1. Шестнадцатеричная система исчисления	42
О преобразовании между десятичной и шестнадцатеричной системами	45
2.1.2. Машинные слова	46
2.1.3. Размеры данных	46
2.1.4. Адресация и упорядочение байтов	48
О присвоении типам данных имени	51
Форматированная печать	51
Указатели и массивы	51
Указатель и разыменование (снятие косвенности)	52
2.1.5. Представление строк	54
2.1.6. Представление кода	55
2.1.7. Булевые алгебры и кольца	56
Какая польза от абстрактной алгебры	58
2.1.8. Операции на уровне бита в С	61
2.1.9. Логические операции в С	63
2.1.10. Операции сдвига в С	64
2.2. Целочисленное представление	65
2.2.1. Типы целого	65
2.2.2. Кодировки со знаком и с двоичным дополнительным кодом	66
2.2.3. Преобразования между числами со знаком и без знака	71
2.2.4. Величины со знаком относительно величин без знака в С	74
2.2.5. Расширение битового представления числа	76
2.2.6. Усечение чисел	79
2.3. Целочисленная арифметика	81
2.3.1. Приращение без знака	81
2.3.2. Приращение в дополнительном коде	85
2.3.3. Отрицание в дополнительном-двоичном коде	88
2.3.4. Умножение без знака	90
2.3.5. Умножение в дополнительном двоичном коде	91
2.3.6. Умножение на степени двух	93
2.3.7. Деление на степени двух	94
2.4. Числа с плавающей точкой	96
2.4.1. Дробные двоичные числа	97
2.4.2. Представление стандарта плавающей точки IEEE	100
Нормализованные значения	101
Ненормализованные значения	101
Особые значения	102
2.4.3. Примерные числа	102
2.4.4. Округление	107
2.4.5. Операции с плавающей точкой	109
2.4.6. Плавающая точка в С	111
Арифметические операции с плавающей точкой Intel IA32	112
2.5. Резюме	117
Библиографические примечания	118

Задачи для домашнего решения.....	119
Решение упражнений.....	128
Глава 3. Представление программ на машинном уровне	145
3.1. Историческая перспектива	148
3.2. Кодирование программ.....	151
3.2.1. Программный код машинного уровня	152
3.2.2. Примеры программных кодов	153
3.2.3. Замечание по форматированию	157
3.3. Форматы данных	159
3.4. Доступ к данным	160
3.4.1. Спецификаторы операндов.....	160
3.4.2. Команды перемещения данных	163
3.4.3. Перемещение данных	166
Несколько примеров указателей	166
3.5. Арифметические и логические операции	169
3.5.1. Команда загрузки исполнительного адреса	170
3.5.2. Унарные и бинарные операции	170
3.5.3. Операции сдвига.....	171
3.5.4. Обсуждение	172
3.5.5. Специальные арифметические операции	174
3.6. Управление.....	175
3.6.1. Коды управления.....	176
3.6.2. Доступ к кодам условия.....	177
3.6.3. Команды перехода и их кодирование	180
3.6.4. Трансляция условных переходов.....	184
3.6.5. Циклы.....	187
Циклы <i>do-while</i>	187
Циклы <i>while</i>	190
Циклы <i>for</i>	194
3.6.6. Операторы выбора.....	196
3.7. Процедуры.....	201
3.7.1. Структура стекового фрейма.....	202
3.7.2. Передача управления	203
3.7.3. Соглашения об использовании регистров	204
3.7.4. Примеры процедур	207
3.7.5. Рекурсивные процедуры	210
3.8. Распределение памяти под массивы и доступ к массивам	213
3.8.1. Базовые принципы	213
3.8.2. Арифметические операции с указателями	215
3.8.3. Массивы и циклы	216
3.8.4. Вложенные циклы	218
3.8.5. Массивы фиксированных размеров	220
3.8.6. Динамически размещаемые массивы	222
3.9. Структуры разнородных данных	225
3.9.1. Структуры	226
Представление объекта как структуры типа <i>struct</i>	226
3.9.2. Объединения	229

3.10. Выравнивание	234
3.11. Как пользоваться указателями	237
3.12. Использование отладчика GDB	241
3.13. Ссылки на ячейку в памяти и переполнение буферов	244
3.14. Коды с плавающей точкой	250
3.14.1. Регистры с плавающей точкой	250
3.14.2. Вычисления выражений с помощью стека	252
3.14.3. Операции перемещения и преобразования данных	255
3.14.4. Арифметические операции с плавающей запятой	257
3.14.5. Использование значений с плавающей точкой в процедурах	261
3.14.6. Тестирование и сравнение значений с плавающей точкой	262
3.15. Встраивание ассемблерных кодов в программы на С	265
3.15.1. Базовый встроенный ассемблер	266
3.15.2. Расширенная форма оператора <i>asm</i>	268
3.16. Резюме	273
Библиографические заметки	274
Задачи для домашнего решения	275
Решение упражнений	282
Глава 4. Архитектура процессора	301
4.1. Архитектура системы команд Y86	304
4.2. Логическое проектирование и язык управления аппаратными средствами HCL	319
4.2.1. Логические шлюзы	320
4.2.2. Комбинационные схемы и булевые выражения HCL	320
4.2.3. Комбинационные схемы на уровне слова и целочисленные выражения в HCL	322
4.2.4. Принадлежность множеству	327
4.2.5. Память и синхронизация	328
4.3. Последовательные реализации Y86	330
4.3.1. Поэтапная организация процессора	330
Выборка	330
Декодирование	331
Выполнение	331
Память	331
Обратная запись	331
Обновление РС	331
Выполнение команды <i>rmmovl</i>	337
Выполнение команды <i>pushl</i>	338
Выполнение команды <i>je</i>	340
Выполнение команды <i>ref</i>	342
4.3.2. Структура аппаратных средств SEQ	343
Выборка	343
Декодирование	343
Выполнение	343
Память	344
Обратная запись	344
4.3.3. Синхронизация SEQ	347

4.3.4. Реализация этапов SEQ	351
Этап выборки	352
Этапы декодирования и обратной записи.....	353
Этап выполнения	355
Этап памяти.....	356
Этап обновления РС	358
Исследование SEQ	358
4.3.5. Реконфигурация этапов вычисления.....	359
4.4. Общие принципы конвейерной обработки	362
4.4.1. Конвейеры вычислений.....	363
4.4.2. Подробное описание конвейерной операции	364
4.4.3. Ограничения конвейерной обработки	367
Неравномерное разбиение	367
Уменьшение возвратов глубокой конвейерной обработки.....	368
4.4.4. Конвейеризация системы с обратной связью	369
4.5. Конвейерные реализации Y86.....	371
4.5.1. Вставка конвейерных регистров	372
4.5.2. Реконфигурация и смена меток сигналов	375
4.5.3. Прогнозирование следующего значения РС	376
4.5.4. Риски конвейерной обработки	378
Перечисление классов рисков по данным	382
Как избежать рисков по данным с помощью останова.....	383
Как избежать рисков по данным с помощью продвижения.....	385
Риски по данным load/use.....	391
4.5.8. Реализации этапов PIPE.....	393
Выбор РС и этап выборки	393
Этапы декодирования и обратной записи.....	395
Этап выполнения	399
Этап памяти.....	399
4.5.9. Управляющая логика конвейера.....	400
Желательная обработка особых контрольных случаев.....	400
Выявление особых условий управления	403
Механизмы управления конвейером	404
Комбинации управляющих условий	406
Реализация управляющей логики.....	409
4.5.10. Анализ эффективности	410
4.5.11. Незаконченная работа	412
Обработка исключительных ситуаций	412
Многократные команды.....	415
Сопряжение с системой памяти.....	415
4.6. Резюме	418
4.6.1. Имитаторы Y86	419
Библиографические примечания	419
Задачи для домашнего решения.....	420
Решение упражнений.....	425
Глава 5. Оптимизация производительности программ	437
5.1. Возможности и ограничения оптимизирующего компилятора.....	439
5.2. Выражение производительности программы	442

5.3. Пример программы	445
5.4. Устранение недостаточности циклов	448
5.5. Сокращение обращений к процедурам	453
5.6. Устранение ненужных ссылок на ячейки памяти	455
5.7. Общее описание современных процессоров	458
5.7.1. Общее функционирование	458
5.7.2. Производительность функционального устройства	463
5.7.3. Более пристальный взгляд на работу процессора	464
Перевод команд в операции	464
Обработка операций устройством выполнения	466
Планирование операций с неограниченными ресурсами	467
Планирование операций с ограничениями ресурсов	469
Выводы о производительности <i>combine4</i>	472
5.8. Снижение непроизводительных издержек циклов	473
5.9. Преобразование в код указателя	477
5.10. Повышение параллелизма	480
5.10.1. Разбиение циклов	480
5.10.2. Вытеснение регистров	485
5.10.3. Ограничения параллелизма	487
5.11. Сводка результатов оптимизации объединяющего кода	489
5.11.1. Аномалия производительности операций с числами с плавающей точкой	489
5.11.2. Изменение платформ	491
5.12. Прогнозирование ветвей и штрафы за некорректное прогнозирование	492
5.13. Понятие производительности памяти	496
5.13.1. Задержка операций загрузки	496
5.13.2. Задержка операций сохранения	498
5.14. Жизнь в реальном мире: методы повышения производительности	504
5.15. Выявление и устранение критических элементов производительности	504
5.15.1. Профилирование программ	505
5.15.2. Использование профайлеров при управлении оптимизацией	507
5.15.3. Закон Эмдала	510
5.16. Резюме	512
Библиографические примечания	513
Задачи для домашнего решения	513
Решение упражнений	518
Глава 6. Иерархия памяти	523
6.1. Технологии сохранения информации	524
6.1.1. Память с произвольной выборкой	524
Статическая RAM	525
Динамическая RAM	525
Стандартные DRAM	526
Модули памяти	528
Расширенные DRAM	529
Энергонезависимая память	530
Доступ к основной памяти	531

6.1.2. Дисковый накопитель	534
Геометрия диска	534
Емкость диска	535
Функционирование диска	536
Логические блоки диска	538
Оценка дисков	539
6.1.3. Направления развития технологий записывающих устройств	544
6.2. Локальность	546
6.2.1. Локальность обращений к данным программы	547
6.2.2. Локальность выборки команд	550
6.2.3. Резюме локальности	550
6.3. Иерархия памяти	552
6.3.1. Кэширование в иерархии памяти	553
Результативные обращения	555
Промахи кэша	555
Виды промахов кэша	555
Управление кэшем	556
6.3.2. Резюме концепции иерархии памяти	557
6.4. Виды кэш-памяти	558
6.4.1. Родовая организация кэш-памяти	559
6.4.2. Кэш прямого отображения	561
Выбор множества в кэше прямого отображения	562
Сопоставление строк в кэше прямого отображения	562
Извлечение слова в кэшах прямого отображения	563
Вытеснение строк при промахах обращения в кэш прямого отображения	563
Окончательная сборка: кэш прямого обращения в действии	563
Конфликтные промахи в кэше прямого отображения	566
6.4.3. Ассоциативные множества кэши	570
Выбор множества в ассоциативных множеству кэшах	570
Сопоставление строк и извлечение слова в ассоциативных множеству кэшах	570
Вытеснение строк при промахах в ассоциативных множеству кэшах	572
6.4.4. Полностью ассоциативные кэши	572
Выбор множества в полностью ассоциативных множеству кэшах	572
Сопоставление строк и извлечение слова в полностью ассоциативных множеству кэшах	573
6.4.5. Работа с операциями записи	576
6.4.6. Кэши команд и унифицированные кэши	577
6.4.7. Влияние параметров кэша на производительность	578
Влияние размера кэша	579
Влияние размера блока	579
Влияние ассоциативности	579
Влияние стратегии записи	580
6.5. Написание кодов, дружественных кэш-памяти	580
6.6. Влияние кэша на производительность программ	586
6.6.1. Гора памяти	586
6.6.2. Реконфигурация циклов для повышения пространственной локальности	592
6.6.3. Использование блокирования для повышения временной локальности	596

6.7. Использование локальности в программах	599
6.8. Резюме	600
Библиографические примечания	601
Задачи для домашнего решения	602
Решение упражнений.....	608
ЧАСТЬ II. ИСПОЛНЕНИЕ ПРОГРАММ В СИСТЕМЕ.....	617
Глава 7. Редактирование связей.....	619
7.1. Драйверы компилятора	621
7.2. Статическое связывание	623
7.3. Объектные файлы.....	624
7.4. Переместимые объектные файлы	624
7.5. Идентификаторы и таблицы имен	626
7.6. Разрешение ссылок	630
7.6.1. Многократно определенные глобальные ссылки	631
7.6.2. Связывание со статическими библиотеками	635
7.6.3. Использование статических библиотек	638
7.7. Перемещение	640
7.7.1. Входы перемещения	640
7.7.2. Перемещение ссылок на имя	641
Перемещение ссылок с относительной адресацией по РС	642
Перемещение абсолютных ссылок.....	643
7.8. Исполняемые объектные файлы.....	645
7.9. Загрузка исполняемых объектных файлов.....	647
7.10. Динамическое связывание с разделяемыми библиотеками.....	649
7.11. Загрузка и связывание с разделяемыми библиотеками из приложений	652
7.12. Переместимый программный код	655
7.12.1. Ссылки на данные РС	656
7.12.2. Вызовы функций РС	657
7.13. Средства управления объектными файлами	659
7.14. Резюме	659
Библиографические замечания	660
Задачи для домашнего решения	661
Решение упражнений	667
Глава 8. Управление исключениями	671
8.1. Исключения	673
8.1.1. Обработка исключений	674
8.1.2. Классы исключений	676
Аппаратные прерывания	677
Системные прерывания	677
Сбои	678
Аварийные завершения	679
8.1.3. Исключения в процессорах Intel	679
8.2. Процессы	681
8.2.1. Логический поток управления	681
8.2.2. Закрытое адресное пространство	683

8.2.3. Пользовательский и привилегированный режимы	684
8.2.4. Контекстные переключатели	685
8.3. Системные вызовы и обработка ошибок.....	687
8.4. Управление процессами	688
8.4.1. Получение ID процесса	688
8.4.2. Порождение и завершение процессов	689
8.4.3. Снятие дочерних процессов	694
Изменение поведения по умолчанию	695
Проверка статуса выхода снятого дочернего процесса	696
Состояния ошибки	696
Константы, связанные с функциями Unix	696
Примеры	697
8.4.4. Перевод процессов в состояние "сна"	699
8.4.5. Загрузка и запуск программ на исполнение	700
8.4.6. Использование функций для запуска программ	703
8.5. Сигналы	706
8.5.1. Терминология, связанная с сигналами	709
8.5.2. Посылка сигналов	710
Группы процессов	710
Посылка сигналов с клавиатуры	710
Посылка сигналов с помощью функций	711
8.5.3. Получение сигналов	714
8.5.4. Некоторые вопросы обработки сигналов	716
8.5.5. Переносимая обработка сигналов	722
8.5.6. Явная блокировка сигналов	724
8.6. Нелокальные передачи управления	727
8.7. Организация управления процессами	730
8.8. Резюме	731
Библиографические замечания	731
Задачи для домашнего решения	732
Решение упражнений	738
Глава 9. Измерение времени исполнения программы	741
9.1. Течение времени в компьютерной системе	743
9.1.1. Планирование процессов и прерывания от таймера	744
9.1.2. Течение времени с точки зрения прикладной программы	745
9.2. Измерение времени подсчетом количества интервалов	748
9.2.1. Роль операционной системы	748
9.2.2. Считывание данных от таймеров	749
9.2.3. Точность таймеров процесса	750
9.3. Счетчики тактов	752
9.3.1. Счетчики тактов в IA32	753
9.4. Измерение времени исполнения программы с помощью счетчиков тактов	755
9.4.1. Эффект переключения контекста	755
9.4.2. Кэширование и другие эффекты	757
9.4.3. Схема измерения K-best	761
Экспериментальная оценка	762
Установка значения K	764
Компенсация обработки прерываний от таймера	767

Вычисление на других машинах.....	768
Результаты наблюдений.....	771
9.5. Измерение по часам реального времени	771
9.6. Протокол эксперимента.....	774
9.7. Взгляд в будущее	775
9.8. Реализация схемы измерения K-best.....	776
9.9. Задания по пройденному материалу	776
9.10. Резюме	777
Библиографические замечания	778
Задачи для домашнего решения.....	778
Решение упражнений	779
Глава 10. Виртуальная память	783
10.1. Физическая и виртуальная адресация	785
10.2. Пространство адресов	786
10.3. Инструментальное средство кэширования	787
10.3.1. Организация кэш в DRAM	788
10.3.2. Таблицы страниц	789
10.3.3. Страница находится в DRAM	791
10.3.4. Обращение к отсутствующей странице	791
10.3.5. Размещение страниц	793
10.3.6. Снова о компактности размещения	793
10.4. Манипулирование памятью.....	794
10.4.1. Упрощение компоновки	795
10.4.2. Упрощение совместного использования.....	796
10.4.3. Упрощение выделения памяти	797
10.4.4. Упрощение загрузки.....	797
10.5. VM как средство защиты памяти.....	797
10.6. Преобразование адресов	799
10.6.1. Интегрирование кэш в виртуальную память	802
10.6.2. Ускорение трансляции адресов с помощью TLB	803
10.6.3. Многоуровневые таблицы страниц.....	805
10.6.4. Непрерывная трансляция адреса.....	807
10.7. Система памяти Pentium/Linux.....	811
10.7.1. Трансляция адресов системой Pentium	812
Таблицы страниц в Pentium	814
Трансляция таблицы страниц в системе Pentium	816
Преобразование TLB в системе Pentium	816
10.7.2. Система виртуальной памяти Linux	817
Области виртуальной памяти в Linux	819
Обработка исключительной ситуации, возникающей при обращении к отсутствующей странице в системе Linux	820
10.8. Отображение в памяти	821
10.8.1. Разделяемые повторно посещаемые объекты.....	822
10.8.2. Еще раз о функции <i>fork</i>	825
10.8.3. Еще раз о функции <i>execve</i>	825
10.8.4. Отображение в памяти на уровне пользователя с помощью функции <i>mmap</i>	826

10.9. Динамическое распределение памяти	828
10.9.1. Функции <i>malloc</i> и <i>free</i>	830
10.9.2. Что дает динамическое распределение памяти	832
10.9.3. Назначение программы распределения памяти и требования к ней	833
10.9.4. Фрагментация	835
10.9.5. Вопросы реализации	836
10.9.6. Неявные списки свободных блоков	837
10.9.7. Размещение распределенных блоков	839
10.9.8. Разбиение свободных блоков	840
10.9.9. Получение дополнительной динамической памяти	841
10.9.10. Объединение свободных блоков	841
10.9.11. Объединение с использованием граничных тегов	842
10.9.12. Реализация простой программы распределения памяти	845
Разработка программы выделения памяти	846
Основные константы и макроопределения для управления	
списком свободных блоков	848
Создание начального списка свободных блоков	849
Освобождение и объединение блоков	851
Выделение блоков	852
10.9.13. Явные списки свободных блоков	854
10.9.14. Раздельные свободные списки	855
Простое разделение памяти	856
Разделение с учетом размера	857
Метод близнецов	858
10.10. Сборка мусора	859
10.10.1. Основные принципы функционирования программ сборки мусора	860
10.10.2. Программы сборки мусора, реализующие алгоритм <i>Mark&Sweep</i>	861
10.10.3. Консервативный алгоритм <i>Mark&Sweep</i> для программ на C	863
10.11. Часто встречающиеся ошибки	864
10.11.1. Разыменование плохих указателей	865
10.11.2. Чтение неинициализированной области памяти	865
10.11.3. Переполнение буфера стека	866
10.11.4. Предположение о размере указателей и объектов	866
10.11.5. Ошибки занижения или завышения на единицу	867
10.11.6. Ссылка на указатель вместо объекта	868
10.11.7. Неправильное понимание арифметических операций	
над указателями	868
10.11.8. Ссылки на несуществующие переменные	869
10.11.9. Ссылка на данные в свободных блоках динамической памяти	869
10.11.10. Представление об утечках в памяти	870
10.12. Сводка некоторых ключевых понятий, связанных	
с виртуальной памятью	871
10.13. Резюме	871
Библиографические замечания	872
Задачи для домашнего решения	873
Решение упражнений	878

ЧАСТЬ III. ВЗАИМОДЕЙСТВИЕ И ВЗАИМОСВЯЗИ ПРОГРАММ	885
Глава 11. Системный уровень ввода-вывода.....	887
11.1. Ввод-вывод Unix.....	888
11.2. Открытие и закрытие файлов.....	889
11.3. Считывание и запись файлов.....	891
11.4. Устойчивое считывание и запись с помощью пакета RIO	893
11.4.1. Небуферизованные функции ввода и вывода RIO.....	893
11.4.2. Буферные функции ввода RIO.....	895
11.5. Считывание метаданных файла	899
11.6. Совместное использование файлов	901
11.7. Переадресация данных ввода-вывода.....	904
11.8. Стандартный ввод-вывод	905
11.9. Окончательная сборка: какие функции ввода-вывода стоит использовать?	906
11.10. Резюме	908
Библиографические примечания	909
Задачи для домашнего решения	909
Решение упражнений	910
Глава 12. Сетевое программирование.....	913
12.1. Программная модель клиент-сервер	913
12.2. Компьютерные сети	914
12.3. Глобальная сеть Internet.....	919
12.3.1. IP-адреса	921
12.3.2. Имена доменов в Internet	923
12.3.3. Internet-соединения	927
12.4. Интерфейс сокетов.....	929
12.4.1. Адресные структуры сокетов	930
12.4.2. Функция <i>socket</i>	931
12.4.3. Функция <i>connect</i>	931
12.4.4. Функция <i>open_clientfd</i>	931
12.4.5. Функция <i>bind</i>	932
12.4.6. Функция <i>listen</i>	933
12.4.7. Функция <i>listenfd</i>	933
12.4.8. Функция <i>accept</i>	934
12.4.9. Примеры эхо-клиента и эхо-сервера	936
12.5. Web-серверы	939
12.5.1. Основные сведения о Web	939
12.5.2. Содержимое Web	940
12.5.3. Транзакции HTTP	941
Запросы HTTP	942
Ответы HTTP	943
12.5.4. Обслуживание динамического содержимого	944
Как клиент передает аргументы программы серверу	945
Как сервер передает аргументы порожденному процессу	945
Как сервер передает порожденному процессу другую информацию.....	945
Куда порожденный процесс отправляет свои выходные данные	946

12.6. Разработка небольшого Web-сервера TINY.....	947
Программа <i>main</i> сервера TINY.....	948
Функция <i>doit</i>	949
Функция <i>clienterror</i>	950
Функция <i>read_requesthdrs</i>	951
Функция <i>parse_uri</i>	952
Функция <i>serve_static</i>	953
Функция <i>serve_dynamic</i>	954
12.7. Резюме	956
Библиографические заметки.....	957
Задачи для домашнего решения.....	957
Решение упражнений	959
Глава 13. Параллельное программирование.....	961
13.1. Параллельное программирование с процессами.....	963
13.1.1. Параллельный сервер, основанный на процессах	965
13.1.2. За и против использования процессов.....	966
13.2. Параллельное программирование с мультиплексированием ввода-вывода	967
13.2.1. Параллельный событийно-управляемый сервер на базе мультиплексирования ввода-вывода.....	971
13.2.2. За и против мультиплексирования ввода-вывода	975
13.3. Параллельное программирование с потоками	976
13.3.1. Модель выполнения потока.....	977
13.3.2. Потоки интерфейса операционной системы Posix.....	978
13.3.3. Создание потоков.....	979
13.3.4. Завершение выполнения потоков	979
13.3.5. Отделение потоков	980
13.3.6. Инициализация потоков	981
13.3.7. Параллельный сервер, основанный на потоках	981
13.4. Совместно используемые переменные в поточных программах	983
13.4.1. Модель памяти для потоков	984
13.4.2. Отображение переменных в память	985
13.4.3. Совместно используемые переменные	986
13.5. Синхронизация потоков с семафорами	986
Упорядочение команд для первой итерации цикла	988
13.5.1. Графы продвижения.....	990
13.5.2. Использование семафоров для доступа к совместно используемым переменным.....	993
13.5.3. Семафоры Posix	995
13.5.4. Использование семафоров для планирования совместно используемых ресурсов.....	996
13.6. Параллельный сервер на базе предварительной организации поточной обработки	998
13.7. Другие вопросы параллелизма	1002
13.7.1. Безопасность потоков	1002
Класс 1: функции, не защищающие совместно используемые переменные	1002
Класс 2: функции, поддерживающие состояние при многократных вызовах	1002

Класс 3: функции, возвращающие указатель статической переменной.....	1003
Класс 4: функции, вызывающие небезопасные по потокам функции	1004
13.7.2. Реентерабельность	1004
13.7.3. Использование существующих библиотечных функций в поточных программах	1006
13.7.4. Гонки.....	1007
13.7.5. Взаимоблокировка (туниковые ситуации)	1009
13.8. Резюме	1012
Библиографические примечания	1013
Задачи для домашнего решения.....	1013
Решение упражнений	1017
ПРИЛОЖЕНИЯ	1023
Приложение 1. Описание управляющей логики процессоров с помощью HCL.....	1025
Справочное руководство HCL	1025
Обявление сигналов	1026
Текст в кавычках.....	1026
Выражения и блоки	1026
Пример HCL	1028
Описание SEQ	1030
Описание SEQ+.....	1034
Конвейер	1039
Приложение 2. Обработка ошибок.....	1047
Обработка ошибок в системе Unix.....	1048
Обработка ошибок в стиле Unix	1048
Обработка ошибок в стиле Posix	1048
Обработка ошибок в стиле DNS	1048
Резюме сообщающих об ошибках функций.....	1049
Интерфейсные программы обработки ошибок.....	1050
Программы обработки ошибок стиля Unix	1050
Программы обработки ошибок стиля Posix	1051
Программы обработки ошибок стиля DNS.....	1051
Заголовочный файл csapp.h	1051
Исходный файл csapp.c.....	1055
Библиография.....	1073
Предметный указатель.....	1079

*Нашим женам — Дженис и Хелен
и нашим детям —
Джейкобу, Клэр, Элизабет,
Майклу, Джозефу, Джону и Николасу*

Предисловие

Данная книга предназначена для программистов, желающих повысить свой профессиональный уровень изучением того, что происходит "под кожухом системного блока" компьютерной системы.

Целью авторов является попытка разъяснения устойчивых концепций, лежащих в основе всех компьютерных систем, а также демонстрация конкретных видов влияния этих идей на корректность, производительность и полезные свойства программного приложения. В отличие от прочих книг, посвященных компьютерным системам и написанных преимущественно для создателей последних, предлагаемый материал предназначен исключительно для программистов и рассматривается с их же позиций.

Изучение и доскональное понимание изложенных в книге концепций позволит читателю со временем превратиться в редкий тип "мощного программиста", знающего самую суть происходящего и способного решить любую проблему. При этом будет заложена основа для изучения таких специфических тем, как работа с компиляторами, архитектура компьютерных систем, операционные системы и использование сетей.

Что нужно знать перед прочтением

Приведенные в книге примеры основаны на процессорах, совместимых с Intel (официальное название — IA32, в быту — x86), выполняющих программы С в операционной системе Unix или совместимых с ней (например, Linux). Для упрощения изложения, в дальнейшем будет использоваться термин Unix для обозначения и таких систем, как Solaris и Linux. В тексте содержатся многочисленные примеры программирования, скомпилированные и выполняемые в системах Linux. Авторы предполагают, что читатели имеют доступ к компьютерам подобного рода.

Если на компьютере установлена система Microsoft Windows, то можно выбрать любой из следующих вариантов. Можно скачать копию Linux (www.linux.org или www.redhat.com) и установить ее как вторую операционную систему, чтобы машина могла работать с любой из двух систем. Либо при установке инструментария Cygwin (www.cygwin.com) в Windows можно получить оболочку, подобную Unix. Однако в Cygwin доступны не все функции Linux.

Также предполагается, что читатель знаком с C или C++. Если весь опыт программиста ограничивается работой с Java, переход потребует от него больше усилий, но авторы окажут всю необходимую помощь. Java и C имеют общий синтаксис и управляющие операторы. Однако в C существуют аспекты (особенно указатели, распределение явной динамической памяти и форматируемый ввод-вывод), которых нет в Java. К счастью, C — не очень сложный язык, он прекрасно и исчерпывающе описан в классическом тексте Брайана Кернигана и Денниса Риччи [40]. Вне зависимости от "подкованности" читателя в области программирования, рекомендуется рассматривать эту книгу в качестве важной части библиотеки.

В начальных главах книги рассматривается взаимодействие между программами, написанными на C, и их аналогами, написанными на машинном языке. Все примеры, написанные на машинном языке, созданы с помощью компилятора GNU GCC на базе процессора Intel IA32. Наличия какого бы то ни было опыта работы с аппаратными средствами, машинными языками или программирования в ассемблере не предполагается.

Совет, относящийся к языку программирования C

В помощь читателям, имеющим слабое представление о программировании на языке C (или не имеющим его вообще), авторами предлагаются примечания, подобные данному, для подчеркивания функций, особенно важных для C. Предполагается, что читатели знакомы с C++ или Java.

Как читать книгу

Изучение с точки зрения программиста того, как в принципе работает компьютерная система, — занятие очень увлекательное в основном потому, что оно происходит оперативно. Как только узнается что-то новое, это можно тут же проверить и получить результат, что называется, из первых рук. На самом деле, авторы полагают, что единственным способом познания систем является их создание: либо путем решения конкретных упражнений, либо написанием и выполнением программ в реально существующих системах.

Система является предметом изучения всей книги. При представлении какой-либо новой концепции в тексте дается иллюстрация одной или несколькими практическими задачами, которые нужно сразу же постараться решить для проверки правильности понимания изложенного. Решение упражнений находится в конце каждой главы. По мере ознакомления с материалом, пытайтесь самостоятельно решать все практические задачи, после чего проверяйте правильность выбранного пути. В конце каждой главы также представлены домашние задания различной степени сложности. Для каждой домашней задачи представлен рейтинг необходимых умственных и прочих затрат:

- ♦ — для решения потребуется всего несколько минут. Требуется минимальный объем программирования (либо его не требуется вообще).
- ♦♦ — для решения потребуется порядка 20 минут. Часто требуется написание и тестирование кодов. Многие из них получены из задач, приведенных в примерах.

♦♦♦ — требует значительных усилий; по времени может занимать до 2 часов. Как правило, включает в себя написание и тестирование большей части кода.

♦♦♦♦ — лабораторная работа, на выполнение которой требуется до 10 часов.

Каждый пример кода в тексте отформатирован автоматически (работы вручную не было) из программы C, скомпилированной с помощью версии GCC 2.95.3, и протестирован на системе Linux с ядром 2.2.16. Весь исходный код доступен на сайте csapp.cs.cmu.edu.

Примечания и реплики

Примечания преследуют несколько целей. Одни представляют собой небольшие исторические экскурсы. Например, откуда взяли свое начало C, Linux и Internet? Другие реплики предназначены для разъяснения каких-либо понятий. Например, чем различаются кэш, ряд и блок? Реплики третьего вида описывают примеры "из жизни". Например, как ошибка с плавающей точкой уничтожила французскую ракету, или что представляет собой геометрия настоящего дисковода IBM. И наконец, некоторые реплики — это всего лишь забавные комментарии.

Происхождение книги

Книга родилась из вводного курса, разработанного в университете Карнеги-Меллона (УКМ) осенью 1998 г. [7]. С тех пор курс читался каждый семестр; слушателями были до 150 студентов факультета вычислительной техники. Данный курс стал предпосылкой большинства курсов по компьютерным системам высшего уровня в университете Карнеги-Меллона.

Идеей было познакомить студентов с компьютерами несколько иначе, нежели это происходит обычным путем. Мало кто из студентов смог бы самостоятельно построить компьютерную систему. С другой стороны, от большинства обучающихся и даже инженеров по вычислительной технике требуется повседневное использование компьютеров в программировании. Поэтому принципом авторов данной книги стало начало обучения работе с системами с точки зрения программиста, с использованием следующего своеобразного фильтра: тема будет освещаться только в том случае, если она имеет отношение к производительности, корректности, либо к полезным свойствам C-программ пользовательского уровня.

К примеру, исключены темы, связанные с сумматорами аппаратных средств и проектированием шин. В свою очередь, в книгу включены темы, посвященные машинному языку, однако, вместо подробного рассмотрения ассемблерного языка, упор делается на построение языком C указателей, циклов, вызовов процедур и возвратов, а операторы выбора переводятся компилятором. Более того, здесь учитывается более широкий и реалистичный взгляд на системы как аппаратных, так и программных средств, охватываются такие темы, как редактирование связей, загрузка, процессы, сигналы, оптимизация эффективности, измерения, ввод-вывод, а также сетевое и параллельное программирование.

Данный подход позволил сделать курс практическим, наглядным и на редкость интересным для студентов. Ответная реакция со стороны последних и коллег по факуль-

тету была незамедлительной и позитивной, и авторы книги поняли, что преподаватели и из других учебных заведений смогут воспользоваться их наработками. Это и явилось предпосылкой появления данной книги, написанной в течение двух лет по лекционным конспектам курса.

Нумерология ВКС

Нумерология курса ВКС слегка необычна. Где-то в середине первого семестра авторы поняли, что присвоенный курсу номер (15-213) также является почтовым индексом университета Карнеги-Меллона; отсюда девиз: 15-213 — курс, который зажигает УКМ! По случайному совпадению, первоначальная версия рукописи была напечатана 13 февраля 2001 года (2/13/01). При презентации курса на образовательной конференции SIGCSE обсуждение было назначено в 213 аудитории. И окончательный вариант книги имеет 13 глав. Как хорошо, что мы не суеверны!

Обзор книги

Книга состоит из 13 глав, разработанных с целью всеобъемлющего охвата основных принципов компьютерных систем:

Глава 1 "Экскурс в компьютерные системы". В первой главе описываются основные идеи и темы, относящиеся к компьютерным системам, посредством исследования жизненного цикла простой программы "hello, world".

Глава 2 "Представление информации и работа с ней". Здесь описывается компьютерная арифметика с упором на свойства представлений числа без знака и числа в дополнительном двоичном коде, которые имеют значение для программистов. В данной главе рассматривается представление чисел и, следовательно, диапазон значений, которые можно запрограммировать для отдельно взятого размера слова. Авторы обсуждают влияние преобразований типов чисел со знаком и без знака, математические свойства арифметических операций. Для читателей становится открытием то, что сумма (дополнительный код) или произведение двух положительных чисел могут быть отрицательными. С другой стороны, арифметика дополнительного кода удовлетворяет свойствам в кольце, и поэтому компилятор может трансформировать умножение посредством константы в последовательность сдвигов и сложений. Для иллюстрации принципов и применения булевой алгебры авторы используют разрядные операции С. Формат IEEE с плавающей точкой описывается в плане представления значений и математических свойств операций с плавающей точкой.

Абсолютное понимание компьютерной арифметики принципиально для написания работающих программ. Арифметическое переполнение является обычным источником ошибок программирования, однако мало в какой книге можно найти описание свойств компьютерной арифметики, сделанное с точки зрения самого программиста.

Глава 3 "Представление программ на машинном уровне". Авторы учат прочтению ассемблерного языка IA32, созданного компилятором С. Здесь представлены основные шаблоны инструкций, созданные для различных управляющих структур, таких как условные операторы, циклы и операторы выбора. Здесь также рассматривается

реализация процедур, включая размещение в стеке, условные обозначения использования реестров и передачу параметров. В главе рассматриваются различные структуры данных, например структуры, объединения и массивы, а также доступ к ним. Изучение положений данной главы помогает повысить профессиональный уровень, потому что возникает понимание представления на компьютере создаваемых программ.

Глава 4 "Архитектура процессора". В данной главе описываются комбинаторные и последовательные логические элементы, после чего демонстрируется то, как эти элементы можно объединить в информационный канал, выполняющий упрощенный набор инструкций IA32, называемый "Y86". Глава начинается с описания проекта информационного канала, который впоследствии расширяется до пятиступенчатого конвейерного проекта. Управляющая логика для проектов процессора описывается в главе с использованием простого языка описания аппаратных средств — HCL. Проекты аппаратного обеспечения, написанные на HCL, можно компилировать и объединять в симуляторы графического процессора.

Глава 5 "Оптимизация производительности программ". Здесь авторами представлены несколько методик повышения производительности кода. Все начинается с трансформаций машинно-независимой программы. Затем осуществляется переход к трансформациям, эффективность которых зависит от характеристик целевой машины и компилятора. Представлена простая операционная модель действий процессора.

Глава 6 "Иерархия памяти". Система памяти для программистов программных приложений является одной из самых "видимых" частей компьютерной системы. До сих пор читатели полагались на концептуальную модель системы памяти как на векторный массив с универсальными значениями времени доступа. На практике система памяти представляет собой иерархию запоминающих устройств разной емкости, цены и быстродействия. В главе рассматриваются разные типы памяти типа ROM и RAM, а также геометрические параметры и устройство существующих современных дисковых накопителей, организация этих запоминающих устройств в иерархию. Авторы показывают возможность иерархии посредством локальности ссылок. Даные идеи конкретизируются представлением уникального взгляда на систему памяти, как на "гору памяти" со "скалами" местной локализации и "склонами" пространственной локализации, как можно повысить производительность программных приложений путем усовершенствования их временной и пространственной локализации.

Глава 7 "Редактирование связей". В данной главе описывается динамическое и статическое связывание, включая понятия переместимых и исполняемых объектных файлов, символьного представления, перераспределения, статических библиотек, библиотек объединенного доступа и непозиционных кодов. Как правило, в большинстве работ по компьютерным системам связывание не рассматривается, но авторы включили его в данную книгу по нескольким причинам. Во-первых, наиболее часто встречающиеся и общие ошибки, с которыми сталкиваются студенты во время процесса связывания, особенно характерны для крупных программных пакетов. Во-вторых, объектные файлы, создаваемые компоновщиками, связаны с такими понятиями, как загрузка, виртуальная память и распределение памяти.

Глава 8 "Управление исключениями". В данной главе однопрограммная модель разделена посредством представления общей концепции исключительной управляющей логики: от исключений аппаратных средств нижнего уровня и прерываний до контекстных переключателей между параллельными процессами, внезапных изменений в управляющей логике, вызванных передачей сигналов Unix, и нелокальных переходов в C, разрывающих стройную структуру стека.

Это — та часть книги, в которой представляются фундаментальные понятия общего процесса. При этом показывается, как программисты могут использовать множественные процессы посредством системных вызовов Unix.

Глава 9 "Измерение времени исполнения программы". В данной главе рассказывается о том, как компьютер считает время (датчики временных интервалов, счетчики циклов и системные тактовые генераторы), об источниках ошибок, когда эти значения времени используются для измерения времени, а также о том, как использовать эти знания для получения точных значений. Насколько можно судить, это — уникальный материал, никогда ранее не представлявшийся в каком бы то ни было систематизированном виде. Эта тема включена потому, что она требует понимания ассемблерного языка, процессов и кэш.

Глава 10 "Виртуальная память". Авторское представление системы виртуальной памяти имеет целью дать некоторое понимание принципов ее работы и характеристик. Мы хотим научить понимать, как разные процессы, происходящие одновременно, могут использовать один и тот же диапазон адресов, совместно использовать одни страницы, но иметь индивидуальные копии других. Здесь также описываются вопросы, связанные с управлением виртуальной памятью и манипуляциями с ней. В частности, внимание уделяется работе с распределителями памяти, такими как операции Unix malloc и free. Изложение данного материала преследует несколько целей. Прежде всего, оно подкрепляет концепцию о том, что пространство виртуальной памяти представляет собой всего лишь массив байтов, который программа может разделить на различные блоки памяти. Материал помогает понять влияние программ, содержащих ошибки обращения к памяти, такие как утечки и неправильные ссылки на указатели. И наконец, многие программисты пишут свои собственные распределители памяти, оптимизированные под требования и характеристики каждого конкретного приложения.

Глава 11 "Системный уровень ввода-вывода". В данной главе рассматриваются основные положения ввода-вывода в системе Unix, такие как файлы и дескрипторы. Авторы описывают совместное использование файлов, принципы работы переадресации ввода-вывода и доступ к файлам метаданных. Здесь же разработан солидный буферизованный пакет ввода-вывода, корректно обрабатывающий короткие единицы счета. В главе описывается стандартная библиотека ввода-вывода и ее связь с вводом-выводом системы Unix с упором на ограничения стандартного ввода-вывода, делающие его непригодным для сетевого программирования. Вообще говоря, темы, охваченные в этой главе, являются компоновочными блоками следующих двух глав, посвященных сетевому и параллельному программированию.

Глава 12 "Сетевое программирование". Сети являются как бы устройствами ввода-вывода для программирования, объединяющими многие из понятий, изученных ра-

нее: процессы, сигналы, упорядочение байтов, распределение памяти и распределение динамических запоминающих устройств. Данная глава — всего лишь тонкий срез глобального предмета сетевого программирования, подводящий студентов к написанию Web-сервера. В главе описывается модель клиент-сервер, являющаяся основой всех сетевых приложений. Авторы представляют сеть Internet с точки зрения программиста и демонстрируют способы написания клиента и сервера Internet, используя интерфейс сокета. И наконец, в главе представлен HTTP и разрабатывается простой итерационный интерфейс.

Глава 13 "Параллельное программирование". В данной главе студентам представлены принципы параллельного программирования с использованием проекта Internet-сервера в качестве рабочего примера. Авторы сравнивают и противопоставляют три базовых механизма, используемых при написании параллельных программ: процессы, уплотнение ввода-вывода и потоки — и показывают возможность их использования при создании параллельных Internet-серверов. Здесь же описаны основные принципы синхронизации с использованием семафорных операций *P* и *V*, безопасности потоков и реентерабельности и взаимоблокировки.

Благодарности

Мы безмерно благодарны всем друзьям и коллегам за их вдумчивую критику и сердечную поддержку. Отдельное спасибо — студентам курса 15-213, чья энергия и энтузиазм "не давали нам засохнуть". Пакет malloc был любезно предоставлен Ником Картером и Винни Фьюриа.

Гай Блеллок, Грек Кесден, Брюс Мэггс и Тодд Маори являлись преподавателями курса на протяжении нескольких семестров, и их неоценимая поддержка помогала нам постоянно совершенствоватьляемый материал. Духовным руководством и постоянным содействием во время работы мы обязаны Хербу Дерби. Аллан Фишер, Гарт Гибсон, Томас Гросс, Сэйша, Питер Стниксте и Хью Жанг дали нам толчок для начала работы над материалами книги. Предложение Гарта "запустило маховик", было поддержано и тщательно проработано командой Алана Фишера. Марк Стелик и Питер Ли оказали неоценимую поддержку в организации данного материала в соответствии с учебным планом. Грэг Кесден предоставил полезную обратную связь, описывающую влияние ICS на курс OS. Грэг Гэнгер и Джири Шиндлер любезно предоставили некоторые характеристики дисководов и ответили на наши вопросы о существующих в настоящее время дисках. Том Стрикер показал нам "гору памяти". Джеймс Хоу выдал полезные идеи о том, как следует изложить материал об архитектуре процессора.

Группа студентов — Халил Амири, Анжела Демке Браун, Крис Колохан, Джейсон Кроуфорд, Питер Динда, Хулио Лопез, Брюс Лоукэмп, Джейфф Пирс, Санджей Рао, Баладжи Сарлешкар, Блейк Шоль, Санжи Сешиа, Грэг Стефан, Тианкай Ту, Кип Уокер и Йинглайн Цзе — помогали в разработке содержания курса. В частности, Крис Колохан выработал развлекательный (и смешной) стиль представления материала, используемый до сих пор, а также разработал легендарную "двоичную бомбу", оказавшуюся превосходным инструментом обучения работе с машинными кодами и концепциям отладки.

Крис Бауэр, Аллан Кокс, Питер Динда, Сандия Дауркадис, Джон Грейнер, Брюс Джейкоб, Барри Джонсон, Дон Хеллер, Брюс Лоукамп, Грэг Моррисетт, Брайан Ноубл, Бобби Отмер, Билл Пью, Майкл Скотт, Марк Сматермен, Грэг Стефан и Боб Уайер потратили кучу времени на ознакомление с черновиками книги. Отдельное спасибо Элу Дэвису (Университет Юты), Питеру Динда (Северо-Западный университет), Джону Грейнеру (Университет Райс), Вей Су (Университет Миннесоты), Брюсу Лоукампу (Уильям и Мэри), Бобби Отмеру (Университет Миннесоты), Майклу Скотту (Университет Рочестера) и Бобу Уайеру (Роки Маунтин колледж) за тестирование бета-версии на принадлежность к классам. Также огромное спасибо всем их студентам!

Нам также хочется поблагодарить наших коллег из Прентис Холл, Марша Хортон, Эрик Фрэнк и Гарольд Стоун оказывали постоянную неослабевающую поддержку в течение всего процесса работы. Гарольд при этом помогал в точном описании исторических фактов по созданию процессорных архитектур RISC и CISC. Джерри Ралия глубоко изучала материал и многому научила нас в плане литературного изложения текста.

И, наконец, хочется выразить свою признательность техническим писателям Брайану Кернингану и покойному В. Ричарду Стивенсу за то, что они доказали нам, что и техническую литературу можно писать красиво.

Огромное спасибо всем.

РЭНДИ БРАЙАНТ
ДЭЙВ О'ХАЛЛАРОН

Информация об авторах

Рэндал Э. Брайант в 1973 г. получил степень бакалавра Мичиганского университета, после чего поступил в аспирантуру Технологического института в Массачусетсе. В 1981 г. получил степень доктора наук по теории вычислительных машин и систем. В течение трех лет работал ассистентом профессора в Калифорнийском технологическом институте; на факультет в Карнеги-Меллон пришел в 1984 г. В настоящее время декан факультета информатики университета Карнеги-Меллон (г. Питсбург).

В течение 20 лет преподает курсы по вычислительной технике как для студентов-первокурсников, так и для выпускников. За долгие годы изучения курсов компьютерной архитектуры Рэндал Брайант постепенно смещал акцент с того, как устроены компьютеры, к тому, как программисты могли бы писать более эффективные и надежные программы посредством более совершенного понимания системы. Вместе с профессором О'Холлароном разработал в университете Карнеги-Меллон учебный курс "Введение в компьютерные системы", являющийся основой данной книги. Также преподавал курсы по алгоритмам и программированию.

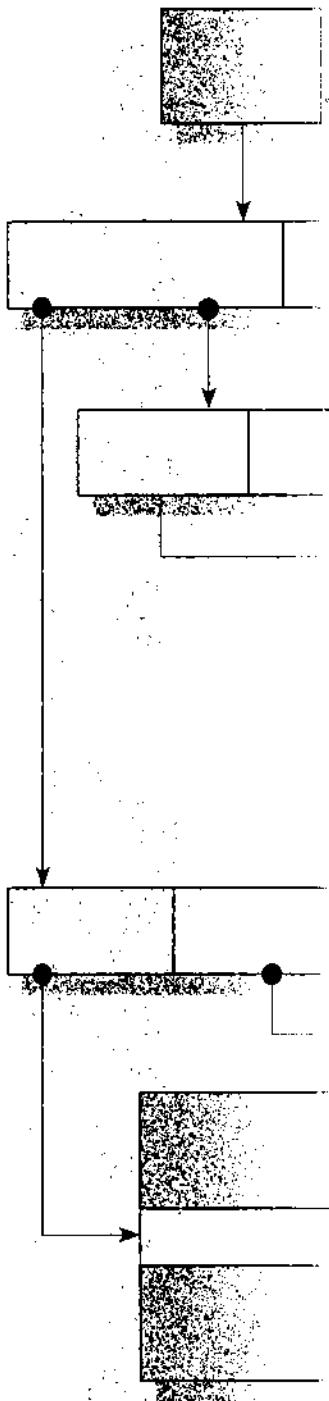
Исследования профессора Брайанта имеют отношение к проектированию инструментальных программных средств в помощь разработчикам аппаратных средств при верификации корректности создаваемых ими систем. Сюда входят несколько видов моделирующих программ, а также инструменты формальной верификации, доказывающие корректность проектирования посредством математических методов. Им опубликовано свыше 100 технических работ. Результаты исследований профессора Брайанта используются ведущими производителями компьютерной техники, включая Intel, Motorola, IBM и Fujitsu. Является лауреатом многих наград за исследования, в числе которых две награды за изобретения, а также награда за технические достижения от Корпорации исследования полупроводников, награда Канеллакиса за теоретические и практические исследования от Ассоциации по вычислительной технике (ACM), награда В.Р.Г. Бейкера и Золотая юбилейная медаль от Института инженеров по электротехнике и электронике (IEEE). Является сотрудником как ACM, так и IEEE.

В 1986 г. Дэвид Р. О'Холларон получил степень доктора наук по вычислительной технике в университете Вирджинии. После работы в компании "Дженерал Электрик" он в 1989 г. перешел на факультет Карнеги-Меллона в качестве преподавателя сис-

темотехники. В настоящее время он является адъюнкт-профессором на отделениях вычислительной техники и электротехники.

Преподавал курсы по компьютерным системам на начальных и выпускных курсах по таким темам, как компьютерная архитектура и введение в компьютерные системы, параллельное проектирование процессоров и Internet. Вместе с профессором Брайантом разработал курс "Введение в компьютерные системы".

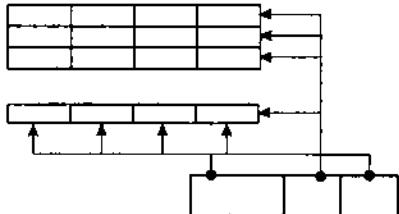
Профессор О'Холларон вместе со своими студентами осуществляет исследования в области компьютерных систем. В частности, они разработали системы программного обеспечения для ученых и инженеров, моделирующие природные явления. Самым известным примером их работы является проект Quake, когда группой специалистов по компьютерной технике, инженерами-строителями и сейсмологами была разработана возможность прогнозирования движений земной коры, сильных землетрясений, включая глобальные катаклизмы в Южной Калифорнии, Японии, Мексике и Новой Зеландии. Вместе с остальными членами проекта Quake профессор О'Холларон получил от университета Карнеги-Меллона медаль Алена Ньювела за исключительный вклад в исследования по компьютерной тематике. Разработанная им точка отсчета для указанного проекта 183.equake была выбрана SPEC для включения в имеющий очень влиятельное значение набор программ SPEC CPU и OMP (Open MP).



ЧАСТЬ вводная

Обзор компьютерных систем

ГЛАВА 1



Экскурс в компьютерные системы

- Информация — это биты + контекст.
 - Программы, которые переводятся другими программами в различные формы.
 - Как работает система компиляции.
 - Процессоры читают и интерпретируют инструкции, сохраняемые в памяти.
 - Различные виды кэш-памяти.
 - Устройства памяти образуют иерархию.
 - Операционная система управляет работой аппаратных средств.
 - Обмен данных в сетях.
 - Следующие шаги.
 - Резюме.
-

Компьютерная система состоит из аппаратных средств и программного обеспечения, которые взаимодействуют, обеспечивая выполнение прикладных программ. Конкретные реализации систем со временем претерпевают изменения, однако идеи, лежащие в их основе, остаются неизменными. Все аппаратные и программные компоненты, из которых состоят вычислительные системы и которые выполняют одни и те же функции, похожи друг на друга. Эта книга предназначена для программистов, которые хотят повысить свою квалификацию за счет лучшего понимания того, как эти компоненты работают и какое влияние они оказывают на правильность функционирования их программ.

Вам предстоит увлекательное путешествие. Если вы не пожалеете времени для изучения понятий, предложенных в этой книге, то вы вступите на путь, который в конечном итоге позволит вам стать представителем немногочисленной категории профессиональных программистов, обладающих четким пониманием принципов работы вычислительной системы и влияния, которое она оказывает на ваши прикладные программы.

Вы приобретете специальные навыки, например, будете знать, как избежать странных числовых ошибок, вызванных особенностями представления чисел конкретным компьютером. Вы узнаете, как получить оптимальный программный код на языке С путем применения специальных приемов, которые используют особенности современных процессоров и систем памяти. Вы получите представление о том, как компилятор реализует вызовы процедур, и как использовать эти знания, чтобы избегать прорех в системе защиты, вызванных сбоями, возникающими в результате переполнения буферов, которые мешают работе сетевого программного обеспечения. Вы научитесь распознавать и избегать неприятных ошибок во время редактирования связей, которые приводят в замешательство программистов средней руки. Вы научитесь писать свои собственные оболочки, собственные пакеты процедур динамического распределения памяти и даже свой собственный Web-сервер!

В своих, ставших классическими книгах по программированию на языке С [40], Керниган (Kernighan) и Ритчи (Ritchie) начинают знакомить читателя с языком программирования с программы `hello` (программы приветствия), представленной в листинге 1.1. И хотя это очень простая программа, тем не менее, все основные части системы должны работать согласованно, чтобы довести ее до успешного завершения. В каком-то смысле цель этой книги заключается в том, чтобы помочь читателю понять, что происходит и как, когда вы осуществляете выполнение программы `hello` на вашей системе.

Мы начнем изучение систем с того, что проследим за программой `hello` на протяжении времени существования, с момента ее написания программистом до момента, когда она выполняется системой, распечатывает свое незатейливое послание и завершается. По мере истечения времени существования этой программы мы будем кратко вводить основные понятия, терминологию и компоненты, которые вступают в действие. В последующих главах мы подробнее остановимся на этих понятиях и идеях.

1.1. Информация — это биты + контекст

Наша программа `hello` начинает свой жизненный путь как *исходная программа* (или *исходный файл*), которую программист создает с помощью редактора текстов и сохраняет в текстовом файле с именем `hello.c`.

Листинг 1.1. Программа `hello`

```
#include <stdio.h>

int main()
{
    printf("hello, world\n");
}
```

Исходная программа представляет собой последовательность битов, каждый из которых принимает значение 0 или 1, организованных в 8-битовые порции, получившие название байтов. Каждый байт представляет собой некоторый символ программы.

Большинство современных систем представляют текстовые символы в стандарте ASCII (American Standard Code for Information Interchange, Американский стандартный код обмена информацией), который представляет каждый символ уникальным восьмизначным целым двоичным числом. Например, в листинге 1.2 дается представление программы `hello.c` в кодах ASCII.

Листинг 1.2. Представление текстового файла `hello.c` в кодах ASCII

#	i	n	c	l	u	d	e	<sp>	<	s	t	d	i	o	.
35	105	110	99	108	117	100	101	32	60	115	116	100	105	111	46
h	>	\n	\n	i	n	t	<sp>	m	a	i	n	()	\n	(
104	62	10	10	105	110	116	32	109	97	105	110	40	41	10	123
	\n	<sp>	<sp>	<sp>	p	r	i	n	t	f	("	h	e	1
10	32	32	32	32	112	114	105	110	116	102	40	34	104	101	108
1	o	,	<sp>	w	o	r	l	d	n	")	:	\n	}	
108	111	44	32	119	111	114	108	100	92	110	34	41	59	10	125

Программа `hello.c` хранится в файле как последовательность байтов. Каждый байт принимает целое значение, которое соответствует некоторому символу. Например, первый байт имеет целое значение 35, которое соответствует символу `#`. Второй байт имеет значение 105, которое соответствует символу `i`, и т. д. Обратите внимание на то обстоятельство, что текстовая строка заканчивается невидимым символом `\n`, который представлен целым значением 10. Такие файлы, как `hello.c`, которые содержат исключительно символы, называются *текстовыми файлами*. Все другие файлы получили название *двоичные файлы*.

Представление файла `hello.c` служит иллюстрацией одной из фундаментальных идей. Вся информация системы, включая файлы на дисках, программы и данные пользователей, хранящиеся в памяти, а также данные, передаваемые по сети, представляются в виде некоторых порций. Единственное, что отличает различные виды данных друг от друга — контекст, в котором мы их рассматриваем. Например, в различных контекстах одна и та же последовательность данных может представлять целое число, число с плавающей точкой, строку символов или машинную инструкцию.

Как программисты, мы должны понимать машинное представление чисел, поскольку они не тождественны целым или вещественным числам. Они суть конечные приближения, которые могут повести себя непредсказуемым образом. Эта фундаментальная идея интенсивно используется в главе 2.

Язык программирования С

Язык С разрабатывался с 1969 года по 1973 год Деннисом Риччи (Dennis Ritchie), сотрудником Bell Laboratories. Национальный Институт Стандартизации США (ANSI — American National Standards Institute) утвердил стандарт ANSI C в 1989 году. Этот стандарт дает определение языка программирования С и набора

библиотечных функций, известных как библиотека стандартных программ на С. Керниган и Риччи описали язык в своей классической книге, которую в программистских кругах любовно называют не иначе как "K&R" [40]. По словам Риччи [64], язык С — это причудливый, порочный и в то же время безусловный успех. Так все-таки почему успех?

- С был тесно связан с операционной системой Unix. Он разрабатывался с самого начала как системный язык программирования для Unix. Большая часть ядра и все вспомогательные средства поддержки и библиотеки были написаны на С. По мере того, как Unix становилась популярной во второй половине семидесятых и в начале восьмидесятых годов прошлого столетия, многим людям приходилось сталкиваться с языком С, и многим он нравился. Поскольку Unix была практически полностью написана на С, ее легко можно было переносить на новые машины, а это, в свою очередь, увеличивало аудиторию пользователей как языка С, так и самой операционной системы Unix.
- С — простой компактный язык. Его разработкой занимался один человек, а не многочисленный комитет, и результатом явился четкий непротиворечивый язык с небольшими добавлениями. В книге дается описание завершенной версии языка и стандартной библиотеки, приводятся многочисленные примеры и упражнения, и это заняло всего лишь 261 страниц. Простота языка С существенно упрощает его изучение и перенос на различные компьютеры.
- С разрабатывался для практических целей. Он первоначально предназначался для реализации операционной системы Unix. Потом другие люди обнаружили, что на нем можно писать программы любого назначения, поскольку сам язык предоставлял такую возможность.

С — язык, на котором особенно удобно писать программы системного уровня, в то же время в нем имеются все средства, обеспечивающие написание программ прикладного уровня. Тем не менее он удовлетворяет некоторую часть программистов и не подходит в равной степени для всех ситуаций. Указатели языка С часто являются причиной различных недоразумений и программных ошибок. Языку не хватает также прямой поддержки таких полезных абстракций, как классы, объекты и исключительные ситуации. Более новые версии этого языка, такие как C++ и Java, позволяют решать подобного рода проблемы и для программ прикладного уровня.

1.2. Программы, которые переводятся другими программами в различные формы

Программа `hello` начинает свою жизнь, как программа на языке высокого уровня, поскольку в этой форме она может быть прочитана и понята человеком. Однако для того, чтобы обработать файл `hello.c` на системе, отдельные операторы языка должны быть преобразованы другими программами в некоторую последовательность инструкций машинного языка низкого уровня. Эти инструкции затем упаковываются в исполняемую объектную программу и сохраняются в двоичном файле на диске. Объектные программы также называются *исполняемыми объектными файлами*.

В операционной системе Unix преобразование исходного файла в объектный файл выполняется драйвером компилятора:

```
unix> gcc -o hello hello.c
```

Здесь драйвер компилятора GCC считывает исходный файл и транслирует его в исполняемый объектный файл hello. Эта трансляция имеет четыре стадии, показанные на рис 1.1. Совокупность программ, которая исполняет эти четыре фазы (препроцессор, компилятор, ассемблер и редактор связей) называется *системой компиляции*.

- Стадия препроцессора (или фаза предварительной обработки). Препроцессор (cpp) изменяет исходную программу в соответствии с директивами, которые начинаются с символа #. Например, команда `#include <stdio.h>` в строке 1 программы hello.c заставляет препроцессор прочитать содержимое системного файла заголовков stdio.h и вставить его непосредственно в программный текст. Результатом является другая программа на языке C, обычно с суффиксом i.

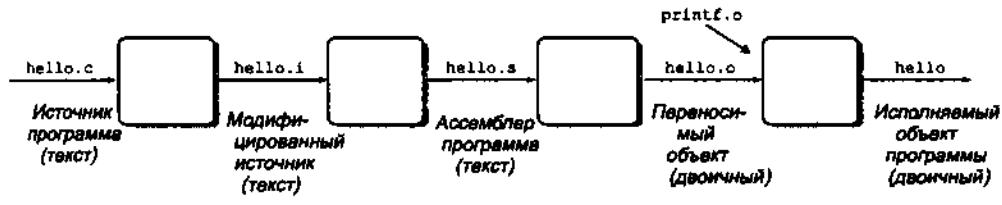


Рис. 1.1. Система компиляции

- Стадия компиляции. Компилятор (cc1) транслирует текстовый файл в текстовый файл `hello.s`, который содержит программу на языке ассемблера. Каждый оператор в программе на языке ассемблера точно описывает одну из машинных инструкций языка нижнего уровня в стандартной текстовой форме. Язык ассемблера полезен, прежде всего, в силу того обстоятельства, что он предоставляет общий выходной язык для компиляторов различных языков высокого уровня. Например, компиляторы языка C и языка Fortran генерируют выходные файлы на одном и том же языке ассемблера.
- Стадия ассемблирования. Ассемблер (as) транслирует файл `hello.s` в инструкции машинного языка, упаковывает их в форму, известную как *перемещаемая объектная программа*, и запоминает результат в объектном файле `hello.o`. Файл `hello.o` есть двоичный файл, байты которого кодируют инструкции машинного языка, но не символы. Если бы мы просмотрели файл с помощью текстового редактора, то увидели бы совершенно непонятную картину.
- Стадия редактирования связей. Обратите внимание на то обстоятельство, что наша программа `hello` обращается к функции `printf` из библиотеки стандартных программ на C, которая предоставляется пользователю каждым компилятором языка C. Функция `printf` постоянно находится в отдельном предварительно скомпилированном объектном файле с именем `printf.o`, который тем или иным способом должен быть слит с нашей программой `hello.o`. Это слияние осуществляется в процессе компиляции.

ляет редактор связей (`ld`). Результатом является файл `hello`, который представляет собой исполнительный объектный файл (или просто *исполнимый файл*), который готов к загрузке и исполнению системой.

О проекте GNU

Gcc — одно из многих полезных инструментальных средств, разработанных в рамках проекта GNU (сокращенно от *GNU's Not Unix*). Проект — освобожденная от налогов благотворительная акция, начатая Ричардом Столлменом (Richard Stallman) в 1984 году с амбициозной целью разработать полную, Unix-подобную систему, исходный код которой не перегружен ограничениями на то, как может модифицироваться или распределяться. В 2002 году проект GNU разработал среду Unix со всеми основными компонентами операционной системы Unix, за исключением ядра, которое разрабатывалось отдельно, а именно в рамках проекта Linux. Среда GNU включает редактор EMACS, компилятор GCC, отладчик GDB, ассемблер, редактор связей, служебные программы для манипуляции двоичными файлами и другими компонентами.

Проект представляет собой замечательное достижение, однако сплошь и рядом ему не уделяют должного внимания. Современная мода на программные продукты с открытым текстом (обычно ассоциируется с Linux) обязана своим интеллектуальным происхождением понятию *открытые программные средства*, возникшему в рамках проекта ("открытые" в смысле "свобода слова", но не в смысле "бесплатное пиво"). Более того, операционная система Linux обязана большей частью своей популярности инструментальным средствам GNU, которые позволяют развернуть среду для ядра системы Linux.

1.3. Как работает система компиляции

Для простых программ, таких как `hello.c`, мы вполне можем рассчитывать на то, что система компиляции построит правильный и эффективный машинный код. Однако существуют важные причины, почему программисты должны понимать, как работает система компиляции.

- Производительность оптимизирующей программы. Современные компиляторы представляют собой инструментальные средства, которые обычно составляют эффективные программные коды. Как программистам, нам не надо знать, каково внутреннее устройство компилятора, и как он работает, чтобы получить эффективные коды. Но для того, чтобы принимать правильные решения, касающиеся кодирования, мы должны в своих программах на языке C иметь четкое представление о языке ассемблера и о том, как компилятор транслирует различные операторы языка C в язык ассемблера. Например, является ли оператор выбора всегда более эффективным, чем некоторая последовательность операторов `if-then-else`? Насколько дорогостоящим является вызов функции? Является ли оператор цикла `while` более эффективным, чем оператор цикла `do`? Являются ли ссылки указателей более эффективными, чем индексы элементов массивов? Почему наш цикл выполняется намного быстрее, если мы будем накапливать сумму в некоторой локальной переменной вместо аргумента, который передается по ссылке?

В главе 3 мы будем рассматривать машинный язык Intel IA32 и опишем, как компиляторы транслируют различные конструкции языка С в этот язык. В главе 5 вы научитесь выполнять настройку ваших программ, выполняя простые преобразования в коде, которые позволили бы компилятору выполнять свою задачу. А в главе 6 вы будете изучать иерархическую структуру системы памяти, как компиляторы языка запоминают в памяти массивы данных, и как ваши программы на С могут использовать эти сведения, чтобы работать более эффективно.

- Понимание ошибок, возникающих в процессе редактирования связей. Наш опыт показывает, что некоторые из наиболее запутанных программных ошибок относятся к функционированию редактора связей, особенно когда вы пытаетесь построить крупные программные системы. Например, что означает ситуация, когда редактор связей сообщает, что он не может разрешить ссылки? Что случится, если вы объявили две глобальные переменные в различных файлах на С с одним и тем же именем? В чем различие между статической библиотекой и динамической библиотекой? Почему имеет значение, в каком порядке мы перечисляем библиотеки в командной строке? И хуже всего, почему ошибки, источником которых является редактор связей, не проявляются раньше, чем начнется выполнение программы? Ответы на все эти вопросы вы получите в главе 7.
- Как избежать пробелов в системе защиты. В течение многих лет ошибка, порождаемая переполнением буфера, была причиной большей части проблем системы защиты сети и в серверах. Такие ошибки существуют в силу того обстоятельства, что многие программисты не имеют понятия о правилах использования стеков, которыми руководствуются компиляторы в процессе генерации кодов функций. Мы дадим описание дисциплины использования стеков и ошибок, порождаемых переполнением буферов, в главе 3 в рамках изучения языка ассемблера.

1.4. Процессоры читают и интерпретируют инструкции, сохраняемые в памяти

На данный момент стадии наша исходная программа `hello.c` прошла через систему компиляции, которая преобразовала ее в исполняемый объектный файл с именем `hello`, который теперь хранится на диске. Чтобы осуществить выполнение исполняемого файла в системе Unix, мы передаем с клавиатуры его имя прикладной программе, известной как оболочка:

```
unix>./hello
hello, world
unix>
```

Эта оболочка представляет собой интерпретатор командной строки, который выводит на экран приглашение на ввод команд, ждет, когда вы заполните с клавиатуры командную строку, а затем выполняет заданную команду. Если первое слово командной строки не является именем какой-либо встроенной команды оболочки, то оболочка делает вывод, что это слово есть имя исполняемого файла, который она должна загрузить и выполнить. Следовательно, в рассматриваемом случае оболочка

загружает и запускает на выполнение программу `hello`, после чего ждет завершения ее выполнения. Программа выводит свое сообщение на экран, после чего завершается. Оболочка затем печатает приглашение на ввод следующей команды и ждет, когда в командную строку будет введена новая команда.

1.4.1. Организация аппаратных средств системы

Чтобы знать, что происходит с нашей программой `hello` во время ее выполнения, мы должны иметь представление о том, как устроена аппаратная часть типичной вычислительной системы, блок-схема которой показана на рис. 1.2. Эта конкретная блок-схема построена для семейства Intel Pentium, однако все вычислительные системы имеют примерно такой же вид и дают те же ощущения от использования. Пусть вас сейчас не беспокоит сложность этой блок-схемы — мы будем рассматривать различные ее детали на протяжении всей книги.

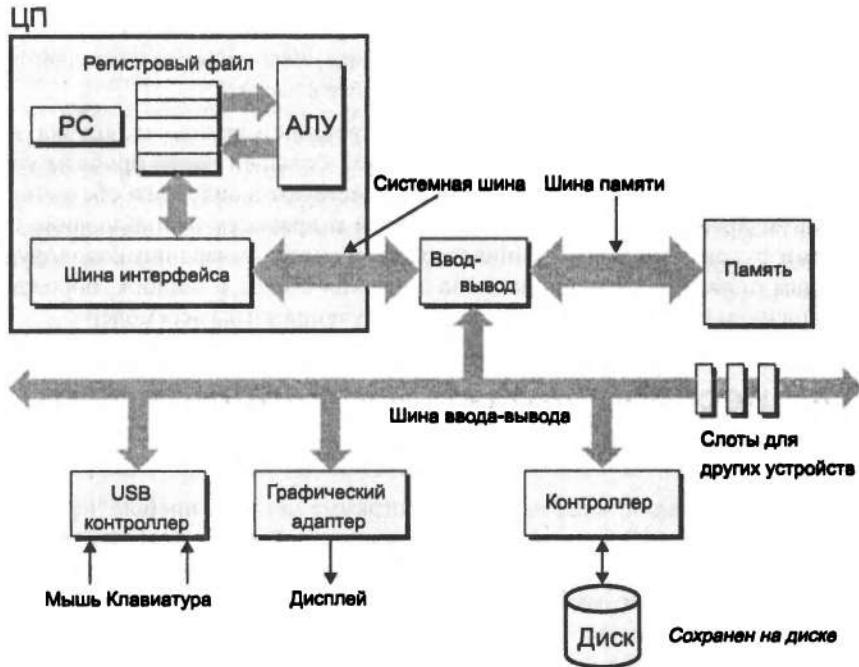


Рис. 1.2. Организация аппаратных средств типичной вычислительной системы

Шины

Вычислительную систему пронизывает совокупность электрических проводников, так называемых *шин*, по которым байты информации циркулируют между компонентами системы. Обычно шины конструируются таким образом, чтобы можно было передавать порции, содержащие фиксированное число байтов, получившие название *слово*. Число байтов в слове (размер слова) является одним из фундаментальных сис-

тенных параметров, которые изменяются от системы к системе. Например, одни системы имеют размер слова 64 байт, а в таких системах серверного класса, как Intel Itaniums, а также в наиболее производительных системах семейства Sun SPARCS размер слова составляет 8 байтов. В системах поменьше, которые используются в качестве устройства управления автомобиля или промышленными установками, размер слова может составлять всего лишь один или два байта. Для простоты полагаем, что размер слова равен 64 байтам, будем считать, что шины передают в одно и то же время только одно слово.

Устройства ввода-вывода

Устройства ввода-вывода представляют собой средства связи с внешним миром. В рассматриваемом примере в системе имеются устройства ввода-вывода четырех видов: клавиатура и мышь для ввода со стороны пользователя, устройство отображения вывода данных для пользователя и накопитель на дисках (или просто диск) для долгосрочного хранения данных и программ. В начальный момент исполняемый файл хранится на диске.

Каждое устройство ввода-вывода подключено кшине ввода-вывода посредством контроллера или адаптера. Различие между ними заключается в их конструктивных особенностях. Контроллеры — наборы плат, установленных в самом устройстве или на главной печатной плате (ее еще часто называют материнской платой). Адаптер представляет собой плату, которая подключается через контактное гнездо в материнской плате. Независимо от конструкции таких устройств, их назначение заключается в том, чтобы передавать информацию между шиной ввода-вывода и устройством ввода-вывода в обоих направлениях.

В главе 6 более подробно описана работа таких устройств ввода-вывода, как диск. В главе 11 вы узнаете, как следует пользоваться интерфейсом ввода-вывода системы Unix, чтобы получить доступ к устройствам из вашей прикладной программы. Мы сосредоточим свое внимание на особо интересном классе устройств, получивших название сетей: специальные технологии обобщают методы их использования также и на другие виды устройств.

Оперативная память

Оперативная память — временное запоминающее устройство, в котором временно хранятся как программа, так и данные, которыми она манипулирует во время выполнения. Физически основная память состоит из совокупности чипов динамических оперативных запоминающих устройств. В логическом плане основная память организована в виде линейной последовательности байтов, каждый из которых имеет свой собственный уникальный адрес (индекс элемента массива); отсчет адресов начинается с нуля. В общем случае все машинные инструкции, составляющие некоторую программу, состоят из переменного числа байтов. Размер элементов данных, соответствующих переменным программы на С, меняется в зависимости от типа. Например, на машине Intel, работающей под Linux, тип данных short требует двух байтов, типы int, float и long требуют четырех байтов, а тип double — восьми байтов.

В главе 6 дается более подробное описание, как работают технологии памяти и как из них конструируется основная память.

Процессор

Центральный процессор (ЦП, CPU), или просто процессор, представляет собой механизм, который интерпретирует (или выполняет) инструкции, хранящиеся в основной памяти. Его ядро составляет устройство памяти размером в одно слово (или регистр), получившее название *счетчик команд* (PC). В любой конкретный момент времени он указывает на адрес некоторой инструкции машинного языка в основной памяти¹.

С момента включения системы и до момента ее выключения процессор слепо и многократно выполняет одну и ту же основную задачу, не прерываясь ни на мгновение. Он считывает из памяти инструкцию, указанную счетчиком команд, интерпретирует биты инструкции, выполняет простые операции, предписанные инструкцией, а затем обновляет значение счетчика, чтобы тот указывал адрес следующей инструкции, которая может быть или не быть соседней в памяти, по отношению к только что выполненной инструкции.

Существует всего несколько таких операций, они циркулируют между основной памятью, регистровым файлом и арифметико-логическим устройством (ALU, ALU). *Регистровый файл* — небольшое запоминающее устройство, которое состоит из совокупности регистров размеров в одно слово, каждый из которых имеет свое уникальное имя. Устройство вычисляет новые значения данных и адресов. Назовем лишь несколько примеров простых операций, которые процессор может выполнить по требованию той или иной инструкции:

- Загрузка. Скопировать байт или слово из основной памяти в регистр, затирая при этом предыдущее содержимое этого регистра.
- Запоминание. Скопировать байт или слово из регистра в некоторую ячейку основной памяти, затирая при этом предыдущее содержимое этой ячейки.
- Обновление данных. Скопировать содержимое двух регистров ALU, которое складывает эти два слова и запоминает результат в одном из регистров, затирая при этом предыдущее содержимое этого регистра.
- Считывание ввода-вывода. Скопировать байт или слово из устройства ввода-вывода в регистр.
- Запись ввода-вывода. Скопировать байт или слово из регистра в устройство ввода-вывода.
- Переход. Извлечь слово из самой инструкции и скопировать это слово в счетчик команд, затирая предыдущее содержимое.

В главе 4 вы узнаете намного больше о том, как работает процессор.

¹ PC также является широко используемым акронимом "персональный компьютер". Различие между понятиями должно быть понятно из контекста.

1.4.2. Выполнение программы *hello*

Ознакомившись с описанным выше простым представлением организации аппаратных средств и операций, мы начинаем понимать, что происходит, когда мы осуществляем выполнение нашего примера программы. Здесь мы должны опустить множество деталей, которые мы учтем позже, но сейчас нас вполне удовлетворит общая картина.

Вначале свои инструкции выполняет программная оболочка, ожидающая, когда мы введем команду с клавиатуры. Как только мы введем символы "./hello", программная оболочка считывает каждый из них в соответствующий регистр, а затем запоминает их в основной памяти, как показано на рис. 1.3.

Затем мы нажимаем клавишу, оболочка воспринимает это как сигнал окончания ввода команды. Загружается исполняемый файл *hello*, выполняя с этой целью последовательность инструкций, которая копирует программные коды и данные, содержащиеся в объектном файле *hello*, с диска в основную память. Данные включают строку символов "hello, world\n", которая в конечном итоге будет выведена на экран.

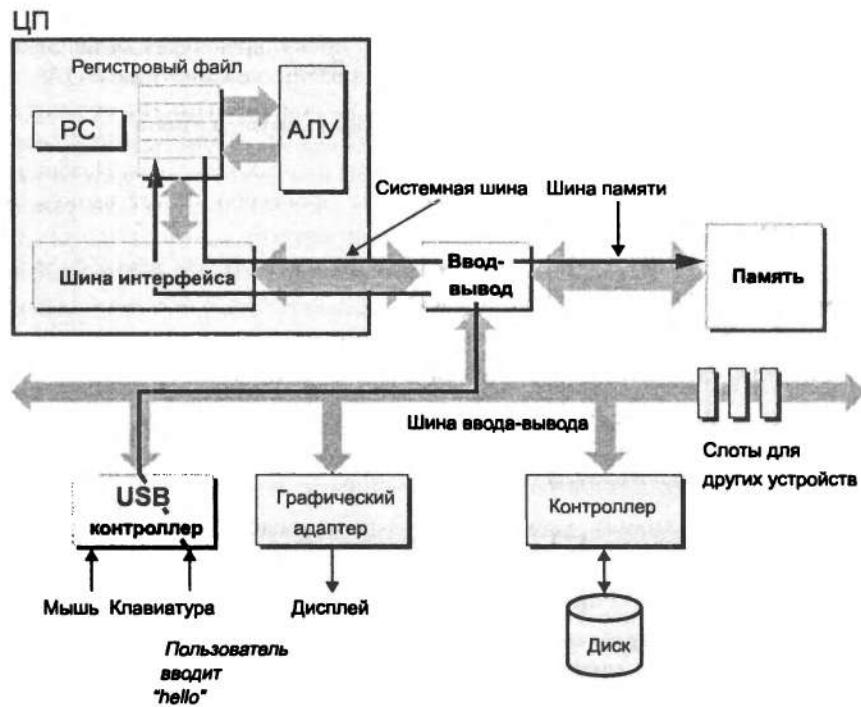


Рис. 1.3. Считывание команды *hello* с клавиатуры

Используя метод, известный как прямой доступ к памяти (DMA, см. главу 6), данные перемещаются с диска непосредственно в оперативную память, не проходя через процессор. Эти действия показаны на рис. 1.4.

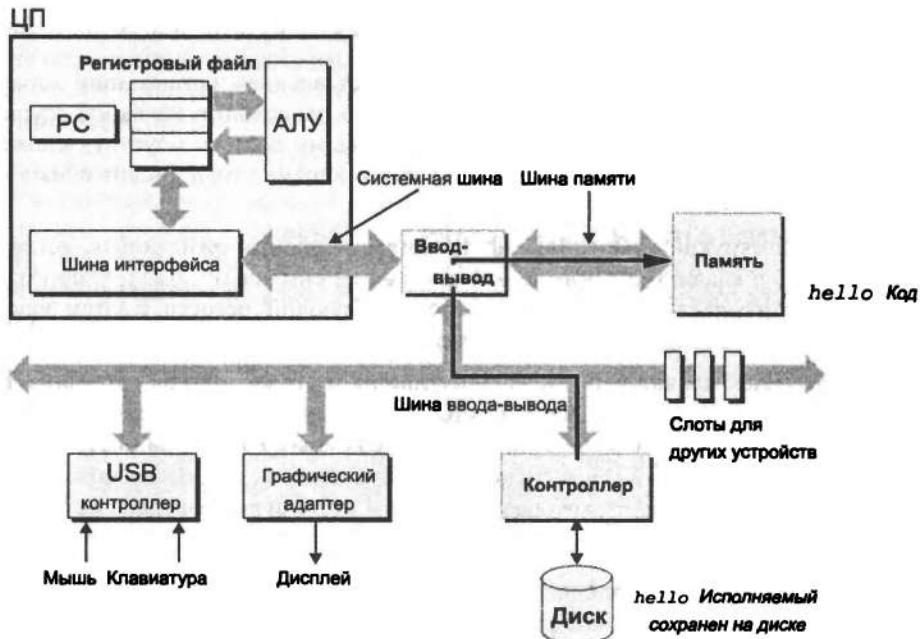


Рис. 1.4. Загрузка исполняемых файлов в основную память

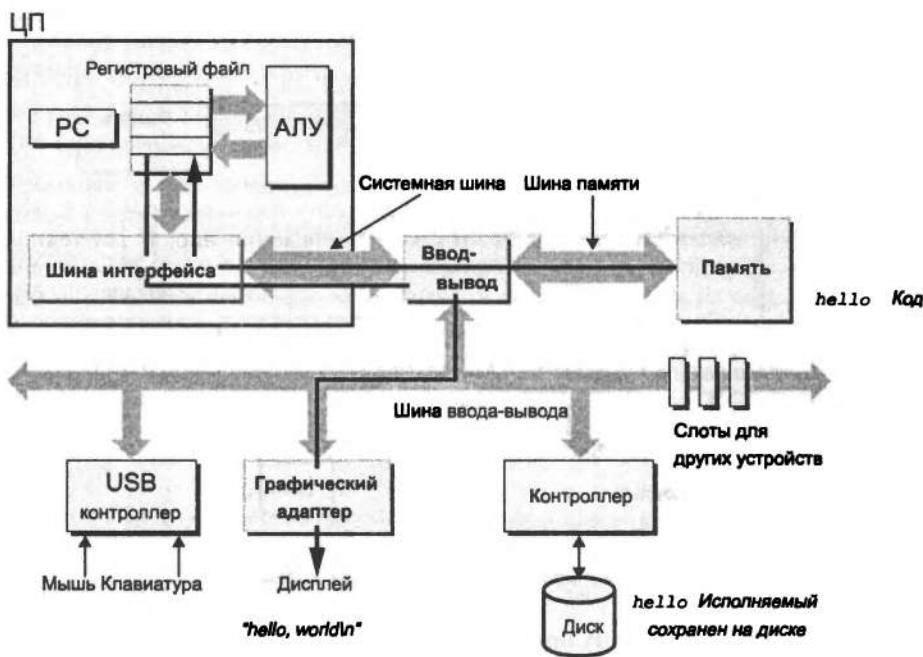


Рис. 1.5. Запись выходной строки из основной памяти на экран дисплея

Как только программный код и объектный файл будут загружены в память, процессор начинает выполнять инструкции на машинном языке подпрограммы main программы hello. Эти инструкции копируют байты строки "hello, world\n" из основной памяти в регистровый файл, а отсюда — на устройство отображения, на экран которой они и выводятся. Эти действия показаны на рис. 1.5.

1.5. Различные виды кэш-памяти

Важный урок, который мы извлекаем из этого простого примера, заключается в том, что система затрачивает уйму времени на перемещение информации с одного места в другое. Машинные инструкции в программе в начальный момент хранятся на диске. Когда производится загрузка программы, они копируются в основную память. По мере выполнения программы процессором инструкции копируются из основной памяти в процессор. Аналогично, строка данных "hello, world\n", которая первоначально хранилась на диске, копируется в основную память, а затем из основной памяти копируется на устройство отображения. С позиций программиста, большая часть такого копирования связана с непроизводительными расходами вычислительных ресурсов, которые существенно снижают "фактическую производительность" программы. Следовательно, главная цель системных проектировщиков заключается в том, чтобы операции по копированию данных происходили как можно быстрее.

В силу чисто физических законов, чем больше запоминающее устройство, тем медленнее оно работает. И в то же время, создание быстродействующих запоминающих устройств обходится дороже, чем более медленных устройств. Например, емкость диска может оказаться в 100 раз больше, чем оперативное запоминающее устройство (память), но на то, чтобы считать слово из дисковой памяти, потребуется времени в 10 млн раз больше, чем из основной памяти.

Аналогично, типичный регистровый файл содержит всего лишь несколько сотен байтов информации, в то время как в основной памяти содержатся миллионы байтов. Однако процессор может считывать данные из регистров примерно в 100 раз быстрее, чем из памяти. Более того, по мере того как на протяжении многих лет технология полупроводников продолжает развиваться, расхождения между процессором и памятью продолжают углубляться. Гораздо проще и дешевле повысить быстродействие процессоров, чем заставить основную память работать быстрее.

Чтобы уменьшить разрыв между процессором и основной памятью, специалисты по проектированию систем используют небольшие быстродействующие устройства, получившие название **кэш-память** (или просто **кэш**) и служащие в качестве временных резервных областей для хранения информации, которая, возможно, потребуется процессору в самом ближайшем будущем. На рис. 1.6. показана кэш-память типичной системы. Кэш L1 на плате процессора содержит десятки тысяч байтов, доступ к ним осуществляется фактически так же быстро, как и к регистровому файлу. Еще больше кэш L2, содержащий от сотен тысяч до миллионов байтов, он соединен с процессором посредством специальной шины. Вычислительному процессору потребуется в 5 раз больше времени для доступа к L2, чем к L1, но это все же в 5—10 раз быстрее, чем доступ к основной памяти. L1 и L2 построены по технологии, известной как статические запоминающие устройства с произвольной выборкой (SRAM).

Один из самых важных уроков этой книги заключается в том, что прикладные программисты, которые знают о наличии кэш-памяти, могут воспользоваться ею, чтобы повысить производительность их программ в несколько порядков. Мы будем изучать эти важные устройства и узнаем, как ими пользоваться, в главе 6.

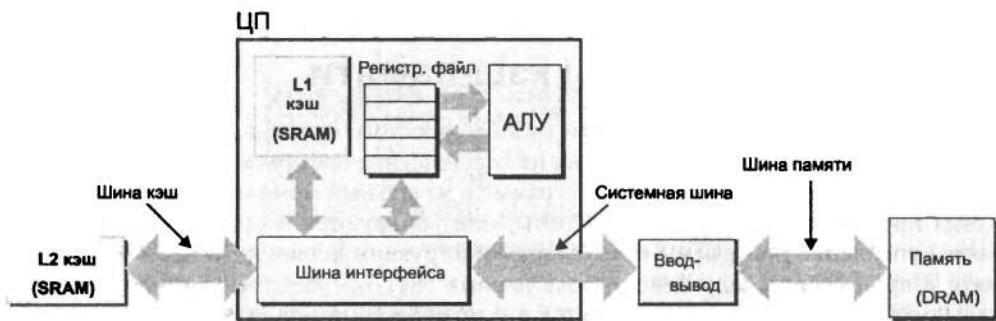


Рис. 1.6. Различные виды кэш-памяти

1.6. Устройства памяти образуют иерархию

Идея поместить небольшое, зато более быстрое запоминающее устройство (кэш-память) между процессором и более крупным, но обладающим меньшим быстродействием запоминающим устройством (основной памятью), оказалась весьма плодотворной. Фактически устройства памяти в каждой вычислительной системе образуют



Рис. 1.7. Пример иерархии памяти

иерархию памяти, подобную изображенной на рис. 1.7. По мере движения по этой иерархии сверху вниз, устройства становятся все медленней, крупнее, а стоимость хранения одного бита уменьшается. Регистровый файл занимает верхнюю позицию в иерархии, которая обозначается как уровень 0 или L0. Кэш-память занимает уровень 1 (отсюда происходит обозначение L1). Кэш-память уровня L2 находится на уровне 2. Основная память занимает уровень 3 и т. д.

Основная идея иерархии памяти заключается в том, что память одного уровня служит кэш-памятью для следующего нижнего уровня. Таким образом регистровый файл — кэш для памяти уровня L1, которая является кэш-памятью для основной памяти, а та, в свою очередь, является кэш-памятью для диска. В некоторых сетевых системах с распределенной файловой системой локальный диск служит кэш-памятью для данных, хранящихся на дисках других систем.

Подобно тому, как программисты используют структуру кэш-памяти уровней L1 и L2 для повышения производительности своих программ, можно использовать структуру всей иерархии памяти. Более подробно эти вопросы будут рассмотрены в главе 6.

1.7. Операционная система управляет работой аппаратных средств

Вернемся к нашему примеру с программой `hello`. Когда оболочка загружала и выполняла программу `hello`, и когда программа `hello` отображала свое сообщение, ни та ни другая программа не имели прямого доступа к клавиатуре, диску или основной памяти. Для этого они пользовались услугами, предоставляемыми операционной системой. Мы можем представлять себе операционную систему как некоторый слой программного обеспечения между прикладной программой и аппаратными средствами, как показано на рис. 1.8. Все попытки прикладной программы манипулировать аппаратными средствами должны проходить через операционную систему.

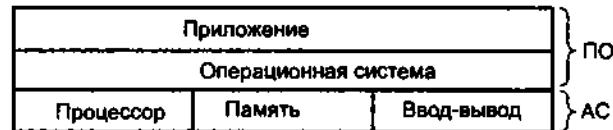


Рис. 1.8. Вычислительная система как многоуровневая система

Операционная система, прежде всего, должна отвечать двум следующим основным требованиям: защитить аппаратные средства от катастрофических действий вышедшей из-под контроля программы и снабжать приложения простыми и единообразными механизмами манипулирования сложными и часто крайне неоднородными низкоуровневыми аппаратными средствами. Операционная система достигает обеих этих целей посредством фундаментальных абстракций, показанных на рис. 1.8: процесса, виртуальной памяти и файлов. Как видно из рис. 1.9, файлы — это абстракции для устройств ввода-вывода, виртуальная память является абстракцией как для основной памяти, так и для дисковых устройств ввода-вывода, а процессы — абстракции для процессора, основной памяти и устройств ввода-вывода.

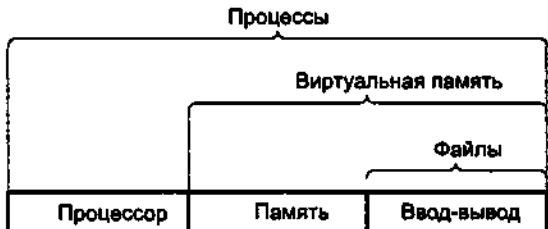


Рис. 1.9. Абстракции, реализованные операционной системой

Unix и стандарты Posix

В шестидесятые годы прошлого столетия господствовали крупные, сложные операционные системы, такие как OS/360, разработанная компанией IBM, и Multics, разработанная компанией Honeywell. И если OS/360 была одной из наиболее успешных операционных систем того периода, то Multics властила жалкое существование в течение многих лет и не смогла добиться широкого признания. Компания Bell Laboratories первоначально была одним из партнеров, разрабатывавших проект Multics, но в 1969 году отказалась от участия в этом проекте по причине чрезмерной сложности проекта и ввиду отсутствия положительных результатов. Полученный при разработке системы отрицательный опыт заставил группу исследователей компании — Кена Томпсона (Ken Thompson), Денниса Риччи (Dennis Ritchie), Дуга Макилоя (Doug McIlroy) и Джо Оссанны (Joe Ossanna) — начать в 1969 году работы над более простой операционной системой для компьютера PDP7 компании DEC, написанной исключительно на машинном языке. Многие из идей новой системы, такие как иерархическая файловая система и интерпретация оболочки как процесса пользовательского уровня, были заимствованы из системы Multics, но были реализованы в виде более простого и компактного пакета программ. В 1970 году Брайан Керниган (Brian Kernighan) выбрал для названия новой системы название "Unix", как противовес названию "Multics", тем самым подчеркивая неповоротливость и тяжеловесность системы Multics¹. Ядро Unix было переписано на языке в 1973 году, сама операционная система была представлена широкой публике в 1974 году [65].

Поскольку компания Bell Labs предоставила высшим учебным заведениям исходные коды на очень выгодных условиях, у операционной системы появилось множество сторонников системы Unix среди студентов и преподавателей различных университетов. Работа, оказавшая большое влияние на дальнейшее развитие, была выполнена в Калифорнийском университете Беркли в конце семидесятых и в начале восьмидесятых годов, когда исследователи из Беркли добавили виртуальную память и протоколы в последовательность версий, получивших название Unix 4.xBSD (Berkeley Software Distribution). Одновременно компания Bell Labs наладила выпуск своих собственных версий Unix, которые стали известны как System V Unix. Версии других поставщиков программного обеспечения, таких как система Sun Microsystems Solaris, были построены на базе исходных версий BSD и System V.

¹ В некотором приближении, Multics можно перевести как многогранный, в том же контексте Unix можно перевести как одногранный. — Прим. перев.

Осложнения возникли в середине восьмидесятых годов, когда поставщики операционной системы предприняли попытки выбрать собственные направления, добавляя новые и часто не совместимые с прежними версиями свойства. Чтобы преодолеть эти сепаратистские тенденции, институт стандартизации IEEE (Institute for Electrical and Electronics Engineers, Институт инженеров по электротехнике и электронике) возглавил усилия по стандартизации системы Unix. Позже Ричард Столлман (Richard Stallman) окрестил продукт этих усилий как "Posix". В результате было получено семейство стандартов, известное как стандарты Posix, которые решали такие проблемы, как интерфейс языка C для системных вызовов в Unix, программные оболочки и утилиты, потоки и сетевое программирование. По мере того как системы достигают все большей совместимости со стандартами Posix, различия между различными версиями системы Unix постепенно сходят на нет.

1.7.1. Процессы

Когда программа, такая как `hello`, работает в современной системе, операционная система создает иллюзию, что только эта программа выполняется системой. Создается впечатление, что только эта программа распоряжается процессором, основной памятью и устройствами ввода-вывода. Процессор как бы выполняет все инструкции программы подряд, одну за другой, без прерываний, и только код программы и ее данные являются единственными объектами, пребывающими в памяти системы. Источником таких иллюзий является понятие процесса, одна из наиболее важных и успешных идей в теории вычислительных машин и систем.

Процесс есть абстракция выполняемой программы в рамках операционной системы. На одной и той же системе одновременно могут выполняться многие процессы, и в то же время создается впечатление, что каждый процесс пользуется исключительными правами на использование аппаратных средств. Под одновременным выполнением мы понимаем то, что инструкции одного процесса чередуются с инструкциями другого процесса. Операционная система выполняет это чередование посредством механизма, известного как *контекстное переключение*.

Операционная система отслеживает информацию, которая нужна процессу для правильного выполнения. Состояние, известное как контекст, содержит такую информацию, как текущее значение счетчика команд, регистрационного файла и содержимое основной памяти. В любой конкретный момент времени в системе выполняется только один процесс. Когда операционная система принимает решение передать управление от текущего процесса некоторому новому процессу, она совершает контекстное переключение, запоминая контекст текущего процесса и восстанавливая контекст нового процесса с последующей передачей управления новому процессу. Новый процесс возобновляет выполнение точно с того места, в котором он его прервал. Рис. 1.10 служит иллюстрацией этой базовой идеи на примере нашего сценария `hello`.

В рассматриваемом нами примере сценария существуют два процесса: процесс оболочки и процесс `hello`. Первоначально система выполняет только один процесс, а именно — процесс оболочки, ожидая ввода в командную строку. Когда мы обращаемся к нему с требованием выполнить программу `hello`, оболочка выполняет наше

требование, вызывая специальную функцию, так называемый *системный вызов*, который передает управление операционной системе. Операционная система сохраняет контекст оболочки, создает новый процесс *hello* и его контекст, а затем передает управление новому процессу *hello*. После завершения *hello* операционная система восстанавливает контекст процесса оболочки и возвращает ему управление, после чего он ждет ввода следующей команды в командную строку.

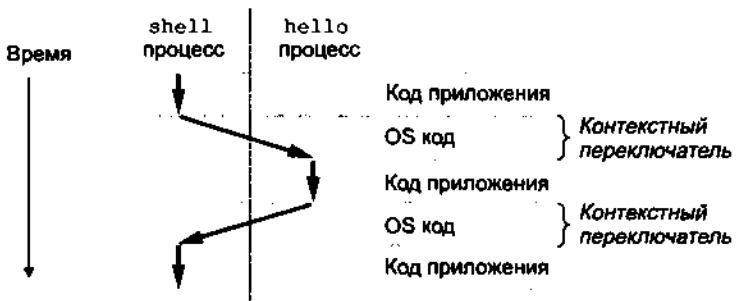


Рис. 1.10. Контекстное переключение процессов

Реализация абстракции процесса требует тесного взаимодействия аппаратных средств нижнего уровня и программ операционной системы. В главе 8 мы выясним, как это делается, а также то, как прикладные программы могут создавать свои собственные процессы и управлять ими.

Одно из осложнений, вносимых реализацией понятия процесс, состоит в том, что при чередовании различных процессов искажается представление о времени, в результате чего программисты испытывают существенные трудности получения точных и систематических данных о времени выполнения их программ. В главе 9 проводится анализ различных способов представления времени в современных системах, и дано описание технологий проведения точных измерений времени.

1.7.2. Потоки

Мы часто представляем себе процесс, как последовательность действий, имеющую единственную управляющую логику, тем не менее, в современных системах процесс фактически может состоять из множества исполнительных элементов, именуемых *потоками* (или нитями), каждая нить выполняется в контексте процесса, использует совместно с процессом те же программные коды и глобальные данные. Роль потоков, как программных моделей, непрерывно возрастает в связи с требованиями параллелизма (одновременности), предъявляемыми сетевыми серверами, поскольку проще организовать совместное использование данных несколькими потоками, чем несколькими процессами, а также в силу того, что потоки обычно значительно эффективнее процессов. Базовое понятие параллелизма, включая организацию поточной обработки, описано в главе 13.

1.7.3. Виртуальная память

Виртуальная память есть абстракция, которая порождает в каждом процессе иллюзию, что лишь он один использует основную память. Каждый процесс имеет одно и то же представление о памяти, которое известно как его *виртуальное адресное пространство*. Виртуальное адресное пространство для процессов операционной системы Linux представлено на рис. 1.11. (Другие Unix-подобные системы используют ту же топологию.) В системе Linux верхняя четверть адресного пространства резервируется для программных кодов и данных этой операционной системы, которая является общей для всех процессов. В нижних трех четвертях адресного пространства содержатся программные коды и данные, порождаемые процессами пользователей. Обратите внимание на тот факт, что адреса на схеме увеличиваются снизу вверх.



Рис. 1.11. Виртуальное адресное пространство процесса

Пространство виртуальных адресов, с точки зрения каждого процесса, состоит из некоторого числа четко определенных областей, каждая из которых выполняет свою задачу. Эти области будут подробно описаны далее в этой книге, однако будет полезно рассмотреть сейчас каждую из них, начиная с низших адресов и продвигаясь в направлении возрастания адресов.

- **Программные коды и данные.** Программный код начинается с одного и того же адреса, за ним следуют ячейки памяти, соответствующие глобальным переменным языка. Области программных кодов и данных инициализируются непосредственно содержимым исполняемого объектного файла, в нашем случае это исполняемый файл *hello*. Более подробно эта часть адресного пространства будет описана в главе 7, в которой мы будем изучать редактирование связей и загрузку.

- **Динамическая память.** Непосредственно за программными кодами и данными следует область динамической памяти программы. В отличие от областей программных кодов и данных, размеры которых фиксируются, как только процесс начнет выполняться, динамическая память может расширяться и сокращаться в размерах во время выполнения программы, как результат обращения к стандартной библиотеке программ, таких как `malloc` и `free`. Мы продолжим подробное изучение динамической памяти после того, как рассмотрим в главе 10 вопросы управления виртуальной памяти.
- **Совместно используемые библиотеки.** Примерно половина адресного пространства представляет собой область, в которой хранятся программные коды и данные для совместно используемых библиотек, таких как библиотека стандартных программ на С или библиотека математических программ. Понятие совместно используемой библиотеки является мощным, но в то же время несколько трудным понятием. Вы узнаете, как нужно с ними работать, когда мы приступим к изучению динамического связывания в главе 7.
- **Стек.** В верхней части виртуального адресного пространства находится стек пользователя, который используется компилятором для реализации вызовов функций. Как и динамическая память, стек пользователя может динамически расширяться и сокращаться в размерах во время исполнения программы. В частности, каждый раз, когда мы вызываем какую-либо функцию, размер стека возрастает. Каждый раз, когда мы возвращаемся из функции, он сокращается. В главе 3 вы узнаете, как компилятор использует стек.
- **Виртуальная память ядра.** Ядро есть часть операционной системы, которое постоянно находится в основной памяти. Верхняя четверть адресного пространства зарезервирована для ядра. Прикладным программам запрещено читать содержимое этой области и заносить в нее записи или непосредственно вызывать функции, записанные в кодах ядра.

Чтобы виртуальная память работала, требуется сложное взаимодействие аппаратных средств с программами операционной системы, включая аппаратную трансляцию каждого адреса, порожденного процессором. Эта базовая идея заключается в том, чтобы сохранить содержимое виртуальной памяти процессора на диске, а затем использовать основную память как кэш-память для диска. В главе 10 показано, как работает этот механизм, и почему это так важно для функционирования современных систем.

1.7.4. Файлы

Файл есть некоторая последовательность, не более и не менее. Каждое устройство ввода-вывода, в том числе диски, клавиатуры, устройства отображения и даже вычислительные сети, моделируется соответствующим файлом. Все операции ввода-вывода системы выполняются путем считывания и записи файлов посредством нескольких системных вызовов, известных как вводы-выводы системы Unix.

Это простое и элегантное понятие файла, тем не менее, оно обладает глубоким смыслом, поскольку обеспечивает унифицированное представление всего разнообразия

файлов, которые могут входить в состав системы. Например, прикладные программисты, манипулирующие содержимым дискового файла, абсолютно не знакомы с дисковой технологией и при этом чувствуют себя вполне комфортно. Более того, одна и та же программа будет выполняться в различных системах, которые исповедуют различные дисковые технологии. Вводы-выводы системы Unix рассмотрены в главе 11.

Отступление. Проект Linux

В августе 1991 года финский аспирант по имени Линус Торвальдс (Linus Torvalds) скромно объявил о завершении разработки ядра новой Unix-подобной операционной системы:

От: torvalds@klaava.Helsinki.FI (Linus Benedict Torvalds)

Сетевая конференция: comp.os.minix

Тема: Что бы вы хотели прежде всего видеть в minix?

Резюме: ограниченный опрос, касающийся моей новой операционной системы

Дата: 25 августа 91 20:57:08 по Гринвичу

Вниманию всех, кто пользуется системой minix:

Я разрабатываю (бесплатную) операционную систему (это всего лишь хобби, система небольшая и спроектирована непрофессионально, в отличие от GNU) для персональных компьютеров AT 386/486. Проект вырвался с апреля, сейчас он приобретает законченный вид. Я хотел бы узнать мнение людей, работающих с minix (одобряют или не одобряют мою систему), поскольку моя операционная система в какой-то степени напоминает minix (то же физическое размещение файловой системы, в силу практических причин, наряду с другими общими чертами).

В настоящий момент я перенес программы bash(1.08) и gcc(1.40), и как ни странно, они работают. Это означает, что через несколько месяцев мне удастся получить кое-что полезное, и мне хотелось бы знать, что бы хотело видеть большинство в моем программном продукте. Благодарен за любые предложения, в то же время я не обещаю, что все выполню.

Linus (torvalds@kruuna.helsinki.fi)

Все остальное, как говорится, уже стало историей. Операционная система Linux стала техническим и культурным явлением. Объединившись с проектом GNU, проект Linux позволил получить полную, совместимую со стандартами Posix версию операционной системы Unix, включая ядро и всю поддерживающую его инфраструктуру. Система Linux успешно работает в широком диапазоне компьютеров, от карманных компьютеров до универсальных вычислительных машин. Группа разработчиков компании IBM умудрилась перенести ее в наручные часы!

1.8. Обмен данных в сетях

До этого момента в нашем экскурсе мы рассматривали системы как изолированную совокупность аппаратных и программных средств. На практике современные системы часто соединены с другими системами посредством компьютерных систем. С точки зрения отдельной системы, сеть можно рассматривать как еще одно устрой-

ство ввода-вывода, как показано на рис. 1.12. Когда система копирует некоторую последовательность байтов из основной памяти в сетевой адаптер, потоки данных устремляются через сеть в другую машину, а не, скажем, в локальный накопитель на магнитных дисках. Аналогично, система может читать данные, отправленные с других машин, и копировать эти данные в свою основную память.

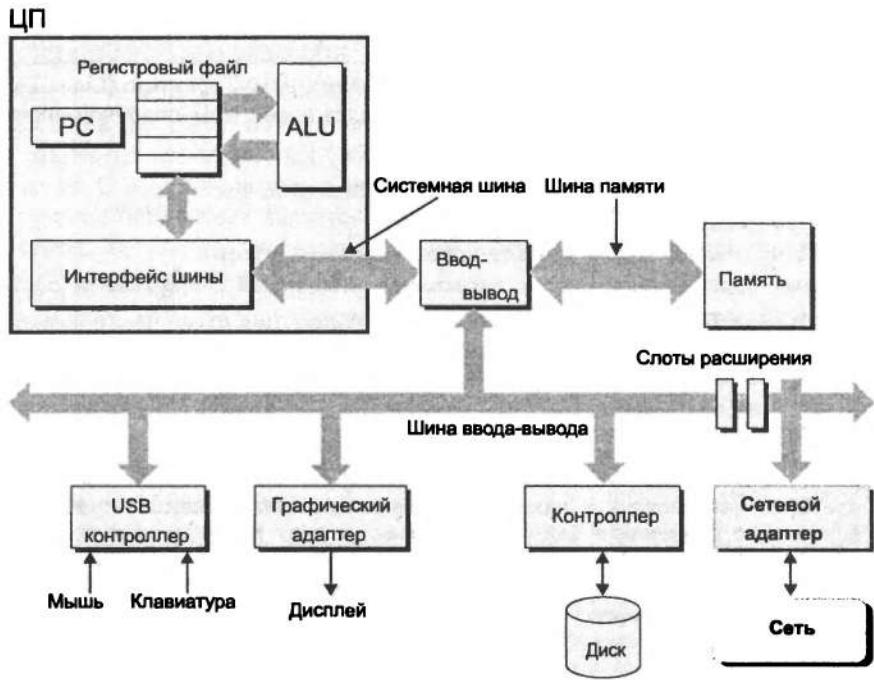


Рис. 1.12. Сеть представляет собой еще одно устройство ввода-вывода

С пришествием глобальных сетей, таких как Internet, копирование информации с одной машины стало одним из наиболее важных применений компьютерных сетей. Например, такие приложения, как электронная почта, обмен сообщениями, FTP (File Transfer Protocol — протокол передачи файлов) и сетевой теледоступ, основаны на копировании информации в сети.

Возвращаясь к нашему примеру `hello`, мы можем воспользоваться знакомым приложением сетевого доступа, чтобы выполнить программу `hello` на удаленной машине. Предположим, что мы воспользовались клиентом сетевого доступа, исполняемым на нашей машине, для подключения к серверу сетевого доступа на удаленной машине. После того как мы зарегистрировались на удаленной машине и запустили оболочку, удаленная оболочка ждет, когда поступит команда ввода. С этого момента дистанционный процесс программы `hello` требует выполнения пяти базовых действий, представленных на рис. 1.13.

После того как мы введем строку "hello" для клиента сетевого доступа и нажмем клавишу ввода, клиент пересыпает эту строку в сервер сетевого доступа. После того

как сервер получит эту строку из сети, он передаст ее программной оболочке. Далее удаленная оболочка выполнит программу `hello` и возвратит выходную строку серверу. В завершение, сервер направляет выходную строку клиенту через сеть, который отображает ее на нашем локальном терминале.

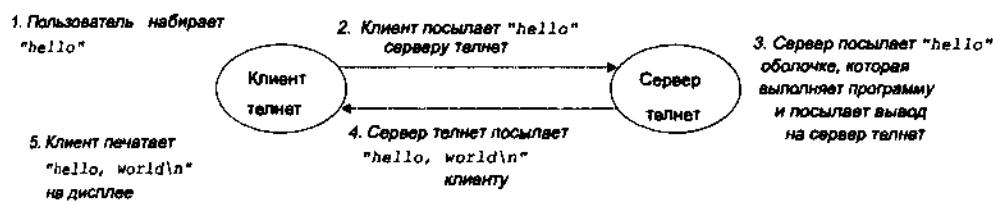


Рис. 1.13. Использование сетевого доступа для выполнения программы `hello` на удаленной машине

Такой тип обмена между клиентами и серверами характерен для всех сетевых приложений. В главе 12 вы узнаете, как создавать сетевые приложения и применять полученные знания для построения простых Web-серверов.

1.9. Следующие шаги

На этом мы завершаем наш первый экскурс в компьютерные системы. Важная причина отказаться от дальнейшего обсуждения заключается в том, что система — это нечто большее, чем только аппаратные средства. Это скорее переплетение аппаратных средств и системного программного обеспечения, которые должны взаимодействовать для достижения окончательной цели выполнения прикладных программ. Все дальнейшее содержание книги посвящено этой теме.

1.10. Резюме

Вычислительную систему составляют аппаратные и программные средства, которые взаимодействуют с целью выполнения прикладных программ. Информация внутри компьютера представлена в виде групп битов, которые интерпретируются в зависимости от контекста. Программы транслируются другими программами в различные формы, сначала они представлены в виде текстов в кодах ASCII, затем они преобразуются компиляторами и редакторами связей в исполняемые файлы.

Процессоры читают и интерпретируют двоичные инструкции, которые размещены в основной памяти. Поскольку большую часть своего времени компьютеры тратят на копирование данных из памяти, с устройств ввода-вывода и регистров центрального процессора, устройства памяти системы организованы в некоторую иерархию, в верхней части которой находятся регистры центрального процессора, далее следуют несколько уровней аппаратно реализованной кэш-памяти, основная DRAM-память и дисковая память. Устройства памяти, расположенные в иерархии выше, обладают большим быстродействием и стоят дороже в пересчете на один бит, чем те, которые

находятся в иерархии ниже. Устройства памяти, которые расположены в иерархии выше, служат кэш-памятью для устройств памяти, расположенных ниже. Программисты имеют возможность оптимизировать производительность своих программ на языке С, изучив и воспользовавшись особенностями иерархии памяти.

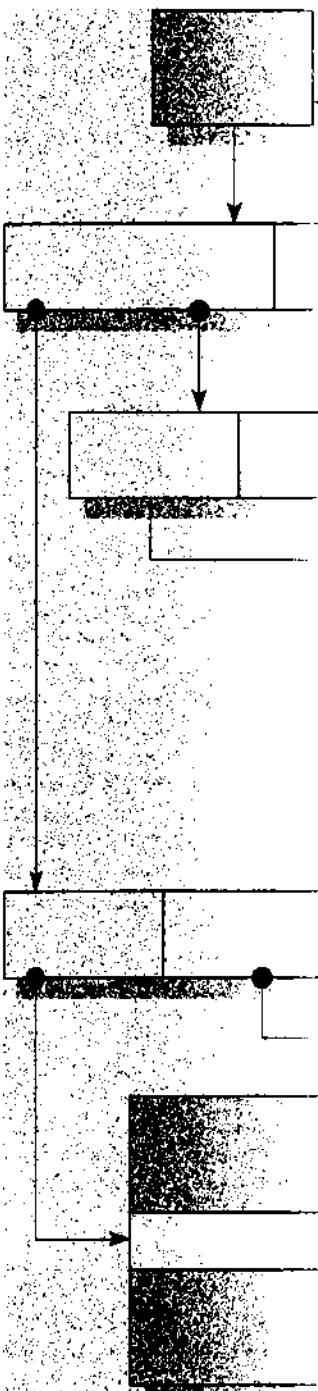
Ядро операционной системы служит посредником между приложением и аппаратными средствами. Оно реализует три фундаментальных абстракций:

- Файлы — это абстракции по отношению к устройствам ввода-вывода.
- Виртуальная память — абстракция как в отношении основной памяти, так и в отношении дисков.
- Процессы — абстракции в отношении процессоров, основной памяти и устройств ввода-вывода.

Наконец, сети предоставляют компьютерным системам возможность обмена данными между собой. С точки зрения конкретной системы, сеть есть не что иное, как устройство ввода-вывода.

Библиографические заметки

Риччи (Ritchie) написал интересные и достоверные отчеты о первых шагах языка С и системы Unix [63, 64]. Риччи и Томпсон (Ritchie and Thompson) впервые опубликовали отчет о системе. Зильбершатц и Гейвин (Silberschatz and Gavin) [70] представили всестороннее описание различных особенностей системы. Web-страницы проекта GNU (www.gnu.org) и операционной системы Linux (www.linux.org) содержат текущую информацию и информацию исторического характера. К сожалению, информация по стандартам Posix недоступна в оперативном режиме. Ее нужно заказывать за дополнительную плату в институте IEEE (standards.ieee.org).



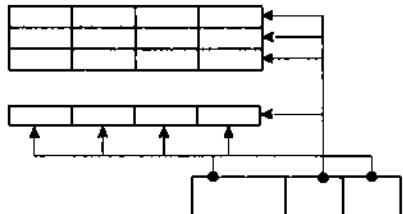
ЧАСТЬ I

Структура и выполнение программы

Исследование компьютерных систем начинается с изучения самого компьютера, состоящего из процессора и подсистемы памяти. По сути дела, нам требуются способы представления основных типов данных как аппроксимации целого и вещественного чисел. Отсюда появится возможность рассмотрения того, как команды машинного уровня обрабатывают эти данные, и как компилятор переводит программы на языке С в команды. Далее будут изучены отдельные процедуры обращения к процессору для лучшего понимания использования аппаратных ресурсов при выполнении команд. Поняв суть работы компиляторов на уровне машинного кода, можно повысить производительность программы написанием эффективно компилируемого исходного текста. Будет рассмотрено моделирование машинной памяти, одного из наиболее сложных компонентов современных компьютерных систем.

Данная часть книги обеспечит читателей глубоким пониманием машинного представления и исполнения прикладных программ. Приобретенные навыки помогут написать стабильные программы, максимально используя компьютерные ресурсы.

ГЛАВА 2



Представление информации и работа с ней

- Хранение информации.
 - Целочисленное представление.
 - Целочисленная арифметика.
 - Числа с плавающей точкой.
 - Резюме.
-

Современные компьютеры хранят и обрабатывают информацию, представленную в виде двоичных сигналов. Эти скромные двоичные знаки, или *биты*, формируют основу цифровой революции. Всем знакомая десятичная система исчисления использовалась в течение 1000 лет; изобретена она была в Индии, в XII веке арабские математики ее усовершенствовали, а на Западе она появилась в XIII веке "с помощью" итальянского математика Леонардо Пизано, более известного под именем Фибоначчи. Использование десятичного исчисления естественно для людей, у которых на руках по десять пальцев, однако при создании машин для хранения и обработки информации более приемлемы двоичные величины. Двоичные сигналы легко представлять, хранить и передавать, к примеру, как наличие или отсутствие отверстия в перфоленте, как высокое или низкое напряжение в электрической цепи, или как электромагнит, ориентированный по часовой стрелке или против нее. Электронные схемы для хранения и осуществления расчетов по двоичным сигналам очень просты и надежны; они позволяют производителям объединять миллионы таких схем в одной кремниевой микросхеме.

Сам по себе одиночный бит не представляет особой ценности. Однако при объединении битов в группы и применения специфической интерпретации, придающей определенное значение различным комбинациям битов, можно представить элементы любого конечного множества. Например, используя двоичную числовую систему, группы битов можно использовать для кодировки неотрицательных чисел. С помощью стандартного кода символов можно кодировать буквы и символы в документе. В данной главе рассматриваются оба этих вида кодировок, а также кодировки для представления отрицательных чисел и для приближения вещественных чисел.

Здесь авторами рассматриваются три важнейшие кодировки чисел. Кодировки без знака основаны на традиционном двоичном представлении чисел больше нуля или равных нулю. Кодировки в дополнительном коде являются наиболее распространенным способом представления целых чисел со знаком, которые могут быть как положительными, так и отрицательными. Кодировки с плавающей точкой — это двоичная версия научного обозначения для представления вещественных чисел. С помощью различных представлений компьютеры выполняют арифметические операции, например, сложение и умножение, подобные соответствующим операциям с целыми и вещественными числами.

Для кодировки числа компьютерные представления используют ограниченное число битов, поэтому некоторые операции могут вызывать *переполнение*, когда числа оказываются слишком большими, чтобы их было возможно представить. Это может привести к поразительным результатам. Например, на большинстве современных компьютеров расчет выражения $200 * 300 * 400 * 500$ дает $-884\ 901\ 888$. Таково вычисление по правилам фиксированной арифметики — расчет произведения положительных чисел привел к отрицательному результату.

С другой стороны, фиксированная арифметика удовлетворяет многим известным правилам арифметики целых чисел. Например, ассоциативность и коммутативность умножения: вычисление любого из следующих выражений на языке С дает $-884\ 901\ 888$:

```
(500 * 400) * (300 * 200)
((500 * 400) * 300) * 200
((200 * 500) * 300) * 400
400 * (200 * (300 * 500))
```

Компьютер может и не выдать ожидаемого результата, но он, по крайней мере, последователен!

Арифметика с плавающей точкой обладает совершенно другими математическими свойствами. Произведение множества положительных чисел всегда будет положительным, хотя переполнение выдаст особую величину: $+\infty$. С другой стороны, арифметика с плавающей точкой не ассоциативна из-за конечной точности представления. Например, представление выражения $(3.14+1e20)-1e20$ на большинстве машин составит 0.0, тогда как $3.14+(1e20-1e20)$ составит 3.14.

Путем изучения фактических числовых представлений можно понять диапазоны величин, которые могут быть представлены, а также свойства различных арифметических операций. Понимание этого принципиально для написания программ, работающих корректно во всем диапазоне числовых величин и переносимых на разные конфигурации машин, операционных систем и компиляторов.

Для кодировки числовых величин в компьютерах используется несколько различных двоичных представлений. По мере углубления в тему программирования на машинном уровне, описываемом в главе 3, читателю будет необходимо ознакомиться с данными представлениями. Кодировки описываются в данной главе и обеспечивают читателю некоторое практическое обоснование систем исчисления.

Для выполнения математических операций непосредственно на уровне битов разработаны несколько способов. Понимание этих методик чрезвычайно важно для понимания кода на машинном уровне, генерированного при компиляции арифметических выражений.

Авторами предлагается математическая трактовка материала. Вначале даются основные определения кодировок, после чего выводятся такие свойства, как диапазон представимых чисел, их представления на двоичном уровне, а также свойства арифметических операций. Авторы полагают, что данный материал будет полезно рассматривать с такой абстрактной точки зрения потому, что программистам необходимо обладать четким пониманием, как компьютерная арифметика соотносится с более знакомой арифметикой целых и вещественных чисел. Это может показаться отпугивающим, однако математическая трактовка требует только знания основ алгебры. Для укрепления связей между формальной трактовкой и некоторыми примерами из реальной жизни рекомендуем решать практические упражнения.

Как читать данную главу

Если какие-либо уравнения и формулы приведут читателя в уныние, не стоит бросать попыток извлечь как можно больше пользы из этой главы! Для полноты здесь приводятся полные выкладки математических идей, однако наилучшим способом для первоначального ознакомления с материалом является их пропуск. Вместо этого попробуйте для развития интуиции разобрать несколько простых примеров (упражнений), а затем проверьте, как математический вывод подкрепляет интуицию.

Язык программирования C++ построен на базе С и использует точно такие же представления данных и операции. Все сказанное в данной главе о С не в меньшей степени относится и к C++. Определение же языка Java создало новый набор стандартов для представления данных и операций. Если стандарт С спроектирован с целью широкого применения, то стандарт Java — довольно специфичен в форматах и кодировках данных. В этой главе в нескольких местах представления и операции, поддерживаемые Java, выделены особо.

2.1. Хранение информации

Вместо доступа к отдельным битам в памяти, в большинстве компьютеров в виде наименьших элементов памяти используются блоки по 8 битов — *байты*. Программа машинного уровня рассматривает память как очень большой массив байтов, называемый *виртуальной памятью*. Каждый байт памяти имеет уникальный номер, называемый *адресом*, а множество всех возможных адресов называется *виртуальным адресным пространством*. Судя по названию, виртуальное адресное пространство — это всего лишь концептуальный образ программы на машинном уровне. Фактическое внедрение, описанное в главе 10, использует комбинацию оперативного запоминающего устройства (RAM), дисковую память, специальные аппаратные средства и программные средства операционной системы для обеспечения программы так называемым непрерывным массивом байтов.

Одной из задач компилятора и системы поддержки выполнения программ является разделение пространства памяти на управляемые сегменты для хранения различных программных объектов, т. е. данных программы, команд и информации управления. Для распределения и управления памятью разных частей программы используются различные механизмы. Все это управление осуществляется в рамках виртуального адресного пространства. Например, значение указателя в С, указывает ли он на целое число, структуру или на какой-либо другой элемент программы, является виртуальным адресом первого байта некоего блока памяти. Компилятор С также ассоциирует информацию о типе с каждым указателем, поэтому он может генерировать различные коды машинного уровня для доступа к значению, хранящемуся в ячейке, обозначенной указателем, в зависимости от типа этого значения. Несмотря на то, что компилятор С поддерживает информацию о типе, фактически генерированная им программа машинного уровня не несет информации о типах данных. Каждый объект программы выступает просто как блок байтов, а сама программа — как последовательность байтов.

Роль указателей в С

Указатели являются основной особенностью С. Они обеспечивают механизм ссылок на элементы структур данных, включая массивы. Подобно переменной, указатель имеет два параметра: *величину* и *тип*. Величина указывает на местоположение (ячейку) определенного объекта, а тип — на то, что за объект (например, целое число или число с плавающей точкой) сохранен в этой ячейке.

2.1.1. Шестнадцатеричная система исчисления

Один байт состоит из восьми битов. В двоичной системе интервал его значений от 00000000 до 11111111. Для десятичного целого числа диапазон от 0 до 255. Двоичное представление слишком громоздкое, а для десятичного очень утомительно выполнять преобразования в битовые комбинации и обратно. Вместо всего этого битовые комбинации записываются как шестнадцатеричные числа. В шестнадцатеричной системе используются числа от 0 до 9 и буквы от А до F. В табл. 2.1 показаны десятичное и двоичное значения шестнадцатеричных знаков. При записи в шестнадцатеричной системе значение каждого байта в диапазоне от 00 до FF.

Таблица 2.1. Шестнадцатеричная система

Шестнадцатеричное число	0	1	2	3	4	5	6	7
Десятичная величина	0	1	2	3	4	5	6	7
Двоичная величина	0000	0001	0010	0011	0100	0101	0110	0111
Шестнадцатеричное число	8	9	A	B	C	D	E	F
Десятичная величина	8	9	10	11	12	13	14	15
Двоичная величина	1000	1001	1010	1011	1100	1101	1110	1111

Численные константы, начинающиеся с 0х или 0Х, интерпретируются как шестнадцатеричные. Буквы от А до F можно использовать как прописные, так и строчные. Например, можно записать число FA1D37B как 0xFA1D37B, как 0xfa1d37b или даже смешивая верхний и нижний регистры написания, например, 0xFa1D37b. В данной книге обозначения С будут использоваться для представления шестнадцатеричных величин.

Обычной задачей при работе с программами машинного уровня является преобразование вручную между десятичным, двоичным и шестнадцатеричным представлениями битовых комбинаций. Преобразование между двоичным и шестнадцатеричным представлениями является непосредственным, поскольку оно может осуществляться только по одному шестнадцатеричному знаку за один раз. Одним из простых способов выполнения преобразований в уме является запоминание десятичных эквивалентов шестнадцатеричных чисел А, С и F. Шестнадцатеричные величины B, D и E можно перевести в десятичные оценкой их величин относительно первых трех.

Например, предположим, что дано число 0x17A4C. Его можно преобразовать в двоичный формат путем расширения каждого шестнадцатеричного числа следующим образом:

шестнадцатеричное	1	7	3	A	4	C
двоичное	0001	0111	0011	1010	0100	1100

Это дает двоичное значение 000101110011101001001100.

И наоборот, имея двоичное число 1111001010110110110011, преобразовать его в шестнадцатеричное можно с первоначальным разбиением его на группы по четыре бита каждая. Однако заметьте, что если общее число битов не кратно четырем, тогда самую левую группу следует сделать размером меньше четырех битов, эффективно заполняя число первыми нулями. Затем, каждая группа из четырех битов переводится в соответствующее шестнадцатеричное число:

двоичное	11	1100	1010	1101	1011	0011
шестнадцатеричное	3	С	А	D	B	3

УПРАЖНЕНИЕ 2.1

Выполните следующие преобразования чисел:

1. 0x8F7A93 — в двоичное.
2. Двоичное 1011011110011100 — в шестнадцатеричное.
3. 0xC4E5D — в двоичное.
4. Двоичное 110101101101111100110 — в шестнадцатеричное.

Когда величина x — степень двойки, т. е. $x = 2^n$ для некоторого n , тогда x можно легко записать в шестнадцатеричной форме, помня, что двоичное представление x — это просто 1, за которой следует n нулей. Шестнадцатеричное число 0 представляет со-

бой четыре двоичных первых нуля. Поэтому для n , записанного в форме $i + 4j$, где $0 \leq i \leq 3$, x можно записать с первым шестнадцатеричным числом 1 ($i = 0$), 2 ($i = 1$), 4 ($i = 2$) или 8 ($i = 3$), за которым следует j шестнадцатеричных нулей. В качестве примера для $x = 2048 = 2^{11}$ имеем $n = 11 = 3 + 4 \cdot 2$, что дает шестнадцатеричное 0x800.

УПРАЖНЕНИЕ 2.2

Заполните пустые клетки в таблице, представляя двоичное и шестнадцатеричное разных степеней двух:

Степень	Десятичное	Шестнадцатеричное
11	2048	0x800
7		
	8192	
		0x20000
16		
	256	
		0x20

В общем случае преобразование между десятичным и шестнадцатеричным требует использования операций умножения или деления. Для преобразования десятичного числа x в шестнадцатеричное можно многократно делить x на 16, получая частное q и остаток r , так, что $x = q \cdot 16 + r$. Затем шестнадцатеричное число, представляющее r , доводится повторением процесса с q до наименьшего значимого. В качестве примера рассмотрим преобразование десятичного числа 314156:

$$314156 = 19634 \cdot 16 + 12 \quad (\text{с})$$

$$19634 = 1227 \cdot 16 + 2 \quad (\text{2})$$

$$1227 = 76 \cdot 16 + 11 \quad (\text{в})$$

$$76 = 4 \cdot 16 + 12 \quad (\text{с})$$

$$4 = 0 \cdot 16 + 4 \quad (\text{4})$$

Из получившегося результата можно рассчитать шестнадцатеричное 0x4CB2C.

И наоборот, для преобразования шестнадцатеричного числа в десятичное можно умножить каждое из шестнадцатеричных чисел на соответствующую степень 16. Например, для числа 0x7AF его десятичный эквивалент рассчитывается следующим образом: $7 \cdot 16^2 + 10 \cdot 16 + 15 = 7 \cdot 256 + 10 \cdot 16 + 15 = 1792 + 160 + 15 = 1967$.

УПРАЖНЕНИЕ 2.3

Один байт можно представить двумя шестнадцатеричными числами. Заполните пустые клетки таблицы, присваивая десятичные, двоичные и шестнадцатеричные величины разным комбинациям байтов:

Десятичная	Двоичная	Шестнадцатеричная
0	00000000	00
55		
136		
243		
	01010010	
	10101100	
	11100111	
		A7
		3E
		BC

О преобразовании между десятичной и шестнадцатеричной системами

При преобразовании больших чисел из десятичного в шестнадцатеричное, и наоборот, лучше всего поручить эту работу калькулятору компьютера. Например, следующий сценарий в языке Perl преобразует список чисел из десятичного счисления в шестнадцатеричное в листинге 2.1.

Листинг 2.1. Преобразования в шестнадцатеричные

```
1 #!/usr/local/bin/perl
2 # Преобразовать список десятичных чисел в шестнадцатеричные
3
4 для ($i = 0; $i < @ARGV; $i++) {
5 printf ("%d\t- 0x%x\n", $ARGV[$i], $ARGV[$i]);
```

Как только этот файл превращен в исполняемый, команда

unix> ./d2h 100 500 751

дает результат:

100 = 0x64
500 = 0x1f4
751 = 0x2ef

Подобным же образом следующий сценарий преобразует шестнадцатеричные числа в десятичные в листинге 2.2.

Листинг 2.2. Преобразование в двоичные

```

1 #!/usr/local/bin/perl
2 # Преобразовать список шестнадцатеричных чисел в десятичные
3
4 для ($i = 0; $i < @ARGV; $i++) {
5   $val = hex ($ARGV[$i]);
6   printf ("0x%x = %d\n", $val, $val);
7 }

```

УПРАЖНЕНИЕ 2.4

Без преобразования чисел из десятичного счисления в двоичное попытайтесь решить следующие арифметические проблемы, давая ответы в шестнадцатеричном счислении. Подсказка: просто модифицируйте методы, используемые для выполнения десятичного сложения и вычитания для использования шестнадцатеричной системы счисления.

0x502c + 0x8 =
 0x502c - 0x30 =
 0x502c + 64 =
 0x50da ~ 0x502c =

2.1.2. Машинные слова

Каждый компьютер имеет *длину машинного слова*, указывающую номинальный размер целого и указателя. Самым важным системным параметром, определяемым длиной слова, является максимальный размер виртуального адресного пространства. То есть, для машины с длиной слова n битов диапазон виртуальных адресов может составлять от 0 до $2^n - 1$, обеспечивая программе доступ максимум к 2^n байтов.

Большинство современных компьютеров имеют размер машинного слова 32 битов. Это ограничивает виртуальное адресное пространство до 4 Гбайт (записывается 4 Гбайт), т. е. свыше 4×10^9 байтов. Несмотря на то, что для многих приложений этого объема более чем достаточно, достигнут предел, когда многие крупные приложения, разработанные для научных целей, или программы баз данных требуют большего объема памяти. Следовательно, по мере снижения стоимости запоминающих устройств, машины высшего класса с длиной слова 64 битов становятся все более распространенными.

2.1.3. Размеры данных

Компьютеры и компиляторы поддерживают форматы множественных данных, используя различные способы кодировки данных, например целых чисел и чисел с плавающей точкой, а также различные длины. Например, многие машины обладают командами для манипуляций отдельными байтами, а также целыми числами длиной 2, 4 и 8 байтов. Они поддерживают числа с плавающей точкой длиной 4 и 8 байтов (табл. 2.2).

Таблица 2.2. Типы данных

Тип в С	Тип процессора 32 битов	Compaq Alpha
char	1	1
short int	2	2
int	4	4
long int	4	8
char *	4	8
float	4	4
double	8	8

Примечание. Число распределенных байтов варьируется в зависимости от машины и компилятора.

Язык С поддерживает смешанные форматы данных, целых и с плавающей точкой. Тип данных `char` имеет длину в один байт. Несмотря на то, что название "`char`" связано с запоминанием одного символа (`character`) в текстовой строке, его также можно использовать для хранения целых величин. Типу данных `int` могут предшествовать спецификаторы `long` и `short` различных длин. В табл. 2.2 показано количество байтов для различных типов данных. Точное количество зависит как от машины, так и от компилятора. Здесь представлены два показательных случая: типичная 32-битовая машина и архитектура Compaq Alpha — 64-битовая машина, рассчитанная на самые современные программные приложения. Обратите внимание, что короткое целое имеет длину 2 байта, тогда как обычное `int` — 4 байта. Длинное целое имеет длину машинного слова.

В табл. 2.2 также показано, что указатель (переменная типа `char *`) занимает полное машинное слово. Большинство машин поддерживают два различных формата плавающей точки: обычной точности, обозначаемый `float`, и двойной точности `double`. Эти форматы используют 4 и 8 байтов, соответственно.

Объявление указателей

Для любого типа данных *T* объявление

*T***p*:

указывает на то, что *p* — это переменная указателя, указывающего на объект типа *T*. Например,

`char *p;`

— объявление указателя на объект типа `char`.

2.1.4. Адресация и упорядочение байтов

Для программных объектов непрерывного распределения необходимо установить два правила: каков будет адрес объекта, и как можно располагать байты в памяти. Виртуально во всех машинах такой объект сохраняется как непрерывная последовательность байтов с адресом, равным наименьшему адресу занятых байтов. Например, предположим, что переменная *x* типа *int* имеет адрес 0x100, т. е. значение адресного выражения *&x* — 0x100. Тогда четыре байта *x* будут сохранены в ячейках памяти 0x100, 0x101, 0x102 и 0x103.

Для упорядочения байтов объекта существуют два общепринятых правила. Рассмотрим целое число длиной *w* битов, имеющее битовое представление $[x_{w-1}, x_{w-2}, \dots, x_0]$, где x_{w-1} — наиболее значимый бит, а x_0 — наименее значимый. Если предположить, что *w* кратно восьми, то эти биты можно сгруппировать в байты, где наиболее значимый байт будет иметь биты $[x_{w-8}, x_{w-16}, \dots, x_{w-24}]$, наименее значимый — биты $[x_7, x_6, \dots, x_0]$, а другие байты будут иметь биты середины. В одних машинах объект сохраняется в памяти, упорядоченной от наименее значимого байта к наиболее значимому, а в других — наоборот. Первое правило — когда наименее значимый байт идет первым — называется правилом *остроконечников*. Этому правилу следует большинство машин бывшей Digital Equipment Corporation (теперь являющейся частью Compaq Corporation), а также Intel. Второе правило — когда наиболее значимый байт идет первым — называется правилом *тупоконечников*. Этого правила придерживается большинство машин от IBM, Motorola и Sun Microsystems. Обратите внимание на слово "большинство". Разделение этих правил по компаниям условно. Например, в персональных компьютерах производства IBM применяются процессоры, совместимые с Intel, следовательно, они являются остроконечниками. Многие микропроцессорные платы, включая Alpha и PowerPC от Motorola, работают в обоих режимах, когда правило упорядочения байтов определяется при подключении к микросхеме питания.

Продолжая разговор на приведенном примере, представим переменную *x* типа *int* с адресом 0x100, имеющую шестнадцатеричное значение 0x1234567. Упорядочение байтов в диапазоне от 0x100 до 0x103 зависит от типа машины:

Тупоконечники

	0x100	0x101	0x102	0x103	
...	01	23	45	67	...

Остроконечники

	0x100	0x101	0x102	0x103	
...	67	45	23	01	...

Обратите внимание, что в слове 0x1234567 старший байт имеет шестнадцатеричное значение 0x01, тогда как младший байт 0x67.

Как правило, всегда возникает много споров относительно того, которое из упорядочений байтов является подходящим. На самом деле, термины "остроконечник" и "ту-

поконечник" заимствованы из книги Джонатана Свифта "Путешествия Гулливера", где описывается вражда двух группировок по поводу того, с какого конца разбивать сваренное всмятку яйцо: с тупого или острого. Точно так же, как и в случае с пресловутым яйцом, не существует технологического смысла ставить один способ упорядочения байтов выше другого и, отсюда, вся полемика сводится к банальному пикированию на общественно-политические темы. Пока будет выбираться один из двух способов, и его приверженцы будут последовательны, до тех пор в этом смысле выбор и будет произвольным.

О происхождении термина "конечники"

Вот как в 1726 году Джонатан Свифт описывал историю противостояния тупо- и остроконечников:

...Лиллипутия и Блефуску... Эти две могущественные державы ведут между собой ожесточенную войну в продолжение тридцати шести лун. Поводом к войне послужили следующие обстоятельства. Всеми разделяется убеждение, что вареные яйца при употреблении их в пищу испокон веков разбивались с тупого конца; но дед нынешнего императора, будучи ребенком, порезал себе палец за завтраком, разбив яйцо означенным древним способом. Тогда император, отец ребенка, обнародовал указ, предписывающий всем его подданным под страхом строгого наказания разбивать яйца с острого конца. Этот закон до такой степени озлобил население, что, по словам наших летописей, был причиной шести восстаний, во время которых один император потерял жизнь, а другой — корону. Мятежи эти постоянно разжигались монархами Блефуску, а после их подавления изгнанники всегда находили приют в этой империи. Насчитывают до одиннадцати тысяч фанатиков, которые в течение этого времени пошли на казнь, лишь бы не разбивать яйца с острого конца. Были напечатаны сотни огромных томов, посвященных этой полемике, но книги Тупоконечников давно запрещены, и вся партия лишена законом права занимать государственные должности.

В свое время Свифт зло ironизировал по поводу непрекращающихся стычек между Англией (Лиллипутия) и Францией (Блефуску). Дэнни Коэн, родоначальник сетевых протоколов, впервые применил эти термины для описания упорядочения байтов [17], после чего они получили весьма широкое распространение.

Вторым случаем, когда упорядочение байтов приобретает важность, является рассмотрение последовательностей байтов, представляющих целочисленные данные. Это часто имеет место при инспекции программ машинного уровня. В качестве примера, в файле текстового представления машинного кода для процессора Intel появляется следующая строка:

```
80483bd: 01 05 64 94 04 08 add    %eax, 0x8049464
```

Эта строка создана *дизассемблером*, инструментом, определяющим последовательность управляющих команд в исполняемом файле программы. В следующей главе эти инструментальные средства, а также способы интерпретации подобных строк, будут рассматриваться более подробно. На данном этапе просто заметим, что эта строка указывает на то, что шестнадцатеричная последовательность байтов 01 05 64 94 04 08 — это представление на уровне байтов команды, добавляющей 0x8049464 к

некоторой программной величине. Если взять последние четыре байта данной последовательности 64 94 04 08 и записать их в обратном порядке, то получится 08 04 94 64. Если не обращать внимания на предшествующий ноль, то получится величина 0x8049464, записанная в правой части. То, что байты отображаются в обратном порядке, обычное дело при считывании машинного кода остроконечников. Нормальный способ записи последовательности байтов следующий: байт с наименьшим числом — слева, с наибольшим — справа, хоть это и противоречит обычному способу написания чисел с наиболее значимой цифрой слева, а с наименее значимой — справа.

В третьем случае упорядочение байтов становится видимым, когда написанные программы, что называется, "обходят" систему нормального типа. В языке C это можно сделать с использованием приведения (cast) для обеспечения ссылки на объект, в соответствии с различными типами данных, из которых он был создан. В общем и целом, подобные "штучки", мягко говоря, не поощряются большинством программистов, однако они могут быть полезными и даже необходимыми для программирования на системном уровне.

В листинге 2.3 показан код C, в котором используется приведение типа для доступа и распечатки байтовых представлений различных программных объектов. Для определения типа данных `byte_pointer` в качестве указателя на объект типа `unsigned char` используется `typedef`. Такой байтовый указатель ссылается на последовательность байтов, в которой каждый байт рассматривается как неотрицательное целое число. Первой рутинной процедуре `show_bytes` дается адрес последовательности байтов, обозначенный байтовым указателем и счетчиком байтов. Она распечатывает отдельные байты в шестнадцатеричной системе счисления. Директива форматирования "%.2x" указывает на то, что целое число должно быть распечатано в шестнадцатеричной системе минимум с двумя цифрами.

Листинг 2.3. Код представления байтов программного объекта

```
1 #include <stdio.h>
2
3 typedef unsigned char *byte_pointer;
4
5 void show_bytes (byte_pointer start, int len)
6 {
7     int i;
8     for (i = 0; i < len; i++)
9         printf(" %.2x", start[i]);
10    printf ("\n");
11 }
12
13 void show_int (int x)
14 {
15     show_bytes ( (byte_pointer) &x, sizeof (int));
16 }
```

```
18 void show_float (float x)
19 {
20 show_bytes ((byte_pointer) &x, sizeoff (float));
21 }
22
23 void show_pointer (void *x)
24 {
25 show_bytes ((byte_pointer) &x, sizeoff (void *));
26 }
```

В листинге 2.3 использовано приведение типа для обхождения системы типов.

О присвоении типам данных имени

Объявление `typedef` в С обеспечивает способ присваивания имен типам данных. Это может заметно улучшить читаемость кода, поскольку глубоко вложенные объявления типов порой очень сложно расшифровывать.

Синтаксис `typedef` точно такой же, как и синтаксис объявления переменной, за исключением того, что в данном случае используется имя типа, а не переменной. Поэтому объявление `byte_pointer` в листинге 2.3 имеет ту же форму, какую бы имело объявление переменной для типа `unsigned char`.

Например, объявление

```
typedef int *int_pointer;
int_pointer ip;
```

определяет тип `int_pointer` в качестве первого для `int` и объявляет переменную `ip` этого типа. С другой стороны, эту переменную можно было объявить напрямую как:

```
int *ip;
```

Форматированная печать

Функция `printf` (наряду со своими "сестрами" `fprintf` и `sprintf`) обеспечивает способ распечатки информации с тщательным контролем всех подробностей форматирования. Первое — это *форматирующая строка*, а все остальное — величины, которые необходимо напечатать. В рамках форматирующей строки каждая последовательность символов, начинающаяся с "%", указывает на то, как форматировать следующий аргумент. В набор типичных примеров входит "%d" для печати десятичного целого числа, "%f" — для печати числа с плавающей точкой и "%c" — для печати символа с кодом символа, представленным аргументом.

Указатели и массивы

В функции `show_bytes` (см. листинг 2.3) видна тесная взаимосвязь между указателями и массивами (*подробно рассматривается в разд. 3.8*). Видно, что эта функция имеет аргумент `start` типа `byte_pointer` (определенный как указатель `unsigned char`). Однако ссылка `start[i]` на массив видна в строке 9. В языке С можно разнести указа-

тель и описание массива и сделать ссылку на указатель элементов массива. В данном примере ссылка `start[i]` указывает на то, что необходимо прочитать байт *i* по адресу `start`.

Указатель и разыменование (снятие косвенности)

В строках 15, 20 и 25 листинга 2.3 видно использование двух операций, уникальных в С и C++. "Адрес" оператора `&` создает указатель. Во всех трех строках выражение `&x` создает указатель на ячейку, содержащую переменную *x*. Тип этого указателя зависит от типа *x*, и отсюда эти три указателя приобретают типы `int *`, `float` и `void **`, соответственно. (Тип данных `void *` — это особый тип указателя без ассоциируемой информации о типе.)

Оператор приведения типа преобразует один тип данных в другой. Следовательно, приведение (`byte_pointer`) `&x` указывает на то, что какой бы тип указатель `&x` ни имел ранее, теперь он является указателем на данные типа `unsigned char`.

Запустим программу, показанную в листинге 2.4, на нескольких разных машинах с представлением результатов, как показано в табл. 2.3. Использовались следующие машины:

- Linux — Intel Pentium II с Linux;
- NT — Intel Pentium II с Windows-NT;
- Sun — Sun Microsystems UltraSPARC с Solaris;
- Alpha — Compaq Alpha 21164 с Tru64 Unix.

Листинг 2.4. Примеры байтовых представлений

```

1 void test_show_bytes (int val)
2 {
3     int ival = val;
4     float fval = (float) ival;
5     int *pval = &ival;
6     show_int (ival);
7     show_float (fval);
8     show_pointer (pval);
9 }
```

Аргумент 12,345 имеет шестнадцатеричное представление 0x00003039. Для данных `int` получаем идентичные результаты для всех машин, за исключением упорядочения байтов. Можно заметить, что наименее значимая байтова величина 0x39 в первую очередь распечатывается для Linux, NT и Alpha, указывая на остроконечные машины, а уже потом для Sun, указывая на тупоконечную машину. Подобным же образом идентичны байты данных `float`, исключая упорядочение байтов. С другой стороны, значения указателя абсолютно различны. В разных конфигурациях машин и опера-

ционных систем используются разные правила распределения памяти. Стоит отметить одну особенность: машины с Linux и Sun используют четырехбайтовые адреса, а Alpha — восьмибайтовые.

Обратите внимание на то, что хотя как целочисленные данные, так и данные с плавающей точкой закодировали числовую величину 12,345, они имеют разные байтовые шаблоны: 0x00003039 — для целых чисел и 0x4640E400 — для данных с плавающей точкой.

Таблица 2.3. Байтовые представления разных величин данных

Машинна	Величина	Тип	Байты
Linux	12,345	int	39 30 00 00
NT	12,345	int	39 30 00 00
Sun	12,345	int	00 00 30 39
Alpha	12,345	int	39 30 00 00
Linux	12,345.0	float	00 e4 40 46
NT	12,345.0	float	00 e4 40 46
Sun	12,345.0	float	46 40 e4 00
Alpha	12,345.0	float	00 e4 40 46
Linux	&ival	int *	3c fa ff bf
NT	&ival	int *	1c ff 44 02
Sun	&ival	int *	ef ff fc e4
Alpha	&ival	int *	80 fc ff 1f 01 00 00 00

Результаты для int и float идентичны, за исключением упорядочения байтов. Значения указателей зависят от типа машины.

Вообще, в этих двух форматах используются разные схемы кодировки. Если развернуть показанные шестнадцатеричные шаблоны в двоичную форму и соответствующим образом их сдвинуть, то получится последовательность из 13 совпадающих битов, помеченных звездочками:

```

0 0 0 0 3 0 3 9
00000000000000000000110000001111001
*****  

4 6 4 0 E 4 0 0
0100011001000001110010000000000

```

Это — не случайно. При изучении форматов с плавающей точкой авторы намерены вернуться к данному примеру.

УПРАЖНЕНИЕ 2.5

Рассмотрим три следующих вызова `show_bytes`:

```
byte_pointer valp = (byte_pointer) &val;
show_bytes (valp, 1); /*1.*/
show_bytes (valp, 2); /*2.*/
show_bytes (valp, 3); /*3.*
```

Укажите, какие из следующих значений будут распечатаны каждым вызовом на остроконечной и тупоконечной машинах.

1. Остроконечная: Тупоконечная:
2. Остроконечная: Тупоконечная:
3. Остроконечная: Тупоконечная:

УПРАЖНЕНИЕ 2.6

Используя `show_int` и `show_float`, определяем, что целое число 3490593 имеет шестнадцатеричное представление 0x00354321, тогда как число с плавающей точкой 3490593.0 имеет шестнадцатеричное представление 0x4A550C84.

1. Напишите двоичные представления двух данных шестнадцатеричных величин.
2. Сдвиньте две строки относительно друг друга для увеличения числа совпадающих битов.
3. Сколько битов совпадают? Какие части строк не совпадают?

2.1.5. Представление строк

Строка в С кодируется массивом символов, прерываемых нулем (нулевым символом). Каждый символ представлен некой стандартной кодировкой, самой распространенной из которых является ASCII. Следовательно, если запустить процедуру `show_bytes` с аргументами "12345" и 6 (для включения символа прерывания), тогда в результате получится 31 32 33 34 35 00. Обратите внимание, что код ASCII для десятичного числа x будет 0x3x, и что прерывающий байт имеет шестнадцатеричное представление 0x00. Тот же результат будет получен на любой системе, использующей ASCII, независимо от упорядочения байтов и правил длины слова. Вследствие этого, тестовые данные более независимы от платформы, нежели двоичные данные.

Создание таблицы ASCII

Выполнением команды `man ascii` можно отобразить таблицу с кодами символов ASCII.

УПРАЖНЕНИЕ 2.7

Что распечатается в результате следующего вызова `show_bytes`?

```
char *s = "ABCDEF";
show_bytes (s, strlen (s));
```

Обратите внимание, что буквы от A до Z имеют коды ASCII от 0x41 до 0x5A.

Набор символов Unicode

Набор символов ASCII подходит для кодировки документов на английском языке, однако в нем отсутствуют некоторые буквы, например, французского языка. Данный набор символов абсолютно непригоден для кодировки документов на таких языках, как греческий, русский и китайский. Недавно был принят 16-битовый набор символов Unicode для поддержки документов на всех языках. Такое дублирование представлений символов обеспечивает отображение огромного числа символов. В языке программирования Java Unicode используется в представлении символьных строк. Для С также доступны программные библиотеки, в которых предоставляются версии Unicode для функций стандартной строки, таких как `srtlen` и `strcpy`.

2.1.6. Представление кода

Рассмотрим следующую функцию С в листинге 2.5.

Листинг 2.5. Представление суммы

```
1 int sum (int x, int y)
2 {
3     return x + y;
4 }
```

При компилировании на машинах-образцах создается машинный код, имеющий следующие байтовые представления:

Linux	55 89 e5 8b 45 0c 03 45 08 89 ec 5d c3
NT	55 89 e5 8b 45 0c 03 45 08 89 ec 5d c3
Sun	81 C3 E0 08 90 02 00 09
Alpha	00 00 30 42 01 80 FA 6B

На данном этапе обнаруживается, что кодировки команд различны, исключая машины с NT и Linux. В разных типах машин используются различные и несовместимые команды и кодировки. Машины с NT и Linux имеют процессоры Intel и, следовательно, поддерживают одинаковые команды машинного уровня. Впрочем, вообще структура исполняемой программы NT отличается от программы Linux. И поэтому эти машины не полностью совместимы в двоичном отношении. Двоичный код редко можно переносить между разными комбинациями машин и операционных систем.

Фундаментальная концепция компьютерных систем заключается в том, что с точки зрения машины программа представляет собой обычную последовательность байтов. Машина не имеет информации об источнике программы, за исключением, возможно, некоторых вспомогательных таблиц, создаваемых при отладке. Более подробно эта тема раскрывается в главе 3 при рассмотрении программирования на машинном уровне.

2.1.7. Булевы алгебры и кольца

Количество различных булевых алгебр не поддается счету; простейшая из них определяется по множеству двух элементов {0,1}. В табл. 2.4 показано несколько операций в этой булевой алгебре. Символы представления операций подобраны такими, как используемые для операций на уровне С (этот вопрос рассматривается далее). Булева операция \sim соответствует логической операции "НЕ", обозначаемой в логике высказываний символом \neg . То есть, говорится, что $\neg P$ истинно, когда P не истинно, и наоборот. Соответственно, $\neg p$ равно 1, когда p равно нулю, и наоборот. Булева операция $\&$ соответствует логической операции "И", обозначаемой в логике высказываний символом \wedge . Говорится, что $P \wedge Q$ выполняется, когда и P , и Q истинны. Соответственно, $p \& q$ равно единице только тогда, когда $p = 1$ и $q = 1$. Булева операция соответствует логической операции "ИЛИ", обозначаемой в логике высказываний символом \vee . Говорится, что $P \vee Q$ выполняется, когда P или Q истинны. Соответственно, $p \vee q$ равно единице только тогда, когда $p = 1$ или $q = 1$. Булева операция \wedge соответствует логической операции "Исключающее ИЛИ", обозначаемой в логике высказываний символом \oplus . Говорится, что $P \oplus Q$ выполняется, когда истинно P или Q , но не оба. Соответственно, $p \oplus q$ равно единице, когда либо $p = 1$ и $q = 0$, либо $p = 0$, а $q = 1$.

Таблица 2.4. Операции булевой алгебры

\sim		$\&$	0	1		0	1	\wedge	0	1
0	1	0	0	0	0	0	1	0	0	1
1	0	1	0	1	1	1	1	1	1	0

В табл. 2.4 двоичные значения 1 и 0 кодируют логические понятия "истинно" и "ложно", тогда как операции \sim , $\&$, | и \wedge кодируют логические операции "НЕ", "И", "ИЛИ" и "Исключающее ИЛИ", соответственно.

В табл. 2.5—2.7 приведено сравнение кольца целых чисел с булевой алгеброй. Две математические структуры имеют много общих свойств, однако между ними существуют принципиальные различия, особенно между \sim и \neg .

Клод Шэнон, ставший впоследствии основоположником теории информации, впервые обратил внимание на связь булевой алгебры и цифровой логики. В своей магистерской диссертации (в 1937 году) он доказал, что булеву алгебру можно применять к проектированию и анализу сетей электромеханических реле. Несмотря на то, что до начала развития компьютерных технологий прошло очень много времени, булева алгебра по-прежнему играет центральную роль в проектировании и анализе цифровых систем.

Между арифметикой и булевой алгеброй можно провести множество параллелей, точно так же, как и определить принципиальные различия. В частности, множество целых чисел, обозначенное символом Z , образует математическую структуру, называемую кольцом, обозначающимся как $\langle Z, +, \times, -, 0, 1 \rangle$ со сложением, обозначенным

операцией суммы, умножением — операции произведения, вычитанием — аддитивной инверсии, а элементами 0 и 1 — как идентификаторами сложения и умножения. Булева алгебра ($\{0,1\}$, $|$, $\&$, \sim , 0, 1) имеет сходные свойства. В табл. 2.5—2.7 представлены свойства двух структур с указанием как общих свойств, так и уникальных для одной и для другой системы. Одним из важнейших различий является то, что $\sim a$ не является инверсией для a под знаком $|$.

Таблица 2.5. Совместно используемые свойства

Свойство	Кольцо целых чисел	Булева алгебра
Коммутативность	$a + b = b + a$ $a \times b = b \times a$	$a b = b a$ $a \& b = b \& a$
Ассоциативность	$(a + b) + c = a + (b + c)$ $(a \times b) \times c = a \times (b \times c)$	$(a b) c = a (b c)$ $(a \& b) \& c = a \& (b \& c)$
Дистрибутивность	$a \times (b + c) = (a \times b) + (a \times c)$	$a \& (b c) = (a \& b) (a \& c)$
Тождественность	$a + 0 = a$ $a \times 1 = a$	$a 0 = a$ $a \& 1 = a$
Аннулятор	$a \times 0 = 0$	$a \& 0 = 0$
Отмена	$-(-a) = a$	$\sim(\sim a) = a$

Таблица 2.6. Особенности колец

Инверсия	$a + -a = 0$	—
----------	--------------	---

Таблица 2.7. Особенности булевых алгебр

Дистрибутивность	— —	$a (b \& c) = (a b) \& (a c)$
Дополнение	— —	$a \sim a = 1$ $a \& \sim a = 0$
Идемпотентность	— —	$a \& a = a$ $a a = a$
Поглощение	— —	$a (a \& b) = a$ $a \& (a b) = a$
Законы де Моргана	— —	$\sim(a \& b) = \sim a \sim b$ $\sim(a b) = \sim a \& \sim b$

Какая польза от абстрактной алгебры

В область абстрактной алгебры входят выявление и анализ общих свойств математических операций в разных областях приложения. Как правило, любая алгебра характеризуется множеством элементов, некоторыми из их основных операций и некоторыми особо важными элементами. Для примера можно сказать, что арифметические операции над абсолютными величинами чисел (модульная арифметика) также образуют кольцо. Для модуля n алгебра обозначается как $\langle Z_n, +_n, \times_n, -_n, 0, 1 \rangle$ с компонентами, обозначаемыми следующим образом:

$$Z_n = \{0, 1, \dots, n-1\}$$

$$a +_n b = a + b \bmod n$$

$$a \times_n b = a \times b \bmod n$$

$$-_n a = \begin{cases} 0, & a = 0 \\ n - a, & a > 0 \end{cases}$$

Несмотря на то, что при операциях в остаточных классах получаются иные результаты, нежели в целочисленной арифметике, ей присущи многие из тех же свойств. Другие известные кольца включают в себя рациональные и вещественные числа.

Если заменить операцию "ИЛИ" булевой алгебры на операцию "Исключающее ИЛИ" и операцию дополнения \sim — на операцию тождественности I , где $I(a) = a$ для всех a , тогда в результате получится структура $\langle \{0, 1\}, \wedge, \&, I, 0, 1 \rangle$. Эта структура более не является булевой алгеброй; по сути дела она представляет собой кольцо. Его можно рассматривать как крайне упрощенную форму кольца, состоящего из всех целых чисел $\{0, 1, \dots, n-1\}$ со сложением и умножением, выполненным по модулю n . В этом случае имеем $n = 2$. То есть булевые операции "И" и "Исключающее ИЛИ" соответствуют умножению и сложению по модулю 2, соответственно. Одним из любопытных свойств этой алгебры является то, что каждый элемент представлен своей собственной аддитивной инверсией: $a \wedge I(a) = a \wedge a = 0$.

Кого, кроме математиков, интересуют булевые кольца

Всякий раз при прослушивании исключительно чисто записанной музыки на компакт-диске или при просмотре фильма на DVD человек пользуется преимуществами булевых колец. Эти технологии основаны на *кодах исправления ошибок*, призванных для корректного считывания битов даже с загрязненного или поцарапанного диска. Математической основой этих кодов исправления ошибок является линейная алгебра, базирующаяся на булевых кольцах.

Упомянутые четыре булевые операции можно расширить для работы на битовых векторах, строк нулей и единиц определенной длины w . Операции определяются по битовым векторам в соответствии с их применением к совпадающим элементам аргументов. Например, определяем, что $[a_{w-1}, a_{w-2}, \dots, a_0] \& [b_{w-1}, b_{w-2}, \dots, b_0]$ является $[a_{w-1} \& b_{w-1}, a_{w-2} \& b_{w-2}, \dots, a_0 \& b_0]$, и подобным же образом для операций \sim , $|$ и \wedge . Допуская, что $\{0, 1\}^w$ будет обозначать множество всех строк нулей и единиц длины w ,

a^w — строку, состоящую из w повторений символа a , можно будет увидеть, что полученные в результате алгебры: $\langle \{0, 1\}^w, |, \&, \sim, 0^w, 1^w \rangle$ и $\langle \{0, 1\}^w, \wedge, \&, I, 0^w, 1^w \rangle$ формируют булевы алгебры и кольца, соответственно. Каждая величина w определяет отдельную булеву алгебру и отдельное булево кольцо.

Булевые кольца и арифметика остаточных классов — это одно и то же?

Двухэлементное булево кольцо $\langle \{0, 1\}, \wedge, \&, I, 0, 1 \rangle$ идентично кольцу целых чисел по модулю два $\langle \mathbb{Z}_2, +_2, \times_2, -_2, 0, 1 \rangle$. Однако объединение битовых векторов длины w дает кольцо, абсолютно отличающееся от арифметики остаточных классов.

УПРАЖНЕНИЕ 2.8

Заполните следующую таблицу с указанием результатов оценки булевых операций на битовых векторах.

Операция	Результат
a	[01101001]
b	[01010101]
$\sim a$	
$\sim b$	
$a \& b$	
$a b$	
$a \wedge b$	

Одним из полезных применений битовых векторов является представление конечных множеств. Например, любое подмножество $A \subseteq \{0, 1, \dots, w-1\}$ можно представить в виде битового вектора $[a_{w-1}, \dots, a_1, a_0]$, где $a_i = 1$ в том и только том случае, если $i \in A$. Например (вспоминая, что a_{w-1} пишется слева, а a_0 — справа), имеем $a = [01101001]$, представляющее множество $A = \{0, 3, 5, 6\}$, и $b = [01010101]$, представляющее множество $B = \{0, 2, 4, 6\}$. При такой интерпретации булевые операции $|$ и $\&$ соответствуют объединению множества и конъюнкции, \sim соответствует дополнению множества. Например, операция $a \& b$ дает битовый вектор $[01000001]$, тогда как $A \cap B = \{0, 6\}$.

На самом деле, для любого множества S структура $\langle P(S), \cup, \cap, \sim, \emptyset, S \rangle$ формирует булеву алгебру, где $P(S)$ обозначает множество всех подмножеств S , \sim обозначает множество оператора дополнения. То есть, для любого множества A его дополнением является множество $\bar{A} = \{a \in S | a \notin A\}$. Возможность представлять конечные множества и управлять ими, используя операции битового вектора, является практическим выводом из математического принципа.

УПРАЖНЕНИЕ 2.9

Компьютеры создают цветные изображения на мониторе или жидкокристаллическом дисплее путем смешения трех разных цветов светового спектра: красного, зеленого и синего. Представьте простую схему с тремя разными цветами, каждый из которых можно спроектировать на стеклянный экран:

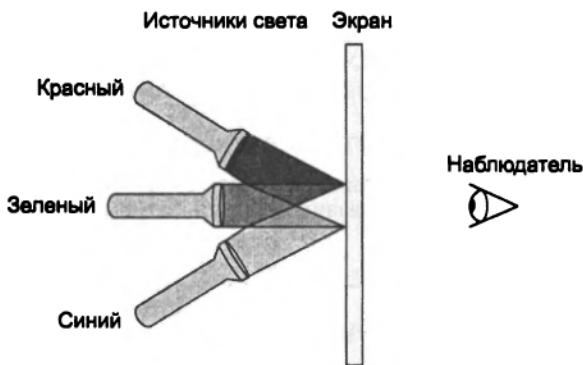


Рис. 2.1. Проекции на экран

Теперь можно создать восемь разных цветов на основании отсутствия (0) или присутствия (1) источников света:

Красный	Зеленый	Синий	Цвет
0	0	0	Черный
0	0	1	Синий
0	1	0	Зеленый
0	1	1	Голубой
1	0	0	Красный
1	0	1	Алый
1	1	0	Желтый
1	1	1	Белый

Данное множество цветов формирует восьмиэлементную булеву алгебру.

1. Дополнение цвета формируется выключением включенного света и включением выключеного. Что будет дополнениями восьми перечисленных цветов?
2. Какие цвета соответствуют булевым величинам 0" и 1" для данной алгебры?

3. Опишите эффект применения булевых операций на следующие цвета:

Синий | Красный =

Алый & Голубой =

Зеленый ^ Белый =

2.1.8. Операции на уровне бита в С

Одной из полезных особенностей С является то, что он поддерживает побитовые булевые операции. На самом деле, символы, использованные для булевых операций, применяются и в С:

- | — ИЛИ;
- & — И;
- ~ — НЕ;
- ^ — Исключительное ИЛИ.

Это применяется для любого целого типа данных, т. е. для объявленного как тип `char` или `int` со спецификаторами (или без них), такими как `short`, `long` или `unsigned`. Вот некоторые примеры оценок выражений (табл. 2.8):

Таблица 2.8. Выражения на битовом уровне

Выражение С	Двоичное выражение	Двоичный результат	Результат С
<code>~0x41</code>	<code>~ [01000001]</code>	<code>[10111110]</code>	<code>0xBE</code>
<code>~0x00</code>	<code>~ [00000000]</code>	<code>[11111111]</code>	<code>0xFF</code>
<code>0x69 & 0x55</code>	<code>[01101001] & [01010101]</code>	<code>[01000001]</code>	<code>0x41</code>
<code>0x69 0x55</code>	<code>[01101001] [01010101]</code>	<code>[01111101]</code>	<code>0x7D</code>

Как показывают наши примеры, лучшим способом определения воздействия выражения на битовом уровне является расширение шестнадцатеричных аргументов до их двоичных представлений, выполнение операции в двоичной системе счисления и осуществление преобразование в шестнадцатеричную.

УПРАЖНЕНИЕ 2.10

Для демонстрации свойств кольца рассмотрим листинг 2.6.

Листинг 2.6. Процедура перестановки

```

1 void inplace_swap (int *x, int *y)
2 {
3     *x = *x ^ y; // Step 1 */

```

```

4   *y = *x ^ y;/ Step 2 */
5   *x = *x ^ y;/ Step 3 */
6 }
```

Уже по названию можно утверждать, что действие этой процедуры есть перестановка величин, сохраненных в ячейках, обозначенными переменными указателя *x* и *y*. Обратите внимание на то, что, в отличие от обычной методики перестановки двух величин, здесь нет необходимости в третьей ячейке для временного хранения одной величины во время перемещения другой. По производительности такого способа перестановки выигрыша нет; выигрыш имеет место лишь в виде интеллектуального развлечения.

Начиная с величин *a* и *b* в ячейках, указанных *x* и *y*, соответственно, заполните следующую таблицу величинами, сохраненными в двух ячейках после каждого шага процедуры. Для демонстрации эффекта пользуйтесь свойствами кольца. Помните о том, что каждый элемент является собственной аддитивной инверсией ($a \wedge a = 0$).

Шаг	*x	*y
Первоначально	<i>a</i>	<i>b</i>
Шаг 1		
Шаг 2		
Шаг 3		

Одним из общих использований операций на уровне бита является реализация операций **маскирования**, где маской называется комбинация битов, указывающая на выбранное множество битов в слове. Например, маска 0xFF (наименее значимые восемь битов — единицы) указывает на байт нижнего уровня в слове. Операция на битовом уровне *x & 0xFF* дает величину, состоящую из наименее значимого байта *x*, но при этом все остальные байты — нулевые. Например, при *x* = 0x89ABCDEF выражение приведет к результату 0x000000EF. Выражение *~0* даст результат маски всех единиц, независимо от длины машинного слова. Несмотря на то, что эту же маску можно записать как 0xFFFFFFFF для 32-битовой машины, такой код не является переносимым.

УПРАЖНЕНИЕ 2.11

- Наименее значимый байт *x* со всеми другими битами, установленными в 1 [0xFFFFFFFBA].
- Дополнение наименее значимого байта *x*; все прочие байты остаются неизменными [0x98FDEC45].
- Все, кроме наименее значимого байта *x*; наименее значимый байт установлен в 0 [0x98FDEC00].

Несмотря на то, что данные примеры предполагают 32-битовую длину слова, полученный код должен работать для любой длины слова *w* ≥ 8 .

УПРАЖНЕНИЕ 2.12

С конца 70-х до конца 80-х годов прошлого века очень популярным был компьютер Digital Equipment VAX. Вместо команд для булевых операций "И" и "ИЛИ" он имел команды `bis` (множество битов) и `bic` (свободный бит). Обе команды принимают информационное слово `x` и слово-маску `m`. Они создают результат `z`, состоящий из битов `x`, измененных в соответствии с битами `m`. При использовании `bis` изменение включает в себя настройку `z` на 1 в каждой позиции бита, где `m` является единицей. При использовании `bic` изменение включает в себя настройку `z` на 0 в каждой позиции бита, где `m` является единицей.

Для расчета воздействия этих двух команд необходимо записать функции `bis` и `bic`. Дополните следующий код, используя операции С на уровне бита:

```
/* Bit Set */
int bis (int x, int m)
{
/* Записать выражение в С для расчета воздействия множества битов */
int result = _____;
return result;
}

/* Bit Clear */
int bic (int x, int m)
{
/* Записать выражение в С для расчета воздействия свободных битов */
int result = _____;
return result;
}
```

2.1.9. Логические операции в С

Язык С также представляет набор логических операторов `|`, `&&` и `!`, соответствующих операциям "ИЛИ", "И" и "НЕ" логики высказываний. Их легко спутать с операциями на уровне битов, однако они выполняют совершенно иные функции. Логические операции рассматривают любой ненулевой аргумент как истинный, а аргумент 0 — как ложный. Они возвращают либо 1, либо 0, указывая либо на истинный, либо на ложный результат, соответственно. Вот несколько примеров оценки выражений:

Выражение	Результат
<code>! 0x41</code>	<code>0x00</code>
<code>!0x00</code>	<code>0x01</code>
<code>!!0x41</code>	<code>0x01</code>
<code>0x69 && 0x55</code>	<code>0x01</code>
<code>0x69 0x55</code>	<code>0x01</code>

Обратите внимание, что побитовая операция соответствует своему логическому двойнику только в особом случае, когда аргументы ограничены значениями 0 или 1.

Вторым важным различием между логическими операторами `&&`, `||` и их двойниками на битовом уровне `&`, `|` является то, что логические операторы не оценивают свой второй аргумент, если результат выражения можно определить оценкой первого аргумента. Следовательно, к примеру, выражение `a && 5/a` никогда не спровоцирует деления на ноль, а выражение `r && *p++` никогда не вызовет разыменования пустого (нулевого) указателя.

УПРАЖНЕНИЕ 2.13

Предположим, что `x` и `y` имеют величины байтов 0x66 и 0x93, соответственно. Заполните следующую таблицу, указывая байтовые величины различных выражений С:

Выражение	Величина	Выражение	Величина
<code>x & y</code>		<code>x && y</code>	
<code>x y</code>		<code>x y</code>	
<code>~x ~y</code>		<code>!x !y</code>	
<code>x & !y</code>		<code>x && ~y</code>	

УПРАЖНЕНИЕ 2.14

Используя только операции на уровне битов и логические, запишите выражение С, эквивалентное `x == y`. Другими словами, результатом будет 1, когда `x` и `y` равны, и 0 — в противном случае.

2.1.10. Операции сдвига в С

В С также представлен набор *операций сдвига* комбинаций битов влево и вправо. Для операнда `x`, имеющего битовое представление $[x_{n-1}, x_{n-2}, \dots, x_0]$ выражение `x << k` дает величину с битовым представлением $[x_{n-k-1}, x_{n-k-2}, \dots, x_0, 0, \dots, 0]$. То есть, `x` смешен на k битов влево с удалением наиболее значимых битов k и заполнением правой стороны k нулями. Сдвиг должен быть в диапазоне от 0 до $n-1$. Операции сдвига группируются в направлении слева направо, поэтому `x << j << k` эквивалентно $(x << j) << k$. Не забывайте о приоритете операций: $1 << 5 - 1$ оценивается как $1 << (5-1)$, а не как $(1 << 5) - 1$.

Существует соответствующая операция сдвига вправо `x >> k`, однако ее поведение имеет отличия. Вообще говоря, все машины поддерживают две формы сдвига вправо: логическую и арифметическую. Логический сдвиг вправо заполняет левый край k нулями, выдавая результат $[0, \dots, 0, x_{n-1}, x_{n-2}, \dots, x_k]$. Арифметический сдвиг вправо заполняет левый край k повторениями наиболее значимого бита, выдавая результат $[x_{n-1}, \dots, x_{n-1}, x_{n-1}, x_{n-2}, \dots, x_k]$. Это правило может показаться незначительным, однако

в дальнейшем будет видно, что оно полезно для работы с целочисленными данными со знаком.

Стандарт С не определяет точно того, какой необходимо использовать тип сдвига вправо. Для данных без знака (т. е. для целых данных, объявленных со спецификатором `unsigned`) сдвиги вправо могут быть логическими. Для данных со знаком (по умолчанию) можно использовать либо арифметические, либо логические сдвиги. К сожалению, это означает, что любой код, принимающий ту или иную форму, может впоследствии столкнуться с проблемами переносимости. Впрочем, как показывает практика, почти все комбинации компиляторов и машин используют для данных со знаком арифметические сдвиги вправо, и большинство программистов принимают это как должное.

УПРАЖНЕНИЕ 2.15

Заполните таблицу, показывая воздействие различных операций сдвига на однобайтовые величины. Лучшим способом рассмотрения операций сдвига является работа с двоичными представлениями. Преобразуйте первоначальные величины в двоичные, выполните сдвиги, после чего выполните обратное преобразование в шестнадцатеричное значение. Каждый из ответов должен состоять из 8 двоичных цифр или двух шестнадцатеричных.

x	$x \ll 3$	$x \gg 2$	$x \gg 2$
Шестн. Двоичное 0xF0	Шестн. Двоичное	Шестн. Двоичное	Шестн. Двоичное
0x0F			
0xCC			
0x55			

2.2. Целочисленное представление

В данном разделе описываются два разных способа того, как можно использовать биты для кодировки целых чисел: один из этих способов может представлять только неотрицательные числа, а другой — отрицательные числа, положительные и ноль. Позже читателю станет понятно, что они взаимозависимы как по математическим свойствам, так и по реализациям на машинном уровне. Авторы также рассматривают воздействие расширения или сжатия кодированного целого числа для того, чтобы оно подходило представлению с разной длиной.

2.2.1. Типы целого

Язык С поддерживает разные типы целого, различаемые рамками диапазонов чисел. Они показаны в табл. 2.9. Каждый тип имеет указатель размера: `char`, `short`, `int` и

`long`, а также признак того, является ли число неотрицательным (`unsigned`). По умолчанию целое может быть отрицательным. Типы представления чисел были описаны в табл. 2.2. Стандарт С определяет минимальный диапазон величин, который способен представить каждый тип данных. В типичной 32-битовой машине используется 32-битовое представление данных типов `int` и `unsigned`, несмотря на то, что стандарт С допускает и 16-битовые представления. Как показано в табл. 2.9, в Compaq Alpha используется слово длиной 64 битов для представления длинного целого `long`, что обеспечивает верхний предел свыше 1.84×10^{19} для величин без знака, и диапазон, превышающий $\pm 9.22 \times 10^{18}$ для величин со знаком.

Таблица 2.9. Типы целого в С

Описание С	Обеспечено		Типичные 32-битовые	
	Минимум	Максимум	Минимум	Максимум
<code>char</code>	-127	127	-128	127
<code>unsigned char</code>	0	255	0	255
<code>short [int]</code>	-32 767	32 767	-32 768	32 767
<code>unsigned short [int]</code>	0	63 535	0	63 535
<code>int</code>	-32 767	32 767	-2 147 483 648	2 147 483 647
<code>unsigned [int]</code>	0	63 535	0	4 294 967 295
<code>long [int]</code>	-2 147 483 647	2 147 483 647	-2 147 483 648	2 147 483 647
<code>unsigned long [int]</code>	0	4 294 967 295	0	4 294 967 295

Числа со знаком и без знака в С, С++ и Java

С и С++ поддерживают числа как со знаком (по умолчанию), так и без знака. Java поддерживает только числа со знаком.

2.2.2. Кодировки со знаком и с двоичным дополнительным кодом

Предположим, что имеется тип целого, состоящий из w битов. Битовый вектор записывается либо как \bar{x} для обозначения полного вектора, либо в $[x_{w-1}, x_{w-2}, \dots, x_0]$ для обозначения отдельных битов в рамках вектора. При рассмотрении \bar{x} как числа, написанного в двоичном счислении, получим интерпретацию \bar{x} без знака. Эта интерпретация выражается как функция $B2U_w$ (двоичного длины w до беззнакового):

$$B2U_w(\bar{x}) = \sum_{i=0}^{w-1} x_i 2^i \quad (2.1)$$

Функция $B2U_w$ преобразует строки нулей и единиц длины w в неотрицательные целые числа. Наименьшая величина представлена битовым вектором $[00 \dots 0]$, имеющим

целое значение 0, а наибольшая величина представлена битовым вектором [11 ... 1], имеющим целую величину $UMax_w = \sum_{i=0}^{w-1} 2^i = 2^w - 1$. Таким образом, функцию $B2U_w$, можно определить как отображение $B2U_w: \{0, 1\}^w \rightarrow \{0, \dots, 2^w - 1\}$. Обратите внимание, что $B2U_w$ — это взаимно однозначное соответствие; оно ассоциирует уникальную величину с каждым битовым вектором длины w ; и наоборот, каждое целое число в диапазоне от 0 до $2^w - 1$ имеет уникальное двоичное представление в виде битового вектора длины w .

Для многих программных приложений также хочется представить и отрицательные величины. Наиболее широко распространенное компьютерное представление чисел со знаками называется формой дополнительного кода. Он определяется интерпретацией наиболее значимого бита слова, который будет иметь отрицательный вес. Подобная интерпретация выражается функцией $B2T_w$ (двоичного длины w до дополнительного кода):

$$B2T_w(\bar{x}) = -x_{w-1} 2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i \quad (2.2)$$

УПРАЖНЕНИЕ 2.16

Предположив, что $w = 4$, каждой возможной шестнадцатеричной цифре можно присвоить числовую величину, предполагая интерпретацию без знака или дополнительный код. Заполните следующую таблицу в соответствии с этими интерпретациями, выписывая ненулевые степени 2, как в уравнениях (2.1) и (2.2):

\bar{x}	$B2U_4(\bar{x})$	$B2T_4(\bar{x})$
Шестн. Двоичное		
A [1010]	$2^3 + 2^1 = 10$	$-2^3 + 2^1 = -6$
0		
3		
8		
C		
F		

В табл. 2.10 показаны комбинации битов и численные значения некоторых характерных величин для различных длин слова. В первых трех строках представлены диапазоны целых чисел. Здесь стоит особо выделить несколько моментов. Во-первых, диапазон дополнительных кодов асимметричен: $|TMax_w| = |TMin_w| + 1$, т. е. для $TMin_w$ не существует положительного эквивалента. Далее мы увидим, что это приводит к проявлению некоторых особых свойств арифметики дополнительных кодов и может оказаться источником небольших программных дефектов. Во-вторых, максимальная

величина без знака несколько превышает удвоенную величину дополнительного кода: $UMax_w = 2TMax_w + 1$. Это вытекает из того факта, что представление дополнительного кода резервирует половину комбинаций битов для представления отрицательных величин. Другими случаями являются константы -1 и 0 . Обратите внимание, что -1 имеет то же битовое представление, что и $UMax_w$ — строку из единиц. Числовое значение 0 представлено в виде строки всех нулей в обоих случаях.

Таблица 2.10. Характерные величины в численном и шестнадцатеричном представлении

Количество	Длина слова w			
	8	16	32	64
$UMax_w$	0xFF	0xFFFF	0xFFFFFFFF	0xFFFFFFFFFFFFFF
	255	65 535	4 294 967 296	18 446 744 073 709 551 615
$TMax_w$	0x7F	0x7FFF	0x7FFFFFFF	0x7FFFFFFFFFFFFF
	127	32 767	2 147 483 647	9 223 372 036 854 775 807
$TMin_w$	0x80	0x8000	0x80000000	0x8000000000000000
	-128	-32 768	-2 147 483 648	-9 223 372 036 854 775 808
-1	0xFF	0xFFFF	0xFFFFFFFF	0xFFFFFFFFFFFFFF
0	0x00	0x0000	0x00000000	0x0000000000000000

Стандарт С не требует представления целых чисел со знаком в форме дополнительного кода, однако практически все машины это делают. Для поддержания переносимости кода не следует предполагать никакого конкретного диапазона представимых величин, либо того, как они представлены, за пределами диапазонов (см. табл. 2.2). Файл `<limits.h>` в библиотеке С определяет множество констант, разграничающих диапазоны различных типов целочисленных данных для каждого конкретного компьютера, на котором запущен компилятор. Например, он определяет константы `INT_MAX`, `INT_MIN` и `UINT_MAX`, описывающие диапазоны целых чисел со знаком и без знака. Для дополнительного кода машин, где тип данных `int` имеет w битов, эти константы соответствуют величинам $TMax_w$, $TMin_w$ и $UMax_w$.

Альтернативные представления чисел со знаком

Для чисел со знаком существуют два других стандартных представления.

Обратный код. То же самое, что и дополнительный код, за исключением того, что самый значимый бит имеет вес $-(2^{w-1} - 1)$, а не -2^{w-1} :

$$B2O_w(\bar{x}) \doteq -x_{w-1}(2^{w-1} - 1) + \sum_{i=0}^{w-2} x'_i 2^i$$

Величина знака. Самый значимый бит — это двоичный разряд, определяющий то, какой вес должен быть определен остальным битам: положительный или отрицательный:

$$B2S_w(\bar{x}) \doteq (-1)^{x_{w-1}} \cdot \left(\sum_{i=0}^{w-2} x'_i 2^i \right)$$

Оба этих представления обладают тем любопытным свойством, что здесь существуют две различные кодировки числа 0. Для обоих представлений [00 ... 0] интерпретируется как +0. Величину –0 можно представить в форме величины знака как [10...0], а в форме обратного кода — как [11...1]. Несмотря на то, что машины, созданные на основе представлений дополнительных кодов, уже устарели, последние используются почти во всех современных компьютерах. Далее будет понятно, что кодировка по величине знака используется с числами с плавающей точкой.

Для примера рассмотрите следующий код:

```
1 short int x = 12345;
2 short int mx = -x;
3
4 show_bytes((byte_pointer) &x, sizeof(short int));
5 show_bytes((byte_pointer) &mx, sizeof(short int));
```

В табл. 2.11 даны представления дополнительного поразрядного кода 12 345 и –12 345 и представление без знака 53 191. Обратите внимание, что битовые представления последних двух идентичны.

При запуске на "тупоконечной" машине данный код распечатывает 30 30 и cf cf, указывая на то, что x имеет шестнадцатеричное представление 0x039, тогда как mx имеет шестнадцатеричное представление 0xCFC7. При их расширении в двоичную систему получим комбинации битов [0011000000111001] — для x и [1100111111000111] — для mx. Как показано в табл. 2.11, для этих двух комбинаций битов уравнение (2.2) дает величины 12 345 и –12 345.

Таблица 2.11. Представления дополнительного поразрядного кода

Вес	12 345		–12 345		53 191	
	Бит	Величина	Бит	Величина	Бит	Величина
1	1	1	1	1	1	1
2	0	0	1	2	1	2
4	0	0	1	4	1	4
8	1	8	0	0	0	0
16	1	16	0	0	0	0
32	1	32	0	0	0	0

Таблица 2.11 (окончание)

Вес	12 345		-12 345		53 191	
	Бит	Величина	Бит	Величина	Бит	Величина
64	0	0	1	64	1	64
128	0	0	1	128	1	128
256	0	0	1	256	1	256
512	0	0	1	512	1	512
1024	0	0	1	1024	1	1024
2048	0	0	1	2048	1	2048
4096	1	4096	0	0	0	0
8192	1	8192	0	0	0	0
16 384	0	0	1	16 384	1	16 384
±32 768	0	0	1	-32 768	1	32 768
Всего	12 345		-12 345		53 191	

УПРАЖНЕНИЕ 2.17

В главе 3 будут рассмотрены листинги, сгенерированные дизассемблером — программой, преобразующей исполняемый программный файл в удобочитаемую форму ASCII. Эти файлы содержат множество шестнадцатеричных чисел, обычно представляющих величины в форме дополнительного поразрядного кода. Возможность распознавания этих чисел и понимание их значимости (например, отрицательные они или положительные) является важным профессиональным навыком для любого программиста.

Для строк, помеченных в следующем листинге как А–К, преобразуйте шестнадцатеричные величины, показанные справа от названий команд (sub, push, mov и add), в их десятичные эквиваленты.

```

80483b7: 81 ec 84 01 00 00 sub $0x184, %esp          A.
80483bd: 53 push %ebp
80483be: 8b 55 08 mov 0x8(%ebp), %edx             B.
80483c1: 8b 5d 0c mov 0xc(%ebp), %ebx             C.
80483c4: 8b 4d 10 mov 0x10(%ebp), %ecx            D.
80483c7: 8b 85 94 fe ff ff mov 0xffffffe94(%ebp), %eax E.
80483cd: 01 cb add %ecx, %ebx
80483cf: 03 42 10 add 0x10(%edx), %eax           F.
80483d2: 89 85 a0 fe ff ff mov %eax, 0xfffffea0(%ebp) G.
80483d8: 8b 85 10 ff ff ff mov 0xfffffff10(%ebp), %eax H.

```

```

80483de: 89 42 1c          mov %eax, 0x1c(%edx)    I.
80483e1: 89 9d 7c ff ff ff mov %ebx, 0xffffffff7c(%ebp) J.
80483e7: 8b 42 18          mov 0x18(%edx), %eax      K.

```

2.2.3. Преобразования между числами со знаком и без знака

Поскольку и $B2U_w$ и $B2T_w$ являются взаимно однозначными соответствиями, то они имеют хорошо определенные инверсии. Определите $U2B_w$ как $B2U_w^{-1}$, а $T2B_w$ — как $B2T_w^{-1}$. Эти функции сопоставляют комбинациям битов без знака или в дополнительном коде числовые значения. При условии, что целое число x находится в диапазоне $0 \leq x < 2^w$, функция $U2B_w(x)$ дает уникальное w -битовое представление x без знака. Подобным же образом, когда x находится в диапазоне $-2^{w-1} \leq x < 2^{w-1}$, функция $T2B_w(x)$ дает уникальное w -битовое представление x дополнительного кода. Обратите внимание, что для величин в диапазоне $0 \leq x < 2^{w-1}$ обе эти функции выдадут одинаковое битовое представление: самый значимый бит будет 0, и не имеет значения, какой вес имеет этот бит, положительный или отрицательный.

Рассмотрим функцию $U2T_w(x) = B2T_w(U2B_w(x))$ для числа в диапазоне от 0 до 2^{w-1} , выдающую число в диапазоне между -2^{w-1} и $2^{w-1} - 1$, где два эти числа имеют одинаковые битовые представления, за исключением того, что аргумент — без знака, а результат имеет представление дополнительного кода. И наоборот, функция $T2U_w(x) = B2U_w(T2B_w(x))$ выдает число без знака, имеющее то же битовое представление, что и величина x дополнительного кода. Например, как показано в табл. 2.11, 16-битовое представление дополнительного кода $-12,345$ идентично 16-битовому представлению без знака числа $53\,191$. Следовательно, $T2U_{16}(-12\,345) = 53\,191$ и $U2T_{16}(53\,191) = -12\,345$.

Может показаться, что данные функции представляют только академический интерес, однако на самом деле они имеют огромную практическую важность: они формально определяют влияние преобразования типа в С между величинами со знаком и без знака. Например, рассмотрим выполнение следующего кода на машине с дополнительным двоичным кодом:

```

1 int x = -1
2 unsigned ux = (unsigned) x;

```

Данный код переведет ux в $UMax_w$, где w — количество битов в типе данных `int`, поскольку в табл. 2.10 видно, что w -битовое представление дополнительного кода -1 имеет то же битовое представление, что и $UMax_w$. Вообще говоря, преобразование типа величины x со знаком в величину x без знака (`unsigned`) эквивалентно применению функции $T2U$. Такое преобразование типа не изменяет битового представления аргумента, т. е. как эти биты интерпретируются в виде числа. Подобным же образом преобразование величины u без знака в величину u со знаком (`int`) эквивалентно применению функции $U2T$.

УПРАЖНЕНИЕ 2.18

Используя таблицу, заполненную при решении упражнения 2.16, заполните следующую таблицу, описывающую функцию $T2U_4$:

x	$T2U_4(x)$
-8	
-6	
-4	
-1	
0	
3	

Для того чтобы лучше понять отношение между числом x со знаком и его эквивалентом $T2U_w(x)$ без знака, можно использовать тот факт, что они имеют идентичные битовые представления. Сравнивая уравнения (2.1) и (2.2), можно увидеть, что для комбинации битов (\bar{x}), если просчитывать разницу между $B2U_w(\bar{x}) - B2T_w(\bar{x})$, взвешенные суммы для битов от 0 до $w - 2$ аннулируют друг друга, оставив величину: $B2U_w(\bar{x}) - B2T_w(\bar{x}) = x_{w-1}(2^{w-1} - 2^{w-1}) = x_{w-1}2^w$. Это дает отношение $B2U_w(\bar{x}) = x_{w-1}2^w + B2T_w(\bar{x})$. Если допустить, что $x = B2T_w(\bar{x})$, тогда получим

$$B2U_w(T2B_w(x)) = T2U_w(x) = x_{w-1}2^w + x \quad (2.3)$$

Это отношение полезно для доказательства отношений между арифметикой чисел без знака и арифметикой дополнительных кодов. В представлении дополнительного кода x бит x_{w-1} определяет, отрицательна ли величина x , выдавая следующее:

$$T2U_w(x) = \begin{cases} x + 2^w, & x < 0 \\ x, & x \geq 0 \end{cases} \quad (2.4)$$

На рис. 2.2 проиллюстрировано поведение функции $T2U$. Как показано, при проекции числа со знаком на его эквивалент без знака отрицательные числа преобразуются в большие положительные числа, тогда как неотрицательные числа остаются без изменений.

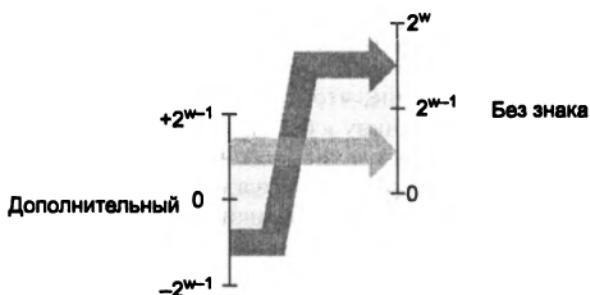


Рис. 2.2. Преобразование из дополнительного кода в величину без знака

УПРАЖНЕНИЕ 2.19

Объясните, как уравнение (2.4) применяется к записям в таблице, созданной в упражнении 2.18.

Двигаясь в обратном направлении, необходимо вывести отношение между числом x без знака и его эквивалентом со знаком $U2T_w(x)$. Если допустить, что $x = B2U_w(\bar{x})$, тогда получим

$$B2T_w(U2B_w(x)) = U2T_w(x) = -x_{w-1}2^w + x \quad (2.5)$$

В представлении без знака x бит x_{w-1} определяет, больше x или равно 2^{w-1} , выдавая следующее:

$$U2T_w(x) = \begin{cases} x, & x < 2^{w-1} \\ x - 2^w, & x \geq 2^{w-1} \end{cases} \quad (2.6)$$

Это поведение проиллюстрировано на рис. 2.3. Для маленьких ($< 2^{w-1}$) чисел преобразование из представления без знака в представление со знаком сохраняет числовую величину. Для крупных величин ($\geq 2^{w-1}$) число преобразуется в отрицательную величину.

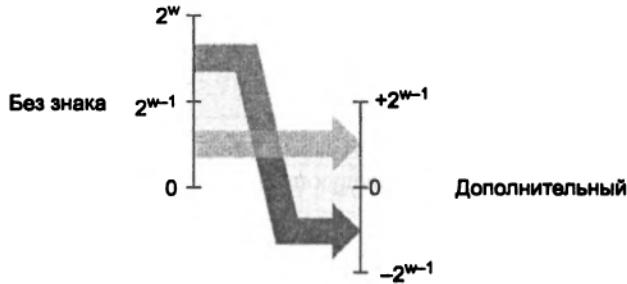


Рис. 2.3. Преобразование величины из представления без знака в представление дополнительного кода

Подводя итог, можно рассматривать эффекты преобразования в обоих направлениях, между представлением числа без знака и представлением в дополнительном коде. Для величин в диапазоне $0 \leq x < 2^{w-1}$ имеем $T2U_w(x) = x$ и $U2T_w(x) = x$. То есть числа в этом диапазоне имеют идентичные представления без знака и в дополнительном коде. Для величин, выходящих за рамки данного диапазона, преобразования либо добавляют, либо вычитают 2^w . Например, $T2U_w(-1) = -1 + 2^w = UMax_w$ — ближайшее к нулю отрицательное число, отображающее самое большое число без знака. В другом крайнем случае видно, что $T2U_w(TMin_w) = -2^{w-1} + 2^w = TMax_w + 1$ — наибольшее по величине отрицательное число отображается на число без знака как раз за пределами диапазона положительных чисел в дополнительном коде. Используя пример, показанный в табл. 2.11, можно заметить, что $T2U_{16}(-12,345) = 65\,536 + -12\,345 = 53\,191$.

2.2.4. Величины со знаком относительно величин без знака в С

Как показано в табл. 2.9, С поддерживает как арифметику со знаком, так и без знака для всех целочисленных типов данных. Хотя стандарт С не задает конкретного представления чисел со знаком, почти на всех машинах используется дополнительный двоичный код. Вообще говоря, большинство чисел имеют знак по умолчанию. Например, при объявлении такой константы, как 12345 или 0x1A2B, считается, что величина имеет знак. Для создания константы без знака в качестве суффикса необходимо добавить символ U или u (например, 12345U или 0x1A2Bu).

В С возможно преобразование из величин без знака в величины со знаком. Правило таково, что лежащее в основе битовое представление не меняется. Таким образом, на машине с использованием дополнительного двоичного кода при преобразовании величины без знака в величину со знаком используется функция $U2T_w$, а наоборот — функция $T2U_w$, где w — число битов типа данных.

Преобразования могут иметь вид явного преобразования типа, как, например, в следующем коде:

```
1 int tx, ty;
2 unsigned ux, uy;
3
4 tx = (int) ux;
5 uy = (unsigned) ty;
```

С другой стороны, они могут быть неявными, когда выражение одного типа присыпывается переменной другого, как в следующем коде:

```
1 int tx, ty;
2 unsigned ux, uy;
3
4 tx = ux; /*Преобразовать в величину со знаком */
5 uy = ty; /*Преобразовать в величину без знака */
```

При печати числовых величин с помощью `printf` необходимо использовать указатели `%d`, `%u` и `%x` для печати числа как десятичного со знаком, десятичного без знака и шестнадцатеричного соответственно. Обратите внимание, что `printf` не использует никакой информации о типе, и поэтому имеется возможность печати величины типа `int` с указателем `%u` и величины типа `unsigned` с указателем `%d`. Например, рассмотрим следующий код:

```
1 int x = -1;
2 unsigned u = 2147483648; /*2 на 31-й */
3
4 printf ("x = %u = %d\n", x, x);
5 printf ("u = %u = %d\n", u, u);
```

При запуске на 32-битовой машине распечатывается следующее:

$x = 4294967295 = -1$

$u = 2147483648 = -2147483648$

В обоих случаях printf сначала распечатывает слово так, как будто оно представило число без знака, а затем — как будто оно представило число со знаком. Процедуры преобразования можно видеть в операции $T2U_{32}(-1) = UMax_{32} = 4,294,967,295$ и $U2T_{32}(2^{31}) = 2^{31} - 2^{32} = -2^{31} = TMin_{32}$.

Определенное особенное поведение возникает вследствие того, что С обрабатывает выражения, содержащие комбинации величин со знаком и без знака. Когда операция выполняется над одним операндом со знаком, а другим — без знака, тогда С неявно преобразует аргумент со знаком в аргумент без знака и выполняет операции, предполагая, что числа неотрицательные. Как будет видно далее, это правило не имеет особых значения для стандартных арифметических операций, однако приводит к не интуитивным результатам для операторов отношения, таких как `<and>`. В табл. 2.12 показаны некоторые образцы выражений отношения и их конечные оценки, при условии 32-битовой машины, использующей представление дополнительного двоичного кода. Рассмотрим сравнение $-1 < 0u$. Поскольку второй операнд — без знака, первый неявно преобразуется в операнд без знака, и отсюда данное выражение эквивалентно сравнению $4294967295u < 0u$ (помните, что $T2U_{32}(-1) = UMax_{32}$), что, разумеется, ложно. С другими случаями можно разобраться путем подобного же анализа.

Таблица 2.12. Правила преобразования С на 32-битовой машине

Выражение	Тип	Оценка
$0 == 0u$	Без знака	1
$-1 < 0$	Со знаком	1
$-1 < 0u$	Без знака	0*
$2147483648 > -2147483647 - 1$	Со знаком	1
$2147483648u > -2147483647 - 1$	Без знака	0*
$2147483647 > (int) 2147483648u$	Со знаком	1*
$-1 > -2$	Со знаком	1
$(unsigned) -1 > -2$	Без знака	1

Примечание

Неочевидные случаи преобразований отмечены в табл. 2.12 звездочкой. Необходимо записывать $TMin_{32}$ как $-2147483647 - 1$, а не как -2147483648 , во избежание переполнения. Компьютер обрабатывает выражение формы $-X$ первым считыванием выражения X с последующим отрицанием его, однако 2147483648 — слишком большое число для его представления как 32-битового числа с дополнительным двоичным кодом.

УПРАЖНЕНИЕ 2.20

Предположив, что выражения оцениваются на 32-битовой машине, использующей арифметику дополнительного двоичного кода, заполните следующую таблицу, описывая влияние преобразований и операций отношения в стиле табл. 2.12.

Выражение	Тип	Оценка
- 2147483647 - 1 == 2147483648U		
- 2147483647 - 1 < - 2147483647		
(unsigned) (- 2147483647 - 1) < - 2147483647		
- 2147483647 - 1 < - 2147483647		
(unsigned) (- 2147483647 - 1) < 2147483647		
2147483647 > (int) 2147483648U		

2.2.5. Расширение битового представления числа

Одной из обычных операций являются преобразования между целыми числами, имеющими различную длину слова, но сохраняющими одно и то же числовое значение. Конечно, это невозможно, когда целевой тип данных слишком мал, для представления нужной величины. Однако преобразование маленького типа данных в большой всегда должно быть возможно. Для преобразования числа без знака в более крупный тип данных можно просто добавлять в представление ведущие нули. Такая операция называется *дополнением нулями*. Для преобразования числа с дополнительным двоичным кодом в более крупный тип данных применяется правило выполнения *дополнительного знакового разряда* с добавлением в представление копий наиболее значимого бита. Следовательно, если оригинальная величина имеет битовое представление $[x_{w-1}, x_{w-2}, \dots, x_0]$, тогда расширенное представление имеет форму $[x_{w-1}, \dots, x_{w-1}, x_{w-2}, \dots, x_0]$.

В качестве примера рассмотрим листинг 2.7.

Листинг 2.7. Пример преобразования

```

1 short sx = val; /* -12345 */
2 unsigned short usx = sx; /* 53191 */
3 int x = sx; /* -12345 */
4 unsigned ux = usx; /* 53191 */
5
6 printf ("sx = %d:\t", sx);
7 show_bytes((byte_pointer) &sx, sizeof(short));
8 printf ("usx = %u:\t", usx);
9 show_bytes((byte_pointer) &usx, sizeof(unsigned short));
10 printf ("x = %d:\t", x);

```

```
11 show_bytes((byte_pointer) &x, sizeof(int));
12 printf ("ux = %u:\t", ux);
13 show_bytes((byte_pointer) &ux, sizeof(unsigned));
```

При запуске на 32-битовой (тупоконечной) машине, использующей дополнительный двоичный код, распечатывается следующее:

```
sx = -12345:cf c7
usx = 53191:cf c7
x = -12345:ff ff cf c7
ux = 53191:00 00 cf c7
```

Несмотря на то, что представление дополнительного кода $-12\ 345$ и представление величины без знака $53\ 191$ идентичны для длины 16-битового слова, они различны для длины 32-битового слова. В частности, $-12\ 345$ имеет шестнадцатеричное представление $0xFFFFFCF7$, тогда как $53\ 191$ имеет шестнадцатеричное представление $0x0000CFC7$. Первое расширено дополнительным знаковым разрядом: 16 копий наиболее значимого бита 1, имеющего шестнадцатеричное представление $0xFFFF$, добавлены в качестве ведущих битов. Последнее расширено 16 ведущими нулями и имеет шестнадцатеричное представление $0x0000$.

Можно ли доказать срабатывание расширения по дополнительному знаковому разряду? Доказать нужно, что

$$B2T_{w+k}([x_{w-1}, \dots, x_{w-1}, x_{w-1}, x_{w-2}, \dots, x_0]) = B2T_w([x_{w-1}, x_{w-2}, \dots, x_0]),$$

где в левой стороне выражения сделано k дополнительных копий бита x_{w-1} . Доказательство выполнено методом математической индукции. То есть, если можно доказать, что расширение знакового разряда на один бит сохраняет числовую величину, тогда это свойство будет выполняться при расширении знакового разряда на произвольное число битов. Итак, задача сокращается до доказательства того, что

$$B2T_{w+1}([x_{w-1}, \dots, x_{w-1}, x_{w-2}, \dots, x_0]) = B2T_w([x_{w-1}, x_{w-2}, \dots, x_0])$$

Расширение левой части выражения уравнением (2.2) дает следующее:

$$\begin{aligned} B2T_{w+1}([x_{w-1}, \dots, x_{w-1}, x_{w-2}, \dots, x_0]) &= -x_{w-1}2^w + \sum_{i=0}^{w-1} x^i 2^i \\ &= -x_{w-1}2^w + x_{w-1}2^{w-1} \sum_{i=0}^{w-2} x^i 2^i \\ &= -x_{w-1}(2^w - 2^{w-1}) + \sum_{i=0}^{w-2} x^i 2^i \\ &= -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x^i 2^i \\ &= B2T_w([x_{w-1}, x_{w-2}, \dots, x_0]) \end{aligned}$$

Основным использованным свойством здесь было то, что $-2^w + 2^{w-1} = -2^{w-1}$. Следовательно, объединенным эффектом добавления бита с весом -2^w и преобразования бита, имеющего вес -2^{w-1} , в бит с весом 2^{w-1} , является сохранение первоначального числового значения.

Стоит отметить, что относительный порядок преобразования данных одного размера в другой и преобразования величин без знака в величины со знаком может повлиять на поведение программы. Рассмотрим следующее дополнение предыдущего примера в листинге 2.8.

Листинг 2.8. Дополнение примера

```
1 unsigned uy = x; /* Мистика! */
2
3 printf ("uy = %u:\t", uy);
4 show_bytes ((byte_pointer) &uy, sizeof (unsigned));
```

Эта часть кода вызывает распечатку следующего:

uy = 4294954951:ff ff cf c7

Этим оказывается, что выражения

(unsigned) (int) sx/* 4294954951 */

и

(unsigned) (unsigned short) sx/* 53191 */

создают разные значения, даже несмотря на то, что типы первоначальных и конечных данных, — одинаковые. В первом выражении 16-битовый short сначала расширяется дополнительным знаковым разрядом в 32-битовый int, тогда как во втором выражении выполняется дополнение нулями.

УПРАЖНЕНИЕ 2.21

Рассмотрим следующие функции C:

```
int fun1 (unsigned word)
{
return (int) ((word << 24) >> 24);
}
```

```
int fun2 (unsigned word)
{
return ((int) word << 24) >> 24;
}
```

Предположим, что они выполняются на машине с длиной слова 32 битов и арифметикой дополнительных кодов. Также предположим, что сдвиги величин со знаком вправо выполняются арифметически, тогда как сдвиги величин без знака вправо — логически.

Заполните следующую таблицу значений этих функций для нескольких примерных аргументов:

w	fun1 (w)	fun1 (w)
127		
128		
255		
256		

Опишите расчеты, выполняемые каждой из этих функций.

2.2.6. Усечение чисел

Предположим, что вместо расширения величины дополнительными битами количество битов, представляющих число, сокращается. К примеру, это происходит в следующем листинге 2.9:

Листинг 2.9. Усечение чисел

```
1 int x = 53191
2 short sx = (short) x; /* -12345 */
3 inty = sx; /* 12345 */
```

На обычной 32-битовой машине, когда тип `x` преобразуется в `short`, 32-битовый `int` усекается до 16-битового `short`. Как уже отмечалось выше, такая 16-битовая комбинация является представлением дополнительного поразрядного кода числа $-12\ 345$. При преобразовании обратно в `int` расширение дополнительным знаковым разрядом переведет старшие 16 битов в единицы, в результате чего получится 32-битовое представление дополнительного поразрядного кода $-12\ 345$.

При усечении w -битового числа (\bar{x}) = $[x_{w-1}, x_{w-2}, \dots, x_0]$ в k -битовое число отбрасываются старшие $w - k$ битов, в результате чего получается битовый вектор $\bar{x}' = [x_{k-1}, x_{k-2}, \dots, x_0]$. Усечение числа может изменить его значение — это своего рода форма переполнения. Теперь рассмотрим числовую величину, которая получится в результате. Для числа без знака x результат усечения на k битов эквивалентен x по модулю 2^k . Это видно через применение операции степени к уравнению (2.1):

$$\begin{aligned} B2U_{w+1}([x_w, x_{w-1}, \dots, x_0]) \bmod 2^k &= \left[\sum_{i=0}^{w-1} x_i 2^i \right] \bmod 2^k \\ &= \left[\sum_{i=0}^{k-1} x_i 2^i \right] \bmod 2^k \\ &= \sum_{i=0}^{k-1} x_i 2^i \\ &= B2U_k([x_k, x_{k-1}, \dots, x_0]) \end{aligned}$$

В таком дифференцировании используется свойство $2^i \bmod 2^k = 0$ для любого $i \geq k$ и $\sum_{i=0}^{k-1} x' 2^i \leq \sum_{i=0}^{k-1} 2^i = 2^k - 1 < 2^k$.

Для числа x в двоичном дополнительном коде подобный же аргумент показывает, что $B2T_{w+1}([x_w, x_{w-1}, \dots, x_0]) \bmod 2^k = B2U_k([x_k, x_{k-1}, \dots, x_0])$. То есть, $x \bmod 2^k$ можно представить числом без знака, имеющим представление на уровне бита $[x_{k-1}, \dots, x_0]$. Однако, вообще говоря, данное усеченное число рассматривается здесь как число со знаком. Оно будет иметь числовую величину $U2T_k(x \bmod 2^k)$.

Подводя итог, эффект усечения выглядит следующим образом:

$$B2U_k([x_k, x_{k-1}, \dots, x_0]) = B2U_w([x_w, x_{w-1}, \dots, x_0]) \bmod 2^k \quad (2.7)$$

$$B2T_k([x_k, x_{k-1}, \dots, x_0]) = U2T_k(B2T_w([x_w, x_{w-1}, \dots, x_0]) \bmod 2^k) \quad (2.8)$$

УПРАЖНЕНИЕ 2.22

Предположим, что величина из четырех битов (представленная шестнадцатеричными цифрами от 0 до F) усекается в величину из трех битов (представленная шестнадцатеричными цифрами от 0 до 7). Заполните следующую таблицу, показывая влияние усечения для некоторых случаев в том, что касается интерпретаций комбинаций битов без знака в дополнительном двоичном коде:

Шестнадцатеричные		Без знака		С дополнительным кодом	
Первонач.	Усеченное	Первонач.	Усеченное	Первонач.	Усеченное
0	0	0		0	
3	3	3		3	
8	0	8		-8	
A	2	10		-6	
F	7	15		-1	

Объясните, как к этим случаям применяются уравнения (2.7) и (2.8).

УПРАЖНЕНИЕ 2.23

Рассмотрим следующий код, описывающий сумму элементов массива a , где количество элементов представлено параметром $length$:

```
1 /*ПРЕДУПРЕЖДЕНИЕ: данный код ошибочен*/
2 float sum_elements (float a [ ], unsigned length)
3 {
4 int i;
```

```

5 float result = 0;
6
7 for (i = 0; i <= length-1; i++)
8 result += a [i];
9 return result;
10 }

```

При запуске с аргументом `length`, равным 0, данный код должен возвращать 0.0. Вместо этого выдается ошибка памяти. Объясните, почему это происходит, и покажите способы исправления кода.

Одним из способов избежать подобных ошибок является отказ от использования чисел без знака. На самом деле, мало какие языки, кроме С, поддерживают целые числа без знака. Очевидно, что разработчики этих языков рассматривали их более серьезно, чем те того заслуживали. Например, Java поддерживает только целые числа со знаком и требует их реализации арифметикой дополнительных кодов. Обычный оператор сдвига вправо `>>` гарантированно выполняет арифметический сдвиг. Специальный оператор `>>>` определен для выполнения логического сдвига вправо.

Величины без знака очень полезны, если слова рассматриваются всего лишь как наборы битов, без какой бы то ни было числовой интерпретации. Это имеет место, например, при снабжении слова флагками, описывающими булевые условия. Адреса не имеют знака, поэтому системные программисты считают, что типы без знака имеют свои преимущества. Величины без знака также полезны при реализации математических пакетов для арифметики сравнений по модулю и арифметических операций с многократно увеличенной точностью, где числа представлены в виде массивов слов.

2.3. Целочисленная арифметика

Многие начинающие программисты с удивлением обнаруживают, что результатом сложения двух положительных чисел может стать число отрицательное, и что сравнение $x < y$ может дать иной результат, нежели сравнение $x - y < 0$. Эти свойства являются порождением конечной природы компьютерной арифметики. Понимание ее нюансов помогает программистам в создании более надежных кодов.

2.3.1. Приращение без знака

Рассмотрим два неотрицательных целых, x и y , таких что $0 \leq x, y \leq 2^n - 1$. При расчете их суммы диапазон $0 \leq x + y \leq 2^{n+1} - 2$. Каждое из этих чисел можно представить в виде чисел без знака, состоящих из w битов. Однако если вычислить их сумму, то, возможно, результатом станет диапазон $0 \leq x + y \leq 2^{w+1} - 2$. Представление этой суммы может потребовать $w + 1$ битов. Например, на рис. 2.4 показана схема функции $x + y$, когда x и y имеют четырехбитовые представления. Диапазон аргументов (показан по горизонтальным осям) составляет от 0 до 15, однако, диапазон суммы — от 0 до 30. Форма данной функции представляет собой наклонную плоскость. Если необходимо поддержать данную сумму в виде количества битов $w + 1$ и прибавить ее к другой величине, тогда могло бы потребоваться $w + 2$ битов и т. д. Такое прогрессирующее

"воспаление" длины слова означает, что для представления результатов арифметических операций в полной мере устанавливать границы длины слова нельзя. Некоторые языки программирования, например Lisp, на самом деле поддерживают арифметику бесконечной точности для обеспечения работы с произвольной (разумеется, в пределах машинной памяти) целочисленной арифметикой. Чаще всего языки программирования поддерживают арифметику фиксированной точности и, следовательно, такие операции, как приращение и умножение, отличаются от эквивалентных операций с целыми числами.

При длине слова 4 бита сумма может потребовать 5 битов, как показывает рис. 2.4. Когда $x + y$ больше 2^w , наступает переполнение (рис. 2.5).

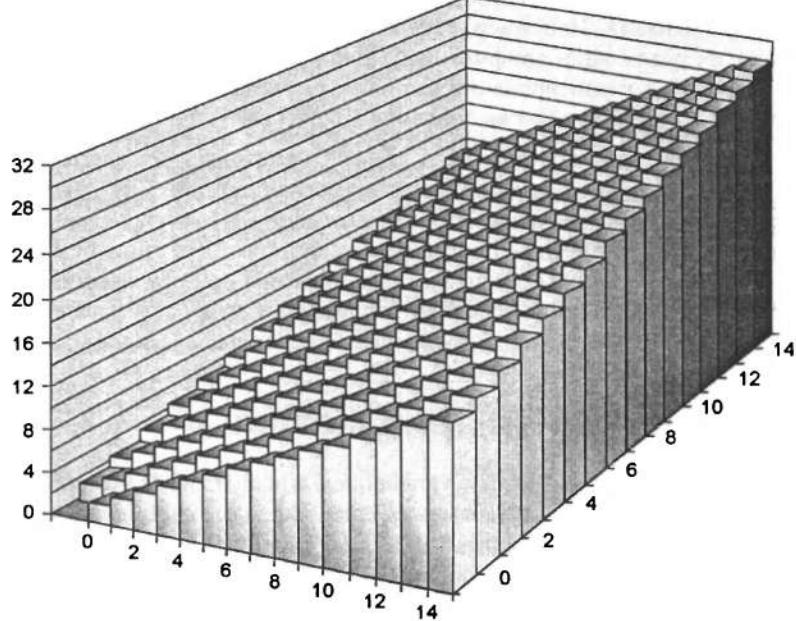


Рис. 2.4. Приращение без знака

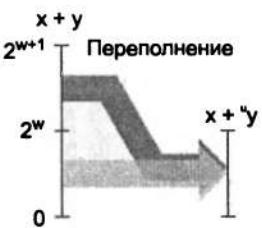


Рис. 2.5. Отношение между приращением целых чисел и чисел без знака

Арифметику чисел без знака можно рассматривать как форму арифметики сравнений по модулю. Сложение величин без знака эквивалентно сумме по модулю 2^w . Эту величину можно рассчитать простым отбрасыванием старшего разряда в $w + 1$ -битовом представлении $x + y$. Рассмотрим представление четырехбитового числа, где $x = 9$, а $y = 12$, имеющие представления [1001] и [1100], соответственно. Их сумма равна 21 и имеет 5-битовое представление [10101]. Однако если отбросить старший разряд, получится [0101], десятичное значение 5. Оно соответствует значению: $21 \bmod 16 = 5$.

Вообще, можно понять, что, если $x + y < 2^w$, тогда ведущий бит в $w + 1$ -битовом представлении суммы будет равен нулю, следовательно, его отбрасывание не изменит числового значения. С другой стороны, если $2^w \leq x + y \leq 2^{w+1}$, тогда ведущий бит в $w + 1$ -битовом представлении суммы будет равен единице, и его отбрасывание равно вычитанию 2^w из суммы. Оба эти случая проиллюстрированы на рис. 2.5. При этом получаем значение в диапазоне $0 \leq x + y - 2^w < 2^{w+1} - 2^w = 2^w$, что в точности является суммой x и y по модулю 2^w . Определим операцию $+_w$ для аргументов x и y так, что $0 \leq x, y < 2^w$ следующим образом:

$$x +_w y = \begin{cases} x + y, & x + y < 2^w \\ x + y - 2^w, & 2^w \leq x + y < 2^{w+1} \end{cases} \quad (2.9)$$

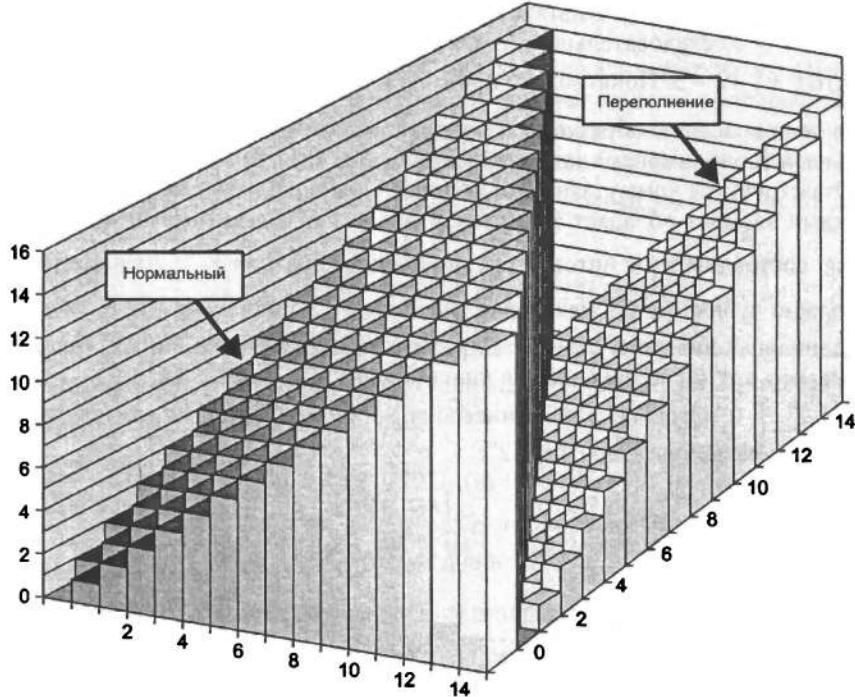


Рис. 2.6. Приращение без знака

При длине слова 4 битов приращение выполняется по модулю 16. Это именно тот результат, который получается в С при выполнении сложения двух w -битовых величин без знака.

Говорится, что арифметическая операция дает *переполнение*, когда полный целочисленный результат выходит за пределы длины слова типа данных. Как указано в уравнении (2.9), переполнение возникает тогда, когда сумма двух операндов составляет 2^w или более. На рис. 2.6 показана схема функции сложения числа без знака для длины слова $w = 4$. Сумма рассчитывается по модулю $2^4 = 16$. Когда $x + y < 16$, тогда переполнения не возникает, а $x + "y$ просто равно $x + y$. Это показано в области, формирующей наклонную плоскость, обозначенную Normal (Норма). Когда $x + y \geq 16$, тогда сложение переполняется с эффектом отрицательного приращения суммы на 16. Это показано в области, формирующей наклонную плоскость, обозначенную Overflow (Переполнение).

При выполнении программ С возникновение переполнений не сигнализируется как ошибка. Однако периодически может возникать необходимость проверки наличия переполнения. Например, предположим, что рассчитывается $s = x + "y$, а необходимо определить, верно ли $s = x + y$. Считается, что переполнение имеет место тогда, и только тогда, когда $s < x$ (или $s < y$). Обратите внимание, что $x + y \geq x$, следовательно, если s не переполнено, то в результате наверняка получится $s \geq x$. С другой стороны, если s переполнено, то в результате получаем $s = x + y - 2^w$. При условии, что $y < 2^w$, имеем $y - 2^w < 0$, следовательно, $s = x + y - 2^w < x$. В примере, приведенном ранее, видно, что $9 + "12 = 5$. Понятно, что произошло переполнение, потому что $5 < 9$.

Модульное приращение образует математическую структуру, известную, как *абелева группа*, названную именем датского математика Нильса Хенрика Абеля (1802—1829). Эта структура коммутативна и ассоциативна. Она имеет нейтральный элемент 0, и каждый элемент обладает аддитивной инверсией. Рассмотрим множество чисел без знака, состоящие из w битов, с операцией наращивания $+_w$. Для каждого значения x должно существовать некоторое значение $-_w x$ такое, что $_w x +_w x = 0$. Когда $x = 0$, аддитивная инверсия равна 0. Для $x > 0$ рассмотрим величину $2^w - x$. Обратите внимание, что это число находится в диапазоне $0 \leq 2^w - x < 2^w$, и $(x + 2^w - x) \bmod 2^w = 2^w \bmod 2^w = 0$. Отсюда, это — инверсия x при $+_w$. Эти два случая приводят к следующему уравнению для $0 \leq x < 2^w$:

$$-_w x = \begin{cases} x, & x = 0 \\ 2^w - x, & x > 0 \end{cases} \quad (2.10)$$

УПРАЖНЕНИЕ 2.24

Комбинацию битов с длиной $w = 4$ можно представить одной шестнадцатеричной цифрой. Для представления этих цифр без знака воспользуйтесь уравнением (2.10) и заполните следующую таблицу, обеспечивая величины и битовые представления (шестнадцатеричные) аддитивных инверсий без знака указанных цифр.

x		$-_w^u x$	
Шестн.	Десятичное	Шестн.	Десятичное
0			
3			
8			
A			
F			

2.3.2. Приращение в дополнительном коде

Аналогичная проблема возникает для приращения в дополнительных кодах. При наличии целых чисел x и y в диапазоне $-2^{w-1} \leq x, y \leq 2^{w-1} - 1$ их сумма находится в диапазоне $-2^{w-1} \leq x + y \leq 2^w - 2$, потенциально испытывая потребность в $w + 1$ битах для наиболее точного представления. Как и ранее, будем избегать бесконечно увеличивающихся размеров данных путем усечения представления до w битов. Впрочем, результат не столь прост в математическом отношении, нежели модульное приращение.

Побитовая сумма в дополнительном коде для двух чисел имеет то же самое представление, что и сумма без знака. Фактически, для выполнения приращения со знаком или без знака в большинстве компьютеров используется одна и та же машинная команда. Следовательно, можно определить приращение в дополнительном двоичном коде для длины слова w , обозначенное как $+_w'$ операндов x и y при $-2^{w-1} \leq x, y \leq 2^{w-1}$ как

$$x +_w' y = U2T_w(T2U_w(x) + {}_w'' T2U_w(y)) \quad (2.11)$$

По уравнению (2.3) $T2U_w(x)$ можно записать, как $x_{w-1}2^w + x$, а $T2U_w(y)$ как $y_{w-1}2^w + y$. Тогда используя свойство, что $+_w''$ просто приращение по модулю 2^w , вместе со свойствами модульного приращения получаем:

$$\begin{aligned} x +_w' y &= U2T_w(T2U_w(x) + {}_w'' T2U_w(y)) \\ &= U2T_w [(-x_{w-1}2^w + x + -y_{w-1}2^w + y) \bmod 2^w] \\ &= U2T_w [(x + y) \bmod 2^w] \end{aligned}$$

Термины $x_{w-1}2^w$ и $y_{w-1}2^w$ выпадают, поскольку они равны 0 по модулю 2^w .

Для лучшего понимания этого количества определим z как целочисленную сумму $z = x + y$, z' как $z' = z \bmod 2^w$, а z'' — как $z'' = U2T_w(z')$. На рис. 2.7 при $x + y$ меньше, чем -2^{w-1} , имеет место отрицательное переполнение. Когда $x + y$ больше, чем $2^{w-1} + 1$, переполнение положительное. Величина z'' равна $x + {}_w'y$.

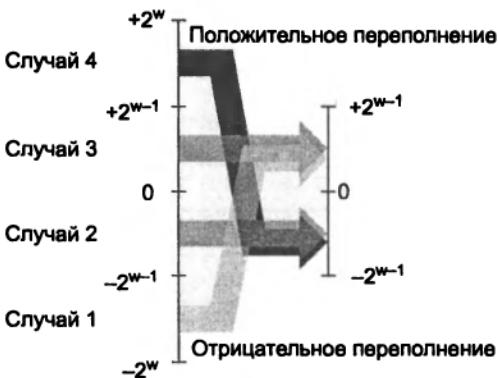


Рис. 2.7. Отношение между целочисленным приращением и приращением в дополнительном коде

Анализ можно разделить на четыре случая, как показано на рис. 2.7.

- $-2^w \leq z < -2^{w-1}$. Тогда имеем $z' = z + 2^w$. Это дает $0 \leq z' < -2^{w-1} + 2^w = 2^{w-1}$. При рассмотрении уравнения (2.6) видно, что z' находится в таком диапазоне, что $z'' = z'$. Такой случай называется *отрицательным переполнением*. Было выполнено сложение двух отрицательных чисел x и y (только так можно получить $z < -2^{w-1}$), и получен неотрицательный результат $z'' = x + y + 2^w$.
- $-2^{w-1} \leq z < 0$. Имеем $z' = z + 2^w$, что дает $-2^{w-1} + 2^w = 2^{w-1} \leq z' < 2^w$. При рассмотрении уравнения (2.6) видно, что z' находится в таком диапазоне, что $z'' = z' - 2^w$, и, следовательно, $z'' = z' - 2^w = z + 2^w - 2^w = z$. То есть, полученная сумма z'' в дополнительном двоичном коде равна целочисленной сумме $x + y$.
- $0 \leq z < 2^{w-1}$. Имеем $z' = z$, что дает $0 \leq z' < 2^{w-1}$. Опять сумма z'' в дополнительном двоичном коде равна целочисленной сумме $x + y$.
- $2^{w-1} \leq z < 2^w$. Имеем $z' = z$, что дает $2^{w-1} \leq z' < 2^w$. Но в этом диапазоне имеем $z'' = z' - 2^w$, что дает $z'' = x + y - 2^w$. Такой случай называется *положительным переполнением*. Было выполнено сложение двух положительных чисел x и y (только так можно получить $z \geq 2^{w-1}$) и получен отрицательный результат $z'' = x + y - 2^w$.

Предварительным анализом было показано, что когда операция $+'_w$ применяется к значениям x и y в диапазоне $-2^{w-1} \leq x, y \leq 2^{w-1} - 1$, тогда получается следующее:

$$x +'_w y = \begin{cases} x + y - 2^w, & 2^{w-1} \leq x + y \\ x + y, & -2^{w-1} \leq x + y < 2^{w-1} \\ x + y + 2^w, & x + y < -2^{w-1} \end{cases} \quad \begin{array}{l} \text{Положительное переполнение} \\ \text{Норма} \\ \text{Отрицательное переполнение} \end{array} \quad (2.12)$$

В качестве иллюстрации в табл. 2.13 несколько примеров четырехбитового приращения в дополнительном двоичном коде. Каждому примеру сопоставлен случай, к которому он относится в выводе уравнения (2.12). Обратите внимание, что $2^4 = 16$, и, следовательно, отрицательное переполнение выдает результат на 16 больше целочис-

ленной суммы, а положительное переполнение — на 16 меньше. Сюда включены представления операндов на битовом уровне и результат. Учтите, что данный результат можно получить выполнением двоичного сложения операндов и усечением его (результата) до четырех битов.

Таблица 2.13. Примеры приращения в дополнительном двоичном коде

x	y	$x + y$	$x + \frac{1}{4}y$	Случай
-8 [1000]	-5 [1011]	-13	3 [0011]	1
-8 [1000]	-8 [1000]	-16	0 [0000]	1
-8 [1000]	5 [0101]	-3	-3 [1101]	2
2 [0010]	5 [0101]	7	7 [0111]	3
5 [0101]	5 [0101]	10	-6 [1010]	4

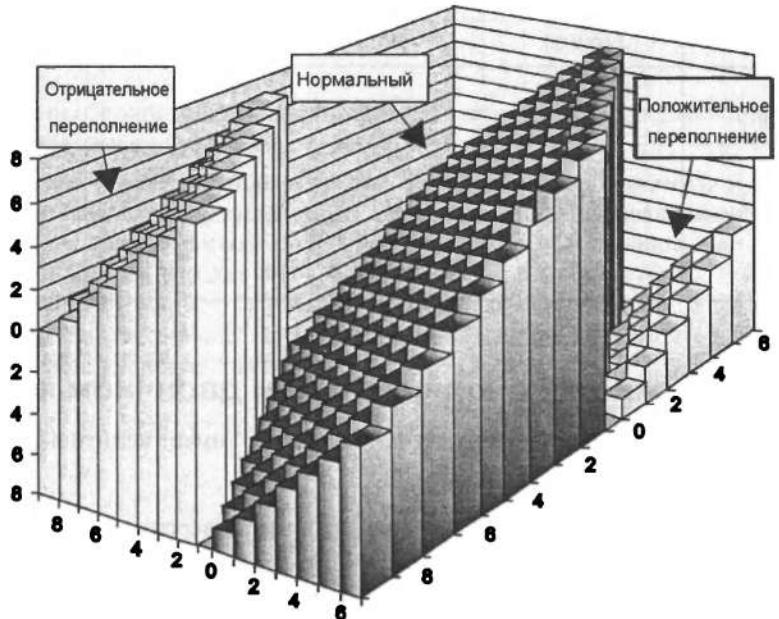


Рис. 2.8. Приращение в дополнительном двоичном коде

На рис. 2.8 представлено приращение в дополнительном двоичном коде длины слова $w = 4$. Диапазон операндов от -8 до 7 . Когда $x + y < -8$, тогда приращение в дополнительном коде имеет отрицательное сокращение разрядов, что вызывает увеличение суммы на 16 . Когда $-8 \leq x + y < 8$, тогда приращение выдает $x + y$. Когда $x + y \geq 8$, тогда приращение имеет положительное переполнение, вызывающее уменьшение суммы на 16 . Каждый из этих трех диапазонов образует на схеме наклонную плоскость.

Уравнение (2.12) позволяет выявить случаи, когда имеет место переполнение. Когда x и y отрицательны, но $x + y \geq 0$, тогда налицо отрицательное переполнение. Когда x и y положительны, но $x + y < 0$, тогда имеет место положительное переполнение.

УПРАЖНЕНИЕ 2.25

Заполните приведенную ниже таблицу в стиле табл. 2.13. Приведите целые величины 5-битовых аргументов, значения их сумм, как целочисленной, так и в дополнительном двоичном коде, представление на битовом уровне суммы в дополнительном коде и случай из вывода уравнения (2.12).

x	y	$x + y$	$x + y'$	Случай
[10000]	[10101]			
[10000]	[10000]			
[11000]	[00111]			
[11110]	[00101]			
[01000]	[01000]			

2.3.3. Отрицание в дополнительном двоичном коде

Видно, что каждое число x в диапазоне $-2^{w-1} \leq x < 2^{w-1}$ имеет аддитивную инверсию при $+y'$.

Во-первых, для $x \neq -2^{w-1}$ видно, что его аддитивная инверсия равна просто $-x$. То есть, имеем $-2^{w-1} < -x < 2^{w-1}$ и $-x + y' = -x + x = 0$. С другой стороны, для $x = -2^{w-1}$ в $TMin_w$ $x = -2^{w-1}$ нельзя представить в виде w -битового числа. Утверждаем, что эта особая величина сама является аддитивной инверсией при $+y'$.

Величина $-2^{w-1} + {}_w' - 2^{w+1}$ представлена третьим случаем уравнения (2.12), поскольку $-2^{w-1} + {}_w' + -2^{w-1} = -2^w$. Это дает $-2^{w+1} + {}_w' - 2^{w+1} = -2^w + 2^w = 0$. Из проведенного анализа операцию отрицания в дополнительном коде $-{}_w'$ для x в диапазоне $-2^{w-1} \leq x < 2^{w-1}$ можно определить следующим образом:

$$-{}_w' x = \begin{cases} -2^{w-1}, x = -2^{w-1} \\ -x, x > -2^{w-1} \end{cases} \quad (2.13)$$

УПРАЖНЕНИЕ 2.26

Комбинацию битов с длиной $w = 4$ можно представить одной шестнадцатеричной цифрой. Для представления этих цифр в дополнительном двоичном коде заполните следующую таблицу для определения аддитивных инверсий показанных цифр.

x		$-\frac{1}{4} x$	
Шестн.	Десятичное	Десятичное	Шестн.
0			
3			
8			
A			
F			

Какие видны особенности комбинаций битов, созданных отрицанием в дополнительном коде и отрицанием без знака (см. упр. 2.24)?

Широко известной методикой выполнения отрицания в дополнительном коде на битовом уровне является дополнение битов с последующим приращением результата. В С это можно записать как $\sim x + 1$. Для доказательства правильности данной методики обратите внимание, что для любого одиночного бита x , имеется $\sim x = 1 - x$. Пусть \bar{x} — вектор длиной w , а $x = B2T_w(\bar{x})$ — представляемое этим вектором число в дополнительном двоичном коде. По уравнению (2.2) дополненный битовый вектор $\sim \bar{x}$ имеет следующее числовое значение:

$$\begin{aligned} B2T_w(\sim \bar{x}) &= -(1 - x_{w-1})2^{w-1} + \sum_{i=0}^{w-2} (1 - x_i)2^i \\ &= \left[-2^{w-1} \sum_{i=0}^{w-2} 2^i \right] - \left[-x_{w-1} 2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i \right] \\ &= [-2^{w-1} + 2^{w-1} - 1] - B2T_w(\bar{x}) \\ &= -1 - x \end{aligned}$$

Основным упрощением в вышеприведенном выводе является то, что $\sum_{i=0}^{w-2} 2^i = 2^{w-1} - 1$.

Из этого следует, что при приращении $\sim \bar{x}$ получаем $-x$.

Для приращения числа x , представленного на битовом уровне как $\bar{x} = [x_{w-1}, x_{w-2}, \dots, x_0]$, определите операцию приращения следующим образом. Пусть k — положение крайнего справа нуля такое, что \bar{x} принимает форму $[x_{w-1}, x_{w-2}, \dots, x_{k+1}, 0, 1, \dots, 1]$. Теперь можно определить, что приращение $\text{incr}(\bar{x})$ принимает форму $[x_{w-1}, x_{w-2}, \dots, x_{k+1}, 1, 0, \dots, 0]$. Для особого случая, когда представление x на битовом уровне — $[1, 1, \dots, 1]$, определите $\text{incr}(\bar{x})$ как $[0, \dots, 0]$. Для доказательства того, что $\text{incr}(\bar{x})$ дает представление на битовом уровне $x + {}'_w 1$, рассмотрите следующие случаи:

- Когда $(\bar{x}) = [1, 1, \dots, 1]$, имеем $x = -1$. Приращенная величина $\text{incr}(\bar{x}) = [0, \dots, 0]$ имеет числовое значение 0.
- Когда $k = w-1$, $(\bar{x}) = [0, 1, \dots, 1]$, имеем $x = TMax_w$. Приращенная величина $\text{incr}(\bar{x}) = [1, 0, \dots, 0]$ имеет числовое значение $TMin_w$. Из уравнения (2.12) видно, что $TMax_w + {}'_w 1$ один из случаев положительного переполнения, дающий значение $TMin_w$.
- Когда $k < w-1$, $x \neq TMax_w$ и $x \neq -1$, можно видеть, что младшие $k+1$ битов $\text{incr}(\bar{x})$ имеют числовое значение 2^k , тогда как младшие $k+1$ битов (\bar{x}) имеют числовое значение $\sum_{i=0}^{k-1} 2^i = 2^k - 1$. Старшие биты $w-k+1$ имеют совпадающие числовые значения. Таким образом, $\text{incr}(\bar{x})$ имеет числовое значение $x + 1$. Кроме этого, для $x \neq TMax_w$ прибавление 1 к x не вызовет переполнения, и следовательно, $x + {}'_w 1$ также имеет числовое значение $x + 1$.

В качестве иллюстрации в табл. 2.14 показано, как дополнение и приращение влияют на числовые значения нескольких четырехбитовых векторов.

Таблица 2.14. Примеры дополнения и приращения четырехбитовых чисел

\bar{x}		$\sim \bar{x}$		$\text{incr}(\sim \bar{x})$	
[0101]	5	[1010]	-6	[1011]	-5
[0111]	7	[1000]	-8	[1001]	-7
[1100]	-4	[0011]	3	[0100]	4
[0000]	0	[1111]	-1	[0000]	0
[1000]	-8	[0111]	7	[1000]	-8

2.3.4. Умножение без знака

Целые числа x и y в диапазоне $0 \leq x, y \leq 2^w - 1$ можно представить как w -битовые числа без знака, однако их произведение $x \cdot y$ может находиться в диапазоне от 0

до $(2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$. Для представления этого может потребоваться, как минимум, $2w$ битов. Вместо этого умножение без знака в С определяется для получения значения, представленного младшими w битами целого произведения 2^w битов. По уравнению (2.7), это эквивалентно расчету произведения по модулю 2^w . Таким образом, воздействие операции w -битового умножения без знака $*_w$:

$$x *_w y = (x \cdot y) \bmod 2^w \quad (2.14)$$

Хорошо известно, что арифметические операции над абсолютными значениями чисел образуют кольцо. Следовательно, можно сделать вывод, что арифметика w -битовых чисел без знака образует кольцо $\langle \{0, \dots, 2^w - 1\}, +_w, *_w, -_w, 0, 1 \rangle$.

2.3.5. Умножение в дополнительном двоичном коде

Целые числа x и y в диапазоне $-2^{w-1} \leq x, y \leq 2^{w-1} - 1$ можно представить как w -битовые числа в дополнительном коде, однако их произведение $x \cdot y$ может находиться в диапазоне от -2^{w-1} до $(2^{w-1} - 1) = 2^{2w-2} + 2^{w-1} - 2^{w-1} = 2^{2w-2}$. Для представления в дополнительном коде это может потребовать как минимум $2w$ битов; большинству случаев подошло бы $2w - 1$ битов, однако особый случай 2^{2w-2} требует полные $2w$ битов (для включения знакового разряда 0). Вместо этого умножение со знаком в С обычно выполняется усечением произведения $2w$ битов до w битов. По уравнению (2.8), воздействие операции w -битового умножения в дополнительном коде $*'_w$:

$$x *'_w y = U2T_w((x \cdot y) \bmod 2^w) \quad (2.15)$$

Утверждается, что представление на битовом уровне операции умножения идентично как для умножения чисел без знака, так и для умножения в дополнительном двоичном коде. То есть, при наличии векторов (\bar{x}) и (\bar{y}) с длиной w представление на битовом уровне произведения без знака $B2U_w(\bar{x}) *_w B2U_w(\bar{y})$ идентично представлению на битовом уровне произведения в дополнительном коде $B2T_w(\bar{x}) *'_w B2T_w(\bar{x})$. Этим подразумевается, что машина использует один тип команды умножения как целых чисел со знаком, так и без знака.

Для демонстрации этого пусть $x = B2T_w(\bar{x})$, а $y = B2T_w(\bar{y})$ величины в дополнительном коде, обозначенные этими комбинациями битов, и пусть $x' = B2U_w(\bar{x})$ и $y' = B2U_w(\bar{y})$ величины без знака. Из уравнения (2.3) имеем

$$\begin{aligned} x' &= x + x_{w-1}2^w \\ y' &= y + y_{w-1}2^w \end{aligned}$$

Расчет произведения этих значений по модулю 2^w дает следующее:

$$\begin{aligned} (x' \cdot y') \bmod 2^w &= [(x + x_{w-1}2^w) \cdot (y + y_{w-1}2^w)] \bmod 2^w \\ &= [x \cdot y + (x_{w-1}y + y_{w-1}x)2^w + x_{w-1}y_{w-1}2^{2w}] \bmod 2^w \\ &= (x \cdot y) \bmod 2^w \end{aligned} \quad (2.16)$$

Таким образом, младшие w битов $x \cdot y$ и $x' \cdot y'$ идентичны.

В качестве иллюстрации в табл. 2.15 показаны результаты умножения различных трехбитовых чисел. Для каждой пары операндов на битовом уровне выполняется как умножение без знака, так и в дополнительном двоичном коде. Обратите внимание, что усеченное произведение без знака всегда равно $x \cdot y \bmod 8$, и что представления на битовом уровне обоих усеченных произведений идентичны.

Таблица 2.15. Умножение трехбитовых чисел без знака и в дополнительном коде

Режим	x	y	$x \cdot y$	Усеченное $x \cdot y$
Без знака	5 [101]	3 [011]	15 [001111]	7 [111]
В доп. коде	-3 [101]	3 [011]	-9 [110111]	-1 [111]
Без знака	4 [100]	7 [111]	28 [011100]	4 [100]
В доп. коде	-4 [100]	-1 [111]	4 [000100]	-4 [100]
Без знака	3 [011]	3 [011]	9 [001001]	1 [001]
В доп. коде	3 [011]	3 [011]	9 [001001]	1 [001]

Несмотря на то, что представления на битовом уровне полных произведений могут различаться, представления усеченных произведений идентичны.

УПРАЖНЕНИЕ 2.27

Заполните следующую таблицу с указанием результатов умножения различных трехбитовых чисел по табл. 2.15:

Режим	x	y	$x \cdot y$	Усеченное $x \cdot y$
Без знака	[110]	[010]		
В доп. коде	[110]	[010]		
Без знака	[000]	[111]		
В доп. коде	[001]	[111]		
Без знака	[111]	[111]		
В доп. коде	[111]	[111]		

Видно, что арифметика над числами w -битов без знака и арифметика дополнительных кодов — изоморфны, т. е. $+_w^r$, $-_w^r$ и $*_w^r$ оказывают тот же самый эффект на битовом уровне, что и $+_w^l$, $-_w^l$ и $*_w^l$. Из этого можно сделать вывод, что арифметика дополнительных кодов образует кольцо $\langle \{-2^{w-1}, \dots, 2^{w-1} - 1\}, +_w^l, *_w^l, -_w^l, 0, 1 \rangle$.

2.3.6. Умножение на степени двух

На большинстве машин команда целочисленного умножения работает довольно медленно; она требует от 12 циклов синхронизации, тогда как другие целочисленные операции, например, сложение, вычитание, операции на уровне бита и сдвиги, требуют только одного цикла синхронизации. Вследствие этого, одной из важнейших оптимизаций, используемых компиляторами, является попытка замещения умножений постоянными множителями в комбинации с операциями сдвига и сложения.

Пусть x — целое число без знака, представленное комбинацией битов $[x_{w-1}, x_{w-2}, \dots, x_0]$. Тогда для любого $k \geq 0$ утверждается, что представление на битовом уровне $x2^k$ дается $[x_{w-1}, x_{w-2}, \dots, x_0, 0, \dots, 0]$, где в правую часть добавлено k нулей. Это свойство можно вывести с помощью уравнения (2.1):

$$\begin{aligned} B2U_{w+k}([x_{w-1}, x_{w-2}, \dots, x_0, 0, \dots, 0]) &= \sum_{i=0}^{w-1} x_i 2^{i+k} \\ &= \left[\sum_{i=0}^{w-1} x_i 2^i \right] \cdot 2^k \\ &= x2^k \end{aligned}$$

Для $k < w$ можно усечь сдвинутый битовый вектор до длины w , что в результате даст $[x_{w-k-1}, x_{w-k-2}, \dots, x_0, 0, \dots, 0]$. По уравнению (2.7) данный битовый вектор имеет числовое значение $x2^k$ по модулю $2^w = x * w 2^k$. Таким образом, для переменной x без знака выражение $x \ll k$ эквивалентно $x * \text{pwr2k}$. В частности, можно рассчитать `pwr2k` как `10 << k`.

На основании тех же доводов можно доказать, что для числа x в дополнительном двоичном коде с комбинацией битов $[x_{w-1}, x_{w-2}, \dots, x_0]$ и любом k в диапазоне $0 \leq k < w$ комбинация битов $[x_{w-k-1}, \dots, x_0, 0, \dots, 0]$ будет представлением в дополнительном коде выражения $x * w 2^k$.

Обратите внимание на то, что умножение на степень двух может вызвать переполнение либо с арифметикой над числами без знака, либо с арифметикой дополнительных кодов. Полученный здесь результат доказывает, что даже в этом случае эффект достигается с помощью сдвигов.

УПРАЖНЕНИЕ 2.28

В главе 3 будет продемонстрировано, как команда `leal` на процессоре, совместимом с Intel, может выполнять расчеты формы $a \ll k + b$, где k — либо 0, 1 или 2, а b — либо 0, либо некоторая программная величина. Компилятор часто пользуется этой командой для выполнения умножения на постоянные множители. Например, можно рассчитать $3 * a$ как $a \ll 1 + a$.

Какие числа, кратные a , можно рассчитать с помощью данной команды?

2.3.7. Деление на степени двух

На многих машинах целочисленное деление осуществляется еще более медленно, чем целочисленное умножение; первое требует до 30 циклов синхронизации. Деление на степень двух также можно выполнить с помощью операций сдвига, но использоваться будет сдвиг вправо, а не влево. Эти два разных сдвига — логический и арифметический — обслуживаются числами без знака и в дополнительном двоичном коде, соответственно.

Целочисленное деление всегда округляется до нуля. Для $x \geq 0$ и $y > 0$ результат должен составить $\lfloor x/y \rfloor$, где для любого вещественного числа a $\lfloor a \rfloor$ определяется как уникальное целое число a' , такое, что $a' \leq a < a' + 1$. Для примера: $\lfloor 3.14 \rfloor = 3$, $\lfloor -3.14 \rfloor = -4$, а $\lfloor 3 \rfloor = 3$.

Рассмотрим эффект применения логического сдвига вправо к числу без знака. Пусть x — целое число без знака, представленное комбинацией битов $[x_{w-1}, x_{w-2}, \dots, x_0]$, а k находится в диапазоне $0 \leq k < w$. Пусть x' — число без знака с $w - k$ -битовым представлением $[x_{w-1}, x_{w-2}, \dots, x_k]$, а x'' — число без знака с k -битовым представлением $[x_{k-1}, \dots, x_0]$. Утверждается, что $x' = \lfloor x/2^k \rfloor$. Для просмотра по уравнению (2.1) имеем $x = \sum_{i=0}^{w-1} x_i 2^i$, $x' = \sum_{i=k}^{w-k-1} x_i 2^{i-k}$ и $x'' = \sum_{i=0}^{k-1} x_i 2^i$. Таким образом, x можно записать как $x = 2^k x' + x''$. Обратите внимание, что $0 \leq x'' \leq \sum_{i=0}^{k-1} 2^i = 2^k - 1$, и отсюда $0 \leq x'' < 2^k$, подразумевая, что $\lfloor x''/2^k \rfloor = 0$. Следовательно, $\lfloor x/2^k \rfloor = \lfloor x' + x''/2^k \rfloor = x' + \lfloor x''/2^k \rfloor = x'$.

Заметьте, что выполнение логического сдвига вправо битового вектора $[x_{w-1}, x_{w-2}, \dots, x_0]$ на k дает битовый вектор $[0, \dots, 0, x_{w-1}, x_{w-2}, \dots, x_k]$.

Данный битовый вектор имеет величину x' . То есть, логически сдвиг числа без знака вправо на k эквивалентен его делению на 2^k . Поэтому, для переменной без знака x выражение $x \gg k$ эквивалентно выражению $x/\text{pwr}2k$, где $\text{pwr}2k$ равно 2^k .

Теперь рассмотрим эффект от применения арифметического сдвига вправо к числу в дополнительном двоичном коде. Пусть x — целое число в дополнительном коде, представленное комбинацией битов $[x_{w-1}, x_{w-2}, \dots, x_0]$, а k находится в диапазоне $0 \leq k < w$. Пусть x' — число в дополнительном коде с $w - k$ -битовым представлением $[x_{w-1}, x_{w-2}, \dots, x_k]$, а x'' — число без знака, представленное младшими k битами $[x_{k-1}, \dots, x_0]$. Посредством более простого анализа, нежели в случае с числом без знака, имеем $x = 2^k x' + x''$ и $0 \leq x'' < 2^k$; в результате получается $x' = \lfloor x/2^k \rfloor$. Более того, обратите внимание на то, что сдвиг битового вектора $[x_{w-1}, x_{w-2}, \dots, x_0]$ арифметически вправо на k дает битовый вектор

$$[x_{w-1}, \dots, x_{w-1}, x_{w-1}, x_{w-2}, \dots, x_k],$$

являющийся дополнительным знаковым разрядом из $w - k$ битов до w битов $[x_{w-1}, x_{w-2}, \dots, x_k]$. Таким образом, данный сдвинутый битовый вектор является представлением в дополнительном двоичном коде $\lfloor x/y \rfloor$.

Для $x \geq 0$ приведенный анализ показывает, что данный сдвинутый результат представляет собой нужное значение. Однако для $x < 0$ и $y > 0$ результат целочисленного деления должен быть $\lceil x/y \rceil$, где для любого вещественного числа a $\lceil a \rceil$ определяется как уникальное целое число a' , такое, что $a' - 1 \leq a < a'$. То есть, целочисленное деление должно округлять отрицательные результаты вверх до нуля. Например, выражение $-5/2$ дает результат -2 . Следовательно, сдвиг отрицательного числа на k не эквивалентно его делению на 2^k , когда имеет место округление. Например, четырехбитовое представление -5 равно $[1011]$. Если арифметически сдвинуть эту величину вправо на единицу, то результатом будет $[1101]$, что является представлением -3 в дополнительном двоичном коде.

Подобное "ненадлежащее" округление можно подкорректировать "смещением" величины перед сдвигом. Данная методика использует свойство $\lceil x/y \rceil = \lfloor (x+y-1)/y \rfloor$ для целых чисел x и y при $y > 0$. Следовательно, для $x < 0$, если перед выполнением сдвига вправо к x прибавить $2^k - 1$, то получим корректно округленный результат. Данный анализ показывает, что для машины в дополнительном коде, использующей арифметические сдвиги вправо, выражение C:

```
(x<0 ? (x + (1<<k - 1) : x) >>k
```

эквивалентно выражению $x/\text{pwr2k}$, где pwr2k равно 2^k . Например, для деления -5 на 2 сначала добавляется смещение $2 - 1 = 1$ с получением $[1100]$. Сдвиг данного выражения вправо на единицу арифметически дает комбинацию битов $[1110]$, что является представлением в дополнительном коде -2 .

УПРАЖНЕНИЕ 2.29

В приведенном коде опущены определения констант M и N:

```
#define M/* Мистическое число 1 */
#define N/* Мистическое число 2 */
int arith (int x, int y)
{
int result = 0
result = x*M + y/N; /* Мистические числа M и N. */
return result;
}
```

Этот код скомпилирован для конкретных величин M и N. Компилятор оптимизировал умножение и деление, используя рассмотренные методы. Далее приведен перевод генерированного машинного кода назад в C:

```
/* Перевод компонующего автокода для арифметических операций */
int optariph (int x, int y)
{
int t = x;
x <= 4;
x -= t;
```

```

if (y < 0) y += 3;
y >>= 2; /* Арифметический сдвиг */
return x+y;
)

```

Каковы значения m и n?

УПРАЖНЕНИЕ 2.30

Предположим выполнение кода на 32-битовой машине с арифметикой дополнительных кодов для величин со знаком. Сдвиги вправо выполняются арифметически для величин со знаком и логически — для величин без знака. Переменные объявлены и инициализированы следующим образом:

```

int x = foo (); /* Произвольная величина */
int y = bar (); /* Произвольная величина */

unsigned ux = x;
unsigned uy = y;

```

Для каждого из следующих выражений С либо аргументируйте истинность при всех значениях x и y, либо приведите значения x и y, при которых оно ложно:

1. $(x \geq 0) \mid\mid ((2*x) < 0)$
2. $(x \& 7) != 7 \mid\mid (x \ll 30 < 0)$
3. $(x * x) \geq 0$
4. $x < 0 \mid\mid -x \leq 0$
5. $x > 0 \mid\mid -x \geq 0$
6. $x*y == ux*uy$
7. $\sim x*y + uy*ux == -y$

2.4. Числа с плавающей точкой

Представление чисел с плавающей точкой рациональных чисел имеет форму $V = x \times 2^y$. Оно полезно для выполнения расчетов, включающих в себя очень большие числа ($|V| > 0$), числа, очень близкие к 0 ($|V| \ll 1$) и, в общем смысле, в качестве приближения к реальной арифметике.

До 80-х годов прошлого века каждый производитель компьютерной техники изобретал как свои собственные правила представления чисел с плавающей точкой, так и подробности выполняемых с ними операций. Кроме того, очень часто они не заботились о точности операций, считая, что скорость и простота реализации имели куда большее значение, нежели числовая точность.

К 1985 году все изменилось с появлением Стандарта 754 (IEEE) — тщательно проработанной методики представления чисел с плавающей точкой и выполняемых с ними

операций. Первые попытки этой деятельности были предприняты в 1976 году (их спонсировала корпорация Intel), результатом стало создание микросхемы 8087, обеспечивающей поддержку чисел с плавающей точкой для процессора 8086. В помощь проектированию стандарта, который бы мог поддерживать числа с плавающей точкой для будущих процессоров, в качестве консультанта был приглашен Уильям Кахан, профессор университета Беркли в Калифорнии. Ему был дан карт-бланш в объединении усилий комиссии по разработке универсального промышленного стандарта под эгидой Института инженеров по электротехнике и электронике (IEEE). Комиссия единогласно приняла стандарт, очень похожий на тот, что Кахан создал для Intel. Сейчас практически все компьютеры поддерживают феномен под названием *плавающая точка IEEE*. Этим был сделан значительный шаг вперед в плане переносимости научных программных приложений между разными компьютерами.

Об институте IEEE

Институт инженеров по электротехнике и электронике (IEEE) — профессиональное сообщество, охватывающее все электронные и компьютерные технологии. Осуществляет издание научных журналов, спонсирует конференции и организует комитеты по формализации стандартов на темы от электропередачи до проектирования программного обеспечения.

В данном разделе будет рассмотрено представление числе в формате плавающей точки IEEE. Также не будут обойдены вниманием вопросы округления, когда число нельзя точно представить в определенном формате, по причине чего его приходится округлять в плюс или минус. Далее обсуждаются математические свойства сложения, умножения и операторов отношения. Многие программисты считают лучшей стороной чисел с плавающей точкой интерес к их решению, а худшей — их запутанность и общую неясность. В данной книге будет показано, что формат IEEE весьма прост и понятен, поскольку он основан на небольшом и постоянном наборе принципов.

2.4.1. Дробные двоичные числа

Первым шагом к пониманию чисел с плавающей точкой является рассмотрение двоичных чисел с дробными значениями. Для начала рассмотрим более знакомую систему десятичного счисления. В последней используется представление в форме: $d_m d_{m-1} \dots d_1 d_0 . d_{-1} d_{-2} \dots d_{-n}$, где каждая десятичная цифра d_i находится в диапазоне от 0 до 9. Такая запись представляет число d , определенное как

$$d = \sum_{i=-n}^m 10^i \times d_i$$

Взвешивание цифр определено относительно символа десятичной точки, означающей, что цифры в левой части взвешиваются по положительным степеням 10, выдавая целочисленные значения, а цифры в правой части взвешиваются по отрицательным степеням 10, выдавая дробные значения. Например, 12.34_{10} представляет число $1 \times 10^1 + 2 \times 10^0 + 3 \times 10^{-1} + 4 \times 10^{-2} = 12 \frac{34}{100}$.

Рассмотрим, по аналогии, запись формы $b_m b_{m-1} \dots b_1 b_0.b_1 b_2 \dots b_n$, где каждый двоичный разряд (бит) b_i находится в диапазоне от 0 до 1. Такая запись представляет число b , определенное как

$$b = \sum_{i=-n}^m 2^i \times b_i \quad (2.17)$$

Теперь символ разделительной точки становится *двоичной точкой*, когда цифры в левой части взвешиваются по положительным степеням двойки, а в правой части — по отрицательным степеням двойки. Например, 101.11_2 представляет число $1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} = 4 + 0 + 1 + \frac{1}{2} + \frac{1}{4} = 5\frac{3}{4}$.

Из уравнения (2.17) можно легко понять, что сдвиг двоичной точки на одну позицию влево оказывает эффект деления данного числа на два. Например, в то время, как 101.11_2 представляет число $5\frac{3}{4}$, 10.111_2 представляет число $2 + 0 + \frac{1}{2} + \frac{1}{4} + 1/8 = 2\frac{7}{8}$.

Подобным же образом, сдвиг двоичной точки на одну позицию вправо оказывает эффект умножения данного числа на два. Например, 1011.1_2 представляет число $8 + 0 + 2 + 1 + \frac{1}{2} = 11\frac{1}{2}$.

Обратите внимание, что в форме $0.11\dots1_2$ представляют числа меньше единицы. Например, 0.111111_2 представляет $\frac{63}{64}$. Для представления таких величин здесь используем сокращение $1.0 - \epsilon$.

При условии, что рассматриваются только кодировки конечной длины, в десятичной системе счисления нельзя точно представить такие числа, как $\frac{1}{3}$ и $\frac{5}{6}$. Подобным же образом в дробном двоичном обозначении можно представить только числа, которые можно записать как $x \times 2^y$. Другие величины выражаются только в приближенном виде. Например, несмотря на то, что число $\frac{1}{5}$ можно аппроксимировать с нарастающей точностью путем удлинения двоичного представления, его нельзя представить в точности, как дробное двоичное число (табл. 2.16).

Таблица 2.16. Двоичное и десятичное представления

Представление	Значение	Десятичное
0.0_2	0	0.0_{10}
0.01_2	$\frac{1}{4}$	0.25_{10}
0.10_2	$\frac{2}{8}$	0.25_{10}
0.0011_2	$\frac{3}{16}$	0.1875_{10}
0.00110_2	$\frac{6}{32}$	0.1875_{10}

Таблица 2.16 (окончание)

Представление	Значение	Десятичное
0.001101 ₂	$\frac{13}{64}$	0.203125 ₁₀
0.0011010 ₂	$\frac{26}{128}$	0.203125 ₁₀
0.00110011 ₂	$\frac{51}{256}$	0/19921875 ₁₀

УПРАЖНЕНИЕ 2.31

Введите недостающую информацию в следующую таблицу:

Дробная величина	Двоичное представление	Десятичное представление
$\frac{1}{4}$	0.01	0.25
$\frac{3}{8}$		
$\frac{23}{16}$		
	10.1101	
	1.011	
		5.625
		3.0625

УПРАЖНЕНИЕ 2.32

Неточность арифметических операций с плавающей точкой может иметь катастрофические последствия. 25 февраля 1991 года во время войны в Персидском заливе батарея American Patriot Missile в Джаране (Саудовская Аравия) не смогла перехватить запущенную иракскую ракету типа "Скад". Ракета взорвалась на территории военной базы США, погубив 28 человек. Генеральное отчетное ведомство США провело подробный анализ сбоя [52] и сделало вывод, что основной его причиной стала неточность числовых расчетов. В данном упражнении читателям предстоит воспроизвести часть анализа этого ведомства.

Система "Patriot" имеет в своем составе внутренний генератор синхронизирующих импульсов, выполненный в виде счетчика с приращением каждую 0.1 секунды. Для определения времени в секундах программе необходимо умножить показания этого

счетчика на 24-битовую величину, являющуюся дробным двоичным приближением $\frac{1}{10}$. В частности, двоичное представление $\frac{1}{10}$ является бесконечной последовательностью

$$0.000110011[0011]..._2$$

где часть, заключенная в скобки, повторяется бесконечно. Компьютер аппроксимировал 0.1, используя только ведущий бит и первые 23 бита данной последовательности справа от двоичной точки. Назовем это число x .

1. Каково двоичное представление $x = 0.1$?
2. Каково приближенное десятичное значение $x = 0.1$?
3. При подаче питания на систему генератор синхронизирующих сигналов начинает работать с нуля; с этой же отметки начинается отсчет. В данном случае система работала порядка 100 часов. Какова была разница между временем, рассчитанным компьютерной программой, и фактическим?
4. Система прогнозирует появление вражеской ракеты, принимая во внимание ее скорость и время последнего показания радара. Если считать, что "Склад" перемещается со скоростью порядка 2000 м/с, каково значение ошибки прогноза?

Обычно незначительная ошибка в абсолютном времени, передаваемом тактовым генератором, не влияет на расчет слежения. Оно, скорее, должно зависеть от относительного времени между двумя последовательными показаниями. Проблема заключалась в том, что программное обеспечение системы "Patriot" было обновлено с целью использования функции считывания показаний повышенной точности, однако в новом коде заменены были не все вызовы функций. В результате программа слежения использовала точное время для одного показания, а неточное — для другого [71].

2.4.2. Представление стандарта плавающей точки IEEE

Позиционное представление, подобное рассмотренному в предыдущем разделе, не сработает при представлении очень больших чисел. Например, представление 5×2^{100} будет состоять из комбинации битов 101, за которой будут стоять 100 нулей. Мы же предлагаем представлять такие числа в форме $x \times 2^y$ путем задания значений x и y .

Стандарт плавающей точки IEEE представляет число в форме $V = (-1)^s \times M \times 2^E$.

- Символ s определяет знак числа: отрицательный ($s = 1$) или положительный ($s = 0$), где интерпретация знакового разряда для числового значения 0 рассматривается как особый случай.
 - Мантисса M — дробное двоичное число в диапазоне либо от 1 до $2 - \epsilon$, либо от 0 до $1 - \epsilon$.
 - Показатель E взвешивает величину по (возможно, отрицательной) степени двойки.
- Для кодировки этих величин битовое представление числа с плавающей точкой делится на три поля:
- Единичный знаковый бит s напрямую кодирует символ s .
 - Поле чисел с плавающей точкой k битов $\text{exp} = e_{k-1} \dots e_1 e_0$ кодирует показатель E .

- Поле n -битовой дроби $f_{\text{frac}} = f_{n-1} \dots f_1 f_0$ кодирует мантиссу M , однако кодированная величина также зависит от того, равно ли поле чисел с плавающей точкой 0 или нет.

В формате плавающей точки с обычной точностью (float в C) поля s , exp и frac — 1, $k = 8$, $n = 23$ битов каждое, что дает 32-битовое представление. В формате плавающей точки с удвоенной точностью (double в C) поля s , exp и frac — 1, $k = 11$, $n = 52$ битов каждое, что дает 64-битовое представление.

Величину, кодированную конкретным битовым представлением, можно разделить на три разных варианта, зависящих от значения exp .

Нормализованные значения

Это самый общий случай. Подобные значения возникают, когда комбинация битов exp ни из одних нулей (числовое значение 0), ни из одних единиц (числовое значение 255 для одинарной точности, 2047 — для удвоенной). В этом случае поле чисел с плавающей точкой рассматривается как представляющее целое число со знаком в смещенной форме. Экспонента $E = e - \text{Bias}$ (смещение), где e — число без знака, имеющее битовое представление $e_{k-1} \dots e_1 e_0$, а Bias — смещенная величина, равная $2^{k-1} - 1$ (127 для одинарной точности, 1023 — для удвоенной). Это разложение дает диапазоны порядков от -126 до $+127$ для одинарной точности и от -1022 до $+1023$ — для удвоенной.

Поле дробных чисел frac рассматривается как представляющее дробную величину f , где $0 \leq f < 1$, имеющую двоичное представление $0.f_{n-1} \dots f_1 f_0$, т. е. с двоичной точкой слева от самого значимого бита. Мантисса определяется как $M = 1 + f$. Иногда это называется представлением неявной ведущей единицы, потому что M можно рассматривать как число с двоичным представлением $1.f_{n-1}, f_{n-2} \dots f_0$. Такое представление является способом свободного получения дополнительного бита точности, поскольку показатель E всегда можно настроить так, чтобы мантисса M находилась в диапазоне $1 \leq M < 2$ (при условии отсутствия переполнения). Поэтому представлять ведущий бит явно необходимости нет, поскольку он всегда равен единице.

Ненормализованные значения

Когда поле чисел с плавающей точкой состоит только из нулей, тогда представленное число находится в ненормализованной форме. В этом случае значение порядка $E = 1 - \text{Bias}$, а значение мантиссы $M = f$, т. е. значение поля дробных чисел без неявной ведущей единицы.

Зачем устанавливать смещение для ненормализованных значений

Наличие значения порядка $1 - \text{Bias}$ вместо Bias может показаться противоречащим интуитивности. Вкратце будет показано, что оно обеспечивает мягкий переход от ненормализованных значений к нормализованным.

Ненормализованные значения служат двум целям. Во-первых, они обеспечивают способ представления числового значения 0, поскольку при нормализованных значе-

ниях всегда необходимо иметь $M \geq 1$, и, следовательно, представить 0 нельзя. Фактически, представление чисел с плавающей точкой +0.00 имеет комбинацию битов, состоящую из одних нулей: знаковый разряд 0, поле чисел с плавающей точкой полностью состоит из нулей (что указывает на ненормализованное значение) и поле дробных чисел также состоит из одних нулей, обеспечивая $M = f = 0$. Интересно, что когда знаковый разряд равен единице, а все другие поля нулю, тогда получаем значение -0.0. В формате IEEE с плавающей точкой значения -0.0 и +0.0 в одних случаях рассматриваются как разные, а в других — как одинаковые.

Второй функцией ненормализованных чисел является представление чисел, очень близких к 0.0. Они обеспечивают свойство равномерного стремления к нулю, при котором возможные числовые значения равномерно распологаются около 0.0.

Особые значения

Последняя категория значений имеет место, когда поле чисел с плавающей точкой состоит из одних единиц. Когда поле дробных чисел состоит из одних нулей, результирующие величины представляют бесконечность: либо $+\infty$, когда $s = 0$, либо $-\infty$, когда $s = 1$. Бесконечность может представлять переполненные результаты, как при умножении очень больших чисел или делении на ноль. Когда поле дробных чисел не равно нулю, тогда результирующее значение называется *NaN* (сокращенно "Not a Number" — "Не число"). Такие значения возвращаются как результат операции, в которой результат не может быть представлен вещественным числом или как бесконечностью, подобно расчетам $\sqrt{-1}$ или $\infty - \infty$. Они также могут быть полезными в некоторых программных приложениях при представлении инициализации данных.

2.4.3. Примерные числа

На рис. 2.9 показан набор значений, которые можно представить в гипотетическом 6-битовом формате, имеющем $k = 3$ разрядов порядка и $n = 2$ битов мантиссы. Смещение — $2^{3-1} - 1 = 3$. В верхней части схемы показаны все представимые значения (отличающиеся от *NaN*). Плюс и минус бесконечность — по обоим концам схемы. Нормализованные числа с максимальным значением ± 14 . Ненормализованные числа сгруппированы вокруг 0. Они более четко видны во второй части схемы, где показаны только числа в диапазоне от -1.0 до +1.0. Два нуля являются особыми случаями ненормализованных чисел. Обратите внимание, что представляемые числа распределены неравномерно.

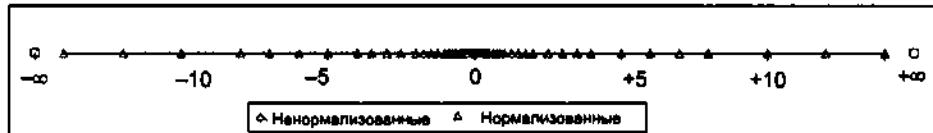
В табл. 2.17 приведено несколько примеров для гипотетического 8-битового формата плавающей точки, имеющего $k = 4$ разрядов порядка и $n = 3$ битов мантиссы. Смещение — $2^{4-1} - 1 = 7$. Таблица разделена на три области, представляющих три класса чисел. Ближе всего к нулю расположены ненормализованные числа, начинающиеся с самого нуля. Ненормализованные числа в этом формате имеют $E = 1 - 7 = -6$, обеспечивая вес $2^E = \frac{1}{64}$. Дроби f находятся в диапазоне 0, $\frac{1}{8}$, ..., $\frac{7}{8}$, выдавая числа V в диапазоне от 0 до $\frac{7}{8} \times 64 = 7/512$.

Таблица 2.17. Пример гипотетического 8-битового формата

Описание	Битовое представление	e	E	f	M	V
Ноль	0 0000 000	0	-6	0	0	0
Наименьшее положительное	0 0000 001	0	-6	$\frac{1}{8}$	$\frac{1}{8}$	$\frac{1}{512}$
	0 0000 010	0	-6	$\frac{2}{8}$	$\frac{2}{8}$	$\frac{2}{512}$
	0 0000 011	0	-6	$\frac{3}{8}$	$\frac{3}{8}$	$\frac{3}{512}$
	...					
	0 0000 110	0	-6	$\frac{6}{8}$	$\frac{6}{8}$	$\frac{6}{512}$
Наибольшее ненормализованное	0 0000 111	0	-6	$\frac{7}{8}$	$\frac{7}{8}$	$\frac{7}{512}$
Наименьшее нормализованное	0 0001 000	1	-6	0	$\frac{7}{8}$	$\frac{8}{512}$
	0 0001 001	1	-6	$\frac{1}{8}$	$\frac{9}{8}$	$\frac{9}{512}$
	...					
	0 0110 110	6	-1	$\frac{6}{8}$	$\frac{14}{8}$	$\frac{14}{16}$
	0 0110 111	6	-1	$\frac{7}{8}$	$\frac{15}{8}$	$\frac{15}{16}$
Единица	0 0111 000	7	0	0	$\frac{8}{8}$	1
	0 0111 001	7	0	$\frac{1}{8}$	$\frac{9}{8}$	$\frac{9}{8}$
	0 0111 010	7	0	$\frac{2}{8}$	$\frac{10}{8}$	$\frac{9}{8}$
	...					
	0 1110 110	14	7	$\frac{6}{8}$	$\frac{14}{8}$	224
Наибольшее нормализованное	0 1110 111	14	7	$\frac{7}{8}$	$\frac{15}{8}$	240
Бесконечность	0 1111 000	-	-	-	-	$+\infty$

В табл. 2.17 самые маленькие нормализованные числа в этом формате также имеют $E = 1 - 7 = -6$, а дроби находятся в диапазоне $0, \frac{1}{8}, \dots, \frac{7}{8}$. Однако мантиссы находятся в диапазоне от $1 + 0 = 1$ до $1 + \frac{7}{8} = \frac{15}{8}$, давая числа V в диапазоне от $\frac{8}{512}$ до $\frac{15}{512}$.

Полный диапазон



Значения между -1.0 и $+1.0$

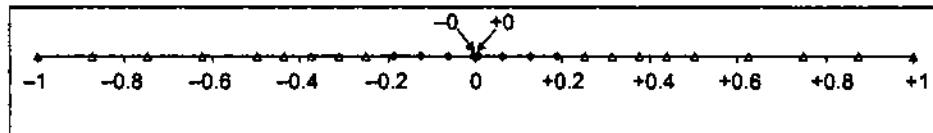


Рис. 2.9. Представимые значения для 6-битового формата плавающей точки

Обратите внимание на мягкий переход между самым большим ненормализованным числом $\frac{7}{512}$ и самым маленьким нормализованным числом $\frac{8}{512}$. Такая мягкость обеспечивается благодаря данному определению E для ненормализованных величин. Установлением смещения $1 - Bias$ вместо $-Bias$ компенсируется тот факт, что мантисса ненормализованного числа не имеет явно ведущей единицы.

По мере увеличения порядка разрядов, получаем гораздо большие нормализованные значения, проходящие через 1.0 к наибольшему нормализованному числу. Это число имеет порядок $E = 7$, обеспечивая вес $2^E = 128$. Дробь равна $\frac{7}{8}$, давая мантиссе $M = \frac{15}{8}$. Таким образом, числовое значение $V = 240$. Выход за пределы диапазона дает переполнение $+\infty$.

Одним из интересных свойств данного представления является то, что если рассматривать битовые представления значений, показанные в табл. 2.17, как целые числа без знака, то они появляются по восходящей, как и значения, представляемые числами с плавающей точкой. И это не случайно: формат IEEE проектировался так, что числа с плавающей точкой стало возможным сортировать, используя рутинную процедуру сортировки целых чисел. Небольшая сложность возникает при обработке отрицательных чисел, поскольку они имеют ведущую единицу и появляются в нисходящем порядке; впрочем, эта трудность преодолима без необходимости операций с числами с плавающей точкой для выполнения сравнений (см. упр. 2.56).

УПРАЖНЕНИЕ 2.33

Рассмотрим 5-битовое представление числа с плавающей точкой, основанное на формате IEEE, с одним знаковым разрядом, двумя разрядами порядка ($k = 2$) и двумя дробными битами ($n = 2$). Смещение порядка составляет $2^{k-1} - 1 = 1$.

В следующей таблице перечислен весь диапазон неотрицательных чисел для данного 5-битового представления числа с плавающей точкой. Заполните таблицу, используя следующие указания:

- e — значение экспоненты как целого числа без знака;
- E — значение порядка после смещения;
- f — значение дроби;
- M — значение мантиссы;
- V — представленное числовое значение.

Выразите значения f , M и V в виде дробей в форме $x/4$. Ячейки с прочерком заполнять не нужно.

Биты	e	E	f	M	V
0 00 00					
0 00 01					
0 00 10					
0 00 11					
0 01 00					
0 01 01					
0 01 10					
0 01 10					
0 01 11					
0 10 00	2	1	$\frac{0}{4}$	$\frac{4}{4}$	$\frac{8}{4}$
0 10 01					
0 10 10					
0 10 11					
0 11 00	—	—	—	—	$+\infty$
0 11 01	—	—	—	—	NaN
0 11 10	—	—	—	—	NaN
0 11 11	—	—	—	—	NaN

В табл. 2.18 показаны представления и числовые значения некоторых важных чисел с плавающей точкой одинарной и удвоенной точности. Как и в случае с 8-битовым форматом, показанным в табл. 2.17, здесь можно увидеть несколько общих свойств для представления чисел с плавающей точкой с k -битовым порядком и дробью длиной l битов.

- Значение +0.0 всегда имеет битовое представление, состоящее из одних нулей.
- Наименьшее положительное ненормализованное значение имеет битовое представление, состоящее из единицы в позиции наименее значимого бита (все остальные — нули). Значение дроби (и мантиссы) равно $M = f = 2^{-n}$, а значение порядка $E = -2^{k-1} + 2$. Следовательно, числовое значение $V = 2^{-n-2^{k-1}+2}$.
- Наибольшее ненормализованное значение имеет представление, состоящее из поля порядка (все нули) и поля дроби (все единицы). Оно имеет значение дроби (и мантиссы) $M = f = 1 - 2^{-n}$ (записано как $1 - \epsilon$), а значение порядка $E = -2^{k-1} + 2$. Следовательно, числовое значение $V = (1 - 2^{-n}) \times 2^{-2^{k-1}+2}$, что наименьшего ненормализованного значения.
- Наименьшее положительное нормализованное значение имеет битовое представление с единицей в наименее значимом бите в поле порядка (все остальные — нули). Значение мантиссы $M = 1$, а значение порядка $E = -2^{k-1} + 2$. Следовательно, числовое значение $V = 2^{-2^{k-1}+2}$.
- Значение 1.0 имеет битовое представление со всеми единицами, кроме наиболее значимого бита поля порядка (все остальные биты — 0). Значение мантиссы $M = 1$, а значение порядка $E = 0$.
- Наибольшее нормализованное значение имеет битовое представление со знаковым разрядом равным 0, наименее значимым битом порядка равным 0 (все остальные биты равны 1). Значение дроби $f = 1 - 2^{-n}$, что дает мантиссу $M = 2 - 2^{-n}$ (что в таблице записано как $(2 - \epsilon)$). Значение порядка $E = 2^{k-1} - 1$, что дает числовое значение $V = (2 - 2^{-n}) \times 2^{2^{k-1}-1} = (1 - 2^{-n-1}) \times 2^{2^{k-1}}$.

Таблица 2.18. Примеры неотрицательных чисел с плавающей точкой

Описание	exp	frac	Одинарная точность		Удвоенная точность	
			Значение	Десятичное	Значение	Десятичное
Ноль	00...00	00...00	0	0.0	0	0.0
Наименьшее ненормализованное	00...00	0...01	$2^{-23} \times 2^{-126}$	1.4×10^{-45}	$2^{-52} \times 2^{-1022}$	4.9×10^{-324}
Наибольшее ненормализованное	00...00	1...11	$(1 - \epsilon) \times 2^{-126}$	1.2×10^{-38}	$(1 - \epsilon) \times 2^{-1022}$	2.2×10^{-108}
Наименьшее нормализованное	00...01	0...00	1×2^{-126}	1.2×10^{-38}	1×2^{-1022}	2.2×10^{-108}
Единица	01...11	0...00	1×2^0	1.0	1×2^0	1.0
Наибольшее нормализованное	11...10	1...11	$(2 - \epsilon) \times 2^{127}$	3.4×10^{38}	$(2 - \epsilon) \times 2^{1023}$	1.8×10^{108}

Одним из полезных упражнений на понимание представлений чисел с плавающей точкой является преобразование выборочных целых значений в форму с плавающей точкой. Например, в табл. 2.11 было видно, что 12,345 имеет двоичное представление [11000000111001]. Создаем его нормализованное представление путем сдвига 13 позиций вправо от двоичной точки, что дает $12345 = 1.100000111001_2 \times 2^{13}$. Для кодировки в формат одинарной точности IEEE построим поле дроби удалением ведущей единицы и добавлением 10 нулей в конце, что дает двоичное представление [100000011100100000000000]. Для построения поля порядка добавляется смещение 127 к 13, в результате чего получится 140, имеющее двоичное представление [10001100]. После объединения со знаковым разрядом 0 для получения представления числа с плавающей точкой в двоичном счислении [01000110010000001110010000000000]. Вспомните из разд. 2.1.4, где рассматривалось соотношение представлений на битовом уровне целого значения 12345 (0x3039) и значения одинарной точности с плавающей точкой 12345.0 (0x4640E400):

0	0	0	0	3	0	3	9	
00000000000000000000011000000111001								

4	6	4	0	E	4	0	0	
010001100100000011100100000000000								

Теперь видно, что область соотношения соответствует младшим битам целого числа, прекращаясь как раз перед наиболее значимым битом, равным 1 (этот бит образует неявно ведущую единицу), совпадающим со старшими битами в дробной части представления числа с плавающей точкой.

УПРАЖНЕНИЕ 2.34

Как уже упоминалось в упр. 2.6, целое число 3490593 имеет шестнадцатеричное представление 0x354321, тогда как число с плавающей точкой одинарной точности 3490593.0 имеет шестнадцатеричное представление 0x4A550C84. Выведите данное представление числа с плавающей точкой и объясните соотношение между битами целочисленного представления и представления с плавающей точкой.

УПРАЖНЕНИЕ 2.35

1. Для формата плавающей точки с порядком k битов и дробью n битов составьте формулу для наименьшего положительного целого числа, которое нельзя представить точно (потому что для полной точности оно потребует дробь $n + 1$ битов).
2. Каково числовое значение этого целого в формате одинарной точности ($k = 8$, $n = 23$)?

2.4.4. Округление

Арифметические операции с плавающей точкой всего лишь аппроксимируют реальную арифметику, поскольку данное представление имеет ограниченные диапазон и точность. Следовательно, для значения x , как правило, необходим систематизирован-

ный метод нахождения "ближайшего" подходящего значения x' , которое можно было бы представить в нужном формате плавающей точки. Это имеет отношение к операции *округления*. Ключевой проблемой здесь является определение направления округления значения, стоящего как бы "на перепутье" между двумя вариантами выбора. Например, некто имеет в кармане 1.50 доллара и хочет округлить их до ближайшего целого доллара. Каков должен быть результат: 1 доллар или 2? Существует альтернативный подход: поддерживать верхнюю и нижнюю границу фактического числа. Например, можно определить представляемые значения x и x' так, что значение x будет гарантированно находиться где-то между ними: $x \leq x \leq x'$. В формате плавающей точки IEEE определены четыре разных режима округления. Метод, использованный по умолчанию, находит самое близкое соответствие; остальные три можно применять для расчета верхних и нижних границ.

В табл. 2.19 представлены четыре режима округления, применимых к задаче округления денежной суммы до ближайшего целого доллара. Округление до четного значения — режим по умолчанию. Здесь делается попытка найти ближайшее соответствие. Поэтому, \$1.40 округляется до \$1, а \$1.60 — до \$2, поскольку это — ближайшие значения целого доллара. Единственным проектировочным решением является определение эффекта округления значений, находящихся между возможными результатами. Режим округления до четного принимает правило, что значение округляется либо "в плюс", либо "в минус" так, что наименее значимая цифра результата является четной. Таким образом, и \$1.50, и \$2.50 округляются до \$2.

Таблица 2.19. Режимы округления

Режим	\$1.40	\$1.60	\$1.50	\$2.50	\$-1.50
Округление до четного значения	\$1	\$2	\$2	\$2	\$-2
Округление до нуля	\$1	\$1	\$1	\$2	\$-1
Округление "в минус"	\$1	\$1	\$1	\$2	\$-2
Округление "в плюс"	\$2	\$2	\$2	\$3	\$-1

Оставшиеся три режима обеспечивают гарантированные границы фактического значения. Их можно использовать в некоторых программных приложениях расчетов. Режим округления до нуля округляет положительные числа "в минус", а отрицательные — "в плюс", выдавая такое значение \hat{x} , что $|\hat{x}| \leq |x|$. Режим округления "в минус" округляет как положительные, так и отрицательные числа, выдавая такое значение x' , что $x \leq x'$. Режим округления "в плюс" округляет как положительные, так и отрицательные числа, выдавая такое значение x' , что $x \leq x'$.

Поначалу кажется, что режим округления до четного значения преследует довольно произвольную цель: какова причина стремления к четным числам? Почему не округлять "в плюс" значения, находящиеся между двумя представимыми значениями? Проблемой этого правила является то, что можно легко представить сценарии, в ко-

торых округление набора значений данных создаст статистическое смещение при расчетах среднего от всех значений. Среднее значение множества чисел, округленных таким способом, будет немного больше, чем среднее значение самих чисел. И наоборот, если бы числа всегда округлялись "в минус", тогда среднее значение множества округленных чисел было бы немного меньше, чем среднее значение самих чисел. При округлении до четного числа в большинстве реальных ситуаций такого статистического смещения можно избежать. Здесь 50% времени округление осуществляется "в плюс" и 50% времени — "в минус".

Округление до четного можно применять даже тогда, когда оно выполняется не до целого числа. Просто обращается внимание на то, какая наименее значимая цифра: четная или нечетная. Например, предположим, что нужно округлить десятичные числа до ближайшей сотой. 1.2349999 будет округлено до 1.23, а 1.2350001 — до 1.24, независимо от режима округления, поскольку эти числа не находятся в диапазоне от 1.23 до 1.24. С другой стороны, до 1.24 будет округлено как число 1.2350000, так и 1.2450000, потому что оба они — четные.

Подобным же образом, округление до четного значения применимо к двоичным дробным числам. Мы считаем, что нулевое значение наименее значимого бита четное, а 1 — нечетная. Вообще говоря, режим округления имеет значение при наличии комбинации битов в форме $XX\dots X.YY\dots Y100\dots$, где X и Y обозначают произвольные значения битов, и крайний правый Y — позиция, до которой необходимо выполнить округление. Комбинации битов только такой формы обозначают значения, которые находятся где-то между возможными результатами. В качестве примера можно рассмотреть проблему округления значений до ближайшей четверти (до 2 битов справа от двоичной точки). 10.00011_2 ($2\frac{3}{32}$) можно округлить до 10.00_2 (2), а 10.00110_2

($2\frac{3}{16}$) — до 10.01_2 ($2\frac{1}{4}$), потому что эти значения не находятся в диапазоне между двумя возможными значениями. 10.11100_2 ($2\frac{7}{8}$) можно округлить до 11.00_2 (3), а 10.10100_2 до 10.10_2 ($2\frac{1}{2}$), поскольку эти значения находятся в диапазоне между двумя возможными значениями, и предпочтительно иметь наименее значимый разряд (бит) равным нулю.

2.4.5. Операции с плавающей точкой

Стандарт IEEE устанавливает простое правило определения результатов таких арифметических операций, как сложение или умножение. Если рассматривать величины с плавающей точкой x и y как вещественные числа, и некую операцию Θ , выполненную над вещественными числами, тогда результатом расчетов должно стать *Round* ($x \Theta y$) — результат применения округления к точному результату реальной операции. На практике же существуют разнообразные "хитроумные штучки", которые используются проектировщиками элементов с плавающей точкой, во избежание выполнения точных расчетов, поскольку высокая степень точности необходима только для гарантии правильно округленного результата. Когда один из аргументов является

частным значением, таким, например, как -0 , ∞ или NaN , то стандарт определяет правила, претендующие на благородство. Например, $1/-0$ определяется для получения результата $-\infty$, тогда как $1/+0$ определяется для получения результата $+\infty$.

Одним из сильных аспектов метода задания поведения операций с плавающей точкой стандарта IEEE является то, что этот метод не зависит ни от реализации аппаратных, ни программных средств. Следовательно, его абстрактные математические свойства можно рассматривать, не заботясь о том, как этот метод реализован.

Ранее уже упоминалось о том, что целочисленное приращение, без знака или в дополнительном двоичном коде, образует абелеву группу. Приращение к вещественным числам также образует абелеву группу, но необходимо иметь в виду влияние, оказываемое округлением на эти свойства. Пусть $x +^f = Round(x + y)$. Эта операция определяется для всех значений x и y , хотя она может давать бесконечность из-за переполнения, несмотря даже на то, что x , и y — вещественные числа. Данная операция коммутативна с $x +^f y = y +^f x$ для всех значений x и y . С другой стороны, данная операция не ассоциативна. Например, выражение с плавающей точкой одинарной точности $(3.14 + 1e10) - 1e10$ выдаст 0.0 : значение 3.14 будет утеряно из-за округления. С другой стороны, выражение $3.14 + (1e10 - 1e10)$ выдаст 3.14 . Как и в абелевой группе, большинство значений имеет инверсию при приращении плавающей точки, т. е. $x +^f -x = 0$. Исключениями являются бесконечности (поскольку $+\infty - \infty = NaN$) и NaN , поскольку $NaN +^f x = NaN$ для любого x .

Недостаток ассоциативности при приращении плавающей точки — это недостаток наиболее важного группового свойства. Оно несет важные импликации как для научного программирования, так и разработчиков компиляторов. Например, предположим, что компилятору предложен следующий фрагмент кода:

```
x = a + b + c;
y = b + c + d;
```

Компилятор может "поддаться искушению" сэкономить на приращении плавающей точки через создание следующего кода:

```
t = b + c;
x = a + t;
y = t + d;
```

Впрочем, подобный расчет может выдать иной результат для x , нежели оригинал, поскольку в нем используется ассоциация, отличная от ассоциации операций сложения. В большинстве программных приложений разница будет предельно незначительной. К сожалению, компиляторы не могут знать, чем готов поступиться пользователь: общими показателями производительности или сохранением лояльности точному поведению программы-оригинала. Поэтому программисты, как правило, очень консервативны и избегают любой оптимизации, которая может оказать даже самое незначительное влияние на функциональность.

С другой стороны, приращение плавающей точки удовлетворяет следующему свойству монотонности: если $a \geq b$, тогда $x + a \geq x + b$ для любых значений a , b и x , отличных от NaN . Это свойство реального (и целочисленного) приращения не относит-

ся ни к приращению чисел без знака, ни к приращению чисел в дополнительном двоичном коде.

Умножение чисел с плавающей точкой также подчиняется многим свойствам, которые обычно ассоциируются с умножением, — со свойствами кольца. Пусть $x *^y u = Round(x \times y)$. При умножении эта операция закрыта (хотя, возможно, даст в результате бесконечность или NaN), она коммутативна и в качестве мультиплликативного тождества имеет 1.0. С другой стороны, она не ассоциативна из-за возможности переполнения или потери точности из-за округления. Например, для числа с плавающей точкой с одинарной точностью выражение $(1e20 * 1e20) * 1e-20$ результатом будет $+\infty$, тогда как результатом $1e20 * (1e20 * 1e-20)$ будет $1e20$. Кроме этого, умножение чисел с плавающей точкой не распространяется на приращение (сложение). Например, для числа с плавающей точкой с одинарной точностью выражение $1e20 * (1e20 - 1e-20)$ результатом будет 0.0, тогда как результатом $1e20 * 1e20 - 1e-20 * 1e20$ будет NaN .

С другой стороны, умножение чисел с плавающей точкой удовлетворяет следующим свойствам монотонности для любых значений a , b и c , отличных от NaN :

$$\square a \geq b \text{ и } c \geq 0 \Rightarrow a *^c c \geq b *^c c$$

$$\square a \geq b \text{ и } c \leq 0 \Rightarrow a *^c c \leq b *^c c$$

Помимо этого, также гарантируется, что $a *^c a \geq 0$, пока $a \neq NaN$. Как уже отмечалось, ни одно из этих свойств монотонности не выполняется для умножения чисел без знака или для умножения чисел в дополнительном двоичном коде.

Такой недостаток ассоциативности и дистрибутивности есть предмет серьезной озабоченности в среде научных программистов и разработчиков компиляторов, потому что решение даже такой, на первый взгляд, простейшей задачи, как написание кода для определения того, пересекаются ли две прямые в трехмерном пространстве, может стать серьезной проблемой.

2.4.6. Плавающая точка в C

В C предусмотрены два типа данных с плавающей точкой: `float` и `double`. На машинах, поддерживающих стандарт плавающей точки IEEE, эти типы данных соответствуют плавающей точке с одинарной и удвоенной точностью. Кроме того, в этих машинах используется режим округления до четного значения. К сожалению, поскольку стандарт C не требует машинного использования стандарта IEEE, то не существует ни стандартных методов смены режима округления, ни способов получения частных значений, таких как -0 , $+\infty$, $-\infty$ или NaN . В большинстве систем для обеспечения доступа к этим функциям предусмотрена комбинация включаемых файлов (`' .h '`) и библиотек процедур, однако в каждой системе существуют "свои тонкости". Например, компилятор GNU GCC определяет макрос `INFINITY` (для $+\infty$) и `NAN` (для NaN), когда в программном файле возникает следующая последовательность:

```
#define _GNU_SOURCE 1
#include <math.h>
```

УПРАЖНЕНИЕ 2.36

Заполните определения макросов для создания следующих значений удвоенной точности: $+\infty$, $-\infty$ и 0.

```
#define POS_INFINITY
#define NEG_INFINITY
#define NEG_ZERO
#endif
```

Включаемые файлы (типа `math.h`) использовать нельзя, однако можно извлечь выгоду из того факта, что наибольшее конечное число, которое можно представить с удвоенной точностью, составляет величину порядка 1.8×10^{308} .

При преобразовании значений между форматами `int`, `float` и `double` программа изменяет числовые значения и битовые представления следующим образом (при условии 32-битового `int`):

- Из `int` в `float` число не может переполниться, но может быть округлено.
- Из `int` или `float` в `double` может быть сохранено точное числовое значение, потому что `double` имеет как больший диапазон (т. е. диапазон представимых значений), так и большую точность (т. е. число значимых разрядов).
- Из `double` в `float` значение может быть переполнено до $+\infty$ или $-\infty$, поскольку диапазон меньше. В противном случае значение может быть округлено из-за меньшей степени точности.
- Из `float` или `double` в `int` значение будет усечено до нуля. Например, 1.999 будет преобразовано в 1, а -1.999 будет преобразовано в -1. Обратите внимание на то, что такое поведение сильно отличается от округления. Более того, данное значение может вызвать переполнение. Для этого случая стандарт C не устанавливает фиксированного результата, однако на большинстве машин результат будет либо `TMaxw` или `TMinw`, где w — количество бит в `int`.

Арифметические операции с плавающей точкой Intel IA32

В следующей главе приступим к углубленному изучению процессоров Intel IA32, которые используются в большинстве современных компьютеров. Здесь выделяются отличительные особенности подобных машин, которые могут серьезно повлиять на поведение программ, работающих с числами с плавающей точкой при компилировании с помощью GCC.

Подобно большинству процессоров, процессоры IA32 имеют особые элементы памяти, называемые *регистрами*, для хранения значений с плавающей точкой по мере их расчета и использования. Хранящиеся в регистрах значения можно считывать и записывать быстрее, чем хранящиеся в основной памяти. Необычной особенностью IA32 является то, что регистры плавающей точки используют особый 80-битовый формат повышенной точности для обеспечения большего диапазона и точности, чем обычные 32-битовый и 64-битовый форматы одинарной и удвоенной точности, соответственно, используемые для хранящихся в памяти значений. Как описано в упр. 2.58,

представление с повышенной точностью похоже на формат плавающей точки IEEE с 15-битовым порядком (т. е. $k = 15$) и 63-битовой дробью (т. е. $n = 63$). Все числа одинарной и удвоенной точности преобразуются в этот формат по мере того, как они загружаются из памяти в регистры чисел с плавающей точкой. Арифметические операции всегда выполняются в режиме повышенной точности. Числа преобразуются из формата повышенной точности в форматы одинарной или удвоенной точности по мере их сохранения в памяти.

Такое расширение до 80 битов для всех данных регистров и последующее сжатие в меньший формат всех данных, хранящихся в памяти, вызывает нежелательные для программистов последствия. Это означает, что сохранение значения в памяти и последующий его вызов могут изменить само значение, из-за округления, исчезновения или переполнения. Такое сохранение и извлечение программист не всегда может видеть, и результаты могут получиться непредсказуемыми.

Приведенный далее пример иллюстрирует это свойство.

Листинг 2.10. Режим повышенной точности

```

1 double recip (int denom)
2 {
3     return 1.0 (double) denom;
4 }
5
6 void do_nothing () { /* Только по имени */
7
8 void test1 (int denom)
9 {
10    double r1, r2;
11    int t1, t2;
12
13    r1 = recip (denom); /* Сохраняется в памяти */
14    r2 = recip (denom); /* Сохраняется в регистре */
15    t1 = r1 == r2; /* Сравнивает регистр с памятью */
16    do_nothing (); /* Вынуждает регистр к сохранению в памяти */
17    t2 = r1 == r2; /* Сравнивает память с памятью */
18    printf ("test1 t1: r1 %f %c= r2 %f\n", r1, t1 ? '=' : '!', r2);
19    printf ("test1 t2: r1 %f %c= r2 %f\n", r1, t2 ? '=' : '!', r2);
20 }
```

Переменные $r1$ и $r2$ рассчитываются по той же функции с тем же аргументом. Можно ожидать, что они будут одинаковыми. Более того, переменные $r1$ и $r2$ рассчитываются по оценке выражения $r1 == r2$, и поэтому можно ожидать, что они будут равны 1. Очевидных скрытых побочных эффектов нет: функция `recip` выполняет сквозной обратный расчет и, судя по названию `do_nothing`, функция ничего не выполняет. Впрочем, когда файл компилируется с флагом оптимизации `-O2` и запускается с аргументом 10, получается следующий результат:

```
test1 t1: r1 0.100000 != r2 0.100000
test1 t2: r1 0.100000 != r2 0.100000
```

Первый тест указывает на то, что две обратные величины различны, тогда как второй тест указывает на то, что они одинаковы! Очевидно, что это — не то, что ожидалось. Понимание всех мелочей данного примера требует изучения кода с плавающей точкой на машинном уровне, генерированного GCC (см. разд. 3.14), однако комментарии в коде подсказывают, почему на выходе получено именно то, что получено. Значение, рассчитанное функцией `recip`, возвращает результат в регистр плавающей точки. Как только процедура `test1` вызывает какую-либо функцию, она должна сохранять любое значение в текущем режиме в регистре плавающей точки основного программного стека, в котором хранятся локальные переменные функции. При выполнении такого сохранения процессор преобразует значения регистра повышенной точности в значения памяти удвоенной точности. Следовательно, до второго вызова `recip` (строка 14) переменная `r1` преобразуется и сохраняется как число удвоенной точности. После второго вызова переменная `r2` имеет значение повышенной точности, возвращенное функцией. При расчете `t1` (строка 15) число удвоенной точности `r1` сравнивается с числом повышенной точности `r2`. Поскольку 0.1 нельзя представить точно ни в каком формате, вывод теста ложен. Перед вызовом функции `do_nothing` (строка 16) `r2` преобразуется и сохраняется в виде числа удвоенной точности. При расчете `t2` (строка 17) сравниваются два числа удвоенной точности, выдавая истинный результат.

Данный пример демонстрирует недостаток GCC при работе на машинах с процессором Intel IA32 (тот же результат имеет место для Linux и Microsoft Windows). Значение ассоциируется с изменениями переменной из-за операций, не видимых для программиста, например, сохранения и восстановления регистров плавающей точки. Эксперименты с компилятором Microsoft Visual C++ указывают на то, что он не сталкивается с подобными проблемами.

Почему нужно обращать внимание на противоречивость?

В главе 5 рассматривается один из фундаментальных принципов оптимизации компиляторов: программы должны выдавать абсолютно одинаковые результаты, независимо от того, активизирована оптимизация или нет. К сожалению, GCC не удовлетворяет этому требованию для кода с плавающей точкой на машинах с процессором Intel IA32.

Существует несколько способов решения данной проблемы, хотя ни один из них не является идеальным. Самое простое: вызвать GCC опцией командной строки `ffloat-store`, указывающей на то, что результат каждого расчета с плавающей точкой должен сохраняться в памяти и считываться перед использованием, а не просто храниться в регистре. В этом случае каждое рассчитанное значение будет преобразовываться в форму с пониженней точностью. В определенной степени это замедляет работу программы, однако ее поведение становится более предсказуемым. К сожалению, обнаружено, что GCC не строго следует правилу чтения после записи даже при наличии опции командной строки. Например, рассмотрим следующую функцию в листинге 2.11.

Листинг 2.11. Функция

```

1 void test2 (int denom)
2 {
3     double r1;
4     int t1;
5     r1 = recip (denom); /* По умолчанию: регистр, принудительное сохранение:
                           память */
6     t1 = r1 == 1.0 / (double) denom; /* Сравнивает регистр или память с регистром */
7     printf ("test2 t1: r1 %f %c= 1.0/10.0\n", r1, t1 ? '=' : '!' );
8 }
```

При компиляции только с опцией `-O2` `t1` приобретает значение 1: сравниваются два значения регистров. При компиляции с флагом `-ffloat-store` `t1` получает значение 0! Несмотря на то, что результат вызова `recip` записывается в память и считывается в регистр, рассчитанное значение `1.0 / (double) denom` сохраняется в регистре. Вообще обнаружено, что на вид незначительные изменения в программе могут вызвать непредсказуемый успех или провал данных тестов.

В качестве альтернативы можно сделать так, что `gcc` будет использовать повышенную точность во всех расчетах, путем объявления всех переменных как `long double` (листинг 2.12).

Листинг 2.12. Сравнение

```

1 long double recip_1 (int denom)
2 {
3     return 1.0 / (long double) denom;
4 }
5
6 void test3 (int denom)
7 {
8     long double r1, r2;
9     int t1, t2, t3;
10
11    r1 = recip_1 (denom); /* Сохраняется в памяти */
12    r2 = recip_1 (denom); /* Сохраняется в регистре */
13    t1 = r1 == r2; /* Сравнивает регистр с памятью */
14    do_nothing (); /* Вынуждает регистр к сохранению в памяти */
15    t2 = r1 == r2; /* Сравнивает память с памятью */
16    t3 = r1 == 1.0 / (long double) denom; /* Сравнивает память с регистром */
17    printf ("test3 t1: r1 %f %c= r2 %f\n",
18           (double) r1, t1 ? '=' : '!', (double) r2);
19    printf ("test3 t2: r1 %f %c= r2 %f\n",
20           (double) r1, t2 ? '=' : '!', (double) r2);
```

```
21 printf ("test3 t3: r1 %f %c= 1.0/10.0\n",
22 (double) r1, t2 ? '=' : '!');
23 }
```

Объявление `long double` допустимо как часть стандарта ANSI C, хотя для большинства машин и компиляторов это объявление эквивалентно обычному `double`. Впрочем, для GCC на машинах IA32 для информации памяти и для данных регистра с плавающей точкой используется формат повышенной точности. Это позволяет в полной мере воспользоваться преимуществами более широкого диапазона и точности, предоставляемых этим форматом, а также избежать отклонений, имевших место в более ранних примерах. К сожалению, это решение довольно дорогостоящее. Для хранения длинного двойного GCC использует 12 байтов, на 50% увеличивая объем занимаемой памяти (10 байт было бы достаточно, однако осуществляется округление до 12 для повышения эффективности работы памяти. То же распределение используется на машинах с Linux и Windows). Перенос более длинных данных между регистрами и памятью занимает больше времени, но это — по-прежнему наилучший выбор для программ, от которых требуются наиболее точные и предсказуемые результаты.

Ariane 5: цена переполнения чисел с плавающей точкой

Преобразование больших чисел с плавающей точкой в целые является основным источником ошибок программирования. Подобные ошибки имели катастрофические последствия для первого запуска ракеты Ariane 5 4 июня 1996 года. Через 37 секунд после старта ракеты отклонилась от курса, переломилась и взорвалась. Стоимость находившихся на ее борту спутников связи составляла порядка 500 млн долларов.

В ходе проведенного расследования [49] выяснилось, что компьютер, управляющий инерционной системой навигации, отправил неверные данные на компьютер, управляющий соплами. Вместо отправки контрольной информации о ходе полета отправлена была диагностическая комбинация битов, указывающая на возникновение переполнения во время преобразования 64-битового числа с плавающей точкой в 16-битовое число со знаком.

Переполненное значение измеряло горизонтальную скорость ракеты, которая могла в пять раз превышать скорость, развитую предыдущей ракетой Ariane 4. При разработке для нее программного обеспечения числовые значения были тщательно проанализированы, и сделан вывод о том, что горизонтальная скорость никогда не переполнит 16-битовое число. К сожалению, в Ariane 5 эта часть программных средств была просто использована повторно, без проверки допущений, на которых она базировалась.

УПРАЖНЕНИЕ 2.37

Предположим, что переменные `x`, `f` и `d` обладают типом `int`, `float` и `double` соответственно. Их значения произвольны, за исключением того, что ни `f`, ни `d` не равны ни $+\infty$, ни $-\infty$, ни NaN . Докажите для каждого из следующих выражений C, что они всегда будут истинны (т. е. равны 1), либо задайте переменным такое значение, что выражение не будет истинным (т. е. равно 0).

1. $x = \text{(int)}(\text{float})x$
2. $x = \text{(int)}(\text{double})x$
3. $f = \text{(float)}(\text{double})f$
4. $d = \text{(float)}d$
5. $f = -(-f)$
6. $2/3 = 2/3.0$
7. $(d >= 0.0) \mid\mid ((d^2) < 0.0)$
8. $(d + f) - d = f$

2.5. Резюме

Компьютеры кодируют информацию в виде битов, составленных в байты. Для представления целых и вещественных чисел и символьных строк используются различные виды кодировок. В разных моделях компьютеров применяются разные правила для кодировки чисел и упорядочивания байтов данных.

Язык С разработан для согласования широкого диапазона различных реализаций в том, что касается длин слов и числовых кодировок. В большинстве современных компьютеров используется длина слова 32 бита; в более мощных машинах все более распространенной становится длина слов 64 бита. В большинстве машин используется кодировка целых чисел в дополнительном двоичном коде и кодировка IEEE чисел с плавающей точкой. Понимание этих кодировок на уровне одного бита, а также понимание математических характеристик арифметических операций важно для написания программ, работающих корректно во всем диапазоне числовых значений.

Стандарт С предписывает то, что при сопоставлении (приведении) целых чисел без знака и со знаком изначальная комбинация битов меняться не должна. На компьютере с дополнительным двоичным кодом такое поведение характеризуется функциями $T2U_w$ и $U2T_w$ для значения w битов. Неявное преобразование в С дает результаты, которых многие программисты просто не ожидают и которые приводят к многочисленным программным ошибкам.

Из-за конечной длины кодировок компьютерная арифметика обладает свойствами, предельно отличающимися от традиционной целочисленной и вещественной арифметики. Конечная длина может вызвать переполнение чисел, когда они выходят за рамки диапазона представления. Значения с плавающей точкой могут терять значимость, когда они настолько близки к 0.0, что превращаются в ноль.

Конечная целочисленная арифметика, реализованная в С, так же, как и в большинстве других языков программирования, обладает некоторыми особыми свойствами, если сравнивать ее с арифметикой точных интегралов. Например, результатом выражения $x*x$ может быть отрицательное число из-за переполнения. Несмотря на это, как арифметика чисел без знака, так и арифметика чисел в дополнительном двоичном коде удовлетворяют свойствам кольца. Это позволяет компиляторам выполнять

множество оптимизаций. Например, при замене выражения $7*x$ на $(x \ll 3) - x$ используются ассоциативные, коммуникативные и дистрибутивные свойства, наряду с соотношением между сдвигами и умножением степеней двойки.

В текущей главе рассматривалось несколько интересных способов использования комбинаций операций на уровне бита и арифметических операций. Например, читатели убедились в том, что с арифметикой в дополнительном коде $-x + 1$ равно $-x$. В качестве другого примера предположим, что нужна комбинация битов в форме $[0, \dots, 0, 1, \dots, 1]$, состоящая из $w - k$ нулей, за которыми следует k единиц. Такие комбинации битов необходимы для операций маскирования. Эту комбинацию можно создать выражением C $(1 \ll k) - 1$, использующим то свойство, что нужная комбинация битов имеет числовое значение $2^k - 1$. Например, выражение $(1 \ll 8) - 1$ генерирует комбинацию битов 0xFF.

Представления чисел с плавающей точкой аппроксимируют вещественные числа путем кодировки чисел, записанных в форме $x \times 2^y$. Наиболее распространенное представление чисел с плавающей точкой определено стандартом 754 IEEE. Он допускает несколько разных степеней точности, наиболее широко используемые из которых, — одинарная (32 битов) и удвоенная (64 битов). Плавающая точка от IEEE также имеет представления для частных значений ∞ и для NaN (не число).

Арифметические операции с плавающей точкой следует использовать с осторожностью, потому что их диапазон и точность весьма ограничены, а также потому, что они не подчиняются общим математическим свойствам, например, ассоциативности.

Библиографические примечания

В справочных руководствах по С [40, 32] рассматриваются свойства различных типов данных и операций. В стандарте С не определены такие подробности, как точные длины слов или числовые кодировки. Они опущены намеренно для того, чтобы С можно было реализовать на как можно большем количестве моделей компьютеров. Существует несколько книг, специально написанных для программистов С [41, 50], в которых содержатся предупреждения о проблемах, связанных с переполнением, неявным преобразованием в число без знака и некоторыми другими тонкостями, описанными в данной главе. В этих книгах также представлены полезные советы по присваиванию имен, стилям кодировок и тестированию кодов. В книгах, посвященных языку кодировки Java (авторы рекомендуют собственную работу, написанную в соавторстве с Джеймсом Гозлингом — создателем данного языка [1]), описываются форматы данных и арифметические операции, поддерживаемые Java.

В большинстве книг по логическому проектированию [86, 39] имеется раздел, в котором рассматриваются кодировки и арифметические операции. В этих книгах описываются различные способы реализации арифметических схем. Книга Овертона, посвященная плавающей точке IEEE [56], приводит подробное описание этого формата, а также его свойств с точки зрения программиста, занимающегося разработкой численных приложений.

Задачи для домашнего решения

Номера домашних упражнений снабжены значками, соответствующими сложности задач:

- ♦ — задача быстрого решения для проверки действенности того или иного положения;
- ♦♦ — 5–15 минут для решения, которое может включать в себя написание или выполнение программ;
- ♦♦♦ — устойчивые проблемы, на решение которых может потребоваться несколько часов;
- ♦♦♦♦ — лабораторная работа, на выполнение которой может потребоваться 1–2 недели.

УПРАЖНЕНИЕ 2.38 ♦

Скомпилируйте и выполните выборочный код, использующий `show_bytes` (файл `show_bytes.c`) на различных машинах, к которым имеется доступ. Определите упорядочивание байтов на этих компьютерах.

УПРАЖНЕНИЕ 2.39 ♦

Попытайтесь выполнить код `show_bytes` для различных выборочных значений.

УПРАЖНЕНИЕ 2.40 ♦

Напишите процедуры `show_short`, `show_long` и `show_double`, распечатывающие байтовые представления объектов С типов `short int`, `long int` и `double`, соответственно. Попытайтесь сделать это на разных компьютерах.

УПРАЖНЕНИЕ 2.41 ♦♦

Напишите процедуру `is_little_endian`, которая при компиляции и запуске на "остроконечном" компьютере будет возвращать 1, а при компиляции и запуске на "тупоконечном" компьютере будет возвращать 0. Эта программа должна работать на компьютерах любого типа, независимо от используемых на них длин слова.

УПРАЖНЕНИЕ 2.42 ♦♦

Напишите выражение С, которое будет выдавать слово, состоящее из наименее значимого байта `x` и остальных байтов — `y`. Для operandов `x = 0x89ABCDEF` и `y = 0x76543210` результат составит `0x765432EE`.

УПРАЖНЕНИЕ 2.43 ♦♦

Используя только операции на уровне бита и логические, напишите выражения С, дающие 1 для описанных условий, и 0 — в противном случае. Данный код должен работать на машине с любой длиной слова. Представьте, что `x` — целое число.

1. Любой бит x равен 1.
2. Любой бит x равен 0.
3. Любой бит в наименее значимом байте x равен 1.
4. Любой бит в наименее значимом байте x равен 0.

УПРАЖНЕНИЕ 2.44 ***

Напишите функцию `int_shifts_are_arithmetic ()`, которая бы выдавала 1 при выполнении на машине, использующей арифметические сдвиги вправо для всех `int`, и 0 — в противном случае. Данный код должен работать на машине с любой длиной слова. Протестируйте написанный код на нескольких машинах. Напишите и протестируйте процедуру `unsigned_shifts_are_arithmetic ()`, определяющую форму сдвигов, используемую для `unsigned int`.

УПРАЖНЕНИЕ 2.45 **

Дано задание написать процедуру `int_size_is_32 ()`, выдающую 1 при выполнении на машине, для которой `int` — 32 бита, и 0 — если нет. В листинге приведена первая попытка:

```

1 /* На некоторых машинах следующий код работает некорректно */
2 int bad_int_size_is_32 ()
3 {
4 /* Задание наиболее значимого бита (msb) 32-битовой машины */
5 int set_msbit = 1 << 31;
6 /* Сдвиг за msb 32-битового слова */
7 int beyond_msbit = 1 << 32;
8
9 /* set_msbit не равно нулю, когда длина слова >= 32
10 beyond_msbit равно нулю, когда длина слова <= 32 */
11 return set_msbit && !beyond_msbit;
12 }
```

Впрочем, при компилировании и выполнении на 32-битовом SUN SPARK данная процедура возвращает 0. Следующее сообщение компилятора указывает на проблему:

`warning: left shift count >= width of type`

1. В чем написанный код не соответствует стандарту C?
2. Модифицируйте код так, чтобы он корректно выполнялся на любом компьютере, для которого `int` — как минимум, 32 бита.
3. Модифицируйте код так, чтобы он корректно выполнялся на любом компьютере, для которого `int` — как минимум, 16 бит.

УПРАЖНЕНИЕ 2.46 ♦

Вы устроились на работу в компанию, занимающуюся имплементацией набора процедур для работы со структурой данных, где четыре байта со знаком упакованы в 32-битовый `unsigned`. Байты в рамках слова пронумерованы от 0 (наименее значимый) до 3 (наиболее значимый). Дано задание реализовать функцию для машины с арифметикой в дополнительном двоичном коде и с арифметическими сдвигами вправо по следующим прототипам:

```
/* Объявление типа данных, где 4 байта упакованы в unsigned (без знака) */
typedef unsigned packed_t;
```

```
/* Извлечение байта из слова. Возврат в виде целого числа со знаком */
int xbyte (packed_t word, int bytenum)
{
return
(word >> (bytenum << 3)) & 0xFF;
}
```

1. Что неправильно в данном коде?

2. Составьте корректную реализацию функции, в которой используются только сдвиги вправо и влево с одним вычитанием.

УПРАЖНЕНИЕ 2.47 ♦

Заполните следующую таблицу, показывая влияние дополнения и инкремента нескольких векторов длиной 5 битов (по типу табл. 2.14). Покажите как битовые векторы, так и числовые значения.

\bar{x}	$\sim \bar{x}$	$\text{incr}(\sim \bar{x})$
[01101]		
[01111]		
[11000]		
[11111]		
[10000]		

УПРАЖНЕНИЕ 2.48 ++

Докажите, что первый декремент и последующее дополнение эквивалентны дополнению с последующим инкрементом. То есть, для любого значения со знаком x выражения $-x$, $\sim x + 1$ и $\sim(x - 1)$ дают одинаковые результаты. На каких математических свойствах сложения в дополнительном коде основан вывод?

УПРАЖНЕНИЕ 2.49 ++

Предположим, что необходимо рассчитать 2_w-битовое представление $x \cdot y$, где x и y — значения без знака, на машине, для которой тип данных `unsigned` w -битовый. Биты младшего разряда произведения можно рассчитать выражением $x * y$, поэтому потребуется только процедура с прототипом

```
unsigned int unsigned_high_prod (unsigned x, unsigned y);
```

рассчитывающим старшие w битов $x \cdot y$ для переменных без знака. Имеется доступ к библиотечной функции со следующим прототипом:

```
int signed_high_prod (int x, int y);
```

рассчитывающим старшие w битов $x \cdot y$ для случая, когда x и y — в форме дополнительного двоичного кода. Напишите код,зывающий эту процедуру, для реализации функции для аргументов без знака. Обоснуйте правильность своего решения.

Подсказка: Обратите внимание на соотношение между произведением $x \cdot y$ со знаком и произведением $x' \cdot y'$ без знака в выводе уравнения (2.16).

УПРАЖНЕНИЕ 2.50 ++

Предположим, что получено задание создать код для умножения целочисленной переменной x на различные постоянные множители k . Для большей эффективности нужно использовать только операции +, - и <<. Напишите выражения умножения для следующих значений k , используя не больше трех операций для одного выражения.

1. $k = 5$;
2. $k = 9$;
3. $k = 14$;
4. $k = -56$.

УПРАЖНЕНИЕ 2.51 ++

Напишите выражения С для создания следующей комбинации битов, где a^k представляет k повторений символа a . Предположим w -битовый тип данных. Код может содержать ссылки на параметры j и k , представляя значения j и k , но не параметр, представляющий w .

1. $1^{w-k} 0^k$;
2. $0^{w-k-j} 1^k 0^j$.

УПРАЖНЕНИЕ 2.52 ++

Предположим, что байты в w -битовом слове пронумерованы от 0 (наименее значимый) до $w/8-1$ (наиболее значимый). Напишите код функции, которая возвращает значение без знака, так что байт i аргумента x заменен байтом b :

```
unsigned replace_byte (unsigned x, int i, unsigned char b);
```

Следующие примеры показывают, как должна работать функция:

```
replace_byte(0x12345678, 2, 0xAB) --> 0x12AB5678
replace_byte(0x12345678, 0, 0xAB) --> 0x123456AB
```

УПРАЖНЕНИЕ 2.53 ◆◆◆

Создайте код для следующих функций С. Функция `srl` выполняет логический сдвиг вправо, используя арифметический сдвиг вправо (представленный значением `xsgt`), за ним следуют остальные операции, за исключением сдвигов вправо или деления. Функция `sra` выполняет арифметический сдвиг вправо, используя арифметический сдвиг вправо (представленный значением `xsrl`), далее следуют остальные операции, за исключением сдвигов вправо или деления. Можно предположить, что длина всех `int` 32 бита. Сумма сдвигов `k` может варьироваться от 0 до 31.

```
unsigned srl (unsigned x, int k)
{
/* Выполнение сдвига арифметически */
unsigned xsra = (int) x >> k;

/* ...*/
}

int sra (int x, int k)
{
/* Выполнение сдвига логически */

/* ...*/
}
```

УПРАЖНЕНИЕ 2.54 ◆

Программы выполняются на машинах, где значения типа `int` равны 32 битам. Они представлены в дополнительном двоичном коде и сдвинуты вправо арифметически. Значения типа `unsigned` также имеют 32 бита.

Сгенерируем произвольные значения `x` и `y` и преобразуем их в другие значения без знака.

```
/* Создать некоторые произвольные значения */
int x = random ();
int y = random ();
/* Конвертирование в значения без знака */
unsigned ux = (unsigned) x;
unsigned uy = (unsigned) y;
```

Для каждого из следующих выражений С необходимо указать, всегда ли данное выражение дает 1. Если да, то опишите математические принципы, лежащие в основе этого. Если нет, приведите пример аргументов, при которых оно выдает 0.

1. $(x < y) == (-x > -y)$
2. $((x+y) \ll 4) + y - x == 17 * y + 15 * x$
3. $\sim x + \sim y == \sim (x+y)$
4. $(\text{int}) (\text{ux} - \text{uy}) == -(y - x)$
5. $((x >> 1) \ll 1) \leq x$

УПРАЖНЕНИЕ 2.55 ++

Рассмотрим числа, имеющие двоичное представление, состоящее из бесконечной строки в виде $0.yuuuu\dots$, где y — k -битовая последовательность. Например, двоичное представление $\frac{1}{3}$ равно $0.01010101\dots$ ($y = 01$), тогда как представление $\frac{1}{5}$ равно $0.001100110011\dots$ ($y = 0011$).

1. Пусть $Y = B2U_k(y)$, т. е. число, имеющее двоичное представление y . Выведите формулу в том, что касается Y и k для значения, представленного бесконечной строкой. *Подсказка:* Рассмотрите влияние сдвига позиций двоичной точки k вправо.
2. Каково числовое значение строки для следующих значений y ?
 - 001
 - 1001
 - 000111

УПРАЖНЕНИЕ 2.56 +

Ведите возвращаемое значение в следующую процедуру (проверяющую, превышает ее первый аргумент второй или равен ему). Предположим, что функция $f2u$ возвращает 32-битовое число без знака, имеющее то же битовое представление, что и его аргумент с плавающей точкой. Можно также предположить, что ни один из аргументов не является Nan . Две разновидности нуля $+0$ и -0 считаются равными.

```
int float_ge (float x, float y)
{
unsigned ux = f2u (x);
unsigned uy = f2u (y);

/* Получение знаковых разрядов */
unsigned sx = ux >> 31;
unsigned sy = uy >> 31;

/* Представить выражения, используя только ux, uy, sx и sy */
return /* ... */;
```

УПРАЖНЕНИЕ 2.57 ♦

Имея формат плавающей точки с k -битовым порядком и n -битовую дробь, напишите формулы для экспоненты E , мантиссы M , дроби f и значения V для следующих чисел. Помимо этого, опишите битовое представление.

- Число 5.0.
- Самое большое нечетное целое число, которой может быть точно представлено.
- Обратную величину наименьшего положительного нормализованного значения.

УПРАЖНЕНИЕ 2.58 ♦

Процессоры, совместимые с Intel, также поддерживают формат плавающей точки "повышенной точности", где 80-битовое слово делится на знаковый бит, $k = 15$ порядковых битов, один целый бит и $n = 63$ дробных битов. Целый бит является явной копией подразумеваемого бита в представлении плавающей точки IEEE. То есть, он равен 1 для нормализованных значений и 0 — для ненормализованных. Заполните таблицу вводом приблизительных значений некоторых "интересных" чисел в данном формате:

Описание	Повышенная точность	
	Значение	Десятичное значение
Наименьшее ненормализованное		
Наименьшее нормализованное	-	
Наибольшее нормализованное		

УПРАЖНЕНИЕ 2.59 ♦

Рассмотрим 16-битовое представление с плавающей точкой, основанное на формате плавающей точки IEEE, с одним знаковым битом, семью порядковыми битами ($k = 7$) и восемью дробными битами ($n = 8$). Отклонение порядка равно $2^{7-1} - 1 = 63$.

Заполните следующую таблицу для каждого из представленных чисел со следующими инструкциями для каждого столбца:

- Четыре шестнадцатеричные цифры, описывающие кодированную форму.
- Значение мантиссы. Это должно быть число в форме x или $\frac{x}{y}$, где x — целое число, а y — целая степень двойки. В число примеров входят: 0, $\frac{67}{64}$ и $\frac{1}{256}$.
- Целое значение порядка.
- Представленные числовые значения. Используйте запись x или $x \times 2^z$, где x и z — целые числа.

В качестве примера, для представления числа $\frac{7}{2}$ имеем $s = 0$, $M = \frac{7}{4}$ и $E = 1$. Следовательно, одно число имеет поле чисел с плавающей запятой, равное 0x40 (десятичное значение $63 + 1 = 64$), а поле мантиссы — 0x80 (двоичное значение 11000000_2), что дает шестнадцатеричное представление — 40C0.

Поля, отмеченные прочерком, заполнять не нужно.

Описание	Hex	<i>M</i>	<i>E</i>	<i>V</i>
-0				—
Наименьшее значение > 1				
256				—
Наибольшее ненормированное				
$-\infty$		—	—	—
Число в шестнадцатеричном представлении ЗААО	—			

УПРАЖНЕНИЕ 2.60 •

Программы выполняются на машине, где значения типа `int` имеют 32-битовое представление в дополнительном коде. Значения типа `float` используют 32-битовый формат IEEE, а значения типа `double` — 64-битовый формат IEEE.

Генерируются произвольные целые значения `x`, `y`, `z` и преобразуются в другие `double` следующим образом:

```
/* Создание некоторых произвольных значений */
int x = random ();
int y = random ();
int z = random ();

/* Преобразование в double */
doubledx = (double) x;
doubledy = (double) y;
doubledz = (double) z;
```

Для каждого из следующих выражений С необходимо указать, всегда или нет данное выражение дает 1. Если — да, то опишите математические принципы, лежащие в основе этого. Если — нет, приведите пример аргументов, при которых оно выдает 0. Заметьте, что для тестирования ответов нельзя пользоваться машиной IA 32 с gcc, поскольку на ней будет применяться 80-битовое представление повышенной точности как для `float`, так и для `double`.

1. `(double) (float) x == dx`
2. `dx + dy == (double) (y+x)`

3. $dx + dy + dz == dz + dy + dx$
4. $dx + dy + dz == dz + dy + dx$
5. $dx / dx == dy / dy$

УПРАЖНЕНИЕ 2.61 •

Дано задание написать функцию C для расчета представления с плавающей точкой 2^x . Понятно, что наилучшим способом будет прямое построение IEEE-представления результата с одинарной точностью. Если x — слишком маленькая величина, тогда рутинная процедура возвратит 0.0. Если x — слишком большая величина, тогда возвратится $+\infty$. Заполните пропущенные места листинга кода для расчета корректного результата. Предположим, что функция $u2f$ возвращает значение с плавающей точкой, имеющее битовое представление, идентичное битовому представлению его аргумента без знака.

```
float fpwr2 (int x)
{
/* Результатирующий порядок и мантисса */
unsigned exp, sig;
unsigned u;

if (x < _____) /* Слишком маленькое. Возврат = 0.0 */
exp = _____;
sig = _____;
} else if (x < _____) /* Ненормализованный результат. */
exp = _____;
sig = _____;
} else if (x < _____) /* Нормализованный результат. */
exp = _____;
sig = _____;
} else /* Слишком большое. Возврат = +∞ */
exp = _____;
sig = _____;
}

/* Упаковка exp и sig в 32 бита */
u = exp << 23 | sig;
/* Возврат в виде float */
return u2f (u);
}
```

УПРАЖНЕНИЕ 2.62 •

Приблизительно в 250 году до н. э. греческий математик Архимед доказал, что

$$\frac{223}{71} < \pi < \frac{22}{7}.$$

Имей он компьютер и стандартную библиотеку <math.h>, он бы смог определить, что приближение плавающей точки одинарной точности π имеет шестнадцатеричное представление 0x40490FDB. Разумеется, все это только приближения, потому что π — число не рациональное.

1. Каково дробное двоичное число, обозначенное этой величиной с плавающей точкой?
2. Каково дробное двоичное представление $\frac{22}{7}$? Подсказка: см. упр. 2.55.
3. В какой позиции двоичного разряда (относительно двоичной точки) эти два приближения π расходятся?

Решение упражнений

РЕШЕНИЕ УПРАЖНЕНИЯ 2.1

Понимание соотношения между шестнадцатеричным и двоичным форматами приобретет особую важность при рассмотрении программ на машинном уровне. Способ осуществления этих преобразований приведен в тексте, однако для его освоения требуется определенный практический навык.

1. 0x8F7A93 в двоичное:

Шестнадцатеричное	8	F	7	A	9	3
Двоичное	1000	1111	0111	1010	1001	0011

2. Двоичное 1011011110011100 в шестнадцатеричное:

Двоичное	1011	0111	1001	1100
Шестнадцатеричное	B	7	9	C

3. 0xC4E5D в двоичное:

Шестнадцатеричное	C	4	E	5	D
Двоичное	1100	0100	1110	0101	1101

4. Двоичное 1101011011011111100110 в шестнадцатеричное:

Двоичное	11	0101	1011	1110	1110	0110
Шестнадцатеричное	3	5	B	7	E	6

РЕШЕНИЕ УПРАЖНЕНИЯ 2.2

Данная проблема дает шанс задуматься о степенях двойки и их шестнадцатеричных представлениях.

<i>n</i>	2^n (Двоичное)	2^n (Шестнадцатеричное)
11	2048	0x800
7	128	0x80
13	8192	0x2000
17	131072	0x20000
16	65536	0x10000
8	256	0x100
5	32	0x20

РЕШЕНИЕ УПРАЖНЕНИЯ 2.3

Данная проблема дает шанс попробовать выполнить преобразования между шестнадцатеричным и десятичным представлением некоторых небольших чисел. Для больших чисел гораздо удобнее и надежнее пользоваться калькулятором или программой преобразования.

Десятичное	Двоичное	Шестнадцатеричное
0	00000000	00
$55 = 3 \cdot 16 + 7$	0011 0111	37
$136 = 8 \cdot 16 + 8$	1000 1000	88
$243 = 15 \cdot 16 + 3$	1111 0011	F3
$5 \cdot 16 + 2 = 82$	0101 0010	52
$10 \cdot 16 + 12 = 172$	1010 1100	A8
$14 \cdot 16 + 7 = 231$	1110 0111	E7
$10 \cdot 16 + 7 = 167$	1010 0111	A7
$3 \cdot 16 + 14 = 62$	0011 1110	3E
$11 \cdot 16 + 12 = 188$	1011 1100	B8

РЕШЕНИЕ УПРАЖНЕНИЯ 2.4

При начале отладки программ машинного уровня обнаружится много случаев, когда будет полезна определенная шестнадцатеричная арифметика. Всегда можно преобразовать числа в десятичные значения, выполнить арифметические операции и преобразовать их обратно, однако возможность работать непосредственно с шестнадцатеричными значениями — более эффективна и информативна.

- $0x502c + 0x8 = 0x5034$. Прибавление 8 к шестнадцатеричному значению дает 4 с переносом 1.
- $0x502c - 0x30 = 0x4ffc$. Вычитание 3 из 2 в позиции второй цифры требует займа от третьей. Поскольку эта цифра — 0, то заем необходим и от четвертой позиции.
- $0x502c + 64 = 0x506c$. Десятичное значение 64 (2^6) равно шестнадцатеричному значению $0x40$.
- $0x51da - 0x502c = 0xae$. Для вычитания шестнадцатеричного с (десятичное 12) из шестнадцатеричного а (десятичное 10) займем 16 от второй цифры, в результате чего получим шестнадцатеричное в (десятичное 14). Теперь во второй цифре вычитаем 2 из шестнадцатеричного с (десятичное 12), в результате чего получим шестнадцатеричное а (десятичное 10).

РЕШЕНИЕ УПРАЖНЕНИЯ 2.5

С помощью данного упражнения проверяется понимание студентом байтового представления данных и двух различных способов упорядочивания байтов.

- | | |
|----------------------------|------------------------|
| 1. Остроконечные: 78 | Тупоконечные: 12 |
| 2. Остроконечные: 78 56 | Тупоконечные: 12 34 |
| 3. Остроконечные: 78 56 34 | Тупоконечные: 12 34 56 |

Вспомните, что `show_bytes` перечисляет серию байтов, начинающуюся с байта самого младшего адреса и перемещающуюся к байту самого старшего адреса. На "остроконечной" машине перечисление байтов будет осуществляться от наименее значимого к наиболее значимому. На "тупоконечной" машине перечисление байтов осуществляется от наиболее значимого — к наименее значимому.

РЕШЕНИЕ УПРАЖНЕНИЯ 2.6

Это упражнение — еще один шанс попрактиковаться в преобразовании из шестнадцатеричного счисления в двоичное. Здесь также разъясняются тонкости целочисленного представления и представления с плавающей точкой. Далее эти вопросы рассматриваются более подробно

- Используя запись представленного в тексте примера, записываем две следующие строки:

```

0 0 3 5 4 3 2 1
00000000001101010100001100100001
*****
4 A 5 5 0 C 8 4
01001010010101010000110010000100

```

- При втором слове, сдвинутом на две позиции относительно первого, получаем последовательность из 21 разрядов совпадения.
- Все биты целого числа оказываются встроенными в число с плавающей точкой, за исключением наиболее значимого бита, имеющего значение 1. Это — случай для

примера, приведенного в тексте. Кроме этого, число с плавающей точкой имеет некоторые не равные нулю старшие биты, не совпадающие с битами целого числа.

РЕШЕНИЕ УПРАЖНЕНИЯ 2.7

Распечатка: 41 42 43 44 45 46. Вспомните о том, что библиотечная программа `strlen` не учитывает прерывающий нулевой символ, поэтому `show_bytes` осуществляла распечатку только через символ F.

РЕШЕНИЕ УПРАЖНЕНИЯ 2.8

Данная проблема является упражнением в помощь ознакомлению с булевыми операциями.

Операция	Результат
a	[01101001]
b	[01010101]
$\sim a$	[10010110]
$\sim b$	[10101010]
$a \& b$	[01000001]
$a b$	[01111101]
$a ^ b$	[00111100]

РЕШЕНИЕ УПРАЖНЕНИЯ 2.9

Данная проблема иллюстрирует использование булевой алгебры для описания и обоснования систем реальных задач. Можно понять, что эта цветовая алгебра идентична булевой по битовым векторам длины 3.

- Цвета дополняются дополнением величин R, G и B. Из этого можно сделать вывод, что Белый является дополнением Черного, Желтый — Синего, Алый — дополнением Зеленого, а Голубой — Красного.
- Черный равен 0, а Белый равен 1.
- Булевые операции выполняются на основе представления битовых векторов цветов. Получаем следующее:

$$\text{Синий (001)} \mid \text{Красный (100)} = \text{Алый (101)}$$

$$\text{Алый (101)} \& \text{Голубой (011)} = \text{Синий (001)}$$

$$\text{Зеленый (010)} \wedge \text{Белый (111)} = \text{Алый (101)}$$

РЕШЕНИЕ УПРАЖНЕНИЯ 2.10

Данная процедура основана на том факте, что ИСКЛЮЧИТЕЛЬНОЕ ИЛИ — коммутативно и ассоциативно, и что $a \wedge a = 0$ для любого значения a . В главе 5 описывается случай некорректной работы кода, когда два указателя x и y равны (указывают на одно и то же местоположение).

Шаг	$*x$	$*y$
Первоначально	a	b
Шаг 1	$a \wedge b$	b
Шаг 2	$a \wedge b$	$(a \wedge b) \wedge b = (b \wedge b) \wedge a = a$
Шаг 3	$(a \wedge b) \wedge a = (a \wedge a) \wedge b = b$	a

РЕШЕНИЕ УПРАЖНЕНИЯ 2.11

Ответом являются выражения:

```
x | ~0xFF
x ^ 0xFF
x & ~0xFF
```

Это — типичные выражения, которые можно получить при выполнении битовых операций на низком уровне. Выражение $\sim 0xFF$ создает маску, в которой 8 наименее значимых битов равны нулю, а все остальные — единице. Обратите внимание на то, что такая маска создается независимо от длины слова. В противоположность этому выражение $0xFFFFFFF00$ будет работать только на 32-битовой машине.

РЕШЕНИЕ УПРАЖНЕНИЯ 2.12

Данные упражнения помогают понять соотношение между булевыми операциями и типичными операциями маскирования. Рассмотрим следующий код:

```
/* Набор битов */
int bis (int x, int m)
{
int result = x | m
return result;
}
/* Стирание битов */
int bic (int x, int m)
{
int result = x & ~m;
return result;
}
```

Легко заметить, что `bis` эквивалентен булевому OR — бит задан в z так же, как если он задан в x ИЛИ в m .

Операция `bit` — более тонкая. Необходимо задать бит z до нуля, если соответствующий бит m равен единице. При дополнении маски добавлением $\sim m$ захочется задать бит z до нуля, если соответствующий бит дополненной маски равен нулю. Это можно сделать с помощью операции AND.

РЕШЕНИЕ УПРАЖНЕНИЯ 2.13

В таблице приведены соответствия между побитовыми операциями и логическими выражениями в С.

Выражение	Значение	Выражение	Значение
<code>x & y</code>	0x02	<code>x && y</code>	0x01
<code>x y</code>	0xF7	<code>x y</code>	0x01
<code>~x ~y</code>	0xFD	<code>!x !y</code>	0x00
<code>x & !y</code>	0x00	<code>x && ~y</code>	0x01

РЕШЕНИЕ УПРАЖНЕНИЯ 2.14

Выражение — `! (x ^ y)`.

То есть, $x ^ y$ будет равно нулю, если, и только если каждый бит x совпадает с соответствующим битом y . После этого исследуется способность `!` к определению того, содержит ли слово биты, не равные нулю.

Для использования данного выражения не существует никакой другой причины, кроме как простого написания `x == y`, но при этом демонстрируются некоторые нюансы логических операций и операций на уровне бита.

РЕШЕНИЕ УПРАЖНЕНИЯ 2.15

Данная проблема является упражнением для понимания различных операций сдвига.

x	$x \ll 3$	$x \gg 2$ Логический	$x \gg 2$ Арифметический
Шестн. Двоичный <code>0xFE</code>	Двоичный Шестн. [11110000]	Двоичный Шестн. [00111100]	Двоичный Шестн. [11111100] 0xFC
Шестн. Двоичный <code>0x0F</code>	Двоичный Шестн. [00001111]	Двоичный Шестн. [00000011]	Двоичный Шестн. [00000011] 0x03
Шестн. Двоичный <code>0xCC</code>	Двоичный Шестн. [11001100]	Двоичный Шестн. [00110011]	Двоичный Шестн. [11110011] 0xF3
Шестн. Двоичный <code>0x55</code>	Двоичный Шестн. [01010101]	Двоичный Шестн. [00010101]	Двоичный Шестн. [00010101] 0x15

РЕШЕНИЕ УПРАЖНЕНИЯ 2.16

Вообще говоря, работа на примерах для очень малых длин слов — очень хороший способ понимания компьютерной арифметики.

Значения без знака соответствуют указанным в табл. 2.1. Для значений в дополнительном коде шестнадцатеричные величины от 0 до 7 имеют наиболее значимый бит 0, дающий неотрицательные значения, тогда как шестнадцатеричные цифры от 8 до F имеют наиболее значимый бит, равный 1, дающий отрицательное значение.

\bar{x}	$B2U_4(\bar{x})$	$B2T_4(\bar{x})$
Шести. Двоичное		
A [1010]	$2^3 + 2^1 = 10$	$-2^3 + 2^1 = -6$
0 [0000]	0	0
3 [0011]	$2^1 + 2^0 = 3$	$2^1 + 2^0 = 3$
8 [1000]	$2^3 = 8$	$-2^3 = -8$
C [1100]	$2^3 + 2^2 = 12$	$-2^3 + 2^2 = -4$
F [1111]	$2^3 + 2^2 + 2^1 + 2^0 = 15$	$-2^3 + 2^2 + 2^1 + 2^0 = -1$

РЕШЕНИЕ УПРАЖНЕНИЯ 2.17

Для 32-битовой машины любое значение, состоящее из восьми шестнадцатеричных цифр, начинающееся с одной из цифр диапазона от 8 до f, представляет отрицательное число. Видеть, что числа начинаются со строки символов f — достаточно распространенное явление, поскольку ведущие биты отрицательного числа — все единицы. Однако нужно быть внимательным. Например, число 0x80483b7 имеет только семь цифр. Ввод в качестве ведущего бита нуля дает 0x080483b7 — положительное число.

80483b7:	81	ec	84	01	00	00	sub	\$0x184, %esp	A.	388
80483bd:	53						push	%ebx		
80483be:	8b	55	08				mov	0x8 (%ebp), %edx	B.	8
80483c1:	8b	5d	0c				mov	0xc (%ebp), %ebx	C.	12
80483c4:	8b	4d	10				mov	0x10 (%ebp), %ecx	D.	16
80483c7:	8b	85	94	fe	ff	ff	mov	0xffffffe94 (%ebp), %eax	E.	-364
80483cd:	01	cb					add	%ecx, %ebx		
80483cf:	03	42	10				add	0x10 (%edx), %eax	F.	16
80483d2:	89	85	a0	fe	ff	ff	mov	%eax, 0xffffffe0 (%ebp)	G.	-352
80483d8:	8b	85	10	ff	ff	ff	mov	0xfffffff10 (%ebp), %eax	H.	-240
80483de:	89	42	1c				mov	%eax, 0x1c (%edx)	I.	28
80483e1:	89	9d	7c	ff	ff	ff	mov	%ebx, 0xfffffff7c (%ebp)	J.	-132
80483e7:	8b	42	18				mov	0x18 (%edx), %eax	K.	24

РЕШЕНИЕ УПРАЖНЕНИЯ 2.18

С математической точки зрения, функции $T2U$ и $U2T$ очень необычны. Важно разобраться в их поведении и понять его.

Данная проблема решается путем переупорядочения строк в решении упражнения 2.16, в соответствии со значением в дополнительном коде с последующей распечаткой значения без знака как результата применения функции. Для конкретизации процесса приводим шестнадцатеричные значения.

\bar{x} (шестн.)	(\bar{x})	$T2U_4(\bar{x})$
8	-8	8
A	-6	10
C	-4	12
F	-1	15
0	0	0
3	3	3

РЕШЕНИЕ УПРАЖНЕНИЯ 2.19

Данным упражнением проверяется понимание уравнения 2.4.

Для записей в первых четырех рядах значения x — отрицательные, а $T2U_4(x) = x + 2^4$.

Для оставшихся двух строк значения x — не отрицательны, и $T2U_4(x) = x$.

РЕШЕНИЕ УПРАЖНЕНИЯ 2.20

Данная проблема закрепляет понимание соотношения между представлением в дополнительном двоичном коде и представлением числа без знака, а также влияние правил С. Вспомните, что $TMin_{32}$ равно -2147483648, и при приведении к значению без знака оно превращается в 2147483648. Кроме этого, если какой-либо из операндов — без знака, тогда другой операнд будет приведен к значению без знака до того, как будет проводиться сравнение.

Выражение	Тип	Оценка
$-2147483647 - 1 == -2147483648U$	Без знака	1
$-2147483647 - 1 < -2147483647$	Со знаком	1
$(unsigned) (-2147483647 - 1) < -2147483647$	Без знака	1
$-2147483647 - 1 < 2147483647$	Со знаком	1
$(unsigned) (-2147483647 - 1) < 2147483647$	Без знака	0

РЕШЕНИЕ УПРАЖНЕНИЯ 2.21

Выражения в этих функциях являются типичными для извлечения значений из слова, в котором поля множественных битов запакованы. Здесь используются свойства заполнения нулей и расширения знака различных операций сдвига. Внимательно сле-

дите за упорядочиванием операций приведения и сдвига. В `fun1` сдвиги выполняются с `word` без знака, следовательно, сдвиги являются логическими. В `fun2` сдвиги выполняются после приведения `word` к `int`, следовательно, сдвиги являются арифметическими.

1. Результаты приведены в следующей таблице:

w	fun1 (w)	fun2 (w)
127	127	127
128	128	-128
255	255	-1
256	0	0

2. Функция `fun1` выделяет величину из низших 8 битов аргумента с выводом целых чисел в диапазоне от 0 до 255. Функция `fun1` также выделяет величину из низших 8 битов аргумента, но при этом выполняет и дополнительный знаковый разряд. В результате получается число в диапазоне от -128 до 127.

РЕШЕНИЕ УПРАЖНЕНИЯ 2.22

Влияние усечения числа без знака достаточно понятно, однако это не относится к числам в дополнительном коде. Данное упражнение помогает исследовать свойства усечения, используя очень маленькие длины слов.

Шестнадцатеричное		Без знака		В дополнительном коде	
Первона-чальное	Усеченное	Первона-чальное	Усеченное	Первона-чальное	Усеченное
0	0	0	0	0	0
3	3	3	3	3	3
8	0	8	0	-8	0
A	2	10	2	-6	2
F	7	15	7	-1	-1

По уравнению (2.7) усечение величины без знака служит только для определения их остатка по модулю 8. Влияние данного усечения на величины со знаком немного более сложное. В соответствии с уравнением (2.8) сначала по модулю 8 рассчитывается остаток аргумента. В результате получаются значения от 0 до 7 для аргументов от 0 до 7, а также для аргументов от -8 до -1. Затем к этим остаткам применяется функция $U2T_3$, в результате чего получаются два повторения последовательностей от 0 до 3 и от -4 до -1.

РЕШЕНИЕ УПРАЖНЕНИЯ 2.23

Упражнение показывает, насколько прост механизм появления ошибок из-за неявного преобразования значения со знаком в значение без знака. Кажется вполне естественным передать параметр `length` в виде числа без знака, поскольку вряд ли кому захочется использовать отрицательную длину. Критерий приостановки `i <= length-1` также представляется вполне натуральным. Однако их объединение дает совершенно неожиданный результат!

Поскольку параметр `length` — без знака, постольку расчет `0 - 1` выполняется с использованием арифметики чисел без знака, что эквивалентно приращению по модулю. Тогда в результате получается $UMax_{32}$ (при условии 32-битовой машины). Сравнение \leq также выполняется с использованием сравнения без знака, и поскольку любое 32-битовое число меньше или равно $UMax_{32}$, данное сравнение всегда выполняется! Таким образом, код стремится "добраться" до неверных элементов массива `a`.

Этот код можно зафиксировать либо объявлением `length` как `int`, либо изменением теста цикла `for` на `i < length`.

РЕШЕНИЕ УПРАЖНЕНИЯ 2.24

Пример является простой демонстрацией арифметики по модулю 16. Самым простым решением будет преобразование шестнадцатеричной комбинации в десятичное значение без знака. Для ненулевых значений x необходимо иметь $(-\frac{1}{4}x) + x = 16$. Затем дополненная величина преобразуется обратно в шестнадцатеричную систему счисления.

x		$-\frac{1}{4}x$	
Шестнадцатеричное	Десятичное	Десятичное	Шестнадцатеричное
0	0	0	0
3	3	13	D
8	8	8	8
A	10	6	6
F	15	1	1

РЕШЕНИЕ УПРАЖНЕНИЯ 2.25

Это упражнение на понимание приращения в дополнительном коде.

x	y	$x + y$	$x + \frac{1}{3}y$	Вариант
-16 [10000]	-11 [10101]	-27	5 [00101]	1

(окончание)

x	y	$x + y$	$x + \frac{t}{5}y$	Вариант
-16 [10000]	-16 [10000]	-32	0 [00000]	1
-8 [11000]	7 [00111]	-1	-1 [11111]	2
-2 [11110]	5 [00101]	3	3 [00011]	3
8 [01000]	8 [01000]	16	-16 [10000]	4

РЕШЕНИЕ УПРАЖНЕНИЯ 2.26

Упражнение помогает понять тонкости отрицания в дополнительном коде с использованием очень маленькой длины слова.

Для $w = 4$ имеем $TMin_4 = -8$. Поэтому -8 является своей собственной аддитивной инверсией, тогда как другие величины отрицаются целочисленным отрицанием.

x		$-\frac{t}{4}x$	
Шестнадцатеричное	Десятичное	Десятичное	Шестнадцатеричное
0	0	0	0
3	3	-3	D
8	-8	-8	8
A	-6	6	6
F	-1	1	1

Для отрицания без знака комбинации битов — те же.

РЕШЕНИЕ УПРАЖНЕНИЯ 2.27

Это упражнение на проверку понимания умножения в дополнительном двоичном коде.

Режим	x		y		$x \cdot y$		Усеченное $x \cdot y$	
Без знака	6	[110]	2	[010]	12	[001100]	4	[100]
В дополнительном коде	-2	[110]	2	[010]	-4	[111100]	-4	[100]

(окончание)

Режим	x		y		$x \cdot y$		Усеченное $x \cdot y$	
Без знака	1	[001]	7	[111]	7	[0000111]	7	[111]
В дополнительном коде	1	[001]	-1	[111]	-1	[1111111]	-1	[111]
Без знака	7	[111]	7	[111]	49	[110001]	1	[001]
В дополнительном коде	-1	[111]	-1	[111]	1	[000001]	1	[001]

РЕШЕНИЕ УПРАЖНЕНИЯ 2.28

В главе 3 будет рассмотрено много примеров команды `leal` в действии. Данная команда предназначена для поддержки арифметики указателей, однако компилятор С часто использует ее в качестве способа выполнения умножения на мелкие константы.

Для каждого значения k можно рассчитать две кратные величины: 2^k (когда b равно 0) и $2^k + 1$ (когда b равно a). Следовательно, можно рассчитать кратные 1, 2, 3, 4, 5, 8 и 9.

РЕШЕНИЕ УПРАЖНЕНИЯ 2.29

Обнаружено, что люди сталкиваются с трудностями при выполнении этого упражнения, при работе непосредственно с компонующим автокодом. Все проясняется при приведении в форму, показанную в `optarith`.

Видно, что M равно 15; $x \cdot M$ рассчитывается как $(x \ll 4) - x$.

Видно, что N равно 4; значение смещения 3 добавляется, когда значение у отрицательное, а значение сдвига вправо 2.

РЕШЕНИЕ УПРАЖНЕНИЯ 2.30

Эти упражнения наглядно демонстрируют необходимость понимания программистами свойств компьютерной арифметики:

1. $(x >= 0) || ((2*x) < 0)$. Ложно. Пусть x будет равным -2147483648 ($TMin_{32}$). Тогда получим, что $2*x$ будет равно 0.
2. $(x \& 7) != 7 || (x \ll 30 < 0)$. Истинно. Если $(x \& 7) != 7$ стремится к нулю, тогда должен быть бит x_2 , равный единице. При сдвиге влево на 30 получится знаковый разряд (бит).
3. $(x * x) >= 0$. Ложно. Когда x равно 65535 ($0xFFFF$), $x * x$ равно -131071 ($0xFFFFE0001$).
4. $x < 0 || -x <= 0$. Истинно. Если x — неотрицательная величина, тогда $-x$ — величина не положительная.
5. $x > 0 || -x >= 0$. Ложно. Пусть x равно -2147483648 ($TMin_{32}$). Тогда x и $-x$ отрицательные величины.

6. $x * y == ux*uy$. Истинно. Умножение в дополнительном коде и умножение числа без знака ведут себя одинаково на уровне бита.

7. $\sim x * y + uy*ux == -y$. Истинно. $\sim x$ равно $-x-1$. $uy*ux$ равно $x*y$. Таким образом, левая часть эквивалентна $-x*y-y+x*y$.

РЕШЕНИЕ УПРАЖНЕНИЯ 2.31

Понимание дробных двоичных представлений является важным шагом на пути к пониманию кодировок с плавающей точкой. Данное упражнение позволяет проверить несколько примеров.

Дробное значение	Двоичное представление	Десятичное представление
$\frac{1}{4}$	0.01	0.25
$\frac{3}{8}$	0.011	0.375
$\frac{23}{16}$	1.0111	1.4375
$\frac{45}{16}$	10.1101	2.8125
$\frac{11}{8}$	1.011	1.375
$\frac{45}{8}$	101.101	5.625
$\frac{49}{16}$	11.0001	3.0625

Одним из простых способов рассмотрения дробных двоичных представлений является представление числа в виде дроби в форме $\frac{x}{2^k}$. Этую дробь можно записать в двоичном счислении, используя двоичное представление x , с двоичной точкой, установленной на k позиций справа. В качестве примера: для $\frac{23}{16}$ имеем $23_{10} = 10111_2$. После этого двоичная точка перемещается на 4 позиции справа для получения 1.0111_2 .

РЕШЕНИЕ УПРАЖНЕНИЯ 2.32

В большинстве случаев ограниченная точность чисел с плавающей точкой не представляет особой проблемы, потому что относительная ошибка расчетов по-прежнему достаточно незначительна. Однако в данном примере система оказалась чувствительной к абсолютной ошибке.

Видно, что $x = 0.1$ имеет следующее двоичное представление:

При сравнении этого с двоичным представлением $\frac{1}{10}$ видно, что величина составляет $2^{-20} \times \frac{1}{10}$, т. е. около 9.54×10^{-8} .

$9.54 \times 10^{-8} \times 100 \times 60 \times 60 \times 10 \approx 0.343$.

$0.343 \times 2000 \approx 687$.

РЕШЕНИЕ УПРАЖНЕНИЯ 2.33

Подробное рассмотрение представлений с плавающей точкой для малых длин слова помогает прояснить принципы работы плавающей точки IEEE. Обратите особое внимание на перенос между нормализованными и ненормализованными величинами.

Разряды	e	E	f	M	V
0 00 00	0	0	0	0	0
0 00 01	0	0	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$
0 00 10	0	0	$\frac{2}{4}$	$\frac{2}{4}$	$\frac{2}{4}$
0 00 11	0	0	$\frac{3}{4}$	$\frac{3}{4}$	$\frac{3}{4}$
0 01 00	1	0	$\frac{0}{4}$	$\frac{4}{4}$	$\frac{4}{4}$
0 01 01	1	0	$\frac{1}{4}$	$\frac{5}{4}$	$\frac{5}{4}$
0 01 10	1	0	$\frac{2}{4}$	$\frac{6}{4}$	$\frac{6}{4}$
0 01 11	1	0	$\frac{3}{4}$	$\frac{7}{4}$	$\frac{7}{4}$
0 10 00	2	1	$\frac{0}{4}$	$\frac{4}{4}$	$\frac{8}{4}$
0 10 01	2	1	$\frac{1}{4}$	$\frac{5}{4}$	$\frac{10}{4}$
0 10 10	2	1	$\frac{2}{4}$	$\frac{6}{4}$	$\frac{12}{4}$
0 10 11	2	1	$\frac{3}{4}$	$\frac{7}{4}$	$\frac{14}{4}$
0 11 00	—	—	—	—	$+\infty$
0 11 01	—	—	—	—	NaN
0 11 10	—	—	—	—	NaN
0 11 11	—	—	—	—	NaN

РЕШЕНИЕ УПРАЖНЕНИЯ 2.34

Шестнадцатеричная величина 0x354321 эквивалентна двоичной величине [1101010100001100100001]. Сдвиг вправо на 21 позицию дает $1.1010101000011001000_2 \times 2^{21}$. При сбросе ведущей единицы и добавлении двух нулей формируется дробное поле, дающее [10101010000110010000100]. Порядок формируется добавлением смещения 127 к 21, что дает 148 (двоичное представление [10010100]). Данное значение объединяется со знаковым полем 0 для получения следующего двоичного представления:

[01001010010101010000110010000100].

Видно, что соотношение между двумя этими представлениями соответствует младшим битам целого числа, до наиболее значимого бита, равного единице, соответствующей 21 старшим битам следующей дроби:

0	0	3	5	4	3	2	1
00000000001101010100001100100001							

4	A	5	5	0	C	8	4
01001010010101010000110010000100							

РЕШЕНИЕ УПРАЖНЕНИЯ 2.35

Данное упражнение помогает задуматься о том, какие числа нельзя представить точно в плавающей точке.

Данное число имеет двоичное представление 1, за которым следует *n* нулей, за которыми следует единица, что дает $2^{n+1} + 1$.

Когда *n* = 23, тогда величина составляет $2^{24} + 1 = 16,777,217$.

РЕШЕНИЕ УПРАЖНЕНИЯ 2.36

Вообще говоря, лучше использовать библиотеку макрокоманд, вместо изобретения собственного кода. Впрочем, данный код работает на множестве различных машин.

Предположим, что значение 1e400 переполняется до бесконечности.

```
1#define POS_INFINITY 1e400
2#define NEG_INFINITY (-POS_INFINITY)
3#define NEG_ZERO (-1.0/POS_INFINITY)
```

РЕШЕНИЕ УПРАЖНЕНИЯ 2.37

Упражнения, подобные этому, помогают в развитии способности обоснования операций с плавающей точкой с точки зрения программиста. Убедитесь в том, что каждый из ответов понятен.

1. $x == (int) (float) x$

Нет. Например, когда $x = TMax$.

2. $x == (\text{int}) (\text{double}) x$

Да, поскольку `double` обладает большей точностью и диапазоном, нежели `int`.

3. $f == (\text{float}) (\text{double}) f$

Да, поскольку `double` обладает большей точностью и диапазоном, нежели `float`.

4. $d == (\text{float}) d$

Нет. Например, когда `d` равно `1e400`, в правой части получаем $+\infty$.

5. $f == -(-f)$

Да, число с плавающей точкой отрицается простой инверсией знака.

6. $2/3 == 2/3.0$

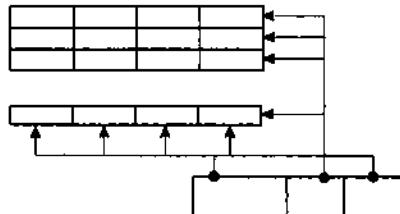
Нет, значение в левой части будет целым значением 0, тогда как значение в правой части будет приближением с плавающей точкой, равное $\frac{2}{3}$.

7. $(d >= 0.0) \mid\mid ((d*2) < 0.0)$

Да, поскольку умножение монотонно.

8. $(d+f) -d == f$

Нет. Например, `d` равно $+\infty$, а `f` — 1, тогда левая часть будет `NaN`, а правая — 1.



ГЛАВА 3

Представление программ на машинном уровне

- Историческая перспектива.
 - Кодирование программ.
 - Форматы данных.
 - Доступ данных.
 - Арифметические и логические операции.
 - Управление.
 - Процедуры.
 - Распределение памяти под массивы и доступ к массивам.
 - Разнородные структуры данных.
 - Выравнивание.
 - Как пользоваться указателями.
 - Использование отладчика.
 - Ссылки на ячейку в памяти и переполнение буферов.
 - Коды с плавающей точкой.
 - Встраивание ассемблерных кодов в программы на языке C.
 - Резюме.
-

При программировании на языках высокого уровня, таких как C, мы не сталкиваемся с необходимостью дальнейшей детализации нашей программы, какой является ее представление на машинном уровне. С другой стороны, разрабатывая программу на языке ассемблера, программист должен точно указать, как программа манипулирует памятью, и указать инструкции нижнего уровня, которые программа использует для вычислений. В большинстве случаев более эффективной и надежной оказывается работа с абстракциями более высокого уровня, которые предоставляют в ваше распоряжение языки программирования высокого уровня. Контроль соответствия типов,

реализуемый компилятором, позволяет обнаружить многие программные ошибки и гарантирует, что мы не ошибаемся, выполняя ссылки на конкретные данные, и правильно манипулируем данными. Программный код, который генерируют современные оптимизирующие компиляторы, по меньшей мере, так же эффективен, как и код, который вручную написал высококвалифицированный программист на языке ассемблера. Более того, программа, написанная на языке программирования высокого уровня, может быть скомпилирована и выполнена на машинах различных типов, в то время как ассемблерный код в значительной степени машинно-ориентированный.

Даже при наличии оптимизирующих компиляторов способность читать и понимать программы на языке ассемблера является существенным преимуществом и признаком высокой квалификации в среде серьезных программистов. Если вызывать компилятор с соответствующими ключами, то он генерирует файл, представляющий собой выходной результат в виде ассемблерного кода. Ассемблерный код очень близок к машинному коду, который исполняет компьютер. Его основное свойство заключается в том, что он представлен в более удобном для чтения формате, по сравнению с двоичным форматом объектного кода. Читая программу на языке ассемблера, вы получаете представление о том, какими возможностями обладает компилятор, и сможете провести анализ неэффективных фрагментов программы. Как вы узнаете из главы 5, программисты, стремящиеся обеспечить максимальную производительность наиболее важных разделов программ, предпринимают попытки использовать различные варианты исходного кода, каждый раз выполняя компонирование и исследование полученного ассемблерного кода, чтобы оценить, насколько эффективной будет исполняемая программа. Более того, имеют место случаи, когда уровни абстракции, обеспечиваемые языками высокого уровня, скрывают информацию о поведении программы во время исполнения, которое мы также должны оценить. Например, когда мы пишем параллельно исполняемые программы, используя поток пакета (thread package), как показано в главе 13, важно знать, какой тип памяти используется для хранения различных переменных программы. Эта информация видна на уровне ассемблерного кода. Потребность программистов изучать ассемблерный код со временем изменилась от способности писать программы непосредственно в кодах ассемблера до способности читать и понимать программные коды, порожденные оптимизирующими компиляторами.

В этой главе мы изучим особенности конкретного языка ассемблера и увидим, как программы на языке C компилируются в эту форму машинного кода. Чтение ассемблерного кода, порожденного компилятором, требует другой квалификации, отличной от способности писать программы на языке ассемблера вручную. Мы должны понимать преобразования, которые осуществляют типичные компиляторы, превращая конструкции языка C в машинные коды. Что касается вычислений, выраженных в программном коде на языке C, то оптимизирующие компиляторы могут поменять порядок вычислений, удалить ненужные вычисления, заменить медленные операции, такие как умножение, реализованное посредством операций сдвига и сложения, и даже изменить рекурсивные вычисления на итеративные. Понимание отношений между исходным программным кодом и порожденным ассемблерным кодом часто может оказаться трудно решаемой проблемой — во многом она подобна составлению мозаики, когда настоящая картинка слегка отличается от той, что изображена на коробке. Это как бы некоторая форма обратного проектирования (reverse engineering) —

попытка понять процесс, в результате которого система была построена путем изучения системы и всех этапов ее построения, двигаясь в обратном порядке. В рассматриваемом случае система представляет собой машинно-генерированную программу на языке ассемблера, а не что-то такое, что было создано человеком. Это упрощает задачу обратного проектирования, поскольку построенный код во многом подчиняется четко определенным правилам, и мы можем проводить эксперименты, в условиях которых компилятор генерирует коды множества различных программ. В данном обзоре технических средств мы приводим множество примеров и упражнений, иллюстрирующих различные аспекты языка ассемблера и компиляторов. Это именно тот случай, когда понимание деталей является предпосылкой для понимания более глубоких фундаментальных понятий. Затраты времени и усилий на изучение примеров и решение упражнений, несомненно, принесут свои плоды.

Далее мы покажем, как развивалась архитектура Intel. Процессоры Intel берут начало от довольно примитивных 16-разрядных процессоров, какими они были в 1978 году, и с течением времени превратились в универсальные вычислительные машины в виде современных настольных компьютеров. Их архитектура развивалась соответствующим образом по мере реализации новых возможностей, и вскоре 16-разрядная архитектура обрела способность поддерживать 32-разрядные данные и адреса. В результате появилась довольно таки своеобразная конструкция, обладающая свойствами, которые имеют смысл, только если рассматривать их в исторической перспективе. Она также перегружена функциональными свойствами, обеспечивающими обратную совместимость, которые не характерны для современных компиляторов и операционных систем. Мы сосредоточим свое внимание на изучении множества свойств, используемых компилятором GCC и Linux. Это позволит нам избежать многих трудностей при изучении скрытых свойств IA-32 (32-разрядной архитектуры для процессоров Intel).

Свой обзор технических средств мы начнем с того, что покажем отношения, которыми связаны между собой язык C, программный код на языке ассемблера и объектный код. Затем мы перейдем к рассмотрению особенностей архитектуры IA-32, начав с изучения представления данных, манипулирования ими и реализации управления. Мы увидим, как реализованы управляющие конструкции языка C, такие как операторы `if`, `while` и `switch`. Затем мы узнаем, как реализуются процедуры, в частности, как стек времени выполнения программ поддерживает передачу данных и управления между процедурами, а также узнаем, как хранятся локальные переменные. Далее мы рассмотрим, как такие конструкции, как массивы, структуры данных и объединения, реализованы на машинном уровне. Обладая такими фундаментальными знаниями в области программирования на машинном уровне, мы можем приступить к исследованию проблем, связанных с выходом за пределы адресного пространства и с уязвимостью систем в отношении атак злоумышленников, использующих метод переполнения буфера. Мы закончим эту часть технического обзора несколькими рекомендациями в отношении использования отладчика GDB с целью исследования поведения программ машинного уровня.

Затем мы перейдем к изучению материала, который предназначен для лиц, проявляющих повышенный интерес к машинным языкам. Мы дадим описание поддержки программ, использующих данные с плавающей точкой. Это специфическое свойство,

характерное только для архитектуры IA-32, поэтому мы рекомендуем изучать материал этого раздела только тем пользователям, которым придется работать с кодами с плавающей точкой. Мы также дадим краткое описание поддержки, которую оказывает GCC встраиванию ассемблерного кода в программы на языке C. В некоторых приложениях программист должен снизойти до ассемблерного кода с тем, чтобы получить доступ к машинным свойствам низкого уровня. Встроенный ассемблер предлагает наилучший способ сделать это.

3.1. Историческая перспектива

Для серии процессоров Intel характерно длительное эволюционное развитие. Серия начинается с одного из первых 16-разрядных микропроцессоров, изготовленного в виде отдельной платы, при разработке которого проектировщикам пришлось пойти на многие компромиссы, обусловленные ограниченными возможностями технологии интегральных схем того времени. С тех пор он претерпел существенные изменения, вбирая в себя технические достижения, позволяющие достигать высокой производительности и поддерживать более совершенные операционные системы.

В приводимом ниже списке представлены модели процессоров Intel и некоторые из их свойств, в порядке возрастания производительности этих процессоров. Мы используем число транзисторов, необходимых для реализации этих процессоров, как показатель того, насколько увеличивалась их сложность (К означает 1000, а М означает 1000000).

8086 (1978 г., 20 К транзисторов), один из первых 16-разрядных однокристальных микропроцессоров. Версия 8088 процессора 8086 с 8-разрядной внешней шиной представляет собой центральное звено подлинных персональных компьютеров корпорации IBM. Корпорация IBM заключила договор с небольшой по тем временам компанией Windows на разработку операционной системы IBM. Первоначальные модели выпускались с основной памятью объемом в 32 768 байтов памяти и с двумя накопителями на гибких магнитных дисках (без жестких дисков). В плане архитектуры эти машины имели ограниченное пространство объемов 655 360 байтов (длина адреса тогда не превышала 20 разрядов, возможность адресации не более 1 048 576 байтов), при этом операционная система резервировала 393 216 байтов для собственных нужд.

- 80286 (1982 г., 134 К транзисторов). Реализовано несколько режимов адресации (они уже устарели). Этот процессор послужил базой для персонального компьютера IBM PC AT, первоначальной платформы MS-Windows.
- i386 (1985 г., 275 К транзисторов). Архитектура расширена до 32 разрядов. Добавлена модель прямой адресации, используемая операционной системой Linux и недавними версиями семейства операционных систем Windows. Это была первая машина серии, которая была способна поддерживать операционную систему Unix.
- i486 (1989 г., 1.9 М транзисторов). Увеличена производительность, в плату процессора интегрирован блок вычислений с плавающей точкой, однако система команд осталась неизменной.

- Pentium (1993 г., 3.1 М транзисторов). Увеличена производительность, однако система команд была расширена незначительно.
- Pentium Pro (1995 г., 6.5 М транзисторов). Совершенно новая конструкция процессора, известная в профессиональных кругах как микроархитектура P6. В систему команд добавлен класс команд "условного перемещения".
- Pentium/MMX (1997 г., 7 М транзисторов). Добавлен новый класс команд в процессор Pentium, позволяющий манипулировать целочисленными векторами. Каждый элемент данных может иметь длину 1, 2 или 4 байта. Каждый вектор состоит из 64 битов.
- Pentium II (1997 г., 7 М транзисторов). Слияние некогда отдельных серий Pentium Pro и Pentium/MMX путем реализации команд MMX в микроархитектуре P6.
- Pentium III (1999 г., 8.2 М транзисторов). Введен еще один класс команд, позволяющий манипулировать целочисленными векторами или данными с плавающей точкой. Каждый элемент данных может иметь длину 1, 2 или 4 байтов, элементы данных упаковываются в 128-разрядный вектор. Более поздние версии этой платы содержат до 24 М транзисторов, благодаря встраиванию кэш-памяти уровня 2.
- Pentium 4 (2001 г., 8.2 М транзисторов). Добавлены 8-байтовый целочисленный формат и формат с плавающей точкой для команд манипулирования векторами, а также 144 новых инструкций для этих форматов. Компания Intel отказалась от римских цифр в своих соглашениях о нумерации.

Каждый очередной процессор разрабатывался с учетом требования обратной совместимости — способности исполнять программу, скомпилированную для более ранних версий процессоров. Как мы увидим далее, существуют множество странных артефактов в системе команд, обусловленных этим эволюционным наследием. Теперь Intel называет систему своих команд "IA32", что означает "32-разрядная архитектура для процессоров Intel". Эта серия процессоров в разговорной речи часто называется "x86", что соответствует соглашению о присвоении имен, действовавшему до появления модели i486.

Почему не i586

Компания Intel прекратила действие соглашения о присвоении имен, поскольку она не смогла обеспечить защиту номеров изготавляемых ею центральных процессоров как товарных знаков. Управление товарных знаков США не разрешает использовать числа в качестве товарных знаков. Вместо чисел использовали имя "Pentium", воспользовавшись греческим словом *pentē* как указанием на то, что соответствующий процессор принадлежит к числу машин пятого поколения. С тех пор Intel использует различные варианты этого имени, не взирая на тот факт, что процессор Pentium Pro есть машина шестого поколения (откуда происходит внутреннее имя P6), а Pentium 4 является машиной седьмого поколения. Каждое новое поколение связано с крупными изменениями в конструкции процессора.

Закон Мура

Если построим график возрастания числа транзисторов в разных процессорах IA32, перечисленных ранее, в зависимости от года выпуска, и воспользуемся логарифмическим масштабом для вертикальной оси, мы убедимся, что этот рост ока-

запся феноменальным. Проведя прямую линию по точкам, соответствующим данным, мы видим, что число транзисторов увеличивается за год примерно на 33%, что означает, что число транзисторов удваивается каждые 30 месяцев. Этот рост сохраняется на протяжении 25-летней истории существования архитектуры IA32.

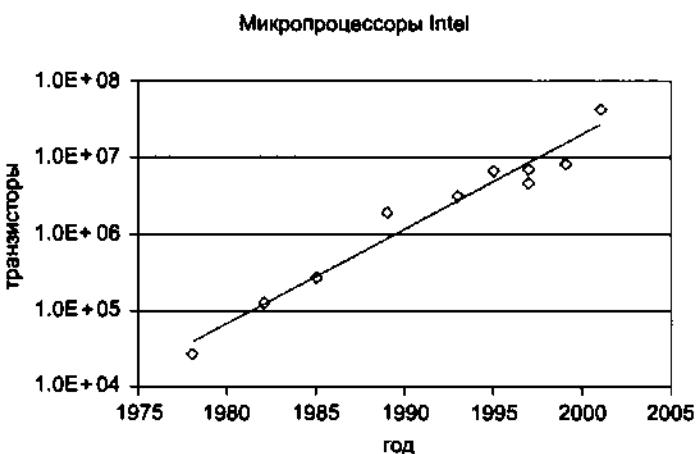


Рис. 3.1. Закон Мура

В 1965 году Гордон Мур (Gordon Moore), основатель компании Intel Corporation, экстраполировал на основании технологий микросхем того времени, когда компания могла производить микросхемы с кристаллами, содержащими примерно 64 транзистора, зависимость, согласно которой число транзисторов в течение следующих 10 лет будет удваиваться каждый год. Это предсказание известно сейчас как закон Мура. Как выяснилось позже, это предсказание несколько оптимистично и в то же время недальновидно. На протяжении своей 40-летней истории полупроводниковая промышленность оказалась способной удваивать число транзисторов каждые 18 месяцев.

Подобные экспоненциальные темпы роста имеют место и в других аспектах компьютерной технологии — емкости дисковой памяти, емкости кристаллов памяти и производительности процессоров. Такие замечательные темпы роста стали основной движущей силой компьютерной революции.

В течение многих лет несколько компаний изготавливали процессоры, совместимые с процессорами компании Intel и способные исполнять те же программы машинного уровня. Главной из них была компания AMD. На протяжении многих лет стратегия компании AMD заключалась в том, чтобы идти в технологии след в след за компанией Intel и производить более дешевые процессоры, хотя и несколько меньшей производительности. Совсем недавно AMD стала производить некоторые из высокопроизводительных процессоров с архитектурой IA32. Они были первыми из серийно выпускаемых микропроцессоров, преодолевшими барьер быстродействия в 1 гигагерц. И хотя настоящий обзор посвящается процессорам производства компании Intel, тем

не менее мы не упускаем из виду и совместимые с ними процессоры, изготавливаемые ее конкурентами.

Большая часть сложных моментов архитектуры IA32, ориентированных на операционную систему Linux, не касается тех, кто проявляет интерес к программам, таким как генерируемые компилятором GCC. Модель памяти, использованная в основоположном процессоре 8086 и его расширении 80286, устарела. Вместо нее Linux использует модель, получившую название *прямой адресации* (flat addressing), в условиях которой все пространство памяти рассматривается программистом как большой массив байтов.

Как следует из приведенного выше списка моделей процессоров, в архитектуру IA32 были добавлены ряд форматов и команд, предназначенных для манипулирования векторами, элементами которых являются небольшие целые числа и числа с плавающей точкой. Эти возможности были добавлены с целью повышения производительности мультимедийных приложений, таких как обработка изображений, кодирование и декодирование видеинформации и трехмерная компьютерная графика. К сожалению, текущие версии компилятора не способны генерировать программный код, в котором используются эти новые свойства. По существу, при вызове по умолчанию компилятор полагает, что он генерирует программный код для i386. Он вообще не предпринимает никаких попыток использовать многочисленные расширения, добавленные в архитектуру, которая сегодня считается устаревшей.

3.2. Кодирование программ

Предположим, что мы написали программу в языке С в виде двух файлов: p1.c и p2.c. Мы компилируем этот программный код, используя для этой цели командную строку Unix:

```
unix> gcc -O2 -o p p1.c p2.c
```

Команда gcc требует использования компилятора GCC языка GNU C. Поскольку этот компилятор используется в Linux по умолчанию, мы можем его вызвать, просто указав cc. Ключ -O2 требует от компилятора применения двухуровневой оптимизации. В общем случае повышение уровня оптимизации повышает скорость выполнения программы, но за счет увеличения времени компиляции и возможного возникновения трудностей при использовании средств отладки. Второй уровень оптимизации можно рассматривать как разумный компромисс между оптимальной производительностью и простотой использования. Все программные коды в этой книге компилировались с использованием этого уровня оптимизации.

Эта команда фактически вызывает некоторую последовательность программ, которые превращают исходный код в рабочую программу. Сначала *препроцессор* (preprocessor) языка C расширяет исходный код с тем, чтобы он включал все файлы, указанные в команде #include, и чтобы он содержал расширения всех использованных макрокоманд. Затем *компилятор* (compiler) генерирует версии ассемблерного кода двух исходных файлов с именами p1.s и p2.s. Далее, *ассемблер* (assembler) преобразует ассемблерный код в двоичные файлы объектных кодов p1.o и p2.o. И, наконец,

нец, *редактор связей* (linker) производит слияние этих двух объектных файлов с кодами, реализующими стандартные функции из библиотеки Unix (например, функция `printf`) и генерирует исполняемый код в его окончательном виде. Более подробно редактирование связей рассматривается в главе 4.

3.2.1. Программный код машинного уровня

Компилятор выполняет почти всю работу общей последовательности действий по компиляции, преобразуя программы, представленные в виде модели исполнения, построенной на языке C, в элементарные инструкции, которые выполняет процессор. Главная особенность этой модели заключается в том, что она представлена в более удобном для чтения формате, по сравнению с двоичным форматом объектного кода. Понимание ассемблерного кода и того, как он соотносится с исходным кодом, является ключевым шагом в понимании того, как компьютеры исполняют программы.

Восприятие машины программистом, работающим на языке ассемблера, существенно отличается от восприятия программиста, работающего на языке C. Ему видны такие состояния процессора, которые скрыты от программиста, работающего на языке C:

- Счетчик команд (с именем `%ip`) указывает адрес в памяти следующей команды, подлежащей исполнению.
- Целочисленный регистровый файл содержит восемь именованных ячеек, в которых хранятся 32-разрядные значения. В этих регистрах могут содержаться адреса (соответствующие указателям языка C) либо целочисленные данные. Некоторые регистры используются для отслеживания наиболее важных частей состояния программы, в то время как другие используются для запоминания промежуточных данных, таких как, например, локальные переменные процедур.
- Регистры кодов условия содержат информацию о последних выполненных арифметических командах. Эти регистры используются для реализации условных изменений управляющей логики программ, которые требуются, например, при реализации операторов `if` или `while`.
- Регистровый файл данных с плавающей точкой состоит из восьми ячеек, предназначенных для хранения соответствующих данных.

В то время как язык C обеспечивает построение модели, объекты которой, представленные различными типами данных, можно объявлять и размещать в памяти, код ассемблера рассматривает память как большой массив с байтовой адресацией. Агрегатные типы данных в языке C, такие как массивы и структуры, представлены в ассемблерном коде как непрерывные совокупности байтов. Даже для скалярных типов данных ассемблерный код не делает различий между целыми со знаком и целыми без знака, между различными типами указателей и даже между указателями и целыми числами.

Память программы содержит объектные коды программ, некоторую информацию, необходимую операционной системе, стек исполняемой программы, используемый для управления вызовами и возвратами процедур, а также блоки памяти, которые распределяет пользователь (например, используя библиотечную процедуру `malloc`).

Адресация памяти программ выполняется с использованием *виртуальных* (virtual) адресов. В любой заданный момент времени только ограниченные субдиапазоны виртуальных адресов можно рассматривать как допустимые. Например, несмотря на то, что 32-разрядные адреса могут потенциально охватывать диапазон значений адресов в 4 Гбайт, обычная программа имеет доступ всего лишь к нескольким мегабайтам. Этим виртуальным пространством адресов управляет операционная система, преобразующая виртуальные адреса в физические адреса со значениями в фактической памяти процессора.

Одна машинная команда может исполнить только очень простую операцию. Например, она может сложить два числа, находящихся в регистрах, передать данные между памятью и регистром или совершить условный переход по адресу новой команды. Задача компилятора заключается в том, чтобы генерировать последовательности команд, которые реализуют такие программные конструкции, как вычисление значений арифметических выражений, циклов или вызовы и возвраты процедур.

3.2.2. Примеры программных кодов

Представим себе, что мы пишем на языке С и помещаем в файл code.c определение следующей процедуры (листинг 3.1):

Листинг 3.1. Определение процедуры

```
1 int accum = 0;
2
3 int sum(int x, int y)
4 {
5     int t = x + y;
6     accum += t;
7     return t;
8 }
```

Чтобы ознакомиться с ассемблерным кодом, составленным компилятором С, мы можем воспользоваться ключом -S в командной строке:

```
unix> gcc -O2 -S code.c
```

По этой команде компилятор генерирует ассемблерный код code.s, но больше ничего не делает. (В нормальной ситуации он далее вызывает ассемблер, чтобы тот построил файл объектного кода.)

Компилятор GCC генерирует ассемблерный код в собственном формате, который называется GAS (сокращение от Gnu ASsembler). В основу используемого нами представления мы положим именно этот формат, который значительно отличается от формата, используемого для документации Intel и компиляторов Microsoft. В библиографических заметках содержится информация о местонахождении документации по различным форматам ассемблерного кода.

Файл ассемблерного кода содержит различные объявления, включая и следующие строки (листинг 3.2):

Листинг 3.2. Ассемблирование процедуры

```
sum:
    pushl %ebp
    movl %esp, %ebp
    movl 12(%ebp), %eax
    addl 8(%ebp), %eax
    addl %eax, accum
    movl %ebp, %esp
    popl %ebp
    ret
```

Каждая строка в указанном программном коде соответствует одной машинной команде. Например, команда `pushl` показывает, что содержимое регистра `%ebp` должно быть протолкнуто в стек программы. Вся информация об именах локальных переменных или типах данных опущена. Все еще остаются ссылки на глобальную переменную `accum`, поскольку компилятор еще не определил, в каком месте памяти эта переменная будет размещена.

Если мы используем параметр `-c` командной строки, компилятор выполнит сразу и компиляцию, и ассемблирование кода:

```
unix> gcc -O2 -c code.c
```

Эта программа генерирует файл с объектным кодом `code.o`, который имеет двоичный формат, и в силу этого обстоятельства его невозможно просматривать непосредственно. В файл `code.o`, содержащий 852 байта, встроена последовательность из 19 байтов, имеющая следующее шестнадцатеричное представление:

```
55 89 e5 8b 45 0c 03 45 08 01 05 00 00 00 00 89 ec 5d c3
```

Эта последовательность есть объектный код, соответствующий командам ассемблера, приведенным выше. Основной урок, который мы должны в этой ситуации усвоить, заключается в том, что программа, которую фактически исполняет машина, есть просто некоторая последовательность байтов, кодирующая соответствующую совокупность команд. Машина получает совсем немного информации об исходном коде, на основании которого эти инструкции были построены.

Как найти представление программы в байтах

Сначала мы использовали обратный ассемблер (краткое описание которого будет приведено далее), чтобы определить, что программный код `sum` содержит 19 байтов. Далее инструментальное GNU-средство отладки GDB мы применим к файлу `code.c` и зададим ему команду

```
(gdb) x/19xb sum
```

проверить (аббревиатура `x`) 19 байтов (аббревиатура `b`) в шестнадцатеричном формате (еще раз аббревиатура `x`). Вы вскоре обнаружите, что инструментальное средство GDB имеет много свойств, полезных при анализе программ машинного уровня, об этом пойдет речь в разд. 3.12.

Незаменимым средством исследования файлов с объектными кодами является класс программ, получивших название *обратных ассемблеров* (*disassemblers*). Эти программы на базе объектного кода генерируют формат, подобный ассемблерному коду. В системах Linux эту задачу решает программа `objdump` (сокращение от `object dump` — дамп объектной программы), если из командной строки задать соответствующую команду с признаком `-d`:

```
unix> objdump -d code.o
```

Результат (мы только добавили нумерацию строк слева и комментарий справа) принимает вид листинга 3.3:

Листинг 3.3. Обратное ассемблирование процедуры

Обратное ассемблирование функции `sum` в файле `code.o`

Смещение	Байты	Эквивалент в языке ассемблера
2 0:	55	push %ebp
3 1:	89 e5	mov %esp,%ebp
4 3:	8b 45 0c	mov 0xc(%ebp),%eax
5 6:	03 45 08	add 0x8(%ebp),%eax
6 9:	01 05 00 00 00 00 00	add %eax,0x0
7 f:	89 ec	mov %ebp,%esp
8 11:	5d	pop %ebp
9 12:	c3	ret
10 13:	90	nop

Слева мы видим 19 шестнадцатеричных значений, представленных в виде последовательности байтов, разбитой на группы, содержащие от 1 до 6 байтов каждая. Каждая из этих групп представляет собой отдельную команду, эквивалент которой в языке ассемблера показан справа. Следует отметить несколько свойств:

- Команды архитектуры IA32 могут отличаться друг от друга по длине и содержать от 1 до 15 байтов. Кодирование команд выполнено таким образом, что часто используемые команды и команды с малым числом операндов требуют меньшего числа байтов, чем редко используемые команды или команды с большим числом операндов.
- Формат команд выбран таким, что с некоторой начальной позиции соответствующие байты однозначно декодируются в машинные команды. Например, только команда `pushl %ebp` начинается с байта со значением 55.
- Обратный ассемблер определяет ассемблерный код, основанный исключительно на последовательностях байтов в объектном файле. Он не требует доступа к исходным кодам или к версии программы в коде ассемблера.
- Обратный ассемблер использует соглашения об именовании команд, несколько отличные от тех, которые использует формат `gas`. В рассматриваемом примере он опускает суффикс `l` во многих командах.

- Мы видим, что по сравнению с ассемблерным кодом из code.s, в конце последовательности стоит команда `por`. Эта команда никогда не будет исполнена (она следует после команды возврата из процедуры), а если бы она выполнялась, то не оказывала бы никакого влияния на дальнейший ход событий (отсюда ее название `por`, что означает "по operation" — никаких операций, в разговорной речи это звучит как "по опр"). Компилятор вставляет эту инструкцию, как один из способов зарезервировать дополнительное пространство памяти для процедуры.

Построение фактической рабочей программы требует прогона редактора связей на некотором множестве файлов объектных кодов, один из которых обязательно должен содержать функцию `main`. Предположим, что в файле `main.c` мы имеем следующую функцию (листинг 3.4):

Листинг 3.4. Прогон процедуры

```
1 int main()
2 {
3     return sum(1, 3);
4 }
```

В таком случае мы можем построить рабочую программу следующим образом:

```
unix> gcc -O2 -o prog code.o main.c
```

Размеры файла `prog` возросли до 11 667 байтов, поскольку он содержит не только коды двух других процедур, но также и информацию, используемую для запуска и останова программы, равно как и для взаимодействия с операционной системой. Мы можем также выполнить обратное ассемблирование файла `prog`:

```
unix> objdump -d prog
```

Обратный ассемблер генерирует различные последовательности кодов, в том числе и следующую (листинг 3.5):

Листинг 3.5. Обратное ассемблирование функции

Обратное ассемблирование функции `sum` в файле `prog`

```
1 080483b4 <sum>:
2 80483b4: 55                      push  %ebp
3 80483b5: 89 e5                   mov   %esp,%ebp
4 80483b7: 8b 45 0c                mov   %ebp,%eax
5 80483ba: 03 45 08                add   %eax,%eax
6 80483bd: 01 05 64 94 04 08      add   %eax,0x8049464
7 80483c3: 89 ec                   mov   %ebp,%esp
8 80483c5: 5d                     pop   %ebp
9 80483c6: c3                     ret
10 80483c7: 90                    nop
```

Обратите внимание на то обстоятельство, что этот код почти полностью идентичен коду, полученному в результате обратного ассемблирования файла code.c. Одним из основных различий является то, что адреса, указанные в колонке слева, другие — редактор связей изменил местоположение этого кода, в результате чего адреса команд изменились. Второе отличие состоит в том, что редактор связей окончательно определился в отношении места хранения глобальной переменной accsum. В строке 6 кода обратного ассемблера для файла code.o (см. листинг 3.3) указан нулевой адрес переменной accsum. В коде обратного ассемблера (см. листинг 3.5) адрес этой переменной есть 0x8049464. В этом можно убедиться из отображения соответствующей команды в ассемблерном коде. Это также видно из последних четырех байтов команды, которые представлены в порядке от наименее значимых до наиболее значимых как 64 94 04 08.

3.2.3. Замечание по форматированию

При чтении ассемблерного кода, построенного компилятором GCC, возникают некоторые трудности. Он содержит определенную информацию, которая для нас не представляет интереса. С другой стороны, в нем не содержится никакой информации, которая могла бы послужить описанием программы или того, как она работает. Например, предположим, что файл simple.c содержит код (листинг 3.6):

Листинг 3.6. Пример кода

```
1 int simple(int *xp, int y)
2 {
3     int t = *xp + y;
4     *xp = t;
5     return t;
6 }
```

Компилятор GCC, инициированный с ключом -S, генерирует следующий файл для simple.c (листинг 3.7):

Листинг 3.7. Пример объектного кода

```
.file      "simple.c"
.version   "01.01"
gcc2_compiled.:          )
.text
.align 4
.globl simple
.type simple,@function
simple:
    pushl %ebp
```

```

movl    %esp, %ebp
movl    8(%ebp), %eax
movl    (%eax), %edx
addl    12(%ebp), %edx
movl    %edx, (%eax)
movl    %edx, %eax
movl    %ebp, %esp
popl    %ebp
ret
.Lf1:
.size simple,.Lf1-simple
.ident "GCC: (GNU) 2.95.3 20010315 (release)"

```

Этот файл содержит больше информации, чем нам нужно на самом деле. Все строки, начинающиеся с точки, суть директивы, управляющие работой ассемблера или редактора связей. Как правило, мы будем их игнорировать. С другой стороны, нет разъяснительных комментариев относительно того, что делают те или иные команды, и какое отношение они имеют к исходному коду.

Чтобы сделать ассемблерный код более понятным, представим его в форме, которая предусматривает нумерацию строк и допускает пояснения различного характера. В нашем примере аннотированная версия кода принимает вид листинга 3.8:

Листинг 3.8. Пример кода с пояснениями

1	simple:	
2	pushl %ebp	Сохранить указатель фрейма
3	movl %esp, %ebp	Установить новый указатель фрейма
4	movl 8(%ebp), %eax	Взять хр
5	movl (%eax), %edx	Извлечь *хр
6	addl 12(%ebp), %edx	Прибавить у, чтобы получить т
7	movl %edx, (%eax)	Сохранить т по адресу *хр
8	movl %edx, %eax	Установить т как возвращаемое значение
9	movl %ebp, %esp	Переустановить указатель стека
10	popl %ebp	Переустановить указатель фрейма
11	ret	Возврат результата

В обычных случаях мы будем показывать только те строки кода, которые имеют отношение к обсуждаемым вопросам. Каждая строка пронумерована слева с целью облегчения ссылок, а справа помещены краткие пояснения относительно результатов выполнения команды и того, какое отношение эта команда имеет к вычислениям, реализуемым исходным кодом на языке С. Именно такой и является стилизованная версия формата, в котором программисты, работающие на языке ассемблера, представляют свои программы.

3.3. Форматы данных

Благодаря своему происхождению как 16-разрядной архитектуры, которая затем расширилась в 32-разрядную архитектуру, Intel использует термин "слово" для обозначения 16-разрядного типа данных. Это послужило причиной того, что 32-разрядные числа иногда называют "двойным словом". Многие команды, с которыми нам придется сталкиваться, оперируют байтами или двойными словами.

В табл. 3.1 показаны машинные представления, используемые для простых типов данных языка С. Обратите внимание на тот факт, что большая часть обычных типов данных хранится в виде двойных слов. Это относится как к обычному типу `int`, так и к типу `long int`, со знаком и без знака. Кроме того, все указатели (в таблице они отмечены звездочкой) хранятся как 4-байтовые слова. Обычно байты используются в случае манипулирования строковыми данными. Числа с плавающей точкой выступают в трех форматах: значения одинарной точности (4 байта), соответствующие типу `float` языка С, значения двойной точности (8 байтов), соответствующие типу `double` языка С, и значения повышенной точности `long double` (10 байтов). Компилятор GCC использует тип данных `long double` для ссылок на значения с плавающей точкой повышенной точности. Он также хранит их как 12-байтовые значения с целью улучшения характеристик памяти, как мы убедимся ниже. И хотя стандарт ANSI C рассматривает `long double` как тип данных, данные этого типа реализуются для большинства сочетаний компилятора и машины с использованием того же 8-байтового формата, что и обычный тип `double`. Поддержка повышенной точности реализована только в сочетании компилятора GCC и архитектуры IA32.

Таблица 3.1. Размеры стандартных типов данных

Описание в С	Тип данных Intel	Суффикс GAS	Размер (в байтах)
<code>char</code>	Байт	b	1
<code>short</code>	Слово	w	2
<code>int</code>	Двойное слово	l	4
<code>unsigned</code>	Двойное слово	l	4
<code>long int</code>	Двойное слово	l	4
<code>unsigned long</code>	Двойное слово	l	4
<code>char *</code>	Двойное слово	l	4
<code>float</code>	Одинарная точность	t	4
<code>double</code>	Двойная точность	l	8
<code>long double</code>	Повышенная точность	t	10/12

Как показывает таблица, каждая операция в формате GAS имеет суффикс из одного символа, обозначающий размер операнда. Например, команда (переслать данные)

может быть представлена в трех вариантах: moveb (переслать байт), movew (переслать слово) и move1 (переслать двойное слово). Суффикс 1 используется для двойных слов, поскольку на многих машинах 32-разрядные значения называются "двойными словами", реликт эпохи, когда 16-разрядные размеры слова были стандартными. Обратите внимание на то обстоятельство, что суффикс 1 используется для обозначения как 4-байтовых целых чисел, так и для обозначения 8-байтовых чисел с плавающей точкой двойной точности. При этом никакая двусмысленность не возникает, поскольку плавающая точка требует использования совершенно другого набора команд и регистров.

3.4. Доступ к данным

Центральное процессорное устройство (CPU — Central Processing Unit) архитектуры IA32 содержит набор из восьми регистров, предназначенных для запоминания 32-разрядных значений. Эти регистры используются для хранения целочисленных данных, а также указателей. На рис. 3.1 представлена диаграмма этих восьми регистров. Имена всех регистров начинаются с %, однако во всем остальном это различные имена. В первоначальной модели 8086 регистры были 16-разрядными, и каждый из них имел собственное назначение. При переходе на прямую адресацию потребность в специализированных регистрах существенно уменьшилась. По большей части, шесть первых регистров можно рассматривать как регистры общего назначения, при этом не существует никаких ограничений на их использование. Мы говорим "по большей части", поскольку некоторые команды используют фиксированные регистры как регистры-источники и регистры-назначения. Кроме того, в разных процедурах соглашения относительно сохранения и восстановления трех первых регистров (%eax, %esi и %edi) отличаются от аналогичных соглашений, касающихся трех следующих регистров (%ebx, %edi и %esi). Этот вопрос мы будем рассматривать в разд. 3.7. Оставшиеся два регистра (%ebp и %esp) содержат указатели на важные места в стеке программы. Их содержимое может быть изменено только в соответствии с системой стандартных соглашений, касающихся управления стеками.

Из рис. 3.2 следует, что информацию из двух младших байтов первых четырех регистров можно отдельно читать и записывать в них данные посредством побайтовых операций. Эта возможность была реализована в модели 8086 в целях обеспечения обратной совместимости с моделями 8008 и 8080 — два 8-разрядных микропроцессора, появившиеся еще в 1974 году. Когда байтовая команда обновляет содержимое одного из таких однобайтовых "регистровых элементов", три остальные байта регистра не меняют свои значения. Аналогично, 16 младших разряда каждого регистра можно читать и менять их значения с помощью операций со словами. Это свойство происходит от 16-разрядных микропроцессоров, от которых берет начало архитектура IA32.

3.4.1. Спецификаторы operandов

Большая часть инструкций принимает один или большее число operandов (operands), описывающих исходные значения, на которые производятся ссылки при выполнении

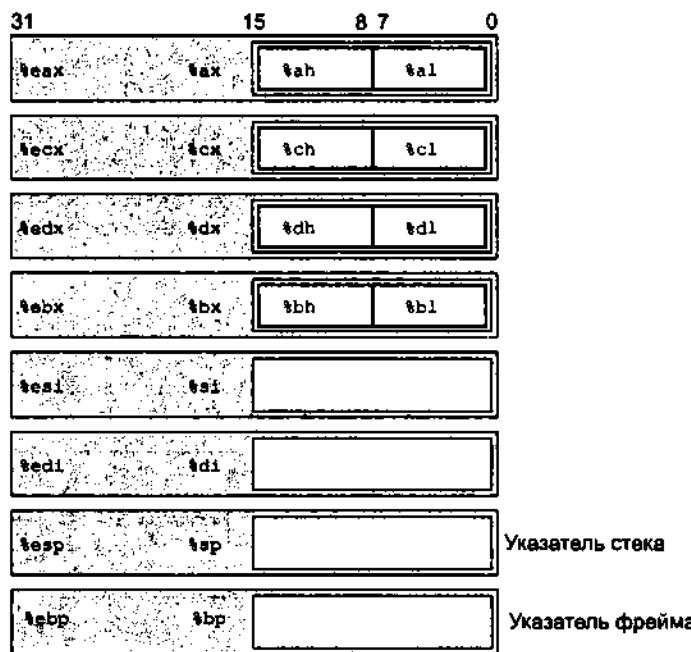


Рис. 3.2. Целочисленные регистры

соответствующей операции, и место назначения, куда записывается полученный результат. Архитектура IA32 поддерживает несколько форматов операторов (табл. 3.2). Исходные значения могут быть представлены в виде констант, либо могут быть считаны из регистров или из памяти. Результаты могут быть помещены в регистр или в память. Следовательно, возможные виды операндов могут быть представлены тремя категориями. Первый тип, *непосредственные операнды* (immediate operands), представляют константы. В формате GAS такие операнды начинаются символом '\$', за которым в стандартной нотации языка C следует целое число, например, \$-577 или \$0x1F. Может быть использовано любое значение, которое представлено 32-разрядным словом, хотя ассемблер будет пользоваться одно- или двухбайтовыми кодами там, где это возможно. Второй тип, *регистр* (register), означает содержимое одного из регистров, либо элементов одного из восьми 32-разрядных регистров (например, %eax), используемого при выполнении двухсловной операции, либо один из восьми однобайтовых элементов регистров (например, %a1) для операций с байтами. В табл. 3.2 мы используем нотацию E_a для обозначения произвольного регистра a , а его значение обозначаем через $R[E_a]$, рассматривая набор регистров как массив R , индексированный идентификаторами регистров.

Третий тип операнда есть *ссылка на ячейку памяти* (memory reference), дающая доступ к некоторой ячейке памяти, адрес которой получен в результате соответствующих вычислений. Этот адрес часто называется *исполнительным адресом* (effective address). Поскольку мы рассматриваем память как большой байтовый массив, мы

рассматриваем обозначение $M_b[Addr]$ как ссылку на b -байтовое значение, сохраненное в памяти, начиная с адреса $Addr$. Чтобы упростить изложение, мы будем опускать индекс b .

Как показывает табл. 3.2, существуют различные *режимы адресации* (addressing modes), благодаря чему возможны несколько форм ссылок на ячейки памяти. Наиболее общая форма показана в нижних строках таблицы, в которых отображается синтаксис $Imm(E_b, E_i, s)$. Такая ссылка состоит из четырех компонентов: непосредственный сдвиг Imm , базовый регистр E_b , индексный регистр E_i и масштабный коэффициент s , при этом s может принимать значения 1, 2, 4 и 8. Исполнительный адрес вычисляется по формуле $Imm + R[E_b] + R[E_i] \times s$. С этой общей формой приходится сталкиваться при ссылках на элементы массивов. Все другие формы являются просто специальными случаями этой общей формы, в которой те или иные компоненты опущены. Как мы увидим далее, более сложные формы адресации полезны при ссылках на элементы массивов и структур.

Таблица 3.2. Формы операндов

Тип	Форма	Значение операнда	Название
Непосредств.	$\$Imm$	Imm	Непосредственное
Регистр	E_a	$R[E_a]$	Регистр
Память	Imm	$M[Imm]$	Абсолютное
Память	(E_a)	$M[R[E_a]]$	Косвенное
Память	$Imm(E_b)$	$M[Imm] + R[E_b]$	База + смещение
Память	(E_b, E_i)	$M[R[E_b]] + R[E_i]$	Индексированное
Память	$Imm(E_b, E_i)$	$M[Imm + R[E_b]] + R[E_i]$	Индексированное
Память	$(, E_i, s)$	$M[R[E_i]] \cdot s$	Нормированно-индексированное
Память	$Imm(, E_i, s)$	$M[Imm + R[E_i]] \cdot s$	Нормированно-индексированное
Память	(E_b, E_i, s)	$M[R[E_b]] + R[E_i] \cdot s$	Нормированно-индексированное
Память	$Imm(E_b, E_i, s)$	$M[Imm + R[E_b]] + R[E_i] \cdot s$	Нормированно-индексированное

Операнды могут обозначать непосредственные значения (константы), значения регистров и значения, хранящиеся в памяти. Масштабный коэффициент может принимать значения 1, 2, 4 или 8.

УПРАЖНЕНИЕ 3.1

Предположим, что следующие значения хранятся в памяти по указанным адресам и в регистрах:

Адрес	Значение
0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11

Регистр	Значение
%eax	0x100
%ecx	0x1
%edx	0x3

Заполните следующую таблицу, показав значения указанных операндов:

Операнд	Значение
%eax	
0x104	
\$0x108	
(%eax)	
4(%eax)	
9(%eax, %edx)	
260(%ecx, %edx)	
0xFC(, %ecx, 4)	
(%eax, %edx, 4)	

3.4.2. Команды перемещения данных

К числу наиболее часто используемых команд относятся и те, которые осуществляют перемещение данных. Универсальность обозначений операндов позволяет простым командам перемещения данных выполнить то, для чего на многих машинах потребуется последовательность из нескольких команд. В табл. 3.3 представлен список наиболее важных команд перемещения данных. Чаще других используется команда `movl` для перемещения двойных слов. Операнд-источник представляет значение, которое является непосредственным значением, либо хранится в регистре или в памяти. Операнд назначения означает ячейку памяти, которая является либо регистром, либо адресом памяти. Архитектура вводит ограничение, согласно которому оба операнда одновременно не могут быть ссылками на ячейки памяти. Копирование значения из одной ячейки памяти в другую требует выполнения двух команд: первая команда загружает значение в регистр, а вторая команда загружает это значение регистра в ячейку назначения.

Таблица 3.3. Команды перемещения данных

Команда	Результат	Описание
movl S, D	$D \leftarrow S$	Переместить двойное слово
movw S, D	$D \leftarrow S$	Переместить слово
movb S, D	$D \leftarrow S$	Переместить байт
movsbl S, D	$D \leftarrow \text{SignExtend}(S)$	Переместить байт дополнения знаком
movzb1 S, D	$D \leftarrow \text{ZeroExtend}(S)$	Переместить байт дополнения нулями
pushl S	$R[\%esp] \leftarrow R[\%esp] - 4;$ $M[R[\%esp]] \leftarrow S$	Протолкнуть
popl D	$D \leftarrow M[R[\%esp]];$ $R[\%esp] \leftarrow R[\%esp] + 4$	Вытолкнуть

Приводимые в листинге 3.9 примеры команды `movl` представляют пять различных сочетаний типов операндов-источников и операндов-назначений. Напомним, что операнд-источник идет первым, а операнд-назначение идет вторым.

Листинг 3.9. Команда перемещения данных

1	movl \$0x4050,%eax	Непосредственное значение – Регистр
2	movl %ebp,%esp	Регистр – Регистр
3	movl (%edi,%ecx),%eax	Память – Регистр
4	movl \$-17,%esp	Непосредственное значение – Память
5	movl %eax,-12(%ebp)	Регистр – Память

Команда `movb` аналогична команде `movl` за исключением того, что она перемещает один байт. Когда один из операндов есть регистр, он должен быть одним из однобайтовых элементов регистров, показанных на рис. 3.2.

Обе команды `movsbl` и `movzb1` предназначены для копирования одного байта и записи оставшихся разрядов в месте назначения. Команда `movsbl` принимает однобайтовый исходный операнд, выполняет знаковое расширение до 32 разрядов (24 старших двоичных разряда принимают значение наибольшего значащего разряда исходного байта) и копирует этот результат в двойное слово по адресу назначения. Аналогично, команда `movzb1` берет однобайтовый исходный операнд, выполняет его расширение до 32 битов, обнуляя 24 старших двоичных разряда, и копирует этот результат в двойное слово по адресу назначения.

Сравнение команд перемещения байтов

Мы видим, что три команды перемещения и отличаются друг от друга совсем не-значительно. Рассмотрим следующие примеры:

Предположим сначала, что `%dh = 8D`, `%eax = 98765432`

1	<code>movb %dh,%al</code>	<code>%eax = 9876548D</code>
2	<code>movsb</code>	<code>%eax = FFFFFFF8D</code>
3	<code>movzbl</code>	<code>%eax = 0000008D</code>

В этих примерах все команды устанавливают значение младшего байта регистра `%eax` равным значению второго байта регистра `%edx`. Команда `movb` не меняет содержимого трех остальных байт. Команда `movsb` устанавливает все разряды трех остальных байт равными единице или нулю, в зависимости от значения наибольшего значащего разряда исходного байта. Команда `movzbl` устанавливает все разряды трех остальных байт равными нулю в любом случае.

Условно изобразим стек вверх дном (рис. 3.3), так что "вершина" стека показана внизу. Стеки архитектуры IA32 возрастают в направлении меньших адресов. Отсюда следует, что проталкивание записи в стек приводит к уменьшению указателя стека (регистр `%esp`) и запоминанию в памяти, в то время как выталкивание записи из стека влечет считывание из памяти и увеличение значения указателя стека.

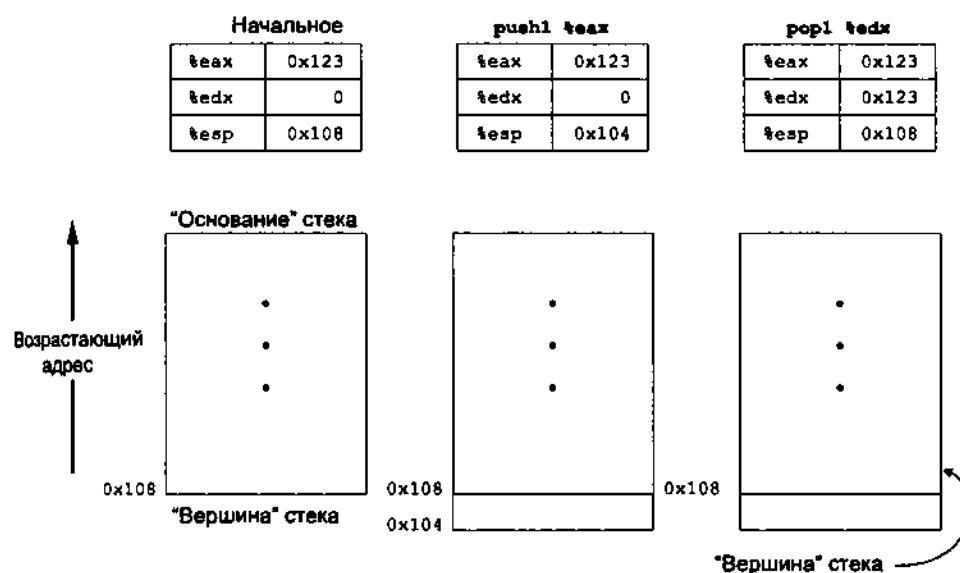


Рис. 3.3. Иллюстрация работы стека

Две последние операции перемещения данных используются для заталкивания данных в программный стек и выталкивания данных из стека. Как мы увидим далее, стек играет жизненно важную роль при обработке вызовов процедур. Обе инструкции `pushl` и `popl` принимают только один operand — источник данных при заталкивании в стек и адрес назначения при выталкивании. Программный стек хранится в специальной области памяти. Как показано на рис. 3.3, стек наращивается вниз таким образом, что элемент в вершине стека имеет наименьший адрес из всех элементов стека.

Указатель стека содержит адрес элемента, находящегося в вершине стека. Следовательно, при заталкивании записи длиной в двойное слово сначала уменьшается на 4 значение указателя стека, а затем это значение записывается по новому адресу вершины стека. В связи с этим результат выполнения команды `pushl %ebp` эквивалентен результату выполнения двух следующих команд:

```
subl $4, %ebp
movl %ebp, (%esp)
```

за исключением того, что команда `pushl` представлена в объектных кодах одним байтом, в то время как пара указанных выше команд требует для себя 6 байтов. Два первых столбца на нашем рисунке показывают результат выполнения команды `pushl %eax`, когда значение `%esp` есть `0x108`, а значение `%eax` есть `0x123`. Сначала значение `%esp` будет уменьшено на 4, после чего `0x123` будет сохранено по адресу `0x104`.

Выталкивание двойного слова выполняется следующим образом: сначала производится считывание ячейки из вершины стека, а затем значение указателя стека увеличивается на 4. Следовательно, команда `popl %eax` эквивалентна двум следующим командам:

```
movl (%esp), %eax
addl $4%, %esp
```

Третий столбец на рис. 3.3 показывает результат выполнения команды `popl %edx` сразу же после выполнения команды `pushl`. Значение `0x123` будет считано из памяти и помещено в регистр `%edx`. Значение регистра `%esp` увеличится до прежнего значения `0x108`. Как показано на рис. 3.3, значение будет оставаться в ячейке памяти с адресом `0x108`, пока он не будет затерт другой операцией проталкивания в стек. В то же время, всегда считается, что вершина стека есть адрес, который указан в регистре `%esp`.

Поскольку стек содержится в той же памяти, что и программный код и другие виды данных программы, то сама программа может осуществлять доступ к любым позициям в стеке, используя для этой цели стандартные методы адресации памяти. Например, предположим, что элемент, находящийся в вершине стека, есть двойное слово, тогда команда `movl 4(%esp), %edx` осуществляет копирование второго двойного слова из стека в регистр `%edx`.

3.4.3. Перемещение данных

Несколько примеров указателей

Функция `exchange` (листинг 3.10) является хорошей иллюстрацией использования указателей в языке С. Аргумент `xp` есть указатель на целое, в то время как `u` есть само это целое. Оператор

```
int x = *xp;
```

указывает на то, что мы должны читать значение, сохраненное в ячейке, обозначенной как `xp`, и запомнить его как локальную переменную с именем `x`. Эта операция

считывания известна как разыменование указателя. Оператор * языка С выполняет разыменование указателя. Оператор

```
*xp = y;
```

выполняет обратное действие — он записывает значение параметра у в ячейке, обозначенной как xp. Это еще одна форма разыменования указателя (а следовательно, и оператора *), здесь он означает операцию записи, т. к. находится в левой части операции присваивания.

Результат действия операции exchange может быть представлен следующим примером:

```
int a = 4;
int b = exchange (&a, 3);
printf ("a = %d, b = %d\n", a, b);
```

Этот программный код печатает

```
a = 3, b = 4
```

Оператор & языка С (адрес переменной) создает указатель, в рассматриваемом случае это указатель на ячейку, содержащую переменную a. Функция exchange затирает значение, сохраняемое в a, и вместо него она возвращает 4 в качестве своего значения. Обратите внимание на то, как, передав указатель функции exchange, он может изменить данные, хранящиеся в дистанционной ячейке.

В качестве примера программы, использующей команды перемещения данных, рассмотрим стандартную программу, которая производит обмен данными. Эта программа представлена как в кодах языка С (листинг 3.10), так и в кодах языка ассемблера (листинг 3.11), порожденных компилятором GCC. Здесь мы опускаем порцию программных кодов на языке ассемблера, которые производят распределение памяти для стека на входе в процедуру, и освобождают память непосредственно перед выходом из нее. Более подробно коды установки стека и завершающие коды мы рассмотрим, когда будем изучать связывание процедур. Программные коды, которые останутся после этого, мы будем называть "телом".

Листинг 3.10. Код С

```
1 int exchange(int *xp, int y)
2 {
3     int x = *xp;
4
5     *xp = y;
6     return x;
7 }
```

Листинг 3.11. Ассемблерный код

```
1    movl  8(%ebp),%eax      Взять xp
2    movl  12(%ebp),%edx     Взять у
```

```

3 movl (%eax),%ecx      Взять x из *хр
4 movl %edx,(%eax)       Взять у из *хр
5 movl %ecx,%eax         Установить x как возвращаемое значение

```

Когда тело процедуры начнет выполняться, параметры процедуры *xp* и *y* хранятся по адресам со сдвигом, соответственно, 8 и 12 относительно адреса, хранящегося в регистре *%ebp*. Команды 1 и 2 переносят эти параметры в регистры *%eax* и *%edx*. Команда 3 выполняет разыменование *xp* и сохраняет это значение в регистре *%ecx*, соответствующем программному значению *x*. Команда 4 сохраняет *y* в *xp*. Команда 5 перемещает *x* в регистр *%eax*. По условию любой функция, возвращающая целое значение или значение указателя, помещает результат в регистр *%eax*, таким образом, указанная команда реализует строку 6 программного кода на языке C. Этот пример показывает, как команда *movl* может быть использована для считывания значений из памяти в регистр (команды 1—3), для записи в память из регистра (команда 4) и для копирования значений из одного регистра в другой (команда 5).

Две особенности программных кодов на языке ассемблера заслуживают особого внимания. Во-первых, мы видим: то, что в языке C мы называем "указателями", есть не что иное, как просто адреса. Разыменование указателя предусматривает размещение этого указателя в регистре с последующим использованием в косвенных ссылках на ячейки памяти. Во-вторых, локальные переменные, такие как *x*, часто содержатся в регистрах, а не в ячейках памяти. Доступ к регистрам осуществляется намного быстрее, чем к ячейкам памяти.

УПРАЖНЕНИЕ 3.2

Вы получаете следующую информацию. Функция с прототипом

```
void decode1(int *xp, int *yp, int *zp);
```

скомпилирована в код в языке ассемблера. Тело кода имеет вид:

```

1 movl 8(%ebp),%edi
2 movl 12(%ebp),%ebx
3 movl 16(%ebp),%esi
4 movl (%edi),%eax
5 movl (%ebx),%edx
6 movl (%esi),%ecx
7 movl %eax,(%ebx)
8 movl %edx,(%esi)
9 movl %ecx,(%edi)

```

Параметры *xp*, *yp* и *zp* хранятся в ячейках памяти со сдвигом, соответственно, 8, 12 и 16 относительно адреса, содержащегося в регистре *%ebp*.

Напишите коды в языке C для функции *decode1*, который вычислял бы те же результаты, что и код на языке ассемблера, приведенный выше. Вы можете протестировать полученный ответ, выполнив компиляцию написанного вами программного кода с параметром *-S*. Ваш компилятор должен породить код, который, возможно, будет

отличаться в использовании регистров и в упорядочении ссылок на ячейки памяти, но тем не менее он должен быть функционально эквивалентным исходному коду.

3.5. Арифметические и логические операции

В табл. 3.4 приводится список целочисленных операций над двойными словами, разбитых на четыре группы. Бинарные операции имеют два операнда, в то время как унарные операции — один operand. Описание этих операций производится в той же нотации, что и в разд. 3.4. За исключением `leal`, каждая из этих команд имеет свой эквивалент, который выполняет операции над словами (16 разрядов) и над байтами. Сuffix `l` заменяется на `w` для операций со словами и на `b` для операций над байтами. Например, команда `addl` становится `addw` или `addb`.

Таблица 3.4. Целочисленные арифметические операции

Команда	Результат	Описание
<code>leal S, D</code>	$D \leftarrow \&S$	Загрузить эффективный адрес
<code>incl D</code>	$D \leftarrow D + 1$	Увеличение на 1
<code>decl D</code>	$D \leftarrow D - 1$	Уменьшение на 1
<code>negl D</code>	$D \leftarrow -D$	Отрицание
<code>notl D</code>	$D \leftarrow \sim D$	Дополнение
<code>addl S, D</code>	$D \leftarrow D + S$	Сложение
<code>subl S, D</code>	$D \leftarrow D - S$	Вычитание
<code>imull S, D</code>	$D \leftarrow D * S$	Умножение
<code>xorl S, D</code>	$D \leftarrow D \wedge S$	Исключающее ИЛИ
<code>orl S, D</code>	$D \leftarrow D S$	ИЛИ
<code>addl S, D</code>	$D \leftarrow D \& S$	И
<code>sall k, D</code>	$D \leftarrow D \ll k$	Сдвиг влево
<code>shll k, D</code>	$D \leftarrow D \ll k$	Сдвиг влево (то же, что и <code>sall</code>)
<code>sarl k, D</code>	$D \leftarrow D \gg k$	Арифметический сдвиг вправо
<code>shrl k, D</code>	$D \leftarrow D \gg k$	Логический сдвиг вправо

Команда загрузки исполнительного адреса `leal` обычно используется для выполнения простых арифметических операций. Остальные операции — это ничем не примечательные стандартные унарные и бинарные операции. Обратите внимание, что упорядочение operandов посредством формата `GAS` не имеет интуитивной основы.

3.5.1. Команда загрузки исполнительного адреса

Команда загрузки исполнительного адреса `leal` фактически представляет собой вариант команды `movl`. Она имеет форму инструкции, которая производит считывание из памяти в регистр и в то же время она вообще не ссылается на память. Ее первый операнд имеет вид ссылки на ячейку памяти, однако, вместо считывания из указанной в команде ячейки, команда считывает исполнительный адрес по указанному адресу. Мы показываем в табл. 3.4 адресный оператор `&` языка C. Эта команда может быть употреблена для создания указателей на последние ссылки на ячейки памяти. Кроме того, она может быть использована для компактного описания обычных арифметических операций. Например, если регистр `%edx` содержит значение `x`, то команда

```
leal 7(%edx, %edx, 4), %eax
```

устанавливает значение регистра `%eax` равным $5x + 7$. Операндом назначения должен быть регистр.

УПРАЖНЕНИЕ 3.3

Предположим, что регистр `%eax` содержит значение `x`, а регистр `%ecx` содержит значение `y`. Заполните приведенную ниже таблицу формулами, указывающими значения, которые будут сохранены в регистре `%edx` для каждой из следующих команд программы на языке ассемблера.

Выражение	Результат
<code>leal 6(%eax), %edx</code>	
<code>leal (%eax,%ecx), %edx</code>	
<code>leal (%eax,%ecx,4), %edx</code>	
<code>leal 7(%eax,%eax,8), %edx</code>	
<code>leal 0xA(%ecx,%ecx,4), %edx</code>	
<code>leal 9(%eax,%ecx,2), %edx</code>	

3.5.2. Унарные и бинарные операции

Операции второй группы суть унарные операции, когда один и тот же операнд служит одновременно и источником, и местом назначения. Этим операндом может быть как регистр, так и ячейка памяти. Например, команда `incl (%esp)` вызывает увеличение значения элемента в вершине стека. Такой синтаксис напоминает операторы инкремента (`++` увеличение на 1) и декремента (`--` уменьшение на 1) языка C.

Третья группа состоит из бинарных операций, в которых второй операнд используется как источник и место назначения. Этот синтаксис напоминает операторы присваивания языка C, такие как `+=`. Однако заметьте, что операнд-источник указан первым,

а операнд назначения указан вторым. Это условие выглядит несколько странным для некоммутативных операций. Например, команда `subl %eax, %edx` уменьшает значение регистра `%eax` на величину, хранящуюся в регистре `%edx`. Первым операндом может быть непосредственное значение, регистр или ячейка памяти. Вторым операндом может быть регистр или ячейка памяти. Однако, как и в случае команды `movl`, оба операнда не могут быть ячейками памяти.

УПРАЖНЕНИЕ 3.4

Предположим, что по указанным адресам памяти и в указанных регистрах хранятся следующие значения:

Адрес	Значение
0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11

Регистр	Значение
<code>%eax</code>	0x100
<code>%ecx</code>	0x1
<code>%edx</code>	0x3

Заполните представленную далее таблицу и покажите результат выполнения указанных в ней команд через значения регистров и ячеек памяти, которые подвергаются обновлению, а также значение результата:

Команда	Назначение	Значение
<code>addl %ecx, (%eax)</code>		
<code>subl %edx, 4(%eax)</code>		
<code>imull \$16, (%eax, %edx, 4)</code>		
<code>incl 8(%eax)</code>		
<code>decl %ecx</code>		
<code>subl %edx, %eax</code>		

3.5.3. Операции сдвига

Завершающую группу операций составляют операции сдвига, в которых величина сдвига задается первой, а значение, подвергающееся сдвигу, задано вторым. Возможен арифметический и логический правый сдвиг. Величина сдвига задается одним байтом, поскольку допустим сдвиг в пределах от 0 до 31. Величина сдвига представлена непосредственно числом, либо однобайтовым регистровым элементом `%cl`. Как показано в табл. 3.4, команда сдвига влево выступает под двумя названиями: `sall` и `shll`. Обе они дают один и тот же результат, заполняя при этом разряды справа нулями. Команды сдвига вправо различаются тем, что команда `sall` выполняет арифметический сдвиг (заполнение производится копиями бита знака), в то время как команда `shrl` выполняет логический сдвиг (заполнение нулями).

УПРАЖНЕНИЕ 3.5

Предположим, что мы хотим разработать программу на языке ассемблера для следующей функции языка C:

```
int shift_left2_rightn(int, x, int n)
{
    x <<= 2;
    x >>= n;
    return x;
}
```

Приводимый далее код, который представляет собой фрагмент программы на языке ассемблера, именно этот фрагмент выполняет сдвиг и оставляет окончательное значение в регистре `%eax`. Две основные команды не показаны. Параметры `x` и `n` хранятся в ячейках памяти со сдвигом, соответственно, 8 и 12 относительно адреса в регистре `%ebp`.

1	<code>movl 12(%ebp), %ecx</code>	Получить <code>n</code>
2	<code>movl 8(%ebp), %eax</code>	Получить <code>x</code>
3	<hr/>	<code>x <<= 2</code>
4	<hr/>	<code>x >>= n</code>

Вставьте отсутствующие команды в соответствии с комментарием справа. Сдвиг вправо выполняется арифметически.

3.5.4. Обсуждение

За исключением операций правого сдвига, никакие команды не делают различий между операндами со знаком и операндами без знака. Арифметические операции над дополнениями на уровне разрядов выполняются так же, как и арифметические операции без знаков для всех команд, представленных в списке.

В листингах 3.12 и 3.13 представлен пример функции, которая выполняет арифметические операции и переводит их в код ассемблера. Как и раньше, здесь не показан фрагмент программы, в котором устанавливается стек, а также завершающий фрагмент программы. Аргументы `x`, `y` и `z` функции хранятся в памяти со сдвигом, соответственно, 8, 12 и 16 относительно адреса, помещенного в регистр `%ebp`.

Команда 3 реализует выражение $x+y$, считывая операнд `y` из регистра (который был загружен по команде 1), а другой операнд — непосредственно из памяти. Команды 4 и 5 выполняют вычисление $z \cdot 4^8$, при этом сначала выполняется команда `leal` с нормированной индексированной адресацией для вычисления выражения $(z+2z = 3z)$ с последующим сдвигом этого значения влево на четыре разряда с целью вычисления выражения $2^4 \cdot 3z = 48z$. Компилятор языка C часто генерирует различные комбинации команд сложения и сдвига с целью выполнения умножения на постоянный множитель, этот вопрос рассматривался в разд. 2.3.6. Команда 6 выполняет операцию `AND`, а инструкция 7 выполняет завершающую операцию умножения. Затем команда 8 записывает возвращаемое значение в регистр `%eax`.

Листинг 3.12. Программный арифметический код на языке C

```

1 int arith(int x,
2             int y,
3             int z)
4 {
5     int t1 = x+y;
6     int t2 = z*48;
7     int t3 = t1 & 0xFFFF;
8     int t4 = t2 * t3;
9
10    .
11    return t4;
12 }
```

Листинг 3.13. Программный арифметический код на языке ассемблера

1 movl 12(%ebp),%eax	Взять у
2 movl 16(%ebp),%edx	Взять z
3 addl 8(%ebp),%eax	Вычислить t1 = x+y
4 leal (%edx,%edx,2),%edx	Вычислить z*3
5 sall \$4,%edx	Вычислить t2 = z*48
6 andl \$65535,%eax	Вычислить t3 = t1&0xFFFF
7 imull %eax,%edx	Вычислить t4 = t2*t3
8 movl %edx,%eax	Установить t4 как возвращаемое значение

В коде на языке ассемблера (листинг 3.13) последовательность значений в регистре `%eax` соответствует значениям переменных программы `y`, `t1`, `t3` и `t4` (возвращаемое значение). В общем случае компиляторы генерируют код, который использует отдельные регистры для многих программных значений и который перемещает программные значения из одного регистра в другой.

УПРАЖНЕНИЕ 3.6

При проведении вычислений в цикле

```
for (i = 0; i<n; i++)
v += i;
```

мы сталкиваемся со следующей строкой в кодах ассемблера:

```
xorl eax, %edx
```

Объясните, почему эта команда находится в этом программном коде на С, ибо в этом случае отсутствует операция EXCLUSIVE-OR (исключающее ИЛИ). Какую операцию в программе на языке С эта команда реализует?

3.5.5. Специальные арифметические операции

В табл. 3.5 описаны команды, которые поддерживают генерацию полного 64-разрядного произведения двух 32-разрядных чисел, а также целочисленное деление.

Команда `imull`, включенная в список, представленный в табл.3.4, известна как бинарная команда умножения. Она выдает 32-разрядное произведение двух 32-разрядных чисел, реализуя операции $*_{w}$ и $*_{u}$, описанные в разд. 2.3.4 и 2.3.5. Напомним, что в процессе усечения этого произведения до 32 разрядов, как команда умножения без знака, так и команда умножения чисел в дополнительном двоичном коде на уровне разрядов ведут себя одинаково. В архитектуре IA32 предусмотрены две унарные команды умножения, позволяющие вычислять 64-разрядное произведение двух 32-разрядных значений, одна для умножения без знака (`mull`), а другая для умножения чисел в дополнительном двоичном коде (`imull`). В обеих этих операциях один аргумент должен находиться в регистре `%eax`, а другой представлен как operand-источник соответствующей команды. Произведение сохраняется в регистрах `%edx` (32 старших разряда) и `%eax` (32 младших разряда). Обратите внимание на тот факт, что хотя одно и то же имя используется для обозначения двух различных операций, ассемблер может различить, какая из них имеется в виду, просчитав число операндов.

Таблица 3.5. Специальные арифметические операции

Команда	Результат	Описание
<code>imull S</code>	$R[\%edx] : R[\%eax] \leftarrow S \times R[\%eax]$	Полное умножение со знаком
<code>mull S</code>	$R[\%edx] : R[\%eax] \leftarrow S \times R[\%eax]$	Полное умножение без знака
<code>cltd S</code>	$R[\%edx] : R[\%eax] \leftarrow \text{SignExtend}(R[\%eax])$	Преобразование в четверное слово
<code>idivl S</code>	$R[\%edx] \leftarrow R[\%edx] : R[\%eax] \bmod S$ $R[\%edx] \leftarrow R[\%edx] : R[\%eax] + S$	Деление со знаком
<code>divl S</code>	$R[\%edx] \leftarrow R[\%edx] : R[\%eax] \bmod S$ $R[\%edx] \leftarrow R[\%edx] : R[\%eax] \div S$	Деление без знака

Эти операции обеспечивают выполнение полного 64-разрядного умножения и деления как для чисел со знаком, так и для чисел без знака. Полученное при умножении 64-разрядное квадратичное слово содержится в паре регистров `%eax` и `%edx`.

В качестве примера предположим, что мы зафиксировали числа x и y в позициях, с смещением 8 и 12 относительно адреса в регистре `%ebp`, а мы хотим сохранить их 64-разрядное произведение в 8 байтах в вершине стека. Эту задачу выполняет следующий программный код (листинг 3.14).

Листинг 3.14. Сохранение произведения

```

адрес x есть %ebp+8, адрес у есть %ebp+12
1  movl    8(%ebp),%eax    Поместить x в %eax
2  imull   12(%ebp)       Умножить на у
3  pushl   %edx           Затолкнуть в стек старшие 32 разряда
4  pushl   %eax           Затолкнуть в стек старшие 32 разряда

```

Заметьте, что порядок, в котором мы проталкиваем данные в стек, правильный для остроконечных машин: стек возрастает в направлении уменьшения адресов (т. е. младшие байты произведения имеют меньшие адреса, чем старшие байты).

Рассмотренный нами выше список арифметических операций (табл. 3.4) не содержит никаких-либо операций деления или операций по модулю. Выполнение этих операций обеспечивается унарными командами деления, подобными унарным командам умножения. Операция деления *idivl* чисел со знаком принимает в качестве делимого 64-разрядную величину из регистров *%edx* (32 старших разряда) и *%eax* (32 младших разряда). Делитель задается в качестве операнда этой команды. Эти инструкции сохраняют частное в регистре *%eax*, а остаток — в регистре *%edx*. Команда *cltd* может быть использована для формирования 64-разрядного делимого из 32-разрядного значения, хранящегося в регистре *%eax*. Знак этой команды расширяет регистр *%eax* в *%edx*.

В качестве примера предположим, что мы имеем числа *x* и *y* со знаком в позициях со смещением, соответственно, 8 и 12 относительно *%ebp*, и мы хотим сохранить значения *x/y* и *x%y* в стеке. Эту задачу выполняет следующий программный код (листинг 3.15):

Листинг 3.15. Операция деления

```

Адрес x есть %ebp+8, адрес у есть %ebp+12
1  movl 8(%ebp),%eax      Поместить x в %eax
2  cltd                   Расширение знака в %edx
3  idivl 12(%ebp)         Разделить на у
4  pushl %eax             Протолкнуть в стек x / у
5  pushl %edx             Протолкнуть в стек x % у

```

Команда выполняет деление без знака. Обычно предварительно в регистр *%edx* записывается 0.

3.6. Управление

До этого момента мы рассматривали способы доступа к данным и манипулирования ими. Другой не менее важной частью программы является управление последовательностью выполняемых операций. По умолчанию для операторов языка C, равно

как в коде ассемблера, принимается последовательная логика, когда команды или операторы выполняются в том порядке, в каком они появляются в программе. Некоторые конструкции языка C, такие как условные операторы, циклы и переключатели, выбирают непоследовательный порядок выполнения, в условиях которой точная последовательность зависит от значений некоторых переменных программы.

Программа на языке ассемблера предоставляет механизмы нижнего уровня для реализации непоследовательной логики управления. Основной операцией является переход на различные части программы, возможно, в зависимости от результатов тех или иных проверок. Компилятор должен генерировать последовательности команд, построенные на базе этих механизмов, предназначенных для реализации управляющих структур языка C.

В этой книге мы сначала рассмотрим механизмы машинного уровня, а затем покажем, как различные управляющие структуры языка C реализуются с их помощью.

3.6.1. Коды управления

Наряду с целочисленными регистрами центральный процессор поддерживает некоторый набор одноразрядных регистров **кодов условий** (*condition code*), описывающих атрибуты последних арифметических и логических операций. Эти регистры затем проверяются с целью выполнения условных переходов. Наиболее часто используемыми условиями являются:

- CF — признак переполнения. Последняя операция породила перенос самого старшего двоичного разряда. Используется для того, чтобы обнаружить переполнение при выполнении операций над числами без знака;
- FZ — признак нуля. Последняя операция, в результате выполнения которой получен ноль;
- SF — признак знака. Последняя операция, в результате выполнения которой получена отрицательная величина.
- OF — признак переполнения. Последняя операция, вызвавшая переполнение дополнения до двух — как положительное, так и отрицательное.

В качестве примера предположим, что мы использовали операцию сложения addl для вычисления выражения, эквивалентного выражению $t = a+b$ в C, где переменные t, a и b имеют тип int. Коды условия будут установлены в соответствии со следующими выражениями в C:

CF: (число без знака a) < (число без знака b) Переполнение без знака

FZ: (t == 0) Ноль

SF: (t<0) Отрицательное значение

OF: (a<0 == b<0) && (t<0 != a<0) Знаковое переполнение

Команда leal не меняет никаких кодов условия, поскольку она предназначена для вычисления адресов. С другой стороны, все команды, приведенные в табл. 3.4, устанавливают коды условий. Для логических операций, таких как xorl, признак перено-

са устанавливается равным 0. Что касается операций сдвига, признак переноса устанавливается равным значению последнего вытесненного разряда, в то время как признак переполнения устанавливается равным 0.

В дополнение к операциям, представленным в табл. 3.4, далее в табл. 3.6 показаны две операции (имеющие 8, 16 и 32-разрядные форматы), которые устанавливают коды условий, не изменяя значений других регистров:

Таблица 3.6. Две операции

Команда	Основана на	Описание
cmpb	S2,SI	$SI - S2$ Сравнить байты
testb	S2,SI	$SI \& S2$ Проверить байт
cmpw	S2,SI	$SI - S2$ Сравнить слова
testw	S2,SI	$SI \& S2$ Проверить слово
cmpl	S2,SI	$SI - S2$ Сравнить двойные слова
testl	S2,SI	$SI \& S2$ Проверить двойное слово

Операции cmpb, cmpw и cmpl устанавливают коды условий в соответствии с разницей обоих их операндов. В формате GAS операнды указаны в обратном порядке, из-за чего программные коды трудно читаются. Эти команды устанавливают признак нуля, если операнды равны. Другие признаки могут быть использованы для определения отношения порядка на двух операндах.

Команды testb, testw и testl устанавливают признаки нуля и признаки отрицательного значения по результату операции AND над двумя операндами. Обычно один и тот же операнд повторяется, например

`testl %eax, %eax`

чтобы узнать, является ли %eax отрицательной, нулев или положительной величиной, либо один из операндов является маской, показывающей, какой разряд должен быть подвергнут проверке.

3.6.2. Доступ к кодам условия

Вместо того чтобы считывать коды условия напрямую, два наиболее широко распространенных метода доступа к ним заключаются в том, что устанавливаются целочисленные регистры или выполняется условный переход на основании некоторого сочетания кодов условия. Различные команды set, представленные в табл. 3.7, устанавливают некоторый разряд в 0 или 1 в зависимости от того или иного сочетания кодов условий. Операнд назначения есть либо один из восьми однобайтовых регистровых элементов (см. рис. 3.2), либо ячейка памяти, предназначенная для хранения одного байта. Чтобы получить 32-разрядный результат, мы должны очистить 24 старших разряда.

Таблица 3.7. Команды установки

Команда	Синоним	Результат	Условие
sete D	setz	$D \leftarrow ZF$	Равно / ноль
setne D	setnz	$D \leftarrow \sim ZF$	Не равно / ноль
sets D		$D \leftarrow SF$	Отрицательное
setns D		$D \leftarrow \sim SF$	Неотрицательное
setg D	setnle	$D \leftarrow \sim (SF \wedge OF) \wedge \sim ZF$	Больше (знак >)
setge D	setnl	$D \leftarrow \sim (SF \wedge OF)$	Больше или равно (знак \geq)
setg D	setnge	$D \leftarrow SF \wedge OF$	Меньше (знак <)
setge D	setng	$D \leftarrow (SF \wedge OF) \mid ZF$	Меньше или равно (знак \leq)
seta D	setnbe	$D \leftarrow CF \wedge \sim ZF$	Выше (без знака >)
setae D	setnb	$D \leftarrow CF$	Выше или равно (без знака \geq)
setb D	setnae	$D \leftarrow \sim CF$	Ниже (без знака <)
setbe D	setna	$D \leftarrow CF \mid ZF$	Ниже или равно (без знака \leq)

Каждая такая команда устанавливает один байт в 0 или 1 в зависимости от некоторой комбинации кодов условий. У некоторых команд имеются "синонимы", т. е. альтернативные имена для одних и тех же машинных команд.

Таким образом, типичная последовательность для предиката языка С (такого как, например, `a < b`) имеет вид, как в листинге 3.16:

Листинг 3.18. Предикат С

Примечание: а находится в `%edx`, б находится в `%eax`

```

1  cmpl    %eax,%edx Сравнить а:б
2  setl    %al    установить младший байт регистра %eax в 0 или 1
3  movzb1 %al,%eax установить остальные байты регистра %eax в 0

```

Команда `movzb1` используется для очистки трех старших байтов.

Для некоторых базовых машинных команд может быть несколько возможных имен, которые мы называем синонимами. Например, как имя `setg` (что означает "установить больше"), так и имя `setngle` (что означает "установить не меньше или равно") относится к одной и той же машинной команде. Компиляторы и обратные ассемблеры по собственному усмотрению выбирают, какие имена следует использовать.

И хотя все арифметические операции устанавливают коды условий, описание различных команд `set` относится к случаю, когда выполняются команды сравнения,

устанавливая при этом коды условий в соответствии с вычислением $t = a - b$. Например, рассмотрим команду `sete` ("установить, если равно"). Если $a = b$, имеем $t = 0$, следовательно, признак нуля означает равенство.

Аналогично, рассмотрим проверку сравнения чисел со знаком, выполняемую командой `setle` ("установить, если меньше"). Если и a и b представлены в виде дополнения до двух, то для $a < b$ имеем $a - b < 0$ в случае, когда разность этих двух значений вычислена правильно. Если переполнения нет, то этот факт будет обозначен установленным признаком знака. Однако имеет место положительное переполнение, поскольку $a - b$ есть большое положительное число, мы получим $t < 0$. Если имеет место отрицательное переполнение в силу того, что $a - b$ есть небольшое отрицательное число, мы имеем $t > 0$. В любом случае, признак знака принимает значение, противоположное значению истинной разности. Следовательно, операция EXCLUSIVE-OR (исключительное ИЛИ) разрядов переполнения и разрядов знака требуют проверки условия $a < b$. Другие тесты на проверку равенства чисел со знаками основаны на других комбинациях признаков SF^OF и ZF.

Для проверки сравнения чисел без знаков признак переноса устанавливается командой `cmpl` в тех случаях, когда целочисленная разность $a - b$ аргументов без знака a и b отрицательна, т. е. когда a (без знака) $<$ b (без знака). Таким образом, эти проверки используют сочетание признаков переноса и признаков нуля.

УПРАЖНЕНИЕ 3.7

В приводимом далее программном коде С мы заменили некоторые операторы сравнения на "___" и опустили типы данных в приведениях типов.

```

1  char ctest(int a, int b, int c)
2  {
3      char t1 =      a ___      b;
4      char t2 =      b ___ ( )      a;
5      char t3 = ( )      c ___ ( )      a;
6      char t4 = ( )      a ___ ( )      c;
7      char t5 =      c ___      b;
8      char t6 =      a ___      0;
9      return t1 + t2 + t3 + t4 + t5 + t6;
10 }
```

Для исходного кода на языке С компилятор генерирует следующий код на языке ассемблера:

1	movl	8(%ebp), %ecx	Взять a
2	movl	12(%ebp), %esi	Взять b
3	cmpl	%esi, %ecx	
4	setl	%al	Вычисление t1
5	cmpl	%ecx, %esi	Сравнение b:a
6	setb	-1(%ebp)	Вычисление t2
7	cmpw	%cx, 16(%ebp)	Сравнение c:a

```

8  setge  -2(%ebp)      Вычисление t3
9  movb   %cl,%dl
10  cmph  16(%ebp),%dl  Сравнение a:c
11  setne  %bl           Вычисление t4
12  cmpl  %esi,16(%ebp) Сравнение c:b
13  setg   -3(%ebp)      Вычисление t5
14  testl  %ecx,%ecx    Проверка a
15  setg   %dl           Вычисление t6
16  addb   -1(%ebp),%al  Прибавить t2 к t1
17  addb   -2(%ebp),%al  Прибавить t3 к t1
18  addb   %bl,%al       Прибавить t4 к t1
19  addb   -3(%ebp),%al  Прибавить t5 к t1
20  addb   %dl,%al       Прибавить t6 к t1
21  movsbl %al,%eax     Перевести sum из типа char в тип int

```

На основании этого кода на языке ассемблера заполните недостающие части (сравнения и преобразования типов) в коде на С.

3.6.3. Команды перехода и их кодирование

В условиях стандартного выполнения команды следуют одна за другой в том порядке, в каком они записаны. Команда *перехода* (jump) может привести к тому, что управление будет передано в совершенно новую позицию программы (табл. 3.8). Места, куда переходит управление, в общем случае указаны *метками*. Рассмотрим следующую последовательность кодов на языке ассемблера (листинг 3.17):

Листинг 3.17. Метки

```

1  xorl  %eax,%eax      Установить регистр %eax в 0
2  jmp   .L1              Перейти на метку .L1
3  movl  (%eax),%edx     Разыменование нулевого указателя
4  .L1:
5  popl  %edx

```

Команда `jmp.L1` приводит к тому, что управление пропускает команду `movl`, и вместо нее возобновляет выполнение программы с команды `popl`. В процессе формирования файла объектного кода ассемблер определяет адреса всех помеченных команд и кодирует *адреса перехода* (jump destination) (адреса команд назначения) как часть команды перехода.

Команда `jmp` осуществляет безусловный переход. Переход может быть *прямой* (direct), когда адрес перехода указан в самой программе, или *непрямой* (indirect), когда адрес перехода считывается из одного из регистров или из ячейки памяти. Непосредственные переходы записываются в языке ассемблера путем указания метки как адреса перехода, т. е. метки `L1` в листинге 3.17. Непрямые переходы отмечены звездочками, за которыми следует описатель операнда, использующий тот же синтаксис,

в каком записана команда `movl`. Например, команда `jmp *%eax` использует значение, хранящееся в регистре `%eax`, в качестве адреса перехода, а команда `jmp *(%eax)` считывает адрес перехода из памяти, используя значение, хранящееся в регистре `%eax` как адрес считываения.

Таблица 3.8. Команды перехода

Команда	Синоним	Результат	Условие
<code>jmp</code> Метка		1	Прямой переход
<code>jmp</code> *Операнд		1	Косвенный переход
<code>je</code> Метка	<code>jz</code>	<code>ZF</code>	Равно / ноль
<code>jne</code> Метка	<code>jnz</code>	<code>~ZF</code>	Не равно / не ноль
<code>js</code> Метка		<code>SF</code>	Отрицательный
<code>jns</code> Метка		<code>~SF</code>	Неотрицательный
<code>jg</code> Метка	<code>jnle</code>	<code>~(SF^OF) & ~ZF</code>	Больше (знак >)
<code>jge</code> Метка	<code>jnl</code>	<code>~(SF^OF)</code>	Больше или равно (знак >=)
<code>jl</code> Метка	<code>jnge</code>	<code>SF^OF</code>	Меньше (знак <)
<code>jle</code> Метка	<code>jng</code>	<code>~(SF^OF) ZF</code>	Меньше или равно (знак <=)
<code>ja</code> Метка	<code>jnb</code>	<code>~CF & ~ZF</code>	Выше (без знака >)
<code>jae</code> Метка	<code>jnb</code>	<code>~CF</code>	Выше или равно (без знака >=)
<code>jb</code> Метка	<code>jnae</code>	<code>CF</code>	Ниже (без знака <)
<code>jbe</code> Метка	<code>jna</code>	<code>CF ZF</code>	Ниже или равно (без знака <=)

Другие команды перехода либо осуществляют переход, либо переходят к выполнению следующей команды в последовательности кодов в зависимости от некоторых комбинаций кодов условий. Обратите внимание на то обстоятельство, что имена этих команд и условия, при выполнении которых осуществляется переход, соответствуют условиям команд `set`. Как и в случае команд `set`, некоторые из базовых машинных команд имеют несколько имен. Условный переход может быть только прямым.

Несмотря на то, что мы не ставим перед собой задачу подробного анализа форматов объектного кода, понимание того, как кодируются адреса команд перехода очень важно для изучения редактирования связей в главе 7. Кроме того, оно полезно при интерпретации выходных данных обратного ассемблера. В программах, написанных в кодах ассемблера, адреса переходов записаны в виде символьных меток. Ассемблер, а позже и редактор связей, генерирует соответствующие обозначения адресов переходов. Существуют несколько различных способов обозначений переходов, но чаще других используется *относительная адресация по счетчику команд* (PC-relative). То есть кодируется разность между адресом команды назначения и адресом команды, которая непосредственно следует за командой перехода. Подобного рода смещения могут быть закодированы с использованием одного, двух или четырех

байтов. Второй метод кодирования заключается в том, чтобы указать "абсолютный" адрес, используя четыре байта для указания точного адреса перехода. Ассемблер и редактор связей сами выбирают соответствующие способы кодирования адресов переходов.

В качестве примера относительной адресации по счетчику команд рассмотрим фрагмент программных кодов (листинг 3.18) на языке ассемблера, который был получен при компилировании файла silly.c. В нем имеют место два перехода: команда `jle` в строке 1 совершает переход вперед по большему адресу, в то время как команда `jg` в строке 8 возвращается по меньшему адресу.

Листинг 3.18. Относительная адресация

```

1 jle .L4           If <=, goto dest2
2 .p2align 4,,7     Выравнивает следующую команду по адресу, кратному 8
3 .L5:             dest1:
4 movl %edx,%eax
5 sarl $1,%eax
6 subl %eax,%edx
7 testl %edx,%edx
8 jg .L5           If >, goto dest1
9 .L4:             dest2:
10 movl %edx,%eax

```

Обратите внимание на то, что в строке 2 находится директива ассемблеру, в соответствии с которой следующая команда должна начинаться с адреса, кратного 16, при этом остаются не использованными, по меньшей мере, 7 байтов. Эта директива применяется с тем, чтобы процессор мог оптимально использовать кэш-память для временного хранения команд.

Версия формата .o, построенного ассемблером, после обработки обратным ассемблером принимает следующий вид (листинг 3.19):

Листинг 3.19. Обработка ассемблером

```

1 8: 7e 11           jle  1b <silly+0x1b> Адрес перехода = dest2
2 a: 8d b6 00 00 00 00 lea   0x0(%esi),%esi Добавлены команды nops
3 10: 89 d0          mov   %edx,%eax      dest1:
4 12: c1 f8 01        sar   $0x1,%eax
5 15: 29 c2          sub   %eax,%edx
6 17: 85 d2          test  %edx,%edx
7 19: 7f f5          jg    10 <silly+0x10> Адрес перехода = dest1
8 1b: 89 d0          mov   %edx,%eax      dest2:

```

Команда `lea 0x0 (%esi),%esi` в строке 2 не производит никаких действий. Она используется как 6-байтовая команда ноп с тем, чтобы следующая команда (строка 3) имела начальный адрес, кратный 16.

В комментарии, который оставляет обратный ассемблер справа, адреса переходов указаны непосредственно как 0x1b для команды 1 и как 0x10 для команды 7. Однако, проанализировав байтовое представление команд, замечаем, что адрес перехода в команде 1 представлен (во втором байте) как 0x11 (десятичное 17). Складывая этот адрес с 0xa (десятичное 10), представляющим адрес следующей команды, получаем адрес перехода 0x1b (десятичное 27), т. е. адрес команды 8.

Аналогично, адрес команды перехода 7 представлен как 0xf5 (десятичное 11) одним байтом в дополнительном коде. Добавляя эту величину к 0x1b (десятичное 27), т. е. к адресу инструкции 8, получаем 0x10 (десятичное 16), адрес команды 3.

Как показывают эти примеры, значение счетчика команд при относительной адресации есть адрес команды, следующей за командой перехода, но не самой команды перехода. Это соглашение относится еще к тем временам, когда процессор обновлял значение счетчика команд как первый этап выполнения конкретной команды.

В листинге 3.20 показана версия обратного ассемблера программы после редактирования связей.

Листинг 3.20. Обратный ассемблер

1	80483c8:	7e 11	jle	80483db <silly+0x1b>
2	80483ca:	bd b6 00 00 00 00	lea	0x0(%esi),%esi
3	80483d0:	89 d0	mov	%edx,%eax
4	80483d2:	c1 f8 01	sar	\$0x1,%eax
5	80483d5:	29 c2	sub	%eax,%edx
6	80483d7:	85 d2	test	%edx,%edx
7	80483d9:	7f f5	jg	80483d0 <silly+0x10>
8	80483db:	89 d0	mov	%edx,%eax

Представленные здесь команды перемещены по различным адресам, однако коды адресов переходов в строках 1 и 7 остаются неизменными. Использование относительной адресации переходов по счетчику команд позволяет получать компактные коды команд (для чего требуется всего лишь два байта), а объектный код можно сдвигать в памяти на различные позиции без изменений.

УПРАЖНЕНИЕ 3.8

В следующих фрагментах двоичного кода, порожденного обратным ассемблером, некоторые данные заменены на x. Ответьте на следующие вопросы, касающиеся команд перехода:

1. Какова цель команды jmp?

8048d1c:	76 da	jbe	xxxxxx
8048d1e:	eb 24	jmp	8048d44

2. Каков адрес команды mov?

xxxxxx:	eb 54	jmp	8048d44
xxxxxx:	c7 44 f8 10 00	mov	\$0x10,0xfffffffff8(%ebp)

3. В приводимом далее программном коде адрес перехода представлен в относительной адресации по счетчику команд в 4-байтовом коде. Байты располагаются в порядке от менее значащего к более значащему, отображая упорядочение байтов, принятые архитектурой IA32. Каким является адрес перехода?

```
8048902: e9 cb 00 00 00      jmp      xxxxxx
8048907: 90                  por
```

4. Объясните, какая связь существует между комментарием справа и кодированием байтов слева. Обе строки являются частью программного кода команды jmp.

```
80483f2: ff 25 e0 a2 04      jmp      *0x804a2e0
8048907: 80
```

Чтобы реализовать управляющие конструкции языка C, компилятор должен использовать различные типы команд перехода, которые мы только что рассмотрели. Далее мы ознакомимся с наиболее употребительными конструкциями, начнем с простых условных переходов, а затем рассмотрим операторы циклов и операторы выбора.

3.6.4. Трансляция условных переходов

Условные операторы в языке C реализуются с использованием различных комбинаций условных и безусловных переходов. Например, в листинге 3.21 показаны программные коды C функции, которая вычисляет абсолютное значение разности двух чисел. Компилятор GCC генерирует код на языке ассемблера, представленный в листинге 3.23. Мы построили версию этой функции под именем `absdiff` (листинг 3.22), которая более строго соблюдает требования управляющей логики этой программы на языке ассемблера. Она использует оператор `goto` языка C, который во многом совпадает с безусловным переходом в языке ассемблера. Оператор `goto less` в строке 6 вызывает переход по метке `less` в строке 9, пропуская оператор, помещенный в строке 7. Обратите внимание на то обстоятельство, что использование операторов `goto` в программировании является дурным тоном, поскольку может создать трудности при чтении программы и при ее отладке. Мы используем ее в своих построениях как способ создания программ на языке C, которые описывают управляющую логику программ на языке ассемблера. Мы будем называть такие программы на C "программами `goto`".

Листинг 3.21. Разность чисел

```
1 int absdiff(int x, int y)
2 {
3     if (x < y)
4         return y - x;
5     else
6         return x - y;
7 }
```

Листинг 3.22. Эквивалентная версия команды

```

1 int gotodiff(int x, int y)
2 {
3     int rval;
4
5     if (x < y)
6         goto less;
7     rval = x - y;
8     goto done;
9 less:
10    rval = y - x;
11 done:
12    return rval;
13 }
```

Листинг 3.23. Ассемблерная версия

1	movl	8(%ebp), %edx	Взять .x
2	movl	12(%ebp), %eax	Взять .y
3	cmpl	%eax, %edx	Сравнение x:y
4	jl	.L3	If <, goto less
5	subl	%eax, %edx	Вычислить x-y
6	movl	%edx, %eax	Установить как возвращаемое значение
7	jmp	.L5	Goto done
8	.	.L3:	less:
9	subl	%edx, %eax	Вычислить y-x как возвращаемое значение
10	.	.L5:	done: Начать выполнения завершающего кода

Процедура `absdiff(a)`, написанная на языке С, содержит оператор `if-else`. Показан построенный код в языке ассемблера, наряду с процедурой `gotodiff` на С, которая воспроизводит управляющую логику этого кода на языке ассемблера. Здесь мы опускаем части программы в ассемблерных кодах на языке, которые устанавливают стек и завершают программу.

Реализация в кодах ассемблера сначала сравнивает два операнда (строка 3), устанавливая коды условий. Если результат сравнения показывает, что x меньше y , то управление переходит к блоку кодов, которые вычисляют разность $y-x$ (строка 9). В противном случае выполняются коды, которые вычисляют разность $x-y$ (строки 5 и 6). В обоих случаях вычисленный результат сохраняется в регистре `%eax`, а программа завершается строкой 10, в которой она выполняет код завершения стека (здесь не показан).

Общая форма оператора `if-else` (если...то) в языке С задается шаблоном

```

if (test-expr)
  then-operator
else
  else-operator
```

где *test-expr* есть целочисленное выражение, которое либо принимает значение 0 (интерпретируется как "ложно") или ненулевое значение (интерпретируется как "истино"). Выполняется только один из двух операторов перехода (*then-оператор* или *else-оператор*).

Для случая этой общей формы реализация в языке ассемблера придерживается следующей формы, в которой мы используем синтаксис языка C для описания управляющей логики программы:

```
t = test-expr;
if (t)
    goto true;
else-оператор
    goto done;
true:
then-оператор
done:
```

То есть компилятор генерирует отдельные блоки кода для *else-оператора* и *then-оператора*. Он вставляет условный переход и безусловный переход с тем, чтобы гарантировать выполнение нужного блока.

УПРАЖНЕНИЕ 3.9

Если на языке C заданы программные коды

```
1 void cond(int a, int *p)
2 {
3     if (p && a > 0)
4         *p += a;
5 }
```

то компилятор генерирует следующие коды на языке ассемблера:

```
1      movl 8(%ebp), %edx
2      movl 12(%ebp), %eax
3      testl %eax, %eax
4      je .L3
5      testl %edx, %edx
6      jle .L3
7      addl %edx, (%eax)
8 .L3:
```

1. Напишите версию команды *goto* в C, которая выполняет те же вычисления и воспроизводит управляющую логику программы в кодах ассемблера в стиле, показанном в листинге 3.22. Возможно, вы посчитаете полезным снабдить эти программные коды комментариями, подобными тем, которые мы делали в наших примерах.

2. Объясните, почему коды в языке ассемблера содержат два условных перехода несмотря на то, что коды в языке С содержат всего лишь один оператор `if`.

3.6.5. Циклы

Язык С предлагает пользователям несколько конструкций циклов, а именно `while`, `for` и `do-while`. Соответствующих им конструкций в языке ассемблера нет. Вместо них используются различные комбинации проверок условий и переходов, обеспечивающие эффект цикла. Интересно отметить, что большинство генерируют коды циклов, беря за основу тип цикла `do-while`, несмотря на то, что этот тип цикла довольно редко встречается в фактических программах. Другие циклы преобразуются в форму `do-while`, после чего компилируются в машинные коды. Мы будем изучать преобразование циклов, рассматривая их как некоторую последовательность, начиная с цикла `do-while`, а затем продвигаться к более сложным циклам с более сложной реализацией.

Циклы `do-while`

Общая форма цикла `do-while` имеет следующий вид:

`do`

body-оператор
while (test-expr);

Назначение цикла состоит в том, чтобы многократно выполнять *body-оператор* (тело цикла), вычислять значение *test-expr* (условие продолжения цикла) и продолжать цикл, если результат вычисления не равен нулю. Обратите внимание на тот факт, что тело цикла выполняется, по меньшей мере, один раз.

Обычно реализация цикла имеет следующую общую форму:

`loop:`

```
body-оператор  
t = test-expr;  
if (t)  
    goto loop;
```

Например, в листинге 3.24 показана реализация стандартной программы для вычисления *n*-го элемента последовательности чисел Фибоначчи с использованием цикла `do-while`. Эта последовательность определяется следующим рекуррентным соотношением:

$$F_1 = 1$$

$$F_2 = 1$$

$$F_n = F_{n-1} + F_{n-2}, n \geq 3$$

Например, первыми десятью элементами последовательности являются числа 1, 1, 2, 3, 5, 8, 13, 34 и 55. Чтобы получить эту последовательность в цикле, мы начинаем ее вычисление со значений $F_0 = 0$ и $F_1 = 1$, а не с F_1 и $F_2 = 1$.

Листинг 3.24. Нисса Фибоначчи

```

1 int fib_dw(int n)
2 {
3     int i = 0;
4     int val = 0;
5     int nval = 1;
6
7     do {
8         int t = val + nval;
9         val = nval;
10        nval = t;
11        i++;
12    } while (i < n);
13
14    return val;
15 }
```

В листинге 3.25 приводится программный код на языке ассемблера, реализующий цикл, в табл. 3.9 показано соответствие между регистрами и программными значениями. В условиях этого примера тело цикла охватывает строки с 8 до 11, в которых осуществляется присваивание значений переменным t , val и $nval$, а также увеличивается значение переменной цикла i . Эти операции выполняются в строках 2—5 кода ассемблера. Выражение $i < n$ представляет собой условие продолжения цикла (*test Expr*). Оно реализуется строкой 6 и тестовым условием в команде перехода в строке 7. Если цикл существует, то значение val копируется в регистр как возвращаемая величина (строка 8).

Листинг 3.25. Ассемблер программы вычисления чисел Фибоначчи

1 .L6:	loop:
2 leal (%edx,%ebx),%eax	Вычислить $t = val + nval$
3 movl %edx,%ebx	Скопируйте $nval$ в val
4 movl %eax,%edx	Скопируйте t к $nval$
5 incl %ecx	Увеличение на i
6 cmpl %esi,%ecx	Сравнение $i:n$
7 jl .L6	If less, goto loop
8 movl %ebx,%eax	Установить val как возвращаемое значение

Создание таблицы использования регистров, подобной табл. 3.9, может оказаться очень полезным шагом при проведении анализа программы на языке ассемблера, особенно в тех случаях, когда в ней используются циклы.

Таблица 3.9. Использование регистра

Регистр	Переменная	Начальное значение
%ecx	i	0
%esi	n	n
%ebx	val	0
%edx	nval	1
%eax	t	-

УПРАЖНЕНИЕ 3.10

Для программы на С

```

1 int dw_loop(int x, int y, int n)
2 {
3     do {
4         x += n;
5         y *= n;
6         n--;
7     } while ((n > 0) & (y < n));
8     return x;
9 }
```

компилятор GCC генерирует следующий код на языке ассемблера:

Сначала x, y, и n смешены на 8, 12 и 16 относительно %ebp

```

1 movl 8(%ebp),%esi
2 movl 12(%ebp),%ebx
3 movl 16(%ebp),%ecx
4 .p2align 4,,7 Используется для оптимизации характеристик кэш
5 .L6:
6 imull %ecx,%ebx
7 addl %ecx,%esi
8 decl %ecx
9 testl %ecx,%ecx
10 setg %al
11 cmpl %ecx,%ebx
12 setl %dl
13 andl %edx,%eax
14 testb $1,%al
15 jne .L6
```

- Постройте таблицу использования регистров, подобную табл. 3.9.
- Выделите *test-expr* и тело оператора, соответствующие им строки в программном коде на языке ассемблера.

3. Введите комментарий в программный код на языке ассемблера, описывающий операции, выполняемые программой, аналогично тому, как это сделано в листинге 3.25.

Циклы *while*

Общая форма цикла *while* имеет следующий вид:

```
while (test-expr)
    body-оператор
```

Он отличается от цикла *do-while* тем, что сначала вычисляется условие продолжения цикла (*test-expr*), при этом существует возможность прекращения цикла, прежде чем наступит очередь выполнения тела цикла (*body-оператор*). Прямой перевод в форму, использующий команду *goto*, придает ему вид:

```
loop
    t = test-expr;
    if (!t)
        goto done;
    body-оператор
    goto loop;
done:
```

Такое преобразование требует наличия двух управляющих операторов во внутреннем цикле, т. е. в той части программного кода, которая выполняется чаще других. Однако большинство компиляторов языка преобразуют этот код в цикл *do-while* с использованием условного перехода, который при необходимости пропускает первое выполнение тела цикла:

```
if (!test-expr)
    goto done;
do
    body-оператор
    while (test-expr);
done:
```

Эти коды, в свою очередь, могут быть преобразованы в коды с операцией *goto* следующим образом:

```
t = test-expr;
if (!t)
    goto done;
loop:
body-оператор
t = test-expr;
```

```

if (t)
    goto loop;
done:

```

В качестве примера покажем реализацию вычисления последовательности чисел Фибоначчи с использованием цикла (листинг 3.26). Обратите внимание на тот факт, что на этот раз мы начали рекурсию с элементов F_1 (`val`) и F_2 (`nval`). Изображенная рядом функция `fib_w_goto`, написанная на языке С, показывает, как этот код транслируется в язык ассемблера. Код на языке ассемблера (листинг 3.28) почти полностью соответствует программным кодам на С функции `fib_w_goto`. Компилятор выполнил несколько интересных операций оптимизации, в чем можно убедиться, ознакомившись с программным кодом операции `goto` в листинге 3.27. Во-первых, вместо того, чтобы использовать `i` в качестве переменной цикла и сравнивать ее значение с `n` на каждой итерации, компилятор ввел новую переменную цикла, которую мы будем называть `nmi`, поскольку по отношению к исходному коду ее значение равно $n - i$. Это позволяет компилятору использовать только три регистра для переменных цикла, в то время как в других случаях таких переменных четыре. Во-вторых, он оптимизировал условие для первоначальной проверки, заменив (`i < n`) на (`val < n`), поскольку исходные значения обеих переменных равны 1. Благодаря этим действиям, компилятор совсем не использует переменную `i`. Часто компилятор использует исходные значения переменных для оптимизации начальной проверки условия цикла. Это существенно усложняет расшифровку кода на языке ассемблера. В-третьих, для успешного выполнения цикла нужно, чтобы $n \leq i$, благодаря чему компилятор делает вывод, что значение переменной `nmi` положительно. В результате компилятор может проверять условие продолжения цикла, руководствуясь условием `nmi != 0`, а не `nmi >= 0`. Это позволяет сэкономить одну команду кода на языке ассемблера.

Листинг 3.26. Вычисление с использованием цикла

```

1 int fib_w(int n)
2 {
3     int i = 1;
4     int val = 1;
5     int nval = 1;
6
7     while (i < n) {
8         int t = val+nval;
9         val = nval;
10        nval = t;
11        i++;
12    }
13
14    return val;
15 }

```

Листинг 3.47. Использование конструкции goto в условном переходе

```

1 int fib_w_goto(int n)
2 {
3     int val = 1;
4     int nval = 1;
5     int nmi, t;
6
7     if (val >= n)
8         goto done;
9     nmi = n-1;
10
11    loop:
12        t = val+nval;
13        val = nval;
14        nval = t;
15        nmi--;
16        if (nmi)
17            goto loop;
18
19    done:
20        return val;
21 }

```

Таблица 3.10. Использование регистра

Регистр	Переменная	Начальное значение
%edx	nmi	n-1
%ebx	val	1
%ecx	nval	1

Листинг 3.48. Составление кода

```

1 movl 8(%ebp),%eax      Взять n
2 movl $1,%ebx           Установить val значение 1
3 movl $1,%ecx           Установить nval значение 1
4 cmpl %eax,%ebx         Сравнение val:n
5 jge .L9                 If >= goto done
6 leal -1(%eax),%edx     nmi = n-1
7 .L10:                  loop:
8 leal (%ecx,%ebx),%eax   Сравнение t = nval+val
9 movl %ecx,%ebx          Присвоить val значение nval
10 ovrl %eax,%ecx         Присвоить nval значение t

```

```

11 decl    %edx           Уменьшить значение пmi
12 jnz    .L10           if != 0, goto loop
13 .L9:                 done :

```

УПРАЖНЕНИЕ 3.11

Для приведенной здесь программы на языке C

```

1 int loop_while(int a, int b)
2 {
3     int i = 0;
4     int result = a;
5     while (i < 256) {
6         result += a;
7         a -= b;
8         i += b;
9     }
10    return result;
11 }

```

компилятор GCC генерирует следующий код на языке ассемблера:

```

1    movl    8(%ebp),%eax
2    movl    12(%ebp),%ebx
3    xorl    %ecx,%ecx
4    movl    %eax,%edx
5    .p2align 4,,7
6    .L5:
7    addl    %eax,%edx
8    ubl    %ebx,%eax
9    addl    %ebx,%ecx
10   cmpl    $255,%ecx
11   jle    .L5

```

1. Постройте таблицу использования регистров в теле цикла, подобную табл. 3.10.
2. Выделите условие продолжение цикла и тело цикла, соответствующие им строки в программном коде на языке ассемблера. Каким оптимизирующими действиям подверг компилятор проверку начального условия?
3. Введите комментарий в программный код на языке ассемблера, описывающий операции, выполняемые программой, аналогично тому, как это сделано в листинге 3.27.
4. Напишите (в языке C) версию этой функции, использующую команду `goto`, которая имеет структуру, аналогичную структуре, представленной в ассемблерном коде, как это сделано в листинге 3.27.

Циклы *for*

Общая форма цикла *for* имеет следующий вид:

```
for (init-expr; test-expr; update-expr)
    body-оператор.
```

Стандарт языка программирования С устанавливает, что поведение такого цикла можно описать следующими программными кодами, в которых используется цикл *while*:

```
init-expr;
while (test-expr) {
    body-оператор
    update-expr;
}
```

То есть, программа сначала выполняет *init-expr* (инициализацию цикла). Затем она входит в цикл, в котором сначала вычисляет *test-expr* (условие продолжения цикла), выходя из цикла, если условие не выполняется, затем выполняет *body-statement* (тело цикла), и в завершение выполняет *update-expr* (обновление переменной цикла).

Компилированная форма этого программного кода основана на переходе от цикла *do-while* к циклу *while*, описанным ранее. Сначала приводим формат с циклом *do-while*:

```
init-expr;
if (!test-expr)
    goto done;
do (
    body-оператор
    update-expr;
) while (update-expr);
done:
```

Этот формат может быть преобразован в следующий программный код, использующий команду *goto*:

```
init-expr;
t = test-expr;
if (!t)
    goto done;
loop:
    body-оператор
    update-expr;
    t = test-expr;
```

```
if (t)
    goto loop;
done:
```

В качестве примера рассмотрим следующую программу, вычисляющую функцию Фибоначчи с применением цикла `for` (листинг 3.29):

Листинг 3.29. Вычисление функции Фибоначчи

```
1 int fib_f(int n)
2 {
3     int i;
4     int val = 1;
5     int nval = 1;
6
7     for (i = 1; i < n; i++) {
8         int t = val+nval;
9         val = nval;
10        nval = t;
11    }
12
13    return val;
14 }
```

Преобразование этого кода в формате цикла позволяет получить программный код, идентичный программе функции `fib_w`, показанный в листинге 3.26. Фактически компилятор GCC порождает для обеих функций идентичные коды в языке ассемблера.

УПРАЖНЕНИЕ 3.12

Рассмотрим следующий код на языке ассемблера:

```
1 movl 8(%ebp),%ebx
2 movl 16(%ebp),%edx
3 xorl %eax,%eax
4 decl %edx
5 js .L4
6 movl %ebx,%ecx
7 imull 12(%ebp),%ecx
8 .p2align 4,,7      Вставлена с целью оптимизации кеш-памяти
9 .L6:
10 addl %ecx,%eax
11 subl %ebx,%edx
12 jns .L6
13 .L4:
```

Предыдущий программный код был получен путем компилирования программы на языке C, представленной в следующем обобщенном виде:

```

1 int loop(int x, int y, int n)
2 {
3     int result = 0;
4     int i;
5     for (i = ____; i ____; i = ____) {
6         result += ____;
7     }
8     return result;
9 }
```

Задача заключается в том, чтобы заполнить отсутствующие части программы на языке C, эквивалентной порожденной компилятором программе на языке ассемблера. Напомним, что результат вычисления функции находится в регистре `%eax`. Чтобы решить эту задачу, вам, возможно, придется немного поразмышлять о том, как лучше использовать регистры, а потом проверить, оправдались ли ваши усилия.

1. В каких регистрах хранятся переменные программы `result` и `i`?
2. Чему равно начальное значение переменной цикла `i`?
3. Каким является условие продолжения цикла, выраженное через переменную цикла `i`?
4. Как осуществляется обновление переменной цикла `i`?
5. Выражение на языке C, описывающее, как увеличивается значение переменной `result` в теле цикла, не меняет значения при переходе от одной итерации к следующей. Компилятор обнаруживает это и производит вычисление данного выражения перед тем, как войти в цикл. Что это за выражение?
6. Заполните все недостающие части в программном коде на C.

3.6.6. Операторы выбора

Операторы выбора (оператор `switch`) обеспечивают возможность многих переходов в зависимости от значения целочисленного индекса. Они особенно полезны при работе с различными тестами, когда существует вероятность большого числа различных результатов. Они не только облегчают чтение программных кодов на языке C, но и позволяют повысить эффективность программы благодаря использованию структуры данных, получившей название *таблицы переходов* (*jump table*). Таблица переходов есть массив, в котором `i` есть адрес сегмента программы, реализующего действия, которые программа должна предпринять, когда оператор выбора принимает значение `i`. Программа осуществляет ссылки на таблицу переходов, чтобы определить адрес назначения команды перехода. Преимущество использования таблицы переходов перед длинной последовательностью операторов `if-else` заключается в том, что время исполнения оператора выбора не зависит от числа переходов. Компилятор выбирает метод транслирования оператора выбора в зависимости от числа случаев и от

размера шага, разделяющего эти случаи. Таблицы переходов используются, когда имеется достаточно большое число случаев (например, четыре и более), и они охватывают небольшой диапазон значений.

В листинге 3.30 представлен пример оператора `switch` языка С. Для этого примера характерен ряд интересных свойств, в том числе и метки случаев, которые не охватывают смежные диапазоны. Например, отсутствуют метки для случаев 101 и 105, есть случаи с несколькими метками 104 и 106, бывает, когда одни случаи проваливаются (*fall through*) в другие (случай 102), поскольку программный код их не заканчивается оператором `break` (останов).

Листинг 3.30. Пример оператора случаев

```
1 int switch_eg(int x)
2 {
3     int result = x;
4
5     switch (x) {
6
7         case 100:
8             result *= 13;
9             break;
10
11        case 102:
12            result += 10;
13            /* Провал случая */
14
15        case 103:
16            result += 11;
17            break;
18
19        case 104:
20        case 106:
21            result *= result;
22            break;
23
24    default:
25        result = 0;
26    }
27
28    return result;
29 }
```

Трансляция в листинге 3.31 показывает структуру таблицы переходов `jt` и то, как следует осуществлять к ней доступ. Такие таблицы и доступы фактически не допускаются в обычном С.

Листинг 3.31. Расширение в расширенный язык C

```

1  /* Следующая строка недопустима в С */
2  code *jt[7] =
3      loc_A, loc_def, loc_B, loc_C,
4      loc_D, loc_def, loc_D
5  };
6
7  int switch_eg_impl(int x)
8  {
9      unsigned xi = x - 100;
10     int result = x;
11
12     if (xi > 6)
13         goto loc_def;
14
15     /* Следующий оператор goto недопустим в С */
16     goto jt[xi];
17
18 loc_A: /* Случай 100 */
19     result *= 13;
20     goto done;
21
22 loc_B: /* Случай 102 */
23     result += 10;
24     /* Провал в другой случай */
25
26 loc_C: /* Случай 103 */
27     result += 11;
28     goto done;
29
30 loc_D: /* Случай 104, 106 */
31     result *= result;
32     goto done;
33
34 loc_def: /* Случай default */
35     result = 0;
36
37 done:
38     return result;
39 }

```

В листинге 3.32 показана программа на языке ассемблера, полученная при компиляции процедуры `switch_eg`. Поведение такого программного кода показано в расширенном языке С на примере процедуры `switch_eg_impl`, представленной в листин-

ге 3.31. Мы говорим "расширенный", поскольку в обычном языке С отсутствуют конструкции, необходимые для поддержки таблицы переходов этого типа, а отсюда следует, что используемый нами код недопустим. Массив `jt` содержит 7 записей, каждая из которых есть адрес соответствующего программного блока. С этой целью мы расширяем язык С, добавив данные типа `code`.

В строках 1—4 устанавливается доступ к таблице переходов. Чтобы убедиться в том, что значения x , которые либо меньше 100, либо больше 106, вызывают вычисления, определенные случаем `default`, программа генерирует значение x_1 без знака, равное $x - 100$. Для значений x в диапазоне между 100 и 106 x_1 принимает значения в пределах от 0 до 6. Все другие значения будут больше 6, поскольку в этом случае отрицательные значения $x - 100$ воспринимаются компьютером как числа без знака очень большой величины. В силу этого обстоятельства, программа использует команду `ja` (больше для чисел без знака) для перехода к коду, который представляет собой случай `default` (по умолчанию), когда x_1 больше 6. Используя команду `jt`, которая употребляется как указатель на таблицу переходов, этот программный код выполняет переход по адресу, указанному в записи x_1 этой таблицы. Обратите внимание на тот факт, что эта форма команды `goto` не допустима в С. Команда в строке 4 реализует переход к соответствующей записи в таблице переходов. Поскольку этот переход непрямой, целевой объект находится в памяти. Исполнительный адрес команды чтения определяется путем сложения базового адреса, описанного меткой `.L10`, и масштабированного значения переменной (это значение, которое хранится в регистре `%eax`, умножается на 4, поскольку длина каждой записи в таблице переходов равна 4 байтам).

Листинг 3.32. Ассемблер оператора выбора

Осуществить доступ к таблице переходов

```

1   leal -100(%edx),%eax          Вычисление выражения x1 = x-100
2   cmpl $6,%eax                 Сравнение x1:6
3   ja .L9 if >,                goto loc_def
4   jmp *.%L10(,%eax,4)          Goto jt[xi]

Случай 100
5   .L4:                           loc_A:
6   leal (%edx,%edx,2),%eax      Вычисление выражения 3*x
7   leal (%edx,%eax,4),%edx      Вычисление выражения x+4*3*x
8   jmp .L3                         Goto done

Случай 102
9   .L5: loc_B:
10  addl $10,%edx result += 10,    Провал в следующий случай

Случай 103
11 .L6:                           loc_C:
12  addl $11,%edx result += 11
13  jmp .L3                         Goto done

```

```

Случай 104, 106
14 .L8:           loc_D:
15 imull %edx,%edx      result *= result
16 jmp .L3          Goto done

Случай по умолчанию
17 .L9:           loc_def:
18 xorl %edx,%edx      result = 0

Возврат результата
19 .L3:           done:
20 movl %edx,%eax      Установить result как возвращаемое значение

```

В кодах языка ассемблера таблица переходов представлена следующими объявлениями в листинге 3.33, которые мы снабдили комментариями:

Листинг 3.33. Объявления

```

1 .section .rodata
2 .align 4          Выровнять адреса с кратностью 4
3 .L10:
4 .long .L4          Случай 100: loc_A
5 .long .L9          Случай 101: loc_def
6 .long .L5          Случай 102: loc_B
7 .long .L6          Случай 103: loc_C
8 .long .L8          Случай 104: loc_D
9 .long .L9          Случай 105: loc_def
10 .long .L8         Случай 106: loc_D

```

Эти объявления устанавливают, что в файле сегмента объектного кода с именем `.rodata` (что означает "только для чтения") должна быть последовательность из семи "длинных" (четырехбайтовых) слов, при этом значение каждого слова задается адресом команды, связанной с указанными метками ассемблерного кода (например, `.L4`). Метка `L10` помечает начало этой последовательности. Адрес, связанный с этой меткой, служит базой для непрямых переходов (команда 4).

Блоки программных кодов с метками `loc_A` до метки `loc_D` и метки `loc_def` в процедуре `switch_eg_impl` в листинге 3.31 реализуют пять различных ветвей оператора выбора. Обратите внимание на то, что блок программного кода с меткой `loc_def` выполняется в тех случаях, когда `x` выходит за пределы диапазона 100—106 (при начальной проверке диапазона) или когда значение этой переменной равно 101 или 105 (на основании таблицы переходов). Следует отметить, что коды программного блока, помеченные как `loc_B`, проваливаются в блок с меткой `loc_C`.

УПРАЖНЕНИЕ 3.13

В функции C, которая приводится далее, мы опустили тело оператора выбора. В соответствующем коде на C метки случаев не охватывают примыкающий диапазон, а некоторые случаи помечены несколькими метками.

```
int switch2(int x) {
    int result = 0;
    switch (x) {
        /* Тело оператора выбора опускается */
    }
    return result;
}
```

Для этой функции компилятор генерирует ассемблерный код, который следует за начальной частью процедуры и за таблицей переходов. Переменная *x* хранится в ячейке, смещённой на 8 относительно адреса в регистре *%ebp*.

Организация доступа к таблице переходов

```
1  movl 8(%ebp),%eax      Поиск x
2  addl $2,%eax
3  cmpl $6,%eax
4  ja .L10
5  jmp * .L11(%eax,4)
```

Таблица переходов для оператора выбора switch2

```
1  .L11:
2  .long .L4
3  .long .L10
4  .long .L5
5  .long .L6
6  .long .L8
7  .long .L8
8  .long .L9
```

Воспользуйтесь полученной информацией, чтобы ответить на следующие вопросы:

1. Какими были значения меток случаев в теле оператора выбора?
2. Какие случаи в программе на С имеют несколько меток?

3.7. Процедуры

Чтобы вызвать процедуру, необходимо ей передать как данные (в виде параметров процедуры и возвращаемых значений), так и управления из одной части программного кода в другую. Кроме того, нужно выделить память для локальных переменных процедуры при входе в нее и освободить эту память при выходе из процедуры. Большинство машин, включая и машины с архитектурой IA32, выполняют только простые команды передачи управления процедурам и получения управления от них. Передача данных, выделение памяти локальным переменным и освобождение этой памяти при выходе из процедуры осуществляется посредством операций манипулирования программным стеком.

3.7.1. Структура стекового фрейма

Программы для машин с архитектурой IA32 используют программный стек для поддержки вызовов процедур. Стек используется для передачи процедуре аргументов, для хранения возвращаемой информации и для хранения содержимого регистров, в целях их последующего восстановления, и как локальная память. Часть стека, выделяемого для одной процедуры, называется *стековым фреймом* (stack frame). На диаграмме рис. 3.4 показана общая структура стекового фрейма. Верхний стековый

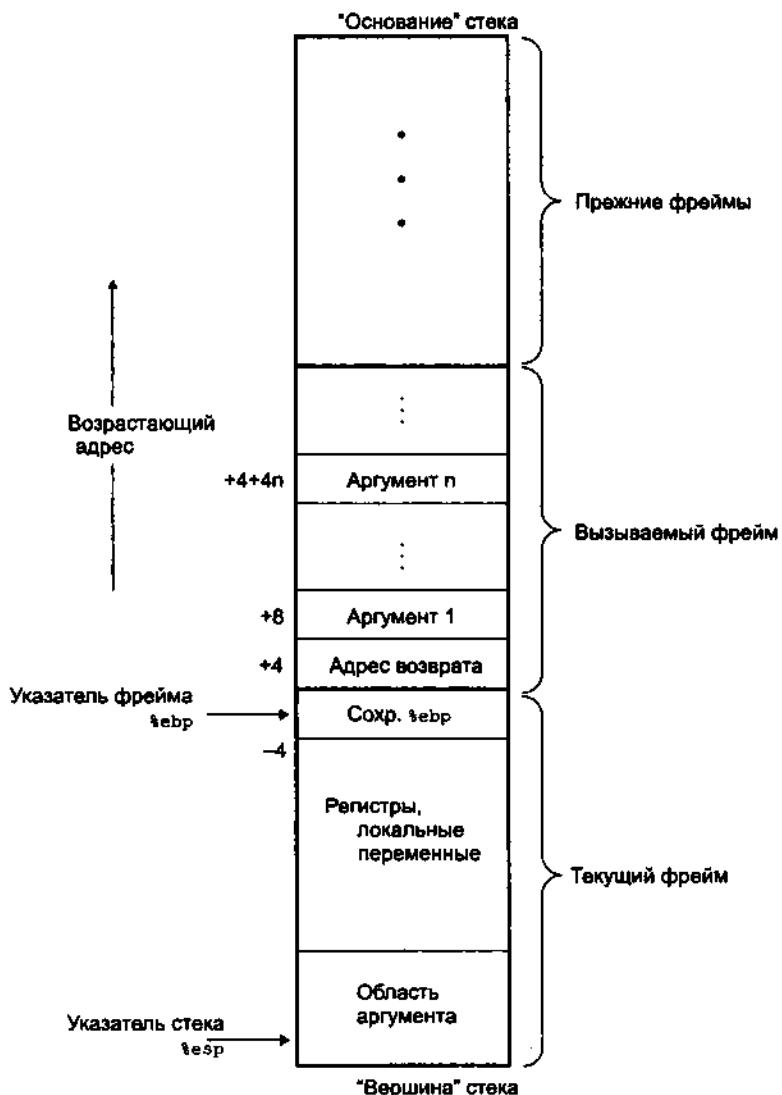


Рис. 3.4. Структура стекового фрейма

фрейм ограничен двумя указателями, при этом регистр `%ebp` служит указателем фрейма (frame pointer), а регистр `%esp` — указателем стека (stack pointer). Указатель стека может перемещаться во время исполнения процедуры, следовательно, большая часть доступов к информации выполняются относительно указателя фрейма.

Предположим, что процедура `R` (caller, вызывающая программа) вызывает процедуру `Q` (callee, вызываемая программа). Аргументы для процедуры `Q` содержатся в стековом фрейме процедуры `R`. Кроме того, когда `R` вызывает `Q`, адрес возврата (return address) внутри процедуры `R`, с которого программа должна возобновить свое выполнение, когда управление возвращается из `Q`, проталкивается в стек, образуя конец стекового фрейма процедуры `R`. Стековый фрейм для процедуры `Q` начинается с сохраненного значения стекового указателя (т. е. `%ebp`), за которым следуют копии любых других значений, сохраненных в регистрах.

Процедура `Q` также использует стек для любых локальных переменных, которые нельзя хранить в регистрах. Это может случиться по следующим причинам:

- Не хватает регистров, чтобы хранить в них все локальные данные.
- Некоторые из локальных переменных суть массивы или структуры, и в силу этого обстоятельства доступ к ним осуществляется с использованием ссылок на массивы и структуры.
- Адресный оператор `&` применяется к одной из локальных переменных, и мы должны быть способными вычислить для нее адрес.

Наконец, `Q` использует стековый фрейм для хранения аргументов для всех процедур, которые она сама вызывает.

Как уже мы отмечали ранее, стек возрастает в сторону уменьшения адресов, указатель стека в регистре `%esp` указывает на элемент в вершине стека. Данные могут сохраняться в стеке и выбираться из него, соответственно, с помощью команд `pushl` и `popl`. Пространство памяти для данных, для которых не указаны конкретные начальные значения, может быть выделено в стеке путем простого уменьшения указателя стека на соответствующий объем памяти. Аналогично, пространство может быть освобождено путем увеличения значения указателя стека.

3.7.2. Передача управления

Команды, поддерживающие вызовы процедур и возврат из процедур, сведены в табл. 3.11.

Таблица 3.11. Вызов процедур

Команда	Описание
<code>call Метка</code>	Вызов процедуры
<code>call *Операнд</code>	Вызов процедуры
<code>leave</code>	Подготовить стек к возврату
<code>ret</code>	Возврат из вызова

Команда `call` в качестве целевого значения имеет адрес команды, с которой стартует вызываемая процедура. Как и переходы, вызов может быть прямым и косвенным. В кодах языка ассемблера цель прямого вызова задается как метка, в то время как цель косвенного вызова задается звездочкой, за которой следует описатель операнда, имеющий тот же синтаксис, какой был использован для операнда команды `movl` (см. табл. 3.2).

Результатом выполнения команды `call` является заталкивание адреса возврата в стек и переход в начало вызываемой процедуры. Адресом возврата является адрес команды, следующей в программе непосредственно за командой `call`, так что выполнение программы возобновляется с этого места, когда вызываемая процедура возвращает управление. Команда `ret` выталкивает адрес из стека и передает управление в эту ячейку. Правильное использование этой команды заключается в том, чтобы указатель стека всегда указывал на то место, в котором предшествующая команда запомнила свой адрес возврата. Команда `leave` может быть использована для подготовки стека для возврата управления. Это эквивалентно следующей последовательности кодов (листинг 3.34):

Листинг 3.34. Подготовка стека

1 <code>movl %ebp, %esp</code>	Установить указатель стека на начало фрейма
2 <code>popl %ebp</code>	Восстановить сохраненное значение регистра <code>%ebp</code>
	Установить указатель стека на фрейм вызывающей процедуры

И наоборот, эта подготовка может быть выполнена прямой последовательностью операций перемещения и выталкивания.

Регистр `%eax` используется для возврата значения любой функции, которая возвращает целое число или указатель.

УПРАЖНЕНИЕ 3.14

Следующий фрагмент программного кода очень часто встречается в компилированных версиях библиотечных программ:

```
1      call next
2  next:
3      popl %eax
```

- Для сохранения каких значений предназначен регистр `%eax`?
- Объясните, почему нет соответствующей команды `ret` для этой команды `call`?
- Какой полезной цели служит этот фрагмент программного кода?

3.7.3. Соглашения об использовании регистров

Набор программных регистров действует как единый ресурс, совместно используемый всеми процедурами. И хотя только одна процедура может быть активной в кон-

крайний момент времени, мы должны сделать так, что если одна процедура (вызывающая процедура) вызывает другую (вызываемую процедуру), то последняя не затирает значения тех или иных регистров, которые запланированы для последующего использования. По этой причине архитектура IA32 принимает набор унифицированных правил по использованию регистров, которые должны соблюдаться всеми процедурами, включая и те, что содержатся в библиотеках программ.

По соглашению регистры `%eax`, `%edx` и `%ecx` относятся к категории регистров *сохранения вызывающей процедуры* (*caller save*). Когда процедура Q вызывается процедурой P, она получает возможность затирать содержимое этих регистров, не уничтожая данных, которые нужны процедуре P. С другой стороны, регистры `%ebx`, `%esi` и `%edi` относятся к категории регистров *сохранения вызываемой процедуры*. Это означает, что процедура Q должна сохранить значения любого из этих регистров в стеке, прежде чем записывать в регистр другие данные, и восстановливать их перед тем, как возвратить управление, поскольку процедуре P (или любой другой процедуре более высокого уровня) эти значения могут потребоваться для будущих вычислений. Кроме того, с регистрами `%ebp` и `%esp` нужно обращаться в соответствии с соглашениями, описанными здесь.

О выборе терминов

Рассмотрим следующий сценарий:

```
int P()
{
    int x = f(); /* Конкретные вычисления */
    Q();
    return x;
}
```

Процедура P хочет, чтобы значение x, которое она вычислила, оставалось неизменным на всем протяжении вызова процедуры Q. Если x находится в регистре *сохранения вызывающей процедуры*, то процедура P (вызывающая процедура) должна сохранить это значение перед тем, как вызывать Q, и восстановить ее после того, как Q возвратит управление. Если x находится в регистре *сохранения вызываемой процедуры*, и Q (вызываемая процедура) хочет использовать этот регистр, то процедура Q должна сохранить это значение перед тем, как использовать этот регистр, и сохранить его перед тем, как возвратить управление. В любом случае запоминание означает заталкивание значения регистра в стек, в то время как восстановление требует выталкивания значения из стека и размещение его в регистре.

В качестве примера рассмотрим следующий программный код (листинг 3.35):

Листинг 3.36. Использование регистров

```
1 int P(int x)
2 {
3     int y = x*x;
```

```

4     int z = Q(y);
5
6     return y + z;
7 }

```

Процедура P вычисляет y, прежде чем вызовет процедуру Q, однако она также должна позаботиться о том, чтобы значение y было доступным и после того, как процедура Q вернет управление. Она может сделать это одним из следующих двух способов:

- Она может запомнить значение в собственном стековом фрейме перед тем, как обратиться к Q; когда Q возвратит управление, она может выбрать значение y из стека.
- Она может запомнить значение y в регистре сохранения вызываемой процедуры. Если или какая-либо другая процедура, вызванная процедурой Q, намеревается использовать этот регистр, она должна сохранить это значение в стековом фрейме и восстановить его, прежде чем передаст управление. Следовательно, когда Q возвращает управление процедуре P, значение уже будет находиться в регистре сохранения вызываемой процедуры, либо по причине того, что этот регистр вообще не подвергался изменениям, либо в силу того, что его значение было сохранено и затем восстановлено.

Чаще всего компилятор GCC использует второе соглашение, поскольку он стремится уменьшить общее число операций записи в регистры и считывания из регистров.

УПРАЖНЕНИЕ 3.15

Следующая последовательность программных кодов возникает непосредственно перед началом программного кода на языке ассемблера, генерированного компилятором GCC для процедуры на языке C.

```

1 pushl %edi
2 pushl %esi
3 pushl %ebx
4 movl 24(%ebp),%eax
5 imull 16(%ebp),%eax
6 movl 24(%ebp),%ebx
7 leal 0(%eax,4,%ecx)
8 addl 8(%ebp),%ecx
9 movl %ebx,%edx

```

Мы видим, что только содержимое трех регистров (%edi, %esi и %ebx) запоминается в стеке. Затем программа изменяет содержимое этих трех регистров, а также еще трех других регистров (%eax, %ecx и %edx). В конце этой процедуры значения регистров %edi, %esi и %ebx восстанавливаются, для чего используется команда popl, в то время как другие регистры остаются в своем измененном состоянии. Объясните, почему возникло такое несоответствие при сохранении и восстановлении содержимого регистров (или короче, восстановления регистров).

3.7.4. Примеры процедур

В качестве примера рассмотрим процедуры на языке С, представленные в листинге 3.36. На рис. 3.5 показаны стековые фреймы для двух процедур. Обратите внимание на то, что процедура `swap_add` выбирает свои аргументы из стекового фрейма для процедуры `caller`. Доступ к этим ячейкам осуществляется относительно указателя фрейма в регистре `%ebp`. Цифры слева от фреймов означают адресные смещения относительно указателя фрейма.

Стековый фрейм для процедуры `caller` содержит память для хранения локальных переменных `arg1` и `arg2` в позициях `-8` и `-4` относительно указателя фрейма. Эти переменные должны быть сохранены в стеке, поскольку мы должны вычислить для них адреса. Приведенный ниже ассемблерный код из компилированной версии процедуры `caller` показывает, как эта процедура вызывает процедуру `swap_add`.

Листинг 3.36. Пример объявления и вызова процедуры

Программный код вызова в вызывающей процедуре

```

1  leal    -4(%ebp),%eax          Вычислить &arg2
2  pushl  %eax                  Протолкнуть &arg2
3  leal    -8(%ebp),%eax          Вычислить &arg1
4  pushl  %eax                  Протолкнуть &arg1
5  call   swap_add              Вызов функции swap_add

1  int swap_add(int *xp, int *yp)
2  {
3      int x = *xp;
4      int y = *yp;
5
6      *xp = y;
7      *yp = x;
8      return x + y;
9  }
10
11 int caller()
12 {
13     int arg1 = 534;
14     int arg2 = 1057;
15     int sum = swap_add(&arg1, &arg2);
16     int diff = arg1 - arg2;
17
18     return sum * diff;
19 }
```

Скомпилированный программный код процедуры `swap_add` можно разбить на три части: установка, которая выполняет инициализацию стекового фрейма, тело, в кото-

ром выполняются содержательные вычисления, и завершающая часть, в которой восстанавливаются первоначальные значения стеков и возвращается управление в вызывающую программу.



Рис. 3.5. Стековые фреймы процедур

В листинге 3.37 приводится код установки процедуры `swap_add`. Напомним, что команда `call` проталкивает адрес возврата в стек.

Листинг 3.37. Код процедуры

Код установки в процедуре `swap_add`

```

1  swap_add:
2  pushl %ebp          Сохранить старое содержимое %ebp
3  movl %esp,%ebp      Установить %ebp как указатель фрейма
4  pushl %ebx          Сохранить %ebx

```

Процедуре `swap_add` для хранения промежуточных данных нужен регистр `%ebx`. Поскольку это регистр сохранения вызываемой процедуры, она заталкивает его прежнее значение в стек как часть операции по установке стекового фрейма.

В листинге 3.38 приводим программный код тела процедуры `swap_add`.

Листинг 3.38. Тело процедуры

Тело процедуры `swap_add`

```

5  movl 8(%ebp),%edx    Взять хр
6  movl 12(%ebp),%ecx    Взять ур
7  movl (%edx),%ebx      Взять х
8  movl (%ecx),%eax      Взять у
9  movl %eax,(%edx)      Сохранить у по адресу *хр
10 movl %ebx,(%ecx)      Сохранить х по адресу *ур
11 addl %ebx,%eax        Установить возвращаемое значение = х+у

```

Этот программный код извлекает свои аргументы из стекового фрейма процедуры `caller`. Так как указатель фрейма сместился, местоположение этих аргументов также сместились с позиции `-12` и `-16` относительно прежнего значения регистра `%ebp`, соответственно, в позиции `+12` и `+8` относительно нового значения регистра `%ebp`. Отметим, что сумма переменных `x` и `y` запоминается в регистре `%eax` для последующей передачи в вызывающую программу в качестве возвращаемого значения.

В листинге 3.39 приводим завершающий код процедуры `swap_add`.

Листинг 3.39. Завершающий код процедуры

Заключительные коды процедуры `swap_add`

12	<code>popl %ebx</code>	Восстановить <code>%ebx</code>
13	<code>movl %ebp,%esp</code>	Восстановить <code>%esp</code>
14	<code>popl %ebp</code>	Восстановить <code>%ebp</code>
15	<code>ret</code>	Возврат в вызывающую процедуру

Этот программный код просто восстанавливает значения трех регистров `%ebx`, `%esp` и `%ebp`, а затем выполняет команду `ret`. Обратите внимание на то, что команды 13 и 14 могут быть заменены одной командой `leave`. Различные версии компилятора GCC, по-видимому, имеют свои предпочтения в этом вопросе.

Непосредственно после команды вызова процедуры `swap_add` в процедуре `caller` идет следующий программный код:

6	<code>movl %eax,%edx</code>	Возобновлять здесь
---	-----------------------------	--------------------

После возврата управления из `swap_add` процедура `caller` возобновляет свое выполнение с этой команды. Заметьте, что эта команда копирует возвращаемое значение из регистра `%eax` в другой регистр.

УПРАЖНЕНИЕ 3.16

Дана функция на языке С:

```
1 int proc(void)
2 {
3     int x,y;
4     scanf("%x %x", &y, &x);
5     return x-y;
6 }
```

Компилятор генерирует следующий ассемблерный код:

```
1 proc:
2 pushl %ebp
3 movl %esp,%ebp
4 subl $24,%esp
5 addl $-4,%esp
```

```

6 leal -4(%ebp),%eax
7 pushl %eax
8 leal -8(%ebp),%eax
9 pushl %eax
10 pushl $.LC0           Указатель на строку "%x %x"
11 call scanf

```

Вычертите диаграмму стекового фрейма в этой точке

```

12 movl -8(%ebp),%eax
13 movl -4(%ebp),%edx
14 subl %eax,%edx
15 movl %edx,%eax
16 movl %ebp,%esp
17 popl %ebp
18 ret

```

Предположим, что процедура начинает выполняться со следующими значениями в регистрах:

%esp	0x800040
%ebp	0x800060

Предположим, что процедура `rgos` вызывает процедуру `scanf` (строка 11) и что `scanf` читает значения 0x46 и 0x53 со стандартного устройства ввода-вывода. Предположим, что строка `%x %x` хранится в ячейке памяти 0x300070.

1. Какое значение устанавливается в регистре `%ebp` строкой 3?
2. По каким адресам запоминаются локальные переменные `x` и `y`?
3. Какое значение хранится в регистре `%esp` после того, как будет выполнена строка 10?
4. Начертите диаграмму стекового фрейма процедуры `rgos` сразу после того, как `scanf` возвратит управление. Отобразите на ней как можно больше информации об адресах и содержимом элементов стекового фрейма.
5. Укажите участки стекового фрейма, которые не используются процедурой `rgos` (эти неиспользуемые области включены с тем, чтобы повысить производительность кэш-памяти).

3.7.5. Рекурсивные процедуры

Стек и соглашения о редактировании связей, описанные в предыдущем разделе, позволяют процедурам вызывать самих себя рекурсивно. Поскольку каждый вызов имеет свое пространство в стеке, то локальные переменные многих незавершенных вызовов никак не мешают друг другу. Более того, дисциплина обслуживания стека обеспечивает применение правильной стратегии распределения локальной памяти при вызове процедуры и освобождения памяти при возвращении управления вызывающей процедуре.

В листинге 3.40 представлен программный код рекурсивной процедуры вычисления чисел Фибоначчи. (Обратите внимание на то обстоятельство, что этот код исключительно незэффективен, но в данном случае нас интересует не столько эффективность алгоритма, сколько иллюстративность программы.) Полностью программа на языке ассемблера также показана в листинге 3.41.

Листинг 3.40. Код рекурсивной процедуры вычисления чисел Фибоначчи на языке C

```

1 int fib_rec(int n)
2 {
3     int prev_val, val;
4
5     if (n <= 2)
6         return 1;
7     prev_val = fib_rec(n-2);
8     val = fib_rec(n-1);
9     return prev_val + val;
10 }
```

Листинг 3.41. Код на языке ассемблера рекурсивной программы

1 fib_rec:	Установочный код
2 pushl %ebp	Сохранение старого значения %ebp
3 movl %esp,%ebp	Установить %ebp как указатель фрейма
4 subl \$16,%esp	Выделить 16 байтов под стек
5 pushl %esi	Сохранить %esi (смещение -20)
6 pushl %ebx	Сохранить %ebx (смещение -24)
7 movl 8(%ebp),%ebx	Получить п
8 cmpl \$2,%ebx	Сравнение n:2
9 jle .L24	if <=, goto terminate
10 addl \$-12,%esp	Выделить 12 под стек
11 leal -2(%ebx),%eax	Вычислить n-2
12 pushl %eax	Протолкнуть как аргумент
13 call fib_rec	Вызов процедуры fib_rec(n-2)
14 movl %eax,%esi	Сохранить результат в регистре %esi
15 addl \$-12,%esp	Выделить 12 байтов под стек
16 leal -1(%ebx),%eax	Вычислить n-1
17 pushl %eax	Протолкнуть как аргумент
18 call fib_rec	Вызов процедуры fib_rec(n-1)
19 addl %esi,%eax	Вычислить выражение val+nval
20 jmp .L25	Goto done

Условия завершения

```
21 .L24:          terminate:  
22     movl $1,%eax      Возвратить значение 1  
23     Завершающие коды  
24 .L25:          done:  
25     leal -24(%ebp),%esp    Установить стек со смещением -24  
26     popl %ebx        Восстановить регистр %ebx  
27     popl %esi        Восстановить регистр %esi  
28     movl %ebp,%esp      Восстановить указатель стека  
29     popl %ebp        Восстановить регистр %ebp  
30     ret.            Возврат
```

Несмотря на то, что этот программный код достаточно длинный, он, тем не менее, заслуживает более внимательного изучения. Установочный код (строки 2—6) создает стековый фрейм, который содержит прежнее значение регистра `%ebp`, 16 неиспользованных байтов и сохраняет для вызываемой процедуры значения регистров сохранения `%esi` и `%ebx`, как это показано на левой диаграмме рис. 3.6. Затем он использует регистр `%ebx` для хранения параметра процедуры `n` (строка 7). В случае терминального условия управления переходит к строке 22, на которой возвращаемой величине присваивается значение 1.

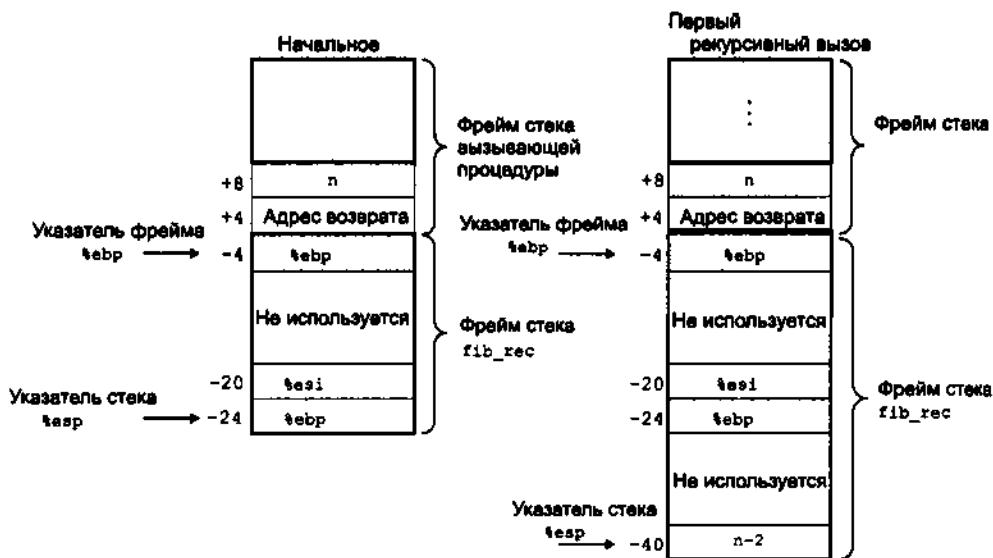


Рис. 3.6. Стековый фрейм рекурсивной функции Фибоначчи

Для промежуточного условия команды 10 и 12 формируют первое рекурсивное обращение. Это обращение предусматривает выделение 12 байтов для стека, которые никогда не используются, затем оно проталкивает в стек вычисленное значение $n-2$. Затем она выполняет рекурсивный вызов, который, в свою очередь, генерирует по-

следовательность вызовов, которые отводят память под стековые фреймы, выполняют различные операции над локальной памятью и т. д. При каждом возврате из процедур освобождается некоторое пространство стека и восстанавливаются модифицированные регистры сохранения вызываемой процедуры. Таким образом, когда мы возвратимся к текущему вызову в строке 14, мы можем не без оснований считать, что регистр `%eax` содержит значение, возвращаемое посредством рекурсивного вызова, и что регистр `%ebx` содержит значения параметра функции `n`. Возвращаемое значение (локальная переменная `prev_val` в программном коде на С) сохранено в `%eax` регистре (строка 14). Используя регистр сохранения вызываемой процедуры, мы можем быть уверены в том, что это значение останется неизменным и после второго рекурсивного вызова.

Команды с 15 по 17 составляют второй рекурсивный вызов. И снова вызов выделяет 12 байтов, которые никогда не будут использованы, заталкивает в стек значение `n-1`. Этому вызову соответствует результат, вычисленный и помещенный в регистр `%eax`, и мы имеем основание ожидать, что результат предыдущего вызова находится в регистре `%esi`. Они складываются, и их сумма дает возвращаемое значение (команда 19).

Код завершения восстанавливает регистры и освобождает память, выделенную под стековый фрейм. Он начинается (строка 24) с того, что образует стековый фрейм в ячейке, указанной в регистре `%ebx`. Заметьте, что если позиция этого стека вычисляется относительно значения регистра `%esp`, то вычисления будут правильными независимо от того, выполнены условия окончания или нет.

3.8. Распределение памяти под массивы и доступ к массивам

Массивы в языке С являются одним из способов агрегирования скалярных данных в более крупные типы данных. Язык С использует особо простые реализации массивов, следовательно, трансляция массивов в машинные коды не сопряжена с трудностями. Одна из интересных особенностей языка С заключается в том, что он допускает построение указателей на элементы массива и позволяет выполнять с указателями различные арифметические операции. Эти указатели в ассемблерном коде транслируются в вычисления соответствующих адресов.

Оптимизирующие компиляторы весьма успешно решают задачу упрощения вычисления адресов при индексировании массивов. Это делает соответствие между кодами на языке С и их трансляцией в машинные коды трудным для расшифровки.

3.8.1. Базовые принципы

Для данных типа T и целочисленной константы N объявление

$T A[N];$

приводит к следующим последствиям. Во-первых, для конкретных целей выделяется непрерывная область памяти размером $L \cdot N$, где L есть размер (в байтах) типа данных

Т. Обозначим начальную ячейку этой области через x_A . Во-вторых, оно вводит в употребление идентификатор A , который может быть использован как указатель на начало массива. Значение этого указателя есть x_A . Доступ к элементам массива может быть реализован путем использования целочисленного индекса, принимающего значения в пределах от 0 до $N-1$. Элемент массива будет храниться по адресу $x_A + iN$.

В качестве примера рассмотрим следующие объявления:

```
char      A[12];
char      *B[8];
double   C[6];
double   *D[5];
```

Эти объявления порождают массив со следующими параметрами (табл. 3.12).

Таблица 3.12. Массивы объявлений

Массив	Размер элемента	Общий размер	Начальный адрес	Элемент i
A	1	12	x_A	$x_A + i$
B	4	32	x_B	$x_B + 4i$
C	8	48	x_C	$x_C + 8i$
D	4	20	x_D	$x_D + 4i$

Массив A состоит из 12 однобайтовых элементов. Массив C состоит из 6 значений с плавающей запятой с двойной точностью, для каждого из них требуется 8 байтов. В и D суть массивы указателей, следовательно, каждый элемент этих массивов занимает 4 байта.

Память для размещения команд архитектуры IA32 сконструирована таким образом, чтобы упростить доступ к массивам. Например, предположим, что E есть массив значений int, а мы хотим вычислить E[i], при этом адрес помещен в регистр %edx, а i хранится в регистре %ecx. Тогда команда

```
movl (%edx,%ecx,4),%eax
```

выполнит вычисление адреса $x_E + 4i$, прочтает эту ячейку памяти и запомнит результат в регистре %eax. Допустимые коэффициенты масштабирования 1, 2, 4 и 8 охватывают размеры простых типов данных.

УПРАЖНЕНИЕ 3.17

Рассмотрим следующие объявления:

```
short      S[7];
short      *T[3];
short      **U[6];
long double V[8];
long double *W[4];
```

Заполните пропуски в приводимой далее таблице, описывающей размеры элементов, общие размеры и адрес элемента i для каждого из указанных массивов:

Массив	Размер элемента	Общий размер	Начальный адрес	Элемент i
S			x_S	
T			x_T	
U			x_U	
V			x_V	
W			x_W	

3.8.2. Арифметические операции с указателями

Язык С допускает выполнение арифметических операций над указателями, при этом вычисленное значение умножается на коэффициент масштабирования, соответствующий типу данных, на которые ссылается указатель. Иначе говоря, если p есть указатель на данные типа T , а значение p есть x_p , то выражение $p+i$ имеет значение $x_p + L \cdot i$, где L есть размер данных типа T . Унарные операторы $\&$ и $*$ позволяют генерировать и разыменовывать указатели. То есть, для выражения $Expr$, обозначающего некоторый конкретный объект, $\&Expr$ есть указатель, значение которого есть адрес этого объекта. Для выражения $Addr-Expr$, обозначающего адрес, значение $*Addr-Expr$ есть значение этого адреса. Следовательно, выражения $Expr$ и $*\&Expr$ эквивалентны. Операцию индексирования массивов можно применять как к массивам, так и к указателям. Ссылка на массив $A[i]$ идентична выражению $*(\&A + i)$. Она вычисляет адрес i -го элемента массива, а затем осуществляет доступ к этой ячейке памяти.

Возвращаясь к предыдущему примеру, предположим, что начальный адрес целочисленного массива E и целочисленный индекс i хранятся, соответственно, в регистрах $%edx$ и $%ecx$. В табл. 3.13 приводятся несколько выражений, в которых фигурирует массив E . Мы также покажем код на языке ассемблера, реализующий каждое из этих выражений, результат вычислений этих выражений запоминается в регистре $%eax$.

Таблица 3.13. Коды выражений

Выражение	Тип	Значение	Код на языке ассемблера
E	int *	x_E	<code>movl %edx, %eax</code>
E[0]	int	$M[x_E]$	<code>movl (%edx), %eax</code>
E[i]	int	$M[x_E + 4i]$	<code>movl (%edx,%ecx,4), %eax</code>
6E[2]	int *	$x_E + 8i$	<code>leal 8(%edx), %eax</code>

Таблица 3.13 (окончание)

Выражение	Тип	Значение	Код на языке ассемблера
E+i-1	int *	$x_E + 4i - 4$	leal -4(%edx,%ecx,4),%eax
$*(\&E[i]+i)$	int	$M[x_E + 4i + 4i]$	movl (%edx,%ecx,8),%eax
$\&E[i]-E$	int	i	movl %ecx,%eax

В этих примерах команда `leal` используется для построения адреса, в то время как команда `movl` — для ссылок в памяти (за исключением первого случая, когда она копирует адрес). Завершающий пример показывает, что мы можем вычислять разность двух указателей в одной и той же структуре данных, при этом результат делится на размер соответствующего типа данных.

УПРАЖНЕНИЕ 3.18

Предположим, что адрес целочисленного массива с элементами типа `short` и целочисленный индекс i хранятся, соответственно, в регистрах `%edx` и `%ecx`. Для каждого из следующих ниже выражений укажите их типы, формулу для вычисления их значений и реализацию в коде языка ассемблера. Результат должен быть помещен в регистр `%edx`, если это указатель, и в элемент регистра `%ax`, если это целое число типа `short`.

Выражение	Тип	Значение	Код на языке ассемблера
S+1			
S[3]			
$\&S[i]$			
S[4*i+1]			
S+i-5			

3.8.3. Массивы и циклы

Ссылки на элементы массивов в цикле часто носят регулярный характер, и эту особенность могут успешно использовать оптимизирующие компиляторы. Например, функция `decimal5` в листинге 3.42 вычисляет целое число, представленное массивом из 5 десятичных цифр. В процессе преобразования этой функции в программу на языке ассемблера компилятор генерирует код, подобный листингу 3.43, в виде функции `decimal5_opt` на языке С. Прежде всего, вместо того, чтобы использовать переменную цикла i , она использует арифметические операции для вычисления значений указателя при последовательном переходе от одного элемента массива к другому. Она вычисляет адрес концевого элемента массива и использует сравнение с этим адресом в качестве условия продолжения цикла. И, наконец, она может воспользоваться циклом `do-while`, поскольку в этом случае имеет место, по меньшей мере, одна итерация.

Листинг 3.42. Пример (недобросовестного) программиста

```

1 int decimal5(int *x)
2 {
3     int i;
4     int val = 0;
5
6     for (i = 0; i < 5; i++)
7         val = (10 * val) + x[i];
8
9     return val;
10 }
```

Листинг 3.43. Пример оптимизированного программиста

```

1 int decimal5_opt(int *x)
2 {
3     int val = 0;
4     int *xend = x + 4;
5
6     do (
7         val = (10 * val) + *x;
8         x++;
9     } while (x <= xend);
10
11    return val;
12 }
```

Ассемблерный код, представленный листингом 3.44, подвергается дальнейшей оптимизации с целью исключить операции целочисленного умножения. В частности, этот код использует операцию `leal` (строка 5) для вычисления $5*val$ как $val+4*val$. Затем он использует множитель 2 (строка 7) для масштабирования $10*val$.

Листинг 3.44. Ассемблерный код

1	<code>movl 8(%ebp),%ecx</code>	Взять базовый адрес массива x
2	<code>xorl %eax,%eax</code>	val = 0;
3	<code>leal 16(%ecx),%ebx</code>	xend = x+4 (16 байтов = 4 двойным словам)
4	<code>.L12:</code>	loop:
5	<code>leal (%eax,%eax,4),%edx</code>	Вычислить 5*val
6	<code>movl (%ecx),%eax</code>	Вычислить *x
7	<code>leal (%eax,%edx,2),%eax</code>	Вычислить *x + 2*(5*val)
8	<code>addl \$4,%ecx</code>	x++
9	<code>cmpl %ebx,%ecx</code>	Сравнение x:xend
10	<code>jbe .L12</code>	if <=, goto loop

Почему следует избегать целочисленного умножения

В более старых моделях процессоров команда целочисленного умножения требует для своего выполнения 30 временных циклов, в силу этого обстоятельства компиляторы стремятся избегать умножения там, где это возможно. В новейших моделях для умножения требуется всего лишь 3 таких цикла, поэтому такого рода оптимизация не оправдана.

3.8.4. Вложенные циклы

Общие принципы размещения в памяти массивов и ссылок на элементы массивов сохраняются в тех случаях, когда мы создаем массивы массивов. Например, объявление `int A[4][3];` эквивалентно объявлению

```
typedef int row3_t[3];
row3_t A[4];
```

Тип данных `row3_t` по определению есть массив из трех целых чисел. Массив `A` содержит четыре таких элемента, и каждый из них требует для размещения трех целых чисел 12 байтов. Общий размер массива в этом случае составляет $4 \times 4 \times 3 = 48$ байтов.

Массив `A` можно просматривать как двумерный массив с четырьмя строками и тремя столбцами, адреса элементов этого массива находятся в диапазоне от `A[0][0]` до `A[3][2]`. Элементы массива упорядочены в памяти по принципу "строка старше", означающему, что сначала идут все элементы строки 0, затем все элементы строки 1 и т. д. (табл. 3.14).

Таблица 3.14. Порядок элементов

Элемент	Адрес
<code>A[0][0]</code>	x_A
<code>A[0][1]</code>	$x_A + 4$
<code>A[0][2]</code>	$x_A + 8$
<code>A[1][0]</code>	$x_A + 12$
<code>A[1][1]</code>	$x_A + 16$
<code>A[1][2]</code>	$x_A + 20$
<code>A[2][0]</code>	$x_A + 24$
<code>A[2][1]</code>	$x_A + 28$
<code>A[2][2]</code>	$x_A + 32$
<code>A[3][0]</code>	$x_A + 36$
<code>A[3][1]</code>	$x_A + 40$
<code>A[3][2]</code>	$x_A + 44$

Такое упорядочение есть следствие нашего вложенного объявления. Рассматривая массив A как массив из четырех элементов, каждый из которых в свою очередь является массивом из трех элементов типа int, имеем A[0] (т. е. строка 0), за которой следует A[1] и т. д.

Чтобы получить доступ к элементам многомерного массива, компилятор генерирует код для вычисления смещения искомого элемента, а затем применяет команду movl, используя начало массива как базовый адрес и (возможно, масштабированное) смещение в качестве индекса. В общем случае массив объявляется как

$T D[R][C];$

Элемент массива $D[i][j]$ есть адрес памяти

$x_D + L(C \cdot i + j),$

где L есть размер типа данных T в байтах.

В качестве примера рассмотрим целочисленный массив A размерности 4×3 , определенный ранее. Предположим, что регистр %eax содержит i , регистр %edx содержит x_A , а регистр %ecx содержит j . Теперь элемент $A[i][j]$ может быть скопирован в регистр посредством программного кода в листинге 3.45.

Листинг 3.45. Копирование в регистр

```
A в регистре %eax, i в регистре %edx, j в регистре %ecx
1  sall $2,%ecx          j * 4
2  leal  (%edx,%edx,2),%edx    i * 3
3  leal  (%ecx,%edx,4),%edx    j * 4 + i * 12
4  movl  (%eax,%edx),%eax      Читать M[x_A + 4(3 * i + j)]
```

УПРАЖНЕНИЕ 3.19

Рассмотрим следующий исходный код, в котором константы M и N определены посредством оператора #define:

```
1  int mat1[M][N];
2  int mat2[N][M];
3
4  int sum_element(int i, int j)
5  {
6      return mat1[i][j] + mat2[j][i];
7 }
```

При компилировании данной программы компилятор генерирует следующий код на языке ассемблера:

```
1  movl  8(%ebp),%ecx
2  movl  12(%ebp),%eax
3  leal  0(%eax,%i),%ebx
```

```

4  leal  0(%ecx,%eax),%edx
5  subl  %ecx,%edx
6  addl  %ebx,%eax
7  sall  $2,%eax
8  movl  mat2(%eax,%ecx,4),%eax
9  addl  mat1(%ebx,%edx,4),%eax

```

Мобилизуйте все свои творческие ресурсы: вычислите значения M и N на основе этого ассемблерного кода.

3.8.5. Массивы фиксированных размеров

Компилятор C обладает многими возможностями оптимизации программ, работающих с многомерными файлами фиксированных размеров. Например, предположим, что мы объявили тип `fix_matrix` как матрицу фиксированного размера 16×16 следующим образом:

```

1  #define N 16
2  typedef int fix_matrix[N][N];

```

Код в листинге 3.46 вычисляет элемент i, k произведения матриц A и B . В этом коде удачная оптимизация выполняется несколько раз. Компилятор C генерирует код, аналогичный листингу 3.47. Процедура оптимизации обнаруживает, что цикл осуществляет доступ к элементам массива A в последовательности $A[i][0], A[i][1], \dots, A[i][15]$. Эти элементы занимают в памяти смежные позиции, начиная с элемента $A[i][0]$ массива. Поэтому программа использует переменную типа ссылки `Aptr` для доступа к этим следующим друг за другом ячейкам памяти. Этот же цикл осуществляет доступ к ячейкам матрицы B в последовательности $B[0][k], B[1][k], \dots, B[15][k]$. Эти элементы занимают позиции в памяти, начиная с элемента с адресом $B[0][k]$, и отделены друг от друга 64 байтами. Поэтому программа может воспользоваться переменной `Bptr` типа ссылки для доступа к этим следующим друг за другом ячейкам. В программе на языке C такой указатель получает приращения, равные 16, но на самом деле фактический указатель увеличивается на $4 \times 16 = 48$. И, наконец, программа может воспользоваться простым счетчиком для подсчета необходимого числа итераций.

Листинг 3.46. Произведение матриц

```

1  #define N 16
2  typedef int fix_matrix[N][N];
3
4  /* Вычислить значения i,k произведения матриц фиксированных размеров*/
5  int fix_prod_ele (fix_matrix A, fix_matrix B, int i, int k)
6  {
7      int j;
8      int result = 0;
9

```

```

10     for (j = 0; j < N; j++)
11         result += A[i][j] * B[j][k];
12
13     return result;
14 }
```

Мы приводим в листинге 3.47 программу `fix_prod_ele_opt` на языке С с тем, чтобы показать, какую оптимизацию осуществляет компилятор С, выполняющий ассемблирование. В листинге 3.48 следует фактический программный код цикла на языке ассемблера.

Листинг 3.47. Оптимизированная версия

```

1  /* Вычислить значения i,k произведения матриц фиксированных размеров */
2  int fix_prod_ele_opt(fix_matrix A, fix_matrix B, int i, int k)
3  {
4      int *Aptr = &A[i][0];
5      int *Bptr = &B[0][k];
6      int cnt = N - 1;
7      int result = 0;
8
9      do {
10          result += (*Aptr) * (*Bptr);
11          Aptr += 1;
12          Bptr += N;
13          cnt--;
14      } while (cnt >= 0);
15
16      return result;
17 }
```

Листинг 3.48. Ассемблерный код цикла

```

Aptr в регистре %edx, Bptr в регистре %ecx, result в %esi, cnt в %ebx
1 .L23:                         loop:
2  movl  (%edx),%eax             Вычислить t = *Aptr
3  imull (%ecx),%eax             Вычислить v = *Bptr * t
4  addl  %eax,%esi              Сложить v result
5  addl  $64,%ecx               Прибавить 64 к Bptr
6  addl  $4,%edx                Прибавить 4 к Aptr
7  decl  %ebx                  Уменьшить значение cnt
8  jns  .L23                   if >=, goto loop
```

Обратите внимание на то, что в этом программном коде все приращения указателя умножены на коэффициент масштабирования 4, по сравнению с программным кодом на С.

УПРАЖНЕНИЕ 3.20

Приводимый программный код на С устанавливает диагональным элементам матрицы значение `val`.

```
1  /* Присвоить всем диагональным элементам значения val */
2  void fix_set_diag(fix_matrix A, int val)
3  {
4      int i;
5      for (i = 0; i < N; i++)
6          A[i][i] = val;
7 }
```

Компилятор GCC генерирует следующий ассемблерный код:

```
1  movl  12(%ebp),%edx
2  movl  8(%ebp),%eax
3  movl  $15,%ecx
4  addl  $1020,%eax
5  .p2align 4,,7      Добавляется для оптимизации работы кэш-памяти
6 .L50:
7  movl  %edx,(%eax)
8  addl  $-68,%eax
9  decl  %ecx
10 jns   .L50
```

Напишите программу `fix_set_diag_opt` на языке C, осуществляющую оптимизацию, подобную той, что была применена к коду на языке ассемблера, и в том же стиле, что был использован в коде, представленном в листинге 3.43.

3.8.6. Динамически размещаемые массивы

Язык C поддерживает многомерные массивы, размеры которых (за исключением, возможно, только одномерных массивов) становятся известными в момент компиляции. Во многих приложениях требуются программные коды, которые будут работать с массивами произвольных размеров, автоматически размещаемых в памяти. Для таких массивов мы должны явно закодировать отображение многомерных массивов в одномерные. Мы можем определить тип `var_matrix` данных просто как `int *`:

```
typedef int *var_matrix
```

Чтобы распределить и инициализировать память для массива размера $n \times n$, мы воспользуемся функцией `calloc` операционной системы Unix (листинг 3.49).

Листинг 3.49. Определение функции

```
1  var_matrix new_var_matrix(int n)
2  {
3      return (var_matrix) calloc(sizeof(int), n * n);
4 }
```

Функция `calloc` (зафиксирована как часть стандарта ANSI C [32, 40]) имеет два аргумента: размер каждого элемента массива и требуемое число элементов массива. Она предпринимает попытку выделить пространство памяти для всего массива. Если эта попытка окажется удачной, она инициализирует всю выделенную область памяти нулями и устанавливает указатель на первый байт. Если не удается выделить нужное пространство памяти, функция возвращает ноль.

Динамическое распределение памяти и освобождение памяти в C, C++ и Java

В языке С динамическая память (пул памяти, выделенный для хранения структур данных) распределяется с помощью библиотечных функций `malloc` и `calloc`. Результат их выполнения эквивалентен операции `new` в С и С++. Как С, так и С++ требуют, чтобы программа освобождала выделенное ей пространство памяти посредством функции `free`. В языке Java освобождение памяти автоматически выполняется системой поддержки исполнения программ с помощью процесса, называемого *сборкой мусора* (*garbage collection*), о чем речь пойдет в главе 10.

Затем для вычисления индексов с учетом упорядочения по принципу "строка старше" с целью определения позиции элемента матрицы i, j мы можем воспользоваться формулой $i \times n + j$.

Листинг 3.50. Определение позиций элемента

```
1 int var_ele(var_matrix A, int i, int j, int n)
2 {
3     return A[(i*n) + j];
4 }
```

Обращение к элементам матрицы в листинге 3.50 транслируется в следующий код ассемблера (листинг 3.51):

Листинг 3.51. Ассемблерный код адреса элемента

1 movl 8(%ebp), %edx	Адрес матрицы A
2 movl 12(%ebp), %eax	Индекс i
3 imull 20(%ebp), %eax	Вычислить $n * i$
4 addl 16(%ebp), %eax	Вычислить $n * i + j$
5 movl (%edx,%eax,4), %eax	Элемент матрицы A[i*n + j]

Сравнивая этот программный код с тем, что мы использовали для индексирования массива фиксированного размера, мы видим, что динамическая версия в некотором смысле сложнее. Следует использовать операцию умножения с тем, чтобы умножить i на коэффициент масштабирования и вместо некоторой последовательности команд сдвига и сложения. Тем не менее для современных процессоров это не приводит к существенному снижению производительности.

Во многих случаях компилятор может упростить вычисление индексов для случаев массивов переменных размеров, используя те же принципы, которые применялись для вычисления индексов элементов массивов фиксированных размеров. Например, в листинге 3.52 показан программный код на С вычисления индексов i, k элементов произведения двух матриц A и B в переменных размеров. В листинге 3.53 представлена оптимизированная версия ассемблерного кода, полученная путем применения метода обратного проектирования к ассемблерному коду, построенного путем компиляции исходной версии. Компилятор способен устраниТЬ из программного кода целочисленное умножение $i * n$ и $j * n$ посредством применения схемы последовательного доступа, определяемой структурой цикла. В этом случае вместо того, чтобы генерировать переменную $Bptr$, имеющую тип указателя, компилятор создает переменную, которую мы назовем $nTjPk$, (n умножить на j плюс k), поскольку ее значение равно $n * j + k$. Первоначально $nTjPk$ равна k , и она увеличивается на n при каждой итерации.

Листинг 3.52. Вычисление элемента произведения матриц

```

1  typedef int *var_matrix;
2
3  /* Вычисление элемента i,k произведения матриц
   переменных размеров */
4  int var_prod_ele(var_matrix A, var_matrix B, int i, int k, int n)
5  {
6      int j;
7      int result = 0;
8
9      for (j = 0; j < n; j++)
10         result += A[i*n + j] * B[j*n + k];
11
12     return result;
13 }
```

Листинг 3.53. Оптимизированная версия

```

1  /* Вычисление элемента i,k of произведения матриц
   переменных размеров */
2  int var_prod_ele_opt (var_matrix A, var_matrix B, int i, int k,
3  int n)
4  {
5      int *Aptr = &A[i*n];
6      int *Bptr = &B[k];
7      int result = 0;
8      int cnt = n;
9
10     if (n <= 0)
11         return result;
```

```

12     do {
13         result += (*Aptr) * (*Bptr);
14         Aptr += 1;
15         nTjPk += n;
16         cnt--;
17     }: while (cnt);
18
19     return result;
20 }
```

Компилятор генерирует программный код циклов, при этом регистр `%edx` содержит значение переменной `cnt`, регистр `%ebx` — значение переменной `Aptr`, регистр `%ecx` содержит значение переменной `nTjPk`, а регистр `%esi` содержит результат (листинг 3.54).

Листинг 3.54. Пример перелива данных

L37:	loop: <code>movl 12(%ebp),%eax</code> Взять В <code>movl (%ebx),%edi</code> Взять * Aptr <code>addl \$4,%ebx</code> Приращение значения Aptr <code>imull (%eax,%ecx,4),%edi</code> Умножить на В[nTjPk] <code>addl %edi,%esi</code> Прибавить к результату <code>addl 24(%ebp),%ecx</code> Прибавить п к nTjPk <code>decl %edx</code> Уменьшить значение cnt <code>jnz .L37</code> If cnt != 0, goto loop
------	--

Обратите внимание на тот факт, что переменные `V` и `n` нужно выбирать из памяти на каждой итерации. Этот код является примером *перелива данных* (register spilling). Для хранения всех необходимых промежуточных данных существующих регистров явно недостаточно, и в силу этого обстоятельства компилятор должен содержать некоторые локальные переменные в памяти. В таких случаях компилятор предпочитает переливать переменные `V` и `n` в память, поскольку они предназначены только для чтения — они не меняются при выполнении цикла. Перелив данных является характерной проблемой для архитектуры IA32, т. к. у процессора имеется слишком малое число регистров.

3.9. Структуры разнородных данных

В языке C существуют два механизма для создания типов данных путем объединения объектов различных типов: *структур* (structures), которые объявляются посредством ключевого слова `struct`, в результате чего многочисленные объекты объединяются в единую конструкцию; и *объединения* (unions), которые объявляются посредством ключевого слова `union` и позволяют ссылаться на объекты, используя с этой целью несколько различных типов.

3.9.1. Структуры

Объявление `struct` в языке C создает тип данных, который группирует объекты, возможно, различных типов, в один объект. К различным компонентам структуры можно обращаться по их именам. Реализация структур подобна реализации массивов в том смысле, что все компоненты структуры расположены в смежных участках памяти, а указателем на структуру служит адрес ее первого байта. Компилятор сохраняет информацию о каждом типе структуры, в частности смещение в байтах каждого поля. Он генерирует ссылки на элементы структуры, используя эти смещения как сдвиги в командах, ссылающихся на ячейки в памяти.

Представление объекта как структуры типа `struct`

Конструктор типа данных `struct` языка C больше, чем какое-либо другое языковое средство приближается к объектам, применяемых в языках C++ и Java. Он позволяет программисту сохранять информацию о том или ином логическом объекте в одной структуре данных и ссылаться на нее по именам элементов.

Например, соответствующая графическая программа может представить прямоугольник как структуру:

```
struct rect {
    int l1x; /* Координата X нижнего левого угла */
    int l1y; /* Координата Y нижнего левого угла */
    int color; /* Кодирование цвета */
    int width; /* Ширина (в пикселях) */
    int height; /* Высота (в пикселях) */
};
```

Мы можем объявить переменную `r` типа `struct rect` и присвоить следующие значения полям:

```
struct rect r;
r.l1x = r.l1y = 0;
r.color = 0xFF00FF;
r.width = 10;
r.height = 20;
```

Выражение `r.l1x` выбирает поле `l1x` структуры `r`.

Обычной практикой является перенос указателя структуры с одного места в другое вместо того, чтобы их копировать. Например, следующая функция вычисляет площадь прямоугольника, при этом указатель структуры `struct` прямоугольника передается функции:

```
int area(struct rect *rp)
{
    return (*rp).width * (*rp).height;
}
```

Выражение `(*rp).width` разыменовывает указатель и выбирает поле `width` полученной структуры. Круглые скобки нужны для того, чтобы компилятор не мог интерпретировать выражение `*rp.width` как `* (rp).width`, поскольку это неправильно. Сочетание разыменования и выбора поля настолько распространено, что в языке для этой цели предусмотрено специальное обозначение `->`. То есть,

`rp->width`

есть эквивалент выражения

`(*rp).width`

Например, мы можем записать функцию, которая вращает прямоугольник влево на 90° , в следующем виде:

```
void rotate_left(struct rect *rp)
{
    /* Поменять местами ширину и высоту */
    int t      = rp->height;
    rp->height = rp->width;
    rp->width = t;
}
```

Объекты языков C++ и Java более сложные по своей природе, чем структуры в языке C, прежде всего, из-за того, что они связывают с объектом набор *методов* (methods), которые можно вызвать, чтобы выполнить те или иные вычисления. В C мы можем записать эти вычисления в виде обычных функций, таких как функция `area` или `rotate_left`, показанных ранее.

В качестве примера рассмотрим следующее объявление структуры

```
struct rec {
    int i;
    int j;
    int a[3];
    int *p;
};
```

Эта структура содержит четыре поля: два четырехбайтовых поля типа `int`, массив, содержащий три четырехбайтовых элемента типа `int`, и четырехбайтовый целочисленный указатель — всего 24 байта.

Обратите внимание на тот факт, что массив `a` встроен в структуру (табл. 3.15). Номера в верхней строке диаграммы показывают смещение полей в байтах относительно начала структуры.

Таблица 3.15. Адресация структуры

Смещение	0	4	8				20
Содержимое	i	j	a[0]	a[1]	a[0]		p

Чтобы получить доступ к тому или иному полю структуры, компилятор генерирует код, который добавляет соответствующее смещение к адресу структуры. Например, предположим, что переменная `r` типа `struct rec *` находится в регистре `%edx`. Затем приводимый ниже код копирует элемент `r->i` в элемент `r->j`:

<code>movl (%edx), %eax</code>	Взять <code>r->i</code>
<code>movl %eax, 4(%edx)</code>	Сохранить <code>r->j</code>

Поскольку смещение `i` поля есть 0, адрес этого поля есть просто значение `r`. Чтобы запомнить значение в поле `j`, код добавляет смещение 4 к адресу `r`.

Чтобы построить указатель на объект в структуре, мы можем просто прибавить смещение соответствующего поля к адресу структуры. Например, построить указатель `r->a[1]`, прибавив смещение $8 + 4 \times 1 = 12$. Для указателя `r` в регистре `%eax` и целочисленной переменной `i` в регистре `%edx` мы можем получить значение указателя с помощью всего лишь одной команды:

`r` содержится в регистре `%eax`, `i` содержится в регистре `%edx`
`leal 8(%edx,%eax,4),%eax` `%eax r->a[i]`

В заключение приводим пример, в котором программный код реализует оператор:

`r->p = &r->a[r->i + r->j];`

Начнем с того, что поместим `r` в регистр `%edx` (листинг 3.55).

Листинг 3.55. Указатель на объект

```

1  movl 4(%edx),%eax      Установить r->j
2  addl (%edx),%eax      Прибавить r->i
3  leal 8(%edx,%eax,4),%eax Вычислить &r->[r->i + r->j]
4  movl %eax,20(%edx)     Сохранить r->p

```

Как показывает листинг 3.55, обработка различных полей структуры выполняется исключительно на стадии компиляции. Машинный код не содержит информации, касающейся объявлений полей или имен полей.

УПРАЖНЕНИЕ 3.21

Рассмотрим объявление следующей структуры:

```

struct prob {
    int *p;
    struct {
        int x;
        int y;
    } s;
    struct prob *next;
};

```

Это объявление показывает, что одна структура может быть встроена в другую аналогично тому, как массивы могут быть встроены в структуры и в другие массивы.

Следующая процедура (некоторые выражения опущены) функционирует на структуре

```
void sp_init(struct prob *sp)
{
    sp->s.x = _____;
    sp->p = _____;
    sp->next = _____;
}
```

1. Каковы смещения (в байтах) следующих полей?

p:
s.x:
s.y:
next:

2. Сколько байтов требуется для размещения этой структуры в памяти?

3. Компилятор генерирует следующий код на языке ассемблера для тела процедуры

```
1  movl 8(%ebp), %eax
2  movl 8(%eax), %edx
3  movl %edx, 4(%eax)
4  leal 4(%eax), %edx
5  movl %edx, (%eax)
6  movl %eax, 12(%eax)
```

На основе этой информации внесите пропущенные выражения в текст процедуры `sp_init`.

3.9.2. Объединения

Объединения предоставляют способ обойти систему типов языка C, позволяя ссылаться на конкретный объект в соответствии с множеством типов. Синтаксис объявления объединения идентичен синтаксису объявления структур, в то же время их семантики существенно различаются. Вместо того чтобы поля ссылались на различные блоки памяти, они осуществляют ссылку на один и тот же блок.

Рассмотрим следующие объявления (листинг 3.56):

Листинг 3.56. Объявление объединения

```
struct S3 {
    char c;
    int i[2];
```

```

    double v;
};

union U3 {
    char c;
    int i[2];
    double v;
};

```

Смещения полей и общие размеры типов данных S3 и U3 показаны в табл. 3.16:

Таблица 3.16. Общие размеры типов данных

Тип	c	i	v	Размер
S3	0	4	12	20
U3	0	0	0	8

(Вскоре мы увидим, почему i имеет смещение 4 в S3, а не 1.) Для указателя p типа union U3 ссылки p->c и p->i[0] указывают на начало структуры данных. Обратите также внимание на то обстоятельство, что общий размер объединения равен размеру самого большого из ее полей.

Объединения могут быть полезны в нескольких контекстах. В то же время они могут оказаться источниками программных ошибок, поскольку обходят средства защиты, предусматриваемые системой типов языка C. Объединения можно использовать в тех случаях, когда известно заранее, что применение одного из двух полей исключает использование другого. Если объявить эти два поля как часть объединения, можно сэкономить некоторое пространство памяти.

Например, предположим, что мы хотим построить структуру данных типа бинарного дерева, в котором каждый лист (концевая вершина) имеет тип данных double, в то время как внутренний узел содержит указатели на два дочерних узла, но не содержит данных. Объявим это дерево как структуру (листинг 3.57).

Листинг 3.57. Объявление структуры

```

struct NODE {
    struct NODE *left;
    struct NODE *right;
    double data
};

```

Для каждого узла потребуется 16 байтов, при этом половина этих байтов будет направлена на каждый тип узла. С другой стороны, если мы объявим узел как объединение (листинг 3.58), то каждый узел потребует 8 байтов.

Листинг 3.58. Узел как объединение

```
union NODE {
    struct {
        union NODE *left;
        union NODE *right;
    } internal;
    double data;
};
```

Если `n` есть указатель на узел типа `union NODE *`, то мы можем ссылаться на данные в листе посредством `n->data`, а на ветви внутреннего узла посредством

`n->internal.left`

и

`n->internal.right`

Однако при таком кодировании невозможно определить, является ли заданный узел листом или внутренним узлом. В этом случае обычно добавляется специальное поле признака (листинг 3.59):

Листинг 3.59. Поле признака

```
struct NODE {
    int is_leaf;
    union {
        struct {
            struct NODE *left;
            struct NODE *right;
        } internal;
        double data;
    } info;
};
```

Поле принимает значение 1 для листа и 0 для внутреннего узла. Такая структура требует в итоге 12 байтов: 4 байта для `is_leaf` и по 4 байта для `info.internal.left` и `info.internal.right`, либо 8 байтов для `info.data`. В этом случае экономия памяти, полученная благодаря использованию объединения, мала из-за громоздкости получившегося программного кода. В случае структур данных с большим числом полей экономия памяти может оказаться более существенной.

Объединения могут быть использованы для доступа к битовым комбинациям различных типов данных. Например, в листинге 3.60 программный код возвращает битовое представление типа `float` как типа `unsigned`:

Листинг 3.60. Представление беззнака

```
unsigned float2bit(float f)
{
    union {
        float f;
        unsigned u;
    } temp;
    temp.f = f;
    return temp.u;
};
```

В этом программном коде мы сохраняем аргумент в объединении, используя один тип данных, и осуществляем к нему доступ с помощью другого типа данных. Интересно отметить, что код, построенный компилятором для этой процедуры, идентичен коду, полученному для процедуры листинга 3.61:

Листинг 3.61. Другой вариант кода

```
1 unsigned copy(unsigned u)
2 {
3     return u;
4 }
```

Телом обеих процедур является всего лишь одна команда:

```
1 movl 8(%ebp),%eax
```

Этот пример показывает, что в ассемблерном коде отсутствует информация о типах. Аргумент процедуры расположен со смещением на 8 байтов относительно значения в регистре `%eax` независимо от того, какой тип он имеет, `float` или `unsigned`. Эта процедура просто копирует свои аргументы как возвращаемые величины, при этом ни один разряд не меняет своего значения.

При использовании объединений с целью комбинирования типов данных различных размеров проблемы упорядочения байтов становятся еще более актуальными. Например, предположим, что мы написали процедуру (листинг 3.62), которая создает 8-байтовое значение `double`, используя битовые комбинации, задаваемые двумя 4-байтовыми значениями типа `unsigned`.

Листинг 3.62. Создание двойного слова

```
1 double bit2double(unsigned word0, unsigned word1) .
2 {
3     union {
4         double d;
```

```

5     unsigned u[2];
6     ) temp;
7
8     temp.u[0] = word0;
9     temp.u[1] = word1;
10    return temp.d;
11 }

```

На остроконечных машинах (первым идет наименьший байт), например IA32, аргумент word0 размещается в четырех младших разрядах значения d, в то время как word1 оказывается в четырех старших байтах. На тупоконечных машинах (первым идет наибольший байт) роли обоих аргументов меняются.

УПРАЖНЕНИЕ 3.22

Рассмотрим следующее объявление объединения

```

union ele {
    struct {
        int *p;
        int y;
    } e1;
    struct {
        int x;
        union ele *next;
    } e2;
};

```

Это объявление показывает, что структуры могут быть встроены в объединения.

Следующая процедура (некоторые выражения в ней опущены) функционирует в некотором связном списке, имеющем эти объединения в качестве своих элементов:

```

void proc (union ele *up)
{
    up->_____ = *(up->_____) - up->_____;
}

```

1. Каким будет смещение (в байтах) следующих полей:

e1.p:
e1.y:
e2.x:
e2.next:

2. Сколько в итоге байтов требуется, чтобы разместить в памяти эту структуру?

3. Компилятор генерирует следующий код на языке ассемблера тела процедуры proc:

```

1  movl  0(%ebp),%eax
2  movl  4(%eax),%edx
3  movl  (%edx),%ecx
4  movl  %ebp,%esp
5  movl  (%eax),%eax
6  movl  (%ecx),%ecx
7  subl  %eax,%ecx
8  movl  %ecx,4(%edx)

```

На основании этой информации внесите пропущенные выражения в текст процедуры `proc`. Некоторые ссылки на объединения могут оказаться неоднозначными интерпретациями. Такие неоднозначности будут устранены, если вы посмотрите, куда направлены эти ссылки. Существует только один ответ, который не выполняет никаких преобразований данных и не нарушает никаких условий.

3.10. Выравнивание

Многие компьютерные системы накладывают ограничения на допустимые адреса для простых типов данных, требующие, чтобы адреса для некоторых типов объектов были кратными некоторому заданному значению (обычно 2, 4 или 8). Соблюдение таких *условий выравнивания* (alignment restrictions) упрощает конструирование аппаратных средств, обеспечивающих интерфейс между процессором и системой памяти. Например, предположим, что процессор при каждом обращении к памяти извлекает из нее 8 байтов по адресу, кратному 8. Если мы можем обеспечить, чтобы любое значение `double` было выровнено таким образом, что его адрес был кратным 8, то это значение можно считать из памяти или записать в память посредством одной операции. В противном случае придется дважды обращаться к памяти, поскольку объект может быть расположен в двух 8-байтовых блоках памяти.

Аппаратные средства с архитектурой IA32 будут работать правильно независимо от выравнивания данных. Тем не менее компания Intel рекомендует выравнивать данные с целью повышения производительности системы памяти. Операционная система Linux следует стратегии выравнивания, согласно которой 2-байтовые типы данных (например, тип `short`) должны иметь адреса, кратные двум, в то время как более крупные типы данных (например, `int`, `int *`, `float` и `double`) должны иметь адреса, кратные 4. Обратите внимание на то, что это требование означает, что наименьший значащий разряд адреса объекта типа `short` должен быть равен 0. Аналогично, любой объект типа `int` или любой указатель должны быть расположены в памяти по адресу, два наименьших значащих разряда которого равны нулю.

Выравнивание в Microsoft Windows

Система Microsoft Windows накладывает более строгое требование выравнивания — любой *k*-байтовый (простой) объект должен иметь адрес, кратный *k*. В частности, требуется, чтобы адрес значения `double` был кратен 8. Соблюдение этого требования улучшают рабочие характеристики памяти за счет некоторого увеличения непроизводительного расхода памяти. Проектные решения, принятые в

Linux, были, возможно, оправданы для процессоров типа i386, когда размеры оперативной памяти были небольшими, а шины памяти были четырехразрядными. В современных процессорах выравнивание, предлагаемое корпорацией Microsoft, является более удачным проектным решением.

Признак командной строки `-falign-double` заставляет компилятор GCC в Linux использовать выравнивание по 8 байтам для типа данных `double`. Это приводит к повышению производительности памяти, но это же становится причиной несоответствий при редактировании связей с библиотечными программными кодами, которые были откомпилированы с соблюдением требования выравнивания по 4 байтам.

Выравнивание автоматически выполняется, если гарантировано, что каждый тип организован и размещен в памяти таким образом, что каждый объект конкретного типа удовлетворяет условиям выравнивания. Компилятор размещает директивы в кодах языка ассемблера, указывая подходящее выравнивание для глобальных данных. Например, объявленная в ассемблерном коде таблица переходов (листинг 3.33) содержит в строке 2 следующую директиву:

```
.align 4
```

Это гарантирует, что данные, которые за ней следуют (в случае начала таблицы переходов), начнутся с адреса, кратного 4. Поскольку каждое вхождение таблицы имеет длину в 4 байта, последующие элементы подчиняются условию выравнивания по 4 байтам.

Библиотечные программы распределения памяти, такие как `malloc`, должны быть спроектированы таким образом, чтобы они могли вернуть указатель, который удовлетворяет самым строгим условиям выравнивания для машин, на которых они работают, обычно это требование соблюдения кратности 4 или 8. Для программных кодов, которые работают со структурами, компилятору часто приходится делать вставки при распределении памяти для полей с тем, чтобы каждый элемент структуры удовлетворял его требованиям к выравниванию. В таких случаях выравнивается также и начальный адрес структуры.

Рассмотрим, например, следующее объявление структуры:

```
struct S1 {
    int i;
    char c;
    int j;
};
```

Предположим, что компилятор использовал минимальное 9-байтовое распределение, представленное в табл. 3.17.

Таблица 3.17. Девятибайтовое распределение

Смещение	0	4	5
Содержимое	i	c	j

В этом случае невозможно выполнить требование 4-байтового выравнивания для обоих полей *i* (смещение 0) и *j* (смещение 5). Вместо этого компилятор вставляет три байта (в табл. 3.18 показаны как *xxx*) между полями *c* и *j*.

Таблица 3.18. Выравнивание по требованию

Смещение	0	4	5	8
Содержимое	<i>i</i>	<i>c</i>	<i>xxx</i>	<i>j</i>

В результате поле *j* имеет смещение 8, а общий размер структуры равен 12 байтам. Более того, компилятор должен сделать так, чтобы любой указатель *p* типа

struct S *

удовлетворял условию выравнивания на 4 байта. Используя прежние обозначения, положим, что указатель *p* имеет значение x_p . Тогда x_p должно быть кратным 4. Это гарантирует, что как $p \rightarrow i$ (адрес x_p), так и $p \rightarrow j$ (адрес $x_p + 4$) будут удовлетворять требованию 4-байтового выравнивания.

В дополнение к этому, перед компилятором может возникнуть необходимость заполнить пробелами окончание структуры, чтобы каждый элемент массива структур удовлетворял требованию выравнивания. Например, рассмотрим следующее объявление структуры:

```
struct S2 {
    int i;
    int j;
    char c;
};
```

Если мы отведем под эту структуру 9 байтов, мы, тем не менее, сможем выполнить условие выравнивания для полей по 4 байтам, если выберем начальный адрес этой структуры кратным 4. Теперь рассмотрим следующее объявление:

```
struct S2 d[4];
```

Если под эту структуру выделяется 9 байтов, то невозможно выполнить условие выравнивания для каждого элемента структуры *d*, поскольку эти элементы будут иметь адреса xd , $xd + 9$, $xd + 18$, $xd + 27$.

Вместо этого компилятор выделит структуре 12 байтов, при этом последние байты не используются (табл. 3.19)

Таблица 3.19. Описание структуры

Смещение	0	4	8	9
Содержимое	<i>i</i>	<i>c</i>	<i>c</i>	<i>xxx</i>

Следовательно, элементы структуры будут иметь адреса xd , $xd + 12$, $xd + 24$, $xd + 36$. Поскольку xd кратен четырем, все условия выравнивания будут выполнены.

УПРАЖНЕНИЕ 3.23

Для каждого из следующих объявлений структуры определите смещение каждого поля, общий размер структуры и предъявляемые к ним требования выравнивания в условиях операционной системы Linux и архитектуры IA32.

1. struct P1 { int i; char c; int j; char d; };
2. struct P2 { int i; char c; char d; int j; };
3. struct P3 { short w[3]; char c[3] };
4. struct P4 { short w[3]; char *c[3] };
5. struct P3 { struct P1 a[2]; struct P2 *p };

3.11. Как пользоваться указателями

Указатели являются одним из основных средств языка программирования С. Они обеспечивают универсальный способ дистанционного доступа к структурам данных. Указатели часто представляют собой источник путаницы для начинающих программистов, однако идеи, положенные в основу этих понятий, достаточно просты. Программный код, представленный в листинге 3.63, позволяет нам проиллюстрировать некоторый список этих идей.

- Каждый указатель имеет тип. Этот тип показывает, на какой тип объекта он указывает. В нашем примере программного кода мы видим следующие типы указателя (табл. 3.20). Обратите внимание на то, что в этой таблице мы указываем тип самого указателя, равно как и на тип объекта, на который он указывает. В общем случае, если объект имеет тип T , то указатель имеет тип $*T$. Специальный тип `void*` представляет групповой указатель. Например, функция `malloc` возвращает групповой указатель, который преобразуется в типизированный указатель посредством соответствующего приведения данных (строка 21 листинга 3.63).

Таблица 3.20. Типы указателя

Тип указателя	Тип объекта	Указатели
<code>int *</code>	<code>int</code>	<code>xp, ip[0], ip[1]</code>
<code>union uni *</code>	<code>union uni</code>	<code>up</code>

- Каждый указатель имеет значение. Этим значением является адрес некоторого объекта заданного типа. Специальное значение `NULL` (0) соответствует ситуации, когда указатель ни на что не указывает. Вскоре мы увидим значение используемых нами указателей.
- Указатели создаются с помощью оператора `&` языка С. Этот оператор может быть применен к любому выражению на языке С, которое характеризуется как lvalue, что означает выражение, которое может появиться в левой части оператора присваивания. Соответствующими примерами могут служить элементы структур,

объединений и массивов. В рассматриваемом нами примере программного кода этот оператор применяется к глобальной переменной *g* (строка 24 листинга 3.63), к элементу структуры *s.v* (строка 32 листинга 3.63) и к элементу объединения *up->v* (строка 33 листинга 3.63), а также к локальной переменной *x* (строка 32 листинга 3.63).

- Указатели разыменовываются посредством оператора `*`. Результатом является значение, имеющее тип, ассоциированный с указателем. Мы видим, что разыменование применяется к объектам *ip* и **ip* (строка 29 листинга 3.63), *ip[1]* (строка 31 листинга 3.63) и *xp* (строка 35 листинга 3.63). Наряду с этим, выражение *up->v* (строка 33 листинга 3.63) не только разыменовывает указатель *up*, но и выбирает поле *v*.
- Массивы и указатели тесно связаны друг с другом. На имя массива можно ссылаться (но его нельзя обновлять), как если бы это была переменная типа указатель. Ссылка на массив (например, *a[3]*) имеет в точности тот же результат, что и арифметические действия над указателями (например, **(a+3)*). Мы с этим сталкиваемся в строке 29 листинга 3.63, в которой мы распечатываем значение указателя на массив *ip* и ссылаемся на его первый элемент (элемент 0) посредством **ip*.
- Указатели могут указывать на функции. Это обеспечивает широкие возможности для сохранения и передачи ссылок на программный код, который может быть вызван в других частях программы. Мы видим это на примере переменной *f* (строка 15 листинга 3.63), которая была объявлена как переменная, которая указывает на функцию, принимающую *int* * как аргумент и возвращающая *void*. Это присваивание приводит к тому, что теперь *f* указывает на функцию *fun*. Когда мы позднее применим *f* (строка 36 листинга 3.63), это будет рекурсивный вызов.

Листинг 3.63. Программа, иллюстрирующая использование указателей

```

1 struct str {                                /* Пример структуры */
2     int t;
3     char v;
4 };
5
6 union uni {                                /* Пример объединения */
7     int t;
8     char v;
9 } u;
10
11 int g = 15;
12
13 void fun(int* xp)
14 {
15     void (*f)(int*) = fun;      /* f есть указатель на функцию */
16

```

```

17  /* Поместить структуру в стек */
18  struct str s = {1,'a'};    /* Инициализация структуры */
19
20  /* Разместить объединение в памяти кучи */
21  union uni *up = (union uni *) malloc(sizeof(union uni));
22
23  /* Локально объявленный массив */
24  int *ip[2] = (xp, &g);
25
26  up->v = s.v+1;
27
28  printf("ip      = %p, *ip     = %p, **ip    = %d\n",
29         ip, *ip, **ip);
30  printf("ip+1    = %p, ip[1]  = %p, *ip[1] = %d\n",
31         ip+1, ip[1], *ip[1]);
32  printf("&s.v   = %p, s.v    = '%c'\n", &s.v, s.v);
33  printf("&up->v = %p, up->v = '%c'\n", &up->v, up->v);
34  printf("f      = %p\n", f);
35  if (--(*xp) > 0)
36      f(xp);                /* Рекурсивный вызов fun */
37 }
38
39 int test()
40 {
41     int x = 2;
42     fun(&x);
43     return x;
44 }
```

Указатели на функции

Синтаксис объявления указателей на функции достаточно сложен для понимания начинающими пользователями. Чтобы понять, о чем идет речь, объявления, подобные следующему:

```
void (*f)(int*);
```

полезно читать, начиная изнутри (с "f") и продвигаться наружу. Таким образом, мы видим, что f есть указатель, о чем свидетельствует часть (*f). Эта часть есть указатель на функцию, у которой имеется один аргумент int*, этот вывод следует из фрагмента

```
(*f)(int*)
```

Наконец, мы видим, что это объявление есть указатель на функцию, которая принимает int * в качестве аргумента и возвращает void.

Круглые скобки, в которые заключена `*f`, необходимы, поскольку в противном случае объявление

```
void *f(int*);
```

будет воспринято как

```
(void *) f(int*);
```

То есть, оно будет интерпретировано как прототип функции, объявляющей функцию `f`, которая в качестве аргумента использует `int*` и возвращает `void *`.

Керниган (Kernighan) и Риччи (Ritchie) [40] представляют весьма полезное учебное пособие, в котором подробно рассматриваются особенности объявлений в языке программирования C.

Приводимые нами программные коды содержат ряд обращений к функции `printf`, которая распечатывает некоторые из указателей (используя директиву `%p`) и значения. В случае ее использования она генерирует следующие выходные данные (листинг 3.64):

Листинг 3.64. Выходные данные

```
1 p      = 0xbffffefa8, *ip    = 0xbffffefe4, **ip    = 2
2 ip+1   = 0xbffffefac, ip[1] = 0x804965c, *ip[1] = 15
3 &s.v   = 0xbffffefb4, s.v    = 'a'
4 &up->v = 0x8049760, up->v = 'b'
5 f      = 0x8048414
6 ip      = 0xbffffef68, *ip    = 0xbffffefe4, **ip    = 1
7 ip+1   = 0xbffffef6c, ip[1] = 0x804965c, *ip[1] = 15
8 &s.v   = 0xbffffef74, s.v    = 'a'
9 &up->v = 0x8049770, up->v = 'b'
10 f     = 0x8048414
```

Мы видим, что эта функция исполняется дважды: сначала выполняется ее прямой вызов из оператора `test` (строка 42 листинга 3.63), затем следует непрямой, рекурсивный вызов (строка 36 листинга 3.63). Мы видим, что все распечатанные значения указателя соответствуют адресам. Те из них, которые указывают на адреса, начинаются со значения `0xbffffef`, указывают на ячейки стека, в то время как остальные являются адресами глобальной памяти (`0x804965c`), частью исполняемого кода (`0x8048414`) или ячейками кучи (`0x8049760` и `0x8049770`).

Экземпляры массива `ip` создаются дважды — по одному разу для каждого обращения к функции `fun`. Второе значение (`0xbffffef68`) меньше, чем первое (`0xbffffefa8`), поскольку стек наращивается книзу. Однако содержимое массива в обоих случаях остается тем же самым. Элемент 0 (`*ip`) есть указатель на переменную `x` в стековом фрейме для функции `test`. Элемент 1 есть указатель на глобальную переменную `g`.

Легко видеть, что структура `s` создается дважды и оба раза в стеке, в то время как указателем на объединение, размещенного в динамически распределяемой области памяти, служит переменная `up`.

И, наконец, переменная *f* есть указатель на функцию *fun*. В программном коде (листинг 3.65), полученном обратным ассемблированием, мы обнаруживаем следующий фрагмент, служащий начальным кодом функции *fun*:

Листинг 3.65. Начальный код

```

1 108048414 <fun>:
2 8048414: 55           push  %ebp
3 8048415: e9 e5        mov    %esp,%ebp
4 8048417: 83 ec 1c     sub    $0x1c,%esp
5 804841a: 57           push   %edi

```

Значение 0x8048414, распечатанное для указателя *f*, есть точный адрес первой команды программного кода функции *fun*.

Передача функции параметров

Другие языки программирования, такие как Паскаль, предлагают два способа передачи параметров процедуре — по значению (value), когда вызывающая процедура предоставляет фактические значения параметров, и по ссылке (reference), когда вызывающая процедура предлагает указатели на фактические значения. В языке С все параметры передаются по значению, но мы можем имитировать результат ссылочного параметра, задавая явным способом указатель на значение и передавая этот указатель процедуре. Мы видели, как это делается на примере функции *fun* с параметром *xp*. Во время начального вызова *fun(&x)* (строка 42) этой функции передается ссылка на локальную переменную *x* в функции *test*. Значение этой переменной уменьшается при каждом вызове функции *fun* (строка 35), благодаря чему рекурсия прекращается после двух обращений.

Язык C++ восстановил понятие передачи параметров по ссылке, но многие считают, что это была ошибка.

3.12. Использование отладчика GDB

GNU-отладчик GDB предлагает ряд полезных средств для поддержки оценки и анализа программ машинного уровня во время их выполнения. В примерах и упражнениях, приводимых в данной книге, мы делаем попытку дать оценку поведению программы на основании анализа программного кода. Использование отладчика GDB позволяет проводить изучение программы, наблюдая за ее исполнением и сохраняя при этом значительный контроль за ее исполнением.

В табл. 3.21 представлены образцы некоторых команд отладчика GDB, которые будут вам полезны при работе с программами машинного уровня, ориентированными на архитектуру IA32. Очень полезно сначала выполнить программу OBJDUMP, чтобы получить версию обратного ассемблера этих программ. В основу наших примеров положено выполнение GDB на файле *prog*, описание которого, а также дизассембли-

рованная версия представлены в листинге 3.5. Мы запускаем GDB в работу посредством следующей командной строки:

```
unix> gdb prog
```

Общая схема заключается в том, что контрольные точки расставляются в местах программы, представляющих интерес. Их можно установить сразу после ввода функции или по адресу программы. Когда во время исполнения программа выходит на одну из контрольных точек, она останавливается и передает управление пользователю. Находясь в контрольной точке, мы можем исследовать различные регистры и ячейки памяти различных форматов. Мы можем также войти в режим пошагового исполнения программы, каждый раз выполняя всего лишь несколько команд, либо сразу переходить к следующей контрольной точке.

Как следует из нашего примера, отладчик GDB обладает не совсем четким синтаксисом команд, однако оперативная справочная информация (вызываемая из GDB посредством команды help) позволяет преодолеть это недостаток.

Таблица 3.21. Команды отладчика

Команда	Результат выполнения
Запуск и останов	
quit	Выйти из отладчика GDB
run	Осуществить прогон вашей программы (в этом месте введите аргументы командной строки)
kill	Останов вашей программы
Точки прерывания	
break sum	Установите точку прерывания на входе функции sum
break 0x80483c3	Установите точку прерывания по адресу 0x80483c3
delete 1	Удалить точку прерывания 1
delete	Удалить все точки прерывания
Выполнение	
stepi	Выполните одну команду
stepi 4	Выполните четыре команды
nexti	Аналогично stepi, но выполняется через вызов функции
continue	Возобновить выполнение
finish	Выполнять, пока не будет получен возврат текущей функции

Таблица 3.21 (окончание)

Команда	Результат выполнения
Анализ программного кода	
disas	Выполнить обратное ассемблирование текущей функции
disas sum	Выполнить обратное ассемблирование функции sum
disas 0x80483b7	Выполнить обратное ассемблирование функции в окрестности адреса 0x80483b7
disas 0x80483b7 0x80483b7	Выполнить обратное ассемблирование программного кода в диапазоне адресов
print /x \$eip	Выполнить распечатку программы в шестнадцатеричном формате
Анализ данных	
print %eax	Распечатать содержимое регистра %eax в десятичном формате
print /x %eax	Распечатать содержимое регистра %eax в шестнадцатеричном формате
print /t %eax	Распечатать содержимое регистра %eax в двоичном формате
print 0x100	Распечатать 0x100 в десятичном представлении
print /x 555	Распечатать 555 в шестнадцатеричном представлении
print /x) (%ebp+8)	Распечатать содержимое регистра (%ebp+8) в десятичном представлении
print *(int *) 0xbfffff890	Распечатать целое число по адресу 0xbfffff890
print *(int *) (%ebp+8)	Распечатать целое число по адресу (%ebp+8)
x/2w 0xbfffff890	Просмотреть два (4-байтовых слова), начинающихся по адресу 0xbfffff890
x/20 b sum	Просмотреть первые 20 байтов функции sum
Полезная информация	
info frame	Информация о текущем стековом фрейме
info registers	Значения всех регистров
help	Предоставление информации об отладчике GDB

3.13. Ссылки на ячейку в памяти и переполнение буферов

Выше мы видели, что язык C не производит никаких граничных проверок ссылок на массивы, и что локальные переменные хранятся в стеке наряду с информацией о состоянии, такой как, например, значения регистров и указатели возврата. Такое сочетание может привести к серьезным программным ошибкам, когда состояние, сохраняемое в стеке, оказывается запорченным по причине записи элементов массива, значения которых выходят за допустимые границы. Когда после этого программа делает попытку перезагрузить регистр или выполнить команду `ret` с подобным запорченным состоянием, это может привести к серьезным последствиям.

Чаще других причиной возникновения запорченного состояния является *переполнение буфера* (buffer overflow). Довольно часто символьные массивы помещаются в стек для запоминания строки, но при этом размер строки превосходит пространство, выделенное под этот массив. Это можно проиллюстрировать на примере следующей программы (листинг 3.66).

Листинг 3.66. Пример программы

```
1 /* Реализация библиотечной функции gets() */
2 char *gets(char *s)
3 {
4     int c;
5     char *dest = s;
6     while ((c = getchar()) != '\n' && c != EOF)
7         *dest++ = c;
8     *dest++ = '\0'; /* Окончание строки */
9     if (c == EOF)
10        return NULL;
11    return s;
12 }
13
14 /* Считывание строки ввода и запись ее назад */
15 void echo()
16 {
17     char buf[4]; /* Путь слишком мал */
18     gets(buf);
19     puts(buf);
20 }
```

Приведенный в листинге 3.66 программный код представляет собой реализацию библиотечной функции `gets`, он показывает, источником каких серьезных проблем может стать эта функция. Она считывает строку со стандартного устройства ввода и останавливается только в тех случаях, когда ей встретится символ новой строки или

некоторая сбойная ситуация. Она копирует строку в место памяти, указанное аргументом `s`, и завершает вводимую строку символом пробела. Мы покажем, как пользоваться функцией `gets`, когда будем рассматривать функцию `echo`, которая просто считывает строку со стандартного устройства ввода и передает ее на стандартное устройство вывода (рис. 3.7).

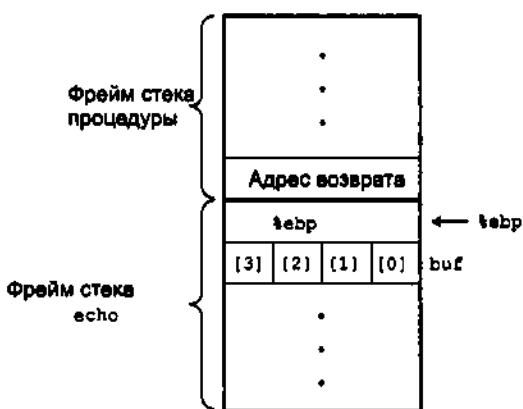


Рис. 3.7. Организация стека для функции `echo`

Проблема с функцией `gets` состоит в том, что для нее не существует способа определить, достаточно ли ей отведено пространства памяти для хранения всей строки. В нашем примере с функцией `echo` мы намеренно выбрали буфер очень маленьким: он рассчитан на хранение всего лишь четырех символов. Любая строка, длина которой больше трех символов, вызовет запись за пределами установленных границ.

Анализ порции кода функции `echo` на языке ассемблера позволяет понять, как организован стек (листинг 3.67).

Листинг 3.67. Организация стека

```

1 echo:
2 pushl %ebp           Сохранить \%ebp в стеке
3 movl %esp,%ebp       Выделить память под стек
4 subl $20,%esp         Сохранить \%ebx
5 pushl %ebx           Выделить больше пространства под стек
6 addl $-12,%esp        Вычислить адрес buf как \%ebp-4
7 leal -4(%ebp),%ebx   Протолкнуть buf в стек
8 pushl %ebx           Вызвать функцию gets
9 call gets

```

Мы можем видеть в этом примере, программа распределила всего 32 байта (строки 4 и 6) под локальную память. Однако адрес массива на 4 байта меньше значения, хранящегося в регистре `%ebp` (строка 7 листинга 3.67). На рис. 3.7 показана полученная

структура стека. Легко видеть, что любая запись в ячейки от `buf[4]` до `buf[7]` вызывает искажение значения регистра `%ebp`. Позднее, когда программа попытается восстановить это значение как указатель фрейма, все последующие ссылки на стек будут неправильными. Любые записи в ячейки от `buf[7]` до `buf[11]` приводят к тому, что адрес возврата будет запорчен. Когда на завершающей стадии выполнения функции будет выполнена команда `ret`, программа "возвратится" по неправильному адресу. Как показывает этот пример, переполнение буфера приводит к серьезным искажениям в поведении программы.

Составленный нами код функции `echo` простой, но сырой. Более совершенная версия предусматривает использование функции `fget`, которая в качестве аргумента использует максимальное число считываемых байтов. Упражнение 3.37 ставит перед вами задачу написать эхо-функцию, которая может обрабатывать входные строки произвольной длины. В общем случае, использование функции `gets` или любой другой функции, которая способна переполнить память, считается дурным тоном в программировании. Компилятор с языка C даже выдает следующее сообщение об ошибке при компилировании файла, содержащего обращение к функции `gets`: функция `gets` опасна и не должна быть использована (листинги 3.68—3.69).

Листинг 3.68. Программный код на С функции

```

1  /* Этот код характеризуется очень низким качеством.
2   Он служит иллюстрацией плохой практики программирования
3   См. практическую задачу 3.24 */
4  char *getline()
5  {
6      char buf[8];
7      char *result;
8      gets(buf);
9      result = malloc(strlen(buf));
10     strcpy(result, buf);
11     return(result);
12 }
```

Листинг 3.69. Код, полученный обратным ассемблированием

```

1  08048524 <getline>:
2  8048524: 55                      push   %ebp
3  8048525: 89 e5                   mov    %esp,%ebp
4  8048527: b3 ec 10                sub    $0x10,%esp
5  804852a: 56                      push   %esi
6  804852b: 53                      push   %ebx
7  Диаграмма стека в этой точке
8  804852c: b3 c4 f4                add    $0xffffffff4,%esp
9  804852f: bd 5d f8                lea    0xfffffff8(%ebp),%ebx
```

```

9 8048532: 53          push  %ebx
10 8048533: e8 74 fe ff ff    call   80483ac <_init+0x50>

```

УПРАЖНЕНИЕ 3.24

Тексты программ листингов 3.68 и 3.69 представляют собой реализацию (невысокого качества) функции, которая считывает временные значения со стандартного устройства ввода, копирует строку во вновь распределенную память и возвращает указатель на результат.

Рассмотрим следующий сценарий: вызывается процедура `getline`, ее адрес возврата есть `0x8048643`, содержимое регистра `%ebp` есть `0xbfffffc94`, содержимое регистра `%esi` есть `0x1`, а содержимое регистра `%ebx` есть `0x2`. Вы вводите с клавиатуры строку `"012345678901"`. Программа останавливается из-за ошибки сегментации. Вы осуществляете прогон отладчика GDB и выясняете, что ошибка произошла во время выполнения команды `ret` программы `getline`.

1. Заполните приводимую ниже таблицу, поместив в нее всю известную вам информацию о состоянии стека после выполнения команды в строке 6 кодов ассемблерного кода. Пометьте соответствующим образом числовые значения, хранящиеся в стеке (например, "адрес возврата"), и укажите в ячейке справа их шестнадцатеричные значения (если таковые известны). Каждая ячейка соответствует 4 байтам. Укажите позицию регистра `%ebp`.
2. Внесите изменения в вашу диаграмму с тем, чтобы показать последствия вызова функции `gets` (строка 10).
3. По какому адресу программа возвратит себе управление?
4. Значение (значения) какого регистра (регистров) окажется запорченным, когда программа вернет себе управление?
5. Наряду с возможностью переполнения буфера, еще какие два недостатка характерны для программы `getline`?

Последствия будут гораздо более разрушительными, чем переполнение буфера, если заставить программу выполнять не своюственную ей функцию. Это один из широко распространенных способов испытания защитных средств системы в компьютерных сетях. Обычно в программу вводится строка, которая содержит представленные в байтах рабочие программные коды, получившие название *подставной код* (*exploit code*), плюс несколько дополнительных байтов, которые затирают в буфере указатель возврата, помещая на его место указатель на программный код в буфере. Результатом выполнения команды `ret` является ее безусловный переход на подставной код.

Как одна из форм подобной атаки системы защиты, подставной код затем обращается к системе с запросом запустить в работу программную оболочку, которая предоставляет злоумышленнику возможность пользоваться некоторым множеством функций операционной системы. В другой форме подставной код выполняет некоторые другие несанкционированные задачи, исправляет искажения, нанесенные стеку, а затем

выполняет команду `ret` второй раз, благодаря чему достигается (кажущаяся) иллюзия нормального возвращения управления вызывающей программе.

В качестве примера можно привести знаменитого программного червя, появившегося в сети Internet в ноябре 1988 года, который использовал четыре различных способа получения доступа к многим компьютерам. Одним из них была атака с переполнением буфера `fingerd`, который обслуживает требования со стороны команды `finger`. При выполнении команды `finger` в соответствующей строке программы-червя могла вызвать у демона на дистанционном сайте переполнение буфера и выполнить программный код, который обеспечивал доступ к дистанционной системе. Как только червь получал доступ к дистанционной системе, он получал возможность тиражировать себя и потреблять практически все вычислительные ресурсы машины. В результате сотни машин были фактически парализованы, пока специалисты по обеспечению безопасности не нашли способа уничтожить червя. Автор этой программы был изобличен и подвергся судебному преследованию. Он был приговорен к трем годам заключения условно, к 400 часам общественно полезных работ и к штрафу в 10 500 долларов. Однако и по сей день находятся люди, ищущие слабые места в системах, которые позволили бы им проводить атаки с переполнением буферов. Все это подчеркивает необходимость более тщательного программирования. Любые интерфейсы со внешними системами должны быть "пуленепробиваемыми", чтобы никакие действия внешних агентов не могли заставить систему изменить линию своего поведения.

Черви и вирусы

Как черви, так и вирусы представляют собой порции программных кодов, которые стремятся распространяться на другие компьютеры. Согласно определению, сформулированному Спаффордом (Spafford) [73], *червь (worm)* есть программа, которая может самостоятельно запускаться в работу и распространять вполне работоспособную версию самой себя на другие машины. *Вирус (virus)* есть порция программных кодов, которая добавляет себя к другим программам, в том числе и к операционным системам. Она не может исполняться самостоятельно. В средствах массовой информации термин "вирус" используется как общее название различных стратегий для распространения злоумышленного программного кода среди систем, так что часто можно слышать, что люди называют вирусом то, что правильнее было назвать "червем".

В упр. 3.38 вы можете получить определенные навыки в проведении атаки типа переполнения буфера. Обратите внимание на то обстоятельство, что мы отнюдь не повторствуем тем или иным методам получения несанкционированного доступа к системам. Вторжение в компьютерную систему подобно незаконному проникновению в чужую квартиру — это преступное деяние, даже если совершивший его неставил перед собой преступных целей. Мы приводим здесь это упражнение по двум причинам: во-первых, это требует глубоких знаний программирования на машинном языке, при этом нужно иметь представление об организации стека, о порядке следования байтов и о кодах команд. Во-вторых, показав, как развивается атака с использованием переполнения буферов, мы надеемся, что вы осознаете всю важность написания программных кодов, которые не допускают проведения такого вида атак.

Битва корпорации Microsoft посредством переполнения буферов

В июле 1999 года корпорация Microsoft представила свою новую систему IM (Instant messaging — система мгновенной передачи сообщений), клиенты которой могли взаимодействовать с популярными серверами AOL (America Online). Однако уже через месяц пользователи внезапно и самым таинственным образом потеряли способность осуществлять интерактивную переписку с пользователями. Корпорация Microsoft выпустила обновленные клиентские программы, которые возобновили оказание услуг системе AOL IM, однако буквально через несколько дней эти клиенты также перестали работать. Каким-то образом компании AOL удалось узнать, что один из пользователей эксплуатировал версию AOL клиента IM, несмотря на то что Microsoft периодически пыталась воспроизвести протокол AOL IM в своих клиентских программах.

Программный код клиентов был уязвим по отношению к атакам типа переполнения буферов. Вполне возможно, что компания AOL заняла такую позицию в отношении этой проблемы непреднамеренно, ибо она использовала эту программную ошибку для разоблачения мошенников посредством атаки на клиента в то время, когда пользователь регистрировался в системе. Использовавшийся компанией AOL программный код выбирал небольшое число ячеек из образа клиента в памяти, упаковывал их в сетевой пакет и отсыпал его обратно на сервер. Если сервер не получал такой пакет, или полученный сервером пакет не соответствовал ожидаемому "отпечатку" клиента AOL, то сервер делал вывод о том, что данный клиент не является клиентом AOL, и отказывал ему в доступе. Таким образом, если другие клиенты, такие как, например, клиенты Microsoft, предпринимали попытку доступа к серверам AOL IM, они должны были не только вобрать в себя программную ошибку переполнения буферов, которая была характерна для клиентов AOL, но и иметь идентичный двоичный код и данные в соответствующих ячейках памяти. Но коль скоро клиенты AOL хранили соответствующее содержимое в этих ячейках и распределяли новые версии своих клиентских программ заказчикам, они легко могли внести такие изменения в подставные коды, чтобы они выбирали другие ячейки в образе клиента в памяти. Это была самая настоящая война, которую не-клиенты AOL не могли выиграть.

Этот эпизод имел целый ряд неожиданных поворотов и осложнений. Информация об ошибке в клиентских программах и о том, как AOL их использовала, впервые стала достоянием гласности, когда человек по имени Фил Бакинг (Phil Bucking), отрекомендовавшийся как независимый консультант, послал ее по электронной почте Ричарду Смиту (Richard Smith), известному специалисту по защите информации. Смит провел некоторые исследования и определил, что источником сообщения электронной почты была корпорация Microsoft. Позже Microsoft признала, что один из ее служащих послал это сообщение Microsoft. Другая сторона этого конфликта, компания AOL, никогда не признавала наличия в ее программных продуктах и использовании программной ошибки, даже несмотря на то, что Джеффом Чепеллом (Geoff Chapell, Австралия) были представлены убедительные доказательства.

Таким образом, кто нарушил и чей кодекс поведения был нарушен в этом инциденте? Во-первых, компания AOL не принимала на себя обязательств отрывать свою систему для не клиентов AOL, следовательно, их блокирующие мероприятия в отношении корпорации Microsoft не были противозаконными. С другой стороны,

использование переполнения буферов представляется сомнительным бизнесом. Небольшая программная ошибка могла полностью вывести из строя компьютеры клиентов, к тому же она делала клиентские системы более уязвимыми для атак со стороны внешних агентов (хотя прямых доказательств того, что они имели место, не было). Корпорация Microsoft поступила бы правильно, если бы публично заявила о намерениях компании AOL использовать переполнение буферов. Однако уловка, предпринятая их представителем Филом Бакингом, оказалась неудачной попыткой распространить эту информацию, как с этической, так и с позиций связи с общественностью.

3.14. Коды с плавающей точкой

Набор инструкций для манипулирования значениями в формате с плавающей точкой является одним из наименее элегантных свойств архитектуры IA32. В первых машинах компании Intel операции со значениями с плавающей точкой выполнялись отдельным *сопроцессором* (coprocessor), устройством с собственными регистрами и вычислительными мощностями, которое выполняло некоторое подмножество команд. Этот сопроцессор был реализован в виде отдельных плат, которым были присвоены названия 8087, 80287 и i387, которые служили приложениями, соответственно, процессоров 8086, 80286 и i386. Мощность плат этого поколения технических средств была недостаточна, чтобы устанавливать на одной плате главный процессор и сопроцессор с плавающей точкой. Кроме того, маломощные машины просто обходятся без операций с плавающей точкой и реализуют их с помощью программного обеспечения (с исключительно низким быстродействием). Начиная с модели i486, аппаратура, выполняющая операции над величинами в формате плавающей точки, стала неотъемлемой частью центрального процессора архитектуры IA32.

Первый сопроцессор 8087 с большой помпой был представлен в 1980 году. Это было первое устройство FPU (Floating point unit, устройство для выполнения операций с плавающей точкой) на одной плате и первая реализация того, что сегодня известно как плавающая точка IEEE. Функционирующее как сопроцессор устройство перехватывает выполнение операций с плавающей точкой после того, как эти операции достанутся главному процессору. Между главным процессором и устройством FPU минимальная связь. Передача данных с одного процессора на другой требует, чтобы процессор-отправитель записывал данные в память, а принимающий процессор читал эти данные. К тому же, технология компиляции в 1980 году была намного проще, чем сегодня. Многие свойства архитектуры IA32, ориентированные на обработку величин с плавающей точкой, представляют собой трудно решаемую задачу для оптимизирующих компиляторов.

3.14.1. Регистры с плавающей точкой

Устройство для выполнения операций с плавающей точкой содержит восемь регистров с плавающей точкой, однако они, в отличие от обычных регистров, трактуются как неглубокий стек. Регистры нумеруются как %reg(0), %reg(1) и т. д. до %reg(7),

при этом в вершине стека находится `%reg(0)`. Если в стек проталкиваются более 8 значений, те, которые находятся внизу, просто исчезают.

Вместо прямой индексации регистров большая часть арифметических команд извлекает свои аргументы из стека, вычисляют результат, а затем проталкивают результат в стек. В 1970 году стековая архитектура считалась передовой идеей, поскольку стеки представляют собой простой механизм для вычисления арифметических команд, к тому же они позволяют проводить очень плотное кодирование команд. По мере совершенствования технологии компиляции и в связи с тем, что память, необходимая для кодирования программ, более не считается критическим ресурсом, эти качества потеряли свою былою важность. Для разработчиков компиляторов было бы намного удобнее, если в их распоряжении имелся больший набор удобных для использования регистров для выполнения операций с плавающей точкой.

Архитектуры, в основу которых были положены стеки, считались в семидесятых годах прошлого столетия передовыми идеями, поскольку они предлагали простой механизм вычисления арифметических операций, при этом они позволяют выполнять интенсивное использование памяти программой. Достижения в компьютерной технологии, а также тот факт, что память, необходимая для кодирования программ, перестала быть критическим ресурсом, эти достоинства отошли на задний план. Программистов, разрабатывающих программы, выполняющие обработку чисел с плавающей точкой, больше бы устроил более широкий набор регистров для работы с числами с плавающей точкой.

Другие языки программирования, использующие регистры

Интерпретаторы, использующие стеки, широко применяют в качестве посредников между языком программирования высокого уровня и его отображения на фактическую машину. Другими примерами блока вычислений, интенсивно использующих стеки, являются коды Java с байтовой организацией и язык PostScript форматирования страниц.

Если регистры для работы с числами с плавающей точкой организованы в виде стека ограниченных размеров, то компиляторам трудно использовать эти регистры для хранения локальных переменных в процедурах, которые вызывают другие процедуры. Мы уже видели, что при хранении целочисленных локальных переменных некоторые регистры общего назначения могут быть использованы для хранения параметров вызывающей процедуры и, следовательно, для сохранения локальных переменных на протяжении времени исполнения вызова процедуры. Такой вид использования невозможен для регистра плавающей точки в архитектуре IA32, поскольку его идентификация меняется по мере проталкивания данных в стек и выталкивания их из стека. В самом деле, операция заталкивания приводит к тому, что содержимое регистра `%st(0)` после этого становится содержимым регистра `%st(1)`.

С другой стороны, иногда желательно рассматривать регистры для работы с числами с плавающей точкой как настоящие регистры, чтобы при каждом вызове процедура могла заталкивать в них свои локальные переменные. К сожалению, такой подход быстро приводит к их переполнению, поскольку они могут вместить всего лишь восемь значений. Чтобы решить эту проблему, компиляторы генерируют программный

код, который сохраняет каждое значение с плавающей точкой в стек главной программы, прежде чем будет вызвана очередная процедура, а по возвращении из этой процедуры он извлекает эти значения из памяти. Но в этом случае возникает трафик памяти, который может ухудшить производительность программы.

Как показано в разд. 2.4.6, регистры IA32 для работы с числами с плавающей точкой содержат по 80 разрядов каждый. Они кодируют числа в формате с повышенной точностью (extended precision), см. упр. 2.58. Все числа одинарной и двойной точности преобразуются в этот формат во время их загрузки из памяти в регистры с плавающей точкой. Арифметические операции всегда выполняются с двойной точностью. Числа переводятся из формата с повышенной точностью в формат с одинарной или двойной точности в зависимости от того, в каком формате они хранились в памяти.

3.14.2. Вычисления выражений с помощью стека

Чтобы понять, как архитектура IA32 использует регистры для работы с числами с плавающей точкой в качестве стека, рассмотрим вычисления с использованием стека на более высоком уровне абстракции. Предположим, что у нас есть арифметическое устройство, которое использует стек для хранения промежуточных результатов вычислений, с набором команд, представленных в табл. 3.22. Например, так называемая запись RPN (Reverse Polish Notation, обратная польская запись), используемая в карманных калькуляторах, обладает этим свойством. В дополнение к этому стеку, рассматриваемое арифметическое устройство имеет память, которая может принимать на хранение значения, к которым мы обращаемся по имени, такие как *a*, *b* и *x*. Как следует из табл. 3.22, мы заталкиваем значения из памяти в этот стек посредством команды *load*. Операция *storep* выталкивает из стека элемент, находящийся в его вершине и сохраняет результат в памяти. Унарная операция, такая как *neg* (отрицание), использует элемент в вершине стека как свой аргумент и на его место записывает результат. Такие бинарные операции, как *addp* и *multp*, используют два аргумента в вершине стека в качестве своих аргументов. Они выталкивают оба аргумента из стека, а затем заталкивают результат операции обратно в стек. Мы используем суффикс *p* с операциями сохранения, сложения, вычитания, умножения и деления с тем, чтобы подчеркнуть тот факт, что эти команды выталкивают из стека свои аргументы.

Таблица 3.22. Гипотетический набор команд, использующих стек

Команда	Результат выполнения
<i>load S</i>	Протолкнуть значение из <i>S</i> в стек
<i>storep D</i>	Вытолкнуть из вершины стека элемент и сохранить его в <i>D</i>
<i>neg</i>	Взять отрицание элемента в вершине стека
<i>addp</i>	Вытолкнуть из вершины стека два элемента; протолкнуть в стек их сумму
<i>subp</i>	Вытолкнуть из вершины стека два элемента; протолкнуть в стек их разность

Таблица 3.22 (окончание)

Команда	Результат выполнения
multp	Вытолкнуть из вершины стека два элемента; протолкнуть в стек их произведение
divp	Вытолкнуть из вершины стека два элемента; протолкнуть в стек их отношение

В качестве примера рассмотрим выражение $x = (a-b) / (-b+c)$. Мы могли бы перевести это выражение в программный код следующим образом. Наряду с каждой строкой программного кода мы покажем содержимое стека, построенного на базе регистров, предназначенных для работы с числами с плавающей точкой. В соответствии с заключенными выше соглашениями, наращивание стека происходит вниз, так что вершину стека следует искать в нижней части диаграммы на рис. 3.8.

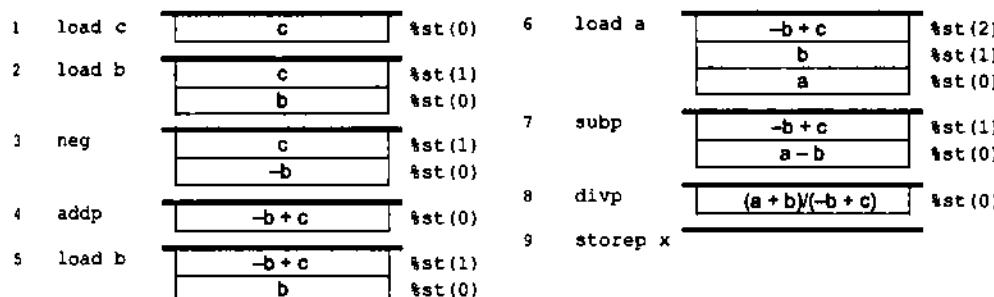


Рис. 3.8. Диаграммы стека

Как показывает этот пример, существует естественная рекурсивная процедура преобразования арифметического выражения в коды стеков. В применяемой нами форме записи используется четыре типа выражения, подчиняющихся следующим правилам преобразования:

- Ссылка на переменную вида *Var*. Этот вид ссылки применяется в команде *load Var*.
- Унарная операция вида *-Expr*. Эта операция реализуется следующим образом: сначала генерируется код для *Expr*, после чего следует команда *neg*.
- Бинарная операция вида *Expr₁ + Expr₂*, *Expr₁ - Expr₂*, *Expr₁ * Expr₂* или *Expr₁ / Expr₂*.

Эти операции реализуются путем генерации кода для *Expr₂*, за которым следует код *Expr₁*, а за ними идет одна из команд *addp*, *subp*, *multp* или *divp*.

- Присваивание вида *Var = Expr*. Эта операция реализуется следующим образом: сначала генерируется код для выражения *Expr*, за которым следует операция *storep Var*.

В качестве примера рассмотрим выражение $x = a - b/c$. Поскольку операция деления имеет преимущество перед операцией вычитания, в этом выражении можно расставить скобки следующим образом: $x = a - (b/c)$. Соответствующая рекурсивная процедура выполняется в следующей последовательности:

1. Постройте код для $Expr = a - (b/c)$.

- Постройте код для $Expr_2 = b/c$: код для $Expr_2 = c$, используя для этой цели команду `load c`, код для $Expr_1 = b$, используя для этой цели команду `load b`, команду `divp`.
- Постройте код для $Expr_1 = a$, используя с этой целью команду `load a`.
- Постройте команду `subp`.

2. Постройте команду `storep x`.

Теперь мы получаем программу, использующую стек, следующим образом (рис. 3.9):

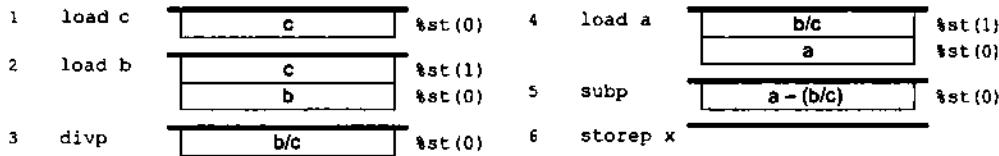


Рис. 3.9. Диаграммы стека рекурсивной процедуры

УПРАЖНЕНИЕ 3.25

Постройте коды, использующие стек, для выражения $x = a * b / c * - (a + b * c)$.

Начертите диаграмму содержимого стека для каждой операции вашей программы. Не забывайте соблюдать правила языка C, определяющие приоритеты операций и свойство ассоциативности.

Вычисление выражений с помощью стека становится более сложным, если мы хотим использовать результат тех или иных вычислений многократно. Например, рассмотрим выражение $x = (a+b)*(-(a+b)+c)$. Для большей эффективности мы вычислим выражение $a+b$ только один раз, однако команды, манипулирующие стеком, не предоставляют нам возможности хранить конкретное значение после того, как оно было использовано. Имея в своем распоряжении команды, фигурирующие в списке, представленном в табл. 3.22, мы в силу этого обстоятельства должны использовать сохранить промежуточный результат $a+b$ в одной из ячеек памяти, скажем, в t , и извлекать это значение из памяти всякий раз, когда мы его используем. Мы получаем следующий программный код, представленный на рис. 3.10.

Недостаток такого подхода заключается в том, что возникает необходимость в памяти генерировать дополнительный трафик, даже если регистровый стек имеет достаточно места для хранения промежуточных результатов. Устройства IA32 с плавающей точкой способны избежать такого неэффективного использования регистров,

вводя такие варианты арифметических операций, которые оставляют свой второй операнд в стеке и которые могут использовать произвольное значение из стека в качестве своего второго операнда. В дополнение к этому вводится команда, которая может заменить элемент в вершине стека на любой другой элемент стека. И хотя эти выражения могут быть использованы для генерирования более эффективного программного кода, простой и в то же время элегантный алгоритм преобразования арифметических выражений в стековый код будет утрачен.

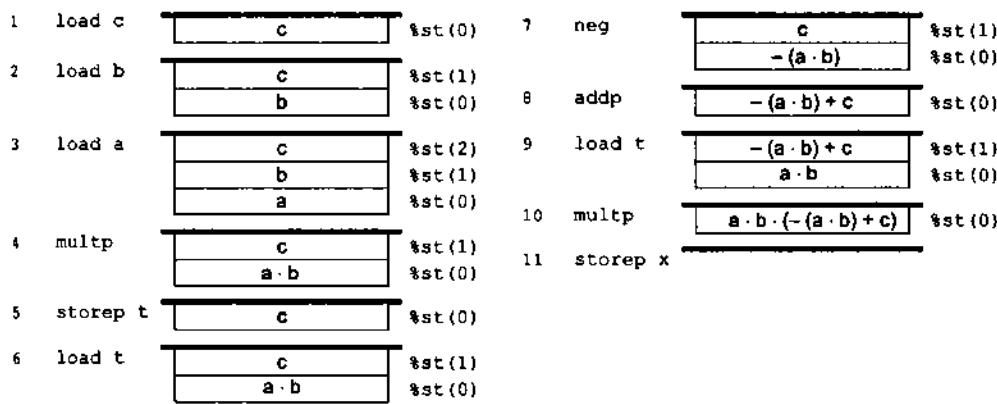


Рис. 3.10. Диаграммы стека промежуточного результата

3.14.3. Операции перемещения и преобразования данных

Для ссылок на регистры данных с плавающей точкой применяется запись $\%st(i)$, где i означает позицию относительно вершины стека. Значение i может находиться в пределах от 0 до 7. Регистр $\%st(0)$ есть элемент, который находится в вершине стека, $\%st(1)$ есть следующий элемент и т. д. На элемент в вершине стека можно ссылаться посредством ссылки $\%st$. Когда в стек заталкивается новое значение, значение $\%st(1)$ теряется. Когда из стека выталкивается значение, новое значение $\%st(7)$ предсказать невозможно. Компиляторы должны генерировать программный код, который работает в условиях ограниченного объема стека регистров.

В табл. 3.23 показан набор команд, использованный для заталкивания значений в стек для чисел с плавающей точкой. Первая группа этих команд считывает соответствующее значение из памяти, при этом аргумент $Addr$ есть адрес в памяти, заданный в одном из operandов, хранимым в памяти, в формате, представленном в табл. 3.2. Эти команды отличаются предполагаемым форматом от исходного операнда и, следовательно, числом байтов, которые должны быть считаны из памяти. Напомним, что запись $M_b[Addr]$ означает доступ к последовательности из b байтов, начинающаяся с адреса $Addr$. Эти команды преобразуют операнд в формат повышенной точности перед тем, как протолкнуть его в стек. Завершающая команда загрузки fld используется для дублирования значения в стеке. Другими словами, она проталкивает копию

`%st(i)` регистра с плавающей точкой в стек. Например, команда `fld %st(0)` проталкивает копию элемента в вершине стека в стек.

Таблица 3.23. Команды загрузки значений с плавающей точкой

Команда		Исходный формат	Исходный адрес
<code>flds</code>	<code>Addr</code>	<code>single</code>	$M_4[Addr]$
<code>fildl</code>	<code>Addr</code>	<code>double</code>	$M_8[Addr]$
<code>fldt</code>	<code>Addr</code>	<code>extended</code>	$M_{10}[Addr]$
<code>fildl</code>	<code>Addr</code>	<code>integer</code>	$M_4[Addr]$
<code>fid</code>	<code>%st(i)</code>	<code>extended</code>	<code>%st(i)</code>

Все данные в табл. 3.24 преобразованы из формата с плавающей точкой повышенной точности в целевой формат.

Таблица 3.24. Команды сохранения данных с плавающей точкой в памяти

Команда	Выталкнуть (Да/Нет)	Целевой формат	Место назначения
<code>fsts</code> <i>Addr</i>	Нет	<code>single</code>	$M_4[Addr]$
<code>fstps</code> <i>Addr</i>	Да	<code>single</code>	$M_4[Addr]$
<code>fstl</code> <i>Addr</i>	Нет	<code>double</code>	$M_8[Addr]$
<code>fstpl</code> <i>Addr</i>	Да	<code>double</code>	$M_8[Addr]$
<code>fstt</code> <i>Addr</i>	Нет	<code>extended</code>	$M_{10}[Addr]$
<code>fstpt</code> <i>Addr</i>	Да	<code>extended</code>	$M_{10}[Addr]$
<code>fistl</code> <i>Addr</i>	Нет	<code>integer</code>	$M_4[Addr]$
<code>fistpl</code> <i>Addr</i>	Да	<code>integer</code>	$M_4[Addr]$
<code>fst</code> <code>%st(i)</code>	Нет	<code>extended</code>	<code>%st(i)</code>
<code>fstp</code> <code>%st(i)</code>	Да	<code>extended</code>	<code>%st(i)</code>

В табл. 3.24 показаны команды, которые запоминают элемент из вершины стека либо в памяти, либо в другом регистре с плавающей точкой. На нем представлены как "выталкивающая" версия, которая выталкивает из стека элемент, находящийся в его вершине (аналогичная команде `storep` для нашего гипотетического вычислителя выражения в стеке), так и невыталкивающая версия, которая оставляет исходное значение в вершине стека. Аналогично командам загрузки значений с плавающей точкой, различные варианты команд генерируют результаты в разных форматах, и в силу этого обстоятельства занимают в памяти разное число байтов. Первая группа этих команд запоминает результаты в памяти. Адрес указывает на операнд с использованием формата, в котором он может храниться в памяти, т. е. в любом из тех, что

представлены в табл. 3.2. Вторая группа копирует элемент в вершине стека в какой-нибудь другой регистр с плавающей точкой.

УПРАЖНЕНИЕ 3.26

Предположим, что для следующего фрагмента программного кода регистр %eax содержит переменную *x* типа integer, и что в двух верхних элементах стека хранятся, соответственно, переменные *a* и *b*. Заполните окна в диаграмме содержимым стека после выполнения каждой команды (рис. 3.11).

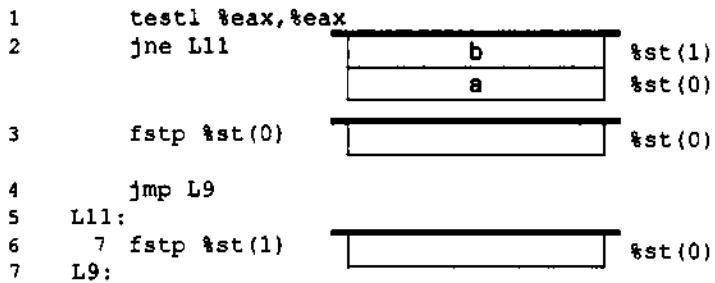


Рис. 3.11. Диаграмма стека упражнения

Представьте на языке С выражение, описывающее содержимое вершины стека, в конце этой последовательности кодов, выразив его через переменные *x*, *a* и *b*.

Завершающая операция перемещения данных с плавающей точкой позволяет двум регистрам с плавающей точкой совершать обмен содержимым. Команда fxch %st(i) осуществляет обмен содержимым регистров с плавающей точкой %st(0) и %st(i). Запись команды fxch без аргументов эквивалентна команде fxch %st(1), т. е. обмен содержимым между двумя верхними элементами стека.

3.14.4. Арифметические операции с плавающей запятой

В табл. 3.25 задокументированы некоторые наиболее часто используемые арифметические операции с плавающей точкой. Команды, включенные в первую группу, вообще не имеет operandов. Они проталкивают в стек некоторые числовые константы, представленные в формате с плавающей точкой. Существуют подобные команды для таких констант, как *π*, *e* и \log_{10} . Команды второй группы имеют один operand.

Этот operand всегда содержится в верхнем элементе стека, подобно операции псевдогипотетического вычислителя выражений в стеке. Они замещают этот элемент вычисленным результатом. Инструкции третьей группы содержат два операнда. Для каждой из этих команд существует множество различных вариантов того, как определяются эти operandы, о чём речь пойдет несколько ниже. Для некоммутативных операций, таких как вычитание и деление, существуют прямые (например, fsub) и обратные (например, fsubr) версии, так что эти аргументы могут быть использованы в любом порядке.

Таблица 3.25. Арифметические операции с плавающей запятой

Команда	Вычисление
fldz	1
fldl	0
fabs	Op
fchs	Отрицание
fcos	cos Op
fsin	sin Op
fsqrt	Квадратный корень
fadd	Сложение
fsub	Вычитание
fsubr	Вычитание
fdiv	Деление
fdivr	Деление
fmul	Умножение

В табл. 3.25 мы показываем всего лишь одну из форм операции вычитания fsub. На самом деле, эта операция выступает в нескольких вариантах, как показано в табл. 3.26. Все они вычисляют разность двух операндов: $Op_1 - Op_2$ и сохраняют результат в одном из регистров с плавающей точкой.

Таблица 3.26. Команды вычитания с плавающей точкой

Команда	Операнд 1	Операнд 2	Формат	Назначение	Вытолкнуть (Да/Нет)
fsub <i>Addr</i>	%st(0)	M ₄ [<i>Addr</i>]	single	%st(0)	Нет
fsubl <i>Addr</i>	%st(0)	M ₈ [<i>Addr</i>]	double	%st(0)	Нет
fsubt <i>Addr</i>	%st(0)	M ₁₀ [<i>Addr</i>]	extended	%st(0)	Нет
fisubl <i>Addr</i>	%st(0)	M ₄ [<i>Addr</i>]	integer	%st(0)	Нет
fsub %st(i), %st	%st(i)	%st(0)	extended	%st(0)	Нет
fsubs %st, %st(i)	%st(0)	%st(i)	extended	%st(i)	Нет
fsubp %st, %st(i)	%st(0)	%st(i)	extended	%st(i)	Да
fsubp	%st(0)	%st(1)	extended	%st(1)	Да

Наряду с простой командой `subp`, которую мы использовали в гипотетическом вычислителе выражений в стеке, в архитектуре IA32 имеются и инструкции, которые считывают свой второй operand из памяти или из одного из регистров с плавающей точкой, отличного от регистра `%st(1)`. Наряду с этим, имеются как вариант с проталкиванием, так и вариант с выталкиванием значений в стек. Первая группа команд считывает второй operand из памяти, который представлен с одинарной точностью, с двойной точностью либо в формате целого числа. Затем она преобразует этот operand в формат повышенной точности, вычитает его из элемента в вершине стека и заменяет им элемент в вершине стека. Эту операцию можно рассматривать как комбинацию загрузки значения с плавающей точкой с последующей операцией вычитания, выполненную над элементами стека.

Все команды сохраняют результат в регистре с плавающей точкой в формате повышенной точности. Команды с суффиксом `p` выталкивают верхний элемент из стека.

Вторая группа команд вычитания использует верхний элемент стека в качестве одного аргумента и некоторые другие элементы стека в качестве другого, но они отличаются друг от друга порядком следования аргументов, местом, куда направляется результат, а также тем, выталкивают ли они верхний элемент стека. Обратите внимание на то обстоятельство, что строка ассемблерного кода `fsubp` есть сокращенная запись оператора

`fsubp %st,%st(1)`

Эта строка соответствует команде `subp` нашего гипотетичного вычислителя выражения в стеке. В самом деле, он вычисляет разность двух верхних элементов стека, запоминая результат в регистре `%st(1)`, а затем выталкивает `%st(0)`, так что вычисленное значение оказывается в вершине стека.

Все бинарные операции, указанные в табл. 3.25, включены во все варианты операции `fsub`, представленные в табл. 3.26. Например, мы можем составить программный код для выражения

$x = (a-b) * (-b+c),$

используя команды IA32. Для большей ясности мы продолжаем использовать символьические имена ячеек памяти и будем предполагать, что все значения имеют двойную точность (рис. 3.12).

В качестве еще одного примера рассмотрим выражение

$x = (a+b) + (- (a+b) + c)$

Обратите внимание на то, как команда `fild %st(0)` используется с целью создания двух копий произведения $a*b$ в стеке, при этом нет необходимости запоминать значение во временной ячейке памяти (рис. 3.13).

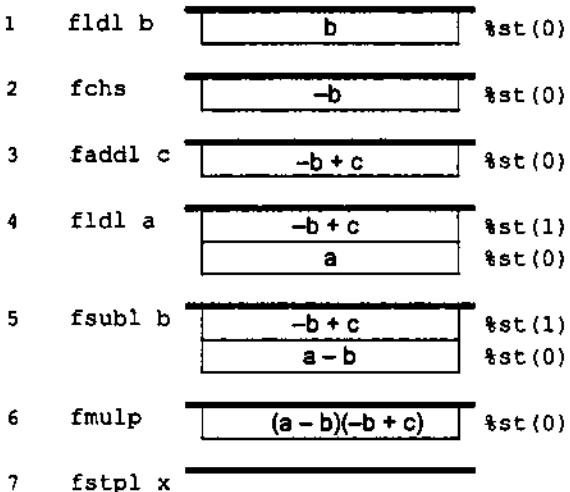


Рис. 3.12. Диаграммы стека двойной точности

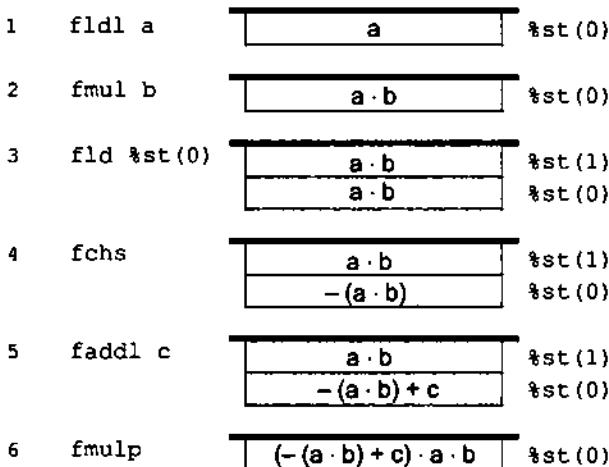


Рис. 3.13. Диаграммы стека двух копий

УПРАЖНЕНИЕ 3.27

Отобразите на диаграмме содержимое стека по выполнении каждого действия следующего программного кода (рис. 3.14).

Составьте программу этих выражений на языке С.

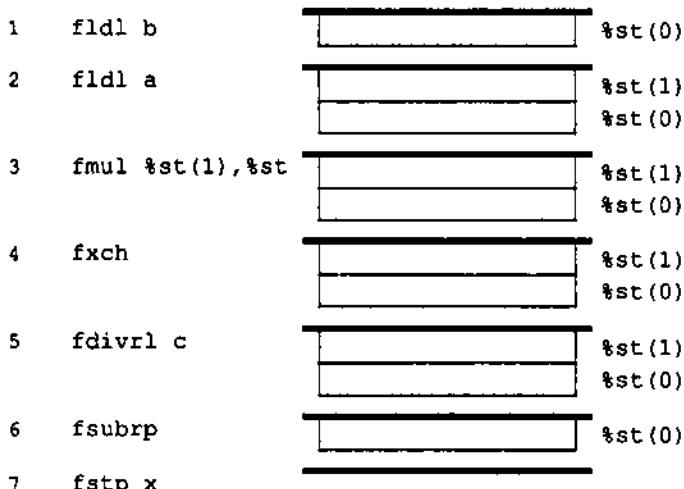


Рис. 3.14. Диаграммы стека упражнения

3.14.5. Использование значений с плавающей точкой в процедурах

Аргументы с плавающей точкой передаются в вызывающую процедуру через стек так же, как и в случае целочисленных аргументов. Каждый параметр типа float требует 4 байтов стекового пространства, в то время как каждый параметр типа double требует 8 байтов. Для функций, возвращаемые значения которых имеют тип float или double, результат помещается в вершину регистра с плавающей точкой в формате расширенной точности.

В качестве примера рассмотрим следующую функцию:

```
1 double funct(double a, float x, double b, int i)
2 {
3     return a*x - b/i;
4 }
```

Аргументы a, x, b и i имеют смещения, соответственно, 8, 16, 20 и 28 байтов относительно содержимого регистра %ebp, а именно так, как на рис. 3.15.

УПРАЖНЕНИЕ 3.28

Дана функция funct2 с аргументами a, x, b и i (их объявления отличаются от объявления функции funct), компилятор генерирует следующие коды для тела этой функции:

```
1 movl 8(%ebp),%eax
2 fldl 12(%ebp)
3 flds 20(%ebp)
```

```

4    movl %eax,-4(%ebp)
5    fildl -4(%ebp)
6    fxch %st(2)
7    faddp %st,%st(1)
8    fdivrp %st,%st(1)
9    fld1
10   flds 24(%ebp)
11   faddp %st,%st(1)
12   fsubrp %st,%st(1)

```

Возвращаемое значение имеет тип `double`. Составьте программу вычисления этой функции `funct2` на языке C. Особое внимание обратите на объявление типов аргументов.

Смещение	8	16	20	28
Содержимое	a	x	b	i

1	fildl 28(%ebp)	i	%st(0)
2	fdivrl 20(%ebp)	b/i	%st(0)
3	flds 16(%ebp)	b/i x	%st(1) %st(0)
4	fmul1 8(%ebp)	b/i a · x	%st(1) %st(0)
5	fsubrp %st,%st(1)	a · x - b/i	%st(0)

Рис. 3.15. Диаграммы стека и таблица смещения

3.14.6. Тестирование и сравнение значений с плавающей точкой

Как и в случае целочисленных значений, при определении относительных значений двух чисел с плавающей точкой используется команда сравнения с тем, чтобы установить коды условия с последующим тестированием этих кодов. Однако в случае значений с плавающей точкой эти коды условия являются частью *слова состояния вычислений с плавающей точкой* (*Floating-point status word*), 16-разрядный регистр, который содержит признаки, характеризующие состояние блока вычислений в режиме с плавающей точкой. Такое слово состояния должно быть преобразовано в целочисленное слово, после чего можно производить проверку отдельных разрядов.

Существует целый ряд различных команд сравнения величин с плавающей запятой. Все они выполняют сравнение операндов Op_1 и Op_2 , где Op_i есть элемент вершины стека. В каждой строке таблицы представлены два различных типа сравнения: **упорядоченное (ordered)** сравнение, используемое операциями $<$ и \leq , а также **неупорядоченное (unordered)** сравнение, используемое для проверки равенств. Два вида сравнений отличаются друг от друга трактовкой значений NaN , поскольку прямая зависимость между NaN и другими значениями отсутствует. Например, если переменная x есть NaN , а переменная y есть некоторое другое значение, оба выражения $x < y$ и $x = y$ должны дать в результате 0.

Различные формы команд сравнения различаются метами расположения операнда Op_2 , что характерно для различных форм команд загрузки значений с плавающей точкой и арифметических операций с плавающей точкой. И наконец, различные формы этих команд различаются числом элементов, выталкиваемых из стека после того, как сравнение будет завершено. Команды первой группы, показанные в таблице, не вносят никаких изменений в стек. Даже в случае, когда один из аргументов находится в памяти, его значение по окончании выполнения команды не попадает в стек. Операции второй группы выталкивают элемент из стека. Последняя операция выталкивает оба операнда Op_1 и Op_2 из стека (табл. 3.27).

Таблица 3.27. Закодированные результаты сравнения значений с плавающей точкой

$Op_1 : Op_2$	Двоичное представление	Десятичное представление
$>$	[00000000]	0
$<$	[00000001]	1
$=$	[00100000]	64
Неупорядочено	[00100101]	69

Слово состояния вычислений с плавающей точкой переносится в целочисленный регистр посредством команды `fNSTSW`. Операнд этой команды есть один из 16 идентификаторов регистров, показанных на рис. 3.1, например, `%ax`. Разряды слова состояния, кодирующие результаты сравнения находятся в позициях 0, 2 и 6 старшего байта слова состояния. Например, если вы используете команду `fNSTSW %ax` для передачи слова состояния, то соответствующие биты находятся в регистре `%ah`. Типичная последовательность программных кодов выбора значений этих разрядов имеет вид, как в листинге 3.70.

Листинг 3.70. Последовательность кодов

```
1  fnstsw %ax          Сохранить слово состояния с плавающей точкой в %ax
2  andb $69,%ah        Замаскировать все разряды, кроме 0, 2 и 6
```

Обратите внимание на то, что число 69_{10} в двоичном представлении имеет вид [00100101], т. е. оно имеет 1 в соответствующих разрядах. В табл. 3.28 показаны воз-

можные значения байта $\%ah$, которые могут стать значениями, полученными в результате выполнения данной последовательности программных кодов. Обратите также внимание на то, что при сравнении операндов Op_1 и Op_2 возможны четыре исхода: первый операнд больше, меньше, равен либо не сравним со вторым, при этом последний исход имеет место, когда одно из сравниваемых значений есть NaN .

В качестве примера рассмотрим следующую процедуру (листинг 3.71):

Листинг 3.71. Пример процедуры

```
int less(double x, double y)
{
    return x < y;
}
```

Компилированный код тела этой функции имеет вид, как в листинге 3.72.

Листинг 3.72. Компилированный код

1	fild 16(%ebp)	Протолкнуть у в стек
2	fcompl 8(%ebp)	Сравнение у:х
3	fnstsw %ax	Сохранить слово состояния с пл. точкой в %ax
4	andb \$69,%ah	Замаскировать все разряды, кроме 0, 2 и 6
5	sete %al	Проверка результата 0 сравнения (>)
6	movzbl %al,%eax	Младший байт в результат, остальные в 0

УПРАЖНЕНИЕ 3.29

Теперь, вставив всего лишь одну строку программных кодов на ассемблере в предыдущую последовательность, покажите, что вы можете реализовать следующую функцию:

```
1 int less(double x, double y)
2 {
3     return x > y;
4 }
```

На этом мы закончим изучение программирования на уровне ассемблера операций с числами с плавающей точкой в условиях архитектуры IA32. Даже опытные программисты находят эти коды лишеными наглядности и трудными для чтения. Операции, выполняемые с использованием стека, громоздкие действия при передаче данных о состоянии из устройства FPU (floating point unit, устройство для выполнения операций с плавающей точкой) в главный процессор, и многие другие тонкости вычислений с плавающей точкой все вместе приводят к тому, что соответствующие машинные коды становятся чрезмерно растянутыми и малопонятными. Следует отметить,

что современные процессорные устройства, изготавляемые компанией Intel и ее конкурентами, могут достигнуть достаточно высокой производительности числовых программ.

3.15. Встраивание ассемблерных кодов в программы на С

На ранней стадии развития вычислительной техники большая часть программ была написана в кодах языка ассемблера. Даже крупномасштабные операционные системы тогда писали без помощи языков высокого уровня. Для более-менее сложных программ задача их написания становилась неразрешимой. Поскольку в программных кодах на языке ассемблера практически отсутствовали какие-либо средства контроля типов, очень легко было сделать существенные ошибки, такие как, например, использовать указатель в качестве целого значения вместо разыменования этого указателя. Еще хуже, запись в кодах ассемблера фактически обрекает всю программу на исполнение только на каком-либо одном классе вычислительных машин. Переделка программ, написанных на языке ассемблера, для их исполнения на другом типе машин требует, по сути дела, тех же усилий, что и написание программы с нуля.

Написание больших программ в кодах ассемблера

Фредерик Брукс младший (Frederick Brooks, Jr.), один из первых разработчиков компьютерных систем, написал прекрасный отчет о разработке OS/360, одной из первых операционных систем для машин IBM [5], которая и сегодня служит важным предметным уроком. Тщательно изучив соответствующий материал, он стал твердым сторонником использования языков программирования высокого уровня в целях разработки крупных систем. Как ни странно, но в то же время существует активная группа программистов, которым доставляет удовольствие писать в кодах ассемблера программы для машин с архитектурой IA32. Они обмениваются между собой данными через группу новостей comp.lang.asm.x86. Большинство из них разрабатывают компьютерные игры для операционной системы DOS.

Ранние компиляторы языков программирования высокого уровня не могли генерировать высокоэффективные коды и не обеспечивали доступ к низкоуровневым объектным представлениям, как это часто требуют системные программисты. Программы, которые должны обладать максимальной производительностью и которые требуют доступа к представлениям объектов, все еще пишутся на языке ассемблера. В настоящее время, однако, появились оптимизирующие компиляторы, которые сняли с повестки дня оптимизацию производительности программы как причину необходимости ее написания в ассемблерных кодах. Программные коды, генерируемые высококачественным компилятором, в общем случае ни в чем не уступают, а во многих случаях и превосходят код, написанный вручную. Язык С практически полностью исключил машинный доступ как причину написания программ в кодах ассемблера. Возможность доступа к низкоуровневым представлениям данных посредством объединений и арифметических операций над указателями, наряду с возможностью выполнять операции над представлениями данных на уровне битов, предоставляет большей части программистов достаточный доступ

к машине. Например, практически каждая часть современной операционной системы, такой как, например, Linux, написана на C.

Тем не менее время от времени возникают моменты, когда единственный выход состоит в написании программных кодов на языке ассемблера. Это достаточно часто случается при реализации операционных систем. Например, существует множество специальных регистров, сохраняющих информацию о состоянии процессов, к которым операционная система должна иметь доступ. Существуют либо специальные команды, либо специальные области памяти, отведенные под операции ввода-вывода. Даже программисты, разрабатывающие приложения, имеют дело с такими машинными свойствами, как значения кодов условий, к которым невозможен непосредственный доступ из программы, написанной на C.

При этом возникает проблема интегрирования в программный код, состоящий преимущественно из кодов на C, с небольшими объемами программных кодов, написанных на языке ассемблера. Один из методов заключается в том, чтобы написать несколько ключевых функций в ассемблерном коде, используя те же соглашения о передаче аргументов и об использовании регистров, каким подчиняется компилятор программ на C. Эти ассемблерные функции хранятся в отдельном файле, а объединение откомпилированный код с ассемблированным кодом на языке ассемблера производит редактор связей. Например, если файл p1.c содержит коды в языке C, а файл p2.s содержит ассемблерные коды, то команда компиляции

```
unix> gcc -o p p1.c p2.s
```

обеспечивает компиляцию файла p1.c и ассемблирование файла p2.s с последующей компоновкой объектного кода, образующей исполняемую программу p.

3.15.1. Базовый встроенный ассемблер

Компилятор GCC обеспечивает возможность смешивать ассемблерные коды с кодами на C. Встроенный ассемблер позволяет пользователю вставлять код на языке ассемблера непосредственно в последовательность кодов, порожденных компилятором. Предусмотрены средства описания операндов команд, указывающие компилятору, какие регистры определяются командами ассемблера. Разумеется, получаемый при этом программный код жестко зависит от машины, поскольку машины различных типов не имеют машинно-совместимых команд. Директива asm характерна только для компилятора GCC, и в силу этого обстоятельства он несовместим со многими другими компиляторами. Тем не менее такой подход может оказаться полезным способом свести количество машинно-зависимых кодов к абсолютному минимуму.

Встроенный ассемблер задокументирован как часть информационного архива компилятора GCC. Выполнение команды info gcc на любой машине, на которой установлен GCC, приводит к построению иерархической программы считывания документов. Встроенный ассемблер документируется сначала путем следования по ссылке C Extensions, а затем по ссылке Extended Asm. К сожалению, эта документация не полна и не точна.

Основная форма встроенного ассемблера требует написания кода, который выглядит как вызов процедуры:

asm (*code-строка*);

Выражение *code-строка* означает последовательность ассемблерных кодов, заданных в виде строки, заключенной в кавычки. Компилятор вставит эту строку в генерируемый ассемблерный код, таким образом, скомпонованные блоки, полученные компилятором и поставленные пользователем, будут объединены в единое целое. Компилятор не проверяет строку на наличие ошибок, и поэтому первым сигналом о наличии проблем будет сообщение об ошибке, поступающее от ассемблера.

Мы покажем, как используется оператор `asm` на примере, в котором доступ к кодам условий может принести существенную помощь. Рассмотрим функции со следующими прототипами:

```
int ok_smul(int x, int y, int *dest);
int ok_umul(unsigned x, unsigned y, unsigned *dest);
```

Каждая из этих функций предназначена для вычисления перемножения ее аргументов *x* и *y*, а результат сохраняется в ячейке памяти, заданной аргументом *dest*. В качестве возвращаемого значения они возвращают 0, если при выполнении умножения происходит переполнение, и 1, если такое переполнение не имеет места. Мы предлагаем отдельные функции для умножения со знаком и для умножения без знака в связи с тем, что в каждой из них переполнение происходит при различных обстоятельствах.

Изучив документацию по операциям умножения `mul` и `imul` в условиях архитектуры IA32, мы видим, что обе команды устанавливают признак переноса CF (carry flag), когда переполнения имеют место. Изучив табл. 3.7, мы замечаем, что команда `setae` может быть использована для установки младшего байта в 0, когда этот признак установлен, и 1 — в противном случае. Таким образом, мы хотим вставить эту команду в последовательность кодов, порожденную компилятором. В стремлении минимизировать как количество ассемблерных кодов, так и детали анализа, мы попытаемся реализовать команду `ok_smul` с помощью следующего программного кода (листинг 3.73):

Листинг 3.73. Реализация команды

```
1 /* Первая попытка. Не работает */
2 int ok_smull(int x, int y, int *dest)
3 {
4     int result = 0;
5
6     *dest = x*y;
7     asm("setae %al");
8     return result;
9 }
```

Стратегия, применяемая в этом случае, использует тот факт, что регистр `%eax` применяется для хранения возвращаемого значения. Полагая, что компилятор использует этот регистр для заломинания значений переменной *result*, первая строка устанавливает значение *result* равным 0. Вторая строка устанавливает значение регистра `%al` равным 0, если значение регистра `%eax` не равно 0, и 1 в противном случае. Третья строка возвращает значение *result*.

ливают этот регистр в 0. Встроенный ассемблер вставит этот код, который, в свою очередь, устанавливает соответствующее значение младшего байта этого регистра, после чего этот регистр будет использован как возвращаемое значение.

К сожалению, у компилятора GCC свое представление о генерировании программных кодов. Вместо того чтобы устанавливать регистр `%eax` равным 0 в начале функции, порожденный компилятором код делает это в самом конце функции, так что эта функция всегда возвращает 0. Главная проблема заключается в том, что компилятор никак не может знать, каковы намерения программиста и как оператор ассемблирования будет взаимодействовать с остальным кодом, порожденным компилятором.

Применив метод проб и ошибок (более систематизированный подход мы разработали), мы смогли построить работающий код, но этот код далеко не идеальный.

Листинг 3.74. Стратегия использования регистра

```

1  /* Работает в ограниченном контексте */
2  int dummy = 0;
3
4  int ok_smul2(int x, int y, int *dest)
5  {
6     int result;
7
8     *dest = x*y;
9     result = dummy;
10    asm("setae %al");
11    return result;
12 }
```

Программный код в листинге 3.74 использует описанную выше стратегию, но он считывает глобальную переменную `dummy` с тем, чтобы инициализировать `result` значением 0. Обычно компиляторы более консервативны при генерировании кодов, включающих глобальные переменные, поэтому вероятность того, что порядок вычислений будет изменен, достаточно мала.

Предыдущий программный код зависит от капризов компилятора при выборе правильного поведения. Фактически он может работать только с задействованным блоком оптимизации (признак командной строки `-O`). При компилировании без оптимизации он сохраняет переменную `result` в стеке и производит поиск ее значения непосредственно перед возвращением, затирая при этом значение, установленное командой `setae`. У компилятора нет возможности знать, как включенный язык ассемблера соотносится с остальным программным кодом, поскольку мы не предоставили компилятору такой информации.

3.15.2. Расширенная форма оператора `asm`

Компилятор GCC предоставляет в распоряжение программиста расширенную версию оператора `asm`, которая позволяет ему определить, какие программные величины

должны быть использованы как операнды относительно последовательности ассемблерных кодов и какие регистры затерты этим ассемблерным кодом. Имея эту информацию, компилятор будет правильно назначать требуемые исходные значения, выполнять команды ассемблера и использовать вычисленные результаты. Он также получит затребованную им информацию о том, как используются регистры, так что важные программные величины не были затерты командами инструкциями ассемблерного кода.

Общий синтаксис расширенной последовательности кодов на языке ассемблера имеет вид:

```
asm (строка-кодов [ : выходной-список [ : входной-список
[ : список-переопределений ] ]]);
```

где в квадратных скобках обозначены необязательные аргументы. Это объявление содержит строку, описывающую последовательность ассемблерных кодов, за которой следует необязательный список выходных данных (т. е. результатов, порожденных ассемблерным кодом), входных данных (исходных значений для заданного ассемблерного кода) и регистров, значения которых переопределены заданным ассемблерным кодом. Эти списки отделены друг от друга символом двоеточия. Как показывают квадратные скобки, мы всего лишь включаем непустые списки.

Синтаксис этой строки кодов напоминает синтаксис форматирующей строки в операторе `printf`. Она состоит из некоторой последовательности команд ассемблерных кодов, отделенных друг от друга знаком точки с запятой. Операнды ввода и вывода обозначаются как `%0`, `%1` и т. д., возможно, `%9`. Операторы нумеруются в соответствии с порядком их следования сначала в списке входных данных, а затем в списке выходных данных. Такие имена регистров, как `%eax`, должны быть записаны с символом `%`, например, `%eax`.

Ниже приводится более совершенная реализация команды `ok_smul`, использующая расширенный оператор ассемблирования с тем, чтобы указать компилятору, что этот ассемблерный код генерирует значение для переменной `result` в листинге 3.75.

Листинг 3.75. Расширенный оператор

```
1  /* Использует расширенный оператор asm, чтобы получить надежный код */
2  int ok_smul3(int x, int y, int *dest)
3  {
4      int result;
5
6      *dest = x*y;
7
8      /* Вставьте следующий ассемблерный код:
9      setae %bl # Установить значение младшего байта
10     movzbl %bl, result # Нулевое расширение результата
11  */
```

```

12    asm("setae %%bl; movzbl %%bl,%0"
13    : "=r" (result) /* Выход */
14    : /* Нет ввода */
15    : "%ebx" /* Перезапись */
16 );
17
18    return result;
19 }

```

Первая инструкция языка ассемблера сохраняет результат проверки в однобайтовом регистре `%bl`. Затем вторая инструкция осуществляет нулевое расширение и копирует значение в регистр, который компилятор выбирает для хранения значения переменной `result`, указанное оператором `%0`. Список выходных данных состоит из пар значений, отделенных друг от друга пробелами. (В рассматриваемом примере имеется только одна такая пара.) Первый элемент пары есть строка, в которой указан тип операнда, в которой '`r`' указывает на целочисленный регистр, а знак равенства указывает, что ассемблерный код присваивает конкретное значение этому операнду. Второй элемент пары есть операнд, заключенный в круглые скобки. Это может быть назначаемое значение (известное в C `lvalue`). Компилятор генерирует необходимую последовательность кодов, чтобы выполнить это назначение. Список входных данных имеет тот же общий формат, в котором операнд может быть любым выражением языка C. Компилятор создает коды, необходимые для вычисления этого выражения. Список переопределений задает названия регистров (в виде строк, заключенных в кавычки), которые подвергаются переопределению.

Предыдущий код работает независимо от значений признаков компиляции. Как показывает этот пример, достаточно немного творчески подумать, чтобы написать код на языке ассемблера, который позволяет описать операнды в требуемой форме. Например, не существует непосредственных способов описания программного значения с целью его в качестве операнда назначения для команды `setae`, поскольку операнд должен быть в формате одного байта. Вместо этого, мы пишем последовательность кодов некоторого конкретного регистра, а затем используем дополнительную команду перемещения данных с тем, чтобы скопировать результирующее значение в некоторую часть состояния программы.

УПРАЖНЕНИЕ 3.30

Компилятор GCC предоставляет средства для выполнения арифметических операций с повышенной точностью. Они могут быть использованы для реализации функции `ok_smul` и обладают тем преимуществом, что допускают перенос на другие машины. Переменная, объявленная как тип `long long`, будет иметь размер, в два раза превосходящий размер обычной переменной типа `long`. Следовательно, оператор

```
long long prod = (long long) x * y;
```

вычислит полное 64-разрядное произведение чисел `x` и `y`. Воспользовавшись этим средством, напишите версию команды `ok_smul`, которая не использует операторов `asm`.

Можно ожидать, что та же последовательность программных кодов может быть использована и для реализации команды `ok_umul`, однако компилятор GCC использует команду `imull` (умножения со знаком) как для умножения со знаком, так и для умножения без знака. Она дает правильное значение для любого произведения, но при этом устанавливает признак переноса в соответствии с правилами умножения со знаком. Поэтому мы должны включить последовательность ассемблерного кода, который явным образом выполняет умножение без знака с помощью команды `mull`, как показано в табл. 3.5, следующим образом (листинг 3.76):

Листинг 3.76. Умножение без знака

```
1  /* Использует расширенный оператор asm */
2  int ok_umul(unsigned x, unsigned y, unsigned *dest)
3  {
4      int result;
5
6      /* Вставить следующий ассемблерный код:
7      movl x,%eax # Получить x
8      mull y # Умножение без знака на y
9      movl %eax,*dest # Сохранить 4 младших байта в dest
10     setae %dl # Установить значение младшего байта
11     movzbl %dl, result # Результатом будет дополнение нулями
12 */
13     asm("movl %2,%eax; mull %3; movl %eax,%0;
14     setae %dl; movzbl %dl,%1"
15 : "=r" (*dest), "=r" (result) /* Выходные данные */
16 : "r" (x), "r" (y) /* Входные данные */
17 : "%eax", "%edx" /* Перезаписи */
18 );
19
20 return result;
21 }
```

Напомним, что команда `mull` требует, чтобы один из ее аргументов находился в регистре `%eax`, а ее второй аргумент был задан как operand. Вы указываете эти подробности в операторе `asm`, используя команду `movl` с целью переноса программного значения `x` в регистр `%eax` и указывая, что программное значение `y` должно быть аргументом команды `mull`. Эта команда сохраняет 8-байтовое произведение в двух регистрах, при этом в регистре `%eax` хранятся младшие 4 байта, а в регистре `%edx` — старшие байты. Затем мы используем регистр `%edx` для построения возвращаемого значения. Как показывает рассматриваемый пример, символы запятой используются для разделения пар operandов во входных и выходных списках и имен регистров в списке перезаписей. Обратите внимание на тот факт, что вы теперь имеете возможность определить `*dest` как выходной результат второй команды `movl`, поскольку это назначает-

мое значение. Затем компилятор генерирует правильный машинный код, запоми-нающий это значение, хранящееся в этой ячейке памяти, в регистре `%eax`.

Чтобы узнать, как компилятор генерирует коды в связи с оператором `asm`, приведем программный код команды `ok_umul` в листинге 3.77.

Листинг 3.77 Генерация кодов

Установите входы для оператора `asm`

```
1 movl 8(%ebp),%ecx      Загрузить x в регистр %ecx
2 movl 12(%ebp),%ebx      Загрузить у в регистр %ebx
3 movl 16(%ebp),%esi      Загрузить dest в регистр %esi
```

Следующие ниже команды порождены оператором `asm`.

Входные регистры: `%ecx` for `x`, `%ebx` for `y`

Выходные регистры: `%ecx` for `product`, `%ebx` for `result`

```
4 movl %ecx,%eax; mull %ebx; movl %eax,%ecx;
```

```
5 setae %dl; movzbl %dl,%ebx
```

Обработка выходных результатов оператора `asm`

```
6 movl %ecx,(%esi) Сохранить произведение в dest
```

```
7 movl %ebx,%eax Представить результаты как возвращаемое значение
```

Строки 1—3 листинга 3.77 выбирают аргументы процедуры и запоминают их в ре-ги-страх. Обратите внимание на то, что он не использует регистр `%eax` или `%edx`, по-скольку мы объявили, что эти регистры будут перезаписаны. Наш оператор встроен-ного ассемблера появляется в строках 4 и 5, но вместо аргументов подставлены име-ны регистров. В частности, он использует регистры `%ecx` вместо аргумента `%2` (`x`), и `%ebx` вместо аргумента `%3` (`y`). Произведение временно будет храниться в регистре `%ecx`, в то же время он использует регистр `%ebx` вместо аргумента `%1` (`result`). Теперь строка 6 хранит произведение в ячейке `dest`, завершая обработку аргумента `%0` (`*dest`). Стока 7 копирует результат в регистр `%eax` как возвращаемое значение. Та-ким образом, компилятор генерирует не только коды, указанные вашим оператором `asm`, но и коды, устанавливающие вводы для этого оператора (строки 1—3 листин-га 3.77) и использующие выводы (строки 6—7).

И хотя синтаксис оператора `asm` не отличается наглядностью, а его использование ограничивает переносимость программных кодов, этот оператор может быть очень полезен при написании программ, которые осуществляют доступ к функциональным средствам на машинном уровне, используя минимальное количество ассемблерных кодов. Мы убедились в том, что пока мы не можем отказаться от использования ме-тода проб и ошибок, чтобы получить работающий программный код. Наилучшая стратегия заключается в том, чтобы компилировать коды с установленным переклю-чателем `-S` с последующей проверкой полученного компилирующего автокода, полу-чен ли желательный результат. Этот код должен быть проверен при различных уста-новках переключателей, например, при наличии и отсутствии признака `-O`.

3.16. Резюме

В этой главе мы заглянули за уровень абстракции, образованного языком высокого уровня, с целью получить представление о программировании на машинном уровне. Имея в своем распоряжении компилятор, который генерирует представление программы машинного уровня в компилирующем автокоде, мы получаем более подробные сведения как о компиляторе, так и о его оптимизирующих возможностях, а также о самой машине, ее типах данных и о наборе команд. В главе 5 мы увидим, что знание характеристик компилятора может помочь писать эффективные программы, интенсивно взаимодействующие с аппаратными средствами машины. Мы также столкнулись с примерами, когда абстракция языка программирования высокого уровня скрывает важные детали функционирования программ. Например, поведение программного кода с плавающей точкой может зависеть от того, содержатся ли значения в регистрах или в памяти. В главе 13 мы рассмотрим многие примеры, в условиях которых мы должны знать, находится ли программная переменная в стеке для переменных исполняемой программы, в некоторых динамически распределяемых структурах данных или в некоторых глобальных ячейках памяти. Понимание того, как программы взаимодействуют с машиной, позволяет лучше понимать, в чем заключается различие между этими двумя видами памяти.

Язык ассемблера очень отличается от программных кодов на языке С. В программах на языке ассемблера различием между типами данных минимально. Программа представляет собой последовательность команд, каждая из которых выполняет единственную операцию. Некоторые части состояния программы, такие как регистры и стек для переменных исполняемой программы, непосредственно видны программисту. Только операции низкого уровня предназначены для поддержки манипулирования данными и программного управления. Компилятор должен использовать множество различных команд, чтобы генерировать различные структуры данных и выполнять на них всевозможные операции и строить управляющие конструкции, такие как условные операторы, циклы и процедуры. Мы рассмотрели различные аспекты языка С и узнали, как компилируются программы, написанные на этом языке. Мы также видели, что отсутствие граничной проверки в С часто является причиной переполнения буферов, и в силу этого обстоятельства многие системы уязвимы в отношении несанкционированного доступа.

Мы пока исследовали только отображение языка С на архитектуру IA32, однако большая часть тех вопросов, которые удалось рассмотреть, решаются так же, как и многие другие сочетания языков программирования и машины. Например, компиляция C++ во многом подобна компиляции С. По сути дела, ранние реализации языка C++ просто выполняли преобразование типа источник-источник (source-to-source) из C++ в С и генерировали объектный код, осуществляя прогон компилятора С применительно к результату преобразования. Объекты C++ представлены структурами, подобными структурам языка С. Методы представлены указателями на программные коды, реализующие эти методы. В отличие от языков программирования, указанных выше, язык Java реализован совершенно другим способом. Объектный код Java представляет собой специальное двоичное представление, известное как байтовый код Java (Java byte code). Этот код можно рассматривать как программу машинного уровня.

ня для виртуальной машины (*virtual machine*). Как следует из этого названия, такая машина не реализована как аппаратное устройство. Вместо этого программные интерпретаторы обрабатывают байтовые коды, имитируя поведение виртуальной машины. Преимущество данного подхода заключается в том, что тот же байтовый код программы на Java можно выполнять на многих разных машинах, в то время как машинный код, который мы рассматривали выше, исполняется только на машинах с архитектурой IA32.

Библиографические заметки

Лучшие описания архитектуры IA32 изданы компанией Intel. Два полезных издания включены в серию, посвященную разработке программного обеспечения. В учебном пособии по базовой архитектуре [18] изложен обзор этой архитектуры с точки зрения программиста, составляющего программы на языке ассемблера, и справочном руководстве по набору команд [19] даются подробные описания различных команд. Эти источники содержат намного больше информации, чем необходимо для понимания кодов Linux. В частности, в режиме прямой адресации вполне можно игнорировать все сложности схемы адресации в сегментированном пространстве адресов.

Формат ассемблерного кода GAS, используемый ассемблером операционной системы Linux, существенно отличается от стандартного формата, используемого в документации компании Intel и другими компиляторами (в частности, компиляторами, поставляемыми компанией Microsoft). Одно из основных отличий заключается в том, что исходный операнд и операнд назначения представлены в обратном порядке.

На машине, работающей под управлением операционной системы Linux (Linux-машине), по команде `info as` на экран выводится справочная информация по ассемблеру. В одном из подразделов представлена машинно-специфичная информация, включая сравнение GAS с более стандартной нотацией, предложенной Intel. Обратите внимание на тот факт, что GCC использует для обозначения этих машин выражение "i386": он генерирует код, прогон которого возможен даже на машине выпуска 1985 года.

Книга Мучника (Muchnick), посвященная конструкции компилятора [55], считается наиболее всесторонним описанием технологий оптимизации программных кодов, которые рассматриваются в данной книге, таких как соглашения по использованию регистров и преимущества построения циклов на базе формы `do-while`.

Во многих книгах рассматриваются вопросы использования переполнения буферов для несанкционированного доступа через Internet. Подробный анализ программы-червя, появившегося в Internet в 1988 году, был опубликован Спаффордом (Spafford [73]) и другими сотрудниками MIT (Massachusetts Institute of Technologies — Massachusettsкий технологический институт, США), которым удалось остановить его распространение [26]. С тех пор в печати появилось множество статей и проектов, таких как, например, [20], касающихся создания и предотвращения атак, сопровождающихся переполнением буферов.

Задачи для домашнего решения

УПРАЖНЕНИЕ 3.31 ◆

Вам предоставляется следующая информация. Функция с прототипом

```
int decode2(int x, int y, int z);
```

переводится в компонующий автокод. Тело этого кода принимает вид:

```
1  movl 16(%ebp),%eax
2  movl 12(%ebp),%edx
3  subl %eax,%edx
4  movl %edx,%eax
5  imull 8(%ebp),%edx
6  sall $31,%eax
7  sarl $31,%eax
8  xorl %edx,%eax
```

Параметры x , y и z сохраняются в ячейках памяти со смещениями, соответственно, 8, 12 и 16 относительно адреса, сохраняемого в регистре $\$ebp$. Заданный программный код сохраняет возвращаемое значение в регистре $\$eax$. Напишите программу на языке C для функции `decode2`, которая выдает тот же результат, что и программа в ассемблерном коде. Вы можете выполнить тестирование полученного решения, проводя компиляцию вашего кода с переключателем `-S`. Ваш компилятор, возможно, не сможет построить идентичный код, но функционально они будут эквивалентны.

УПРАЖНЕНИЕ 3.32 ◆◆

Рассмотрим программный код, идентичный листингу 3.21.

```
1  int absdiff2(int x, int y)
2  {
3  int result;
4
5  if (x < y)
6  result = y-x;
7  else
8  result = x-y;
9  return result;
10 }
```

В процессе компиляции, однако, мы получаем другую форму ассемблерного кода.

```
1  movl 8(%ebp),%edx
2  movl 12(%ebp),%ecx
3  movl %edx,%eax
4  subl %ecx,%eax
5  cmpl %ecx,%edx
```

```

6 jge .L3
7 movl %ecx,%eax
8 subl %edx,%eax
9 .L3:

```

1. Какие операции вычитания выполняются, когда $x < y$? Когда $x \geq y$?
2. Как этот код отличается от стандартной реализации операции *if-else*, описанной выше?
3. Используя синтаксис конструкций языка C (включая конструкцию *goto*), покажите общую форму этой трансляции.
4. Какие ограничения должны быть установлены на использование трансляции с тем, чтобы было гарантировано поведение функции, описанное кодом на C?

УПРАЖНЕНИЕ 3.33 ◆◆

Приводимый здесь код показывает пример условного перехода на значение перечислимого типа в операторе выбора. Напоминаем вам, что перечислимые типы в C суть просто способ ввести некоторое множество имен, с которыми связаны конкретные целые числа. По умолчанию значения, присваиваемые этим значениям, начинаются с нуля и возрастают с каждым значением. В нашем коде действия, связанные с метками различных случаев, опущены.

```

/* Перечислимый тип создает множество констант, начинающихся с 0
и возрастающих с каждым значением */
typedef enum (MODE_A, MODE_B, MODE_C, MODE_D, MODE_E) mode_t;
int switch3(int *p1, int *p2, mode_t action)
{
int result = 0;
switch(action) {
case MODE_A:
case MODE_B:
case MODE_C:
case MODE_D:
case MODE_E:
default:
}
return result;
}

```

Часть построенного ассемблерного кода, реализующего различные действия, представлена далее. В комментарии указаны значения, хранящиеся в регистрах, и метки случаев различных переходов.

Целевые аргументы перехода

p1 и p2 находятся в регистрах %ebx и %ecx.

1 .L15: MODE_A

```

2    movl (%ecx),%edx
3    movl (%ebx),%eax
4    movl %eax,(%ecx)
5    jmp .L14
6    .p2align 4,,7 Вставляются с целью оптимизировать
7 .L16:      MODE_B
8    movl (%ecx),%eax
9    addl (%ebx),%eax
10   movl %eax,(%ebx)
11   movl %eax,%edx
12   jmp .L14
13   .p2align 4,,7 Вставляются с целью оптимизировать
14 .L17:      MODE_C
15   movl $15,(%ebx)
16   movl (%ecx),%edx
17   jmp .L14
18   .p2align 4,,7 Вставляются с целью оптимизировать
19 .L18:      MODE_D
20   movl (%ecx),%eax
21   movl %eax,(%ebx)
22 .L19:      MODE_E
23   movl $17,%edx
24   jmp .L14
25   .p2align 4,,7 Вставляются с целью оптимизировать
26 .L20:
27   movl $-1,%edx
28 .L14: по умолчанию
29   movl %edx,%eax Установить возвращаемое значение

```

1. Какие регистры соответствуют программной переменной `result`?
2. Вставьте отсутствующие части кода на С. Проверьте, имеются ли выпадающие случаи.

УПРАЖНЕНИЕ 3.34 ♦♦

Операторы выбора представляют особую трудность для специалистов, восстанавливавших программу по ее объектному коду. В приводимой ниже процедуре тело оператора выбора удалено.

```

1 int switch_prob(int x)
2 {
3     int result = x;
4
5     switch(x) {
6

```

```

7  /* Вставить код в этом месте */
8  }
9
10 return result;
11 }
```

Подвергнутый обратному ассемблированию, код задачи представлен следующим кодом:

```

1  080483c0 <switch_prob>:
2  80483c0: 55                      push  %ebp
3  80483c1: 89 e5                  mov   %esp,%ebp
4  80483c3: 8b 45 08                mov   0x8(%ebp),%eax
5  80483c6: 8d 50 ce                lea   0xfffffffce(%eax),%edx
6  80483c9: 83 fa 05                cmp   $0x5,%edx
7  80483cc: 77 1d                  ja    80483eb <switch_prob+0x2b>
8  80483ce: ff 24 95 68 84 04 08   jmp   *0x8048468(%edx,4)
9  80483d5: c1 e0 02                shl   $0x2,%eax
10 80483d8: eb 14                 jmp   80483ee <switch_prob+0x2e>
11 80483da: 8d b6 00 00 00 00      lea   0x0(%esi),%esi
12 80483e0: c1 f8 02                sar   $0x2,%eax
13 80483e3: eb 09                 jmp   80483ee <switch_prob+0x2e>
14 80483e5: 8d 04 40                lea   (%eax,%eax,2),%eax
15 80483e8: 0f af c0                imul %eax,%eax
16 80483eb: 83 c0 0a                add   $0xa,%eax
17 80483ee: 89 ec                  mov   %ebp,%esp
18 80483f0: 5d                   pop   %ebp
19 80483f1: c3                   ret
20 80483f2: 89 f6                  mov   %esi,%esi
```

Нас прежде всего интересует та часть кода, которая показана в строках 4—16 этого кода. В строке 4 мы видим, что параметр *x* (смещение 8 относительно %ebp) загружен в регистр %eax, соответствующий программной переменной *result*. Команда *lea 0x0(%esi),%esi* в строке 11 есть команда *nop*, вставленная с тем, чтобы команда в строке 12 стартовала с адреса, кратного 16.

Таблица переходов содержится в другой области памяти. Используя отладчик GDB, мы можем проверить шесть 4-байтовых слов памяти, начиная с адреса 0x8048468, по которому находится команда *x/6w 0x8048468*. Отладчик GDB печатает следующий текст:

```
(gdb) x/6w 0x8048468
0x8048468: 0x080483d5 0x080483eb 0x080483d5 0x080483e0
0x8048478: 0x080483e5 0x080483e8
(gdb)
```

Заполните тело оператора выбора программным кодом на C, который обеспечивает то же поведение, что и объектный код.

УПРАЖНЕНИЕ 3.35 ◆◆

Программный код, порожденный компилятором языка С для var_prod_ele (см. листинг 3.52), не является оптимальным. Запишите программный код этой функции на основе гибридных процедур fix_prod_ele_opt (см. листинг 3.47) и var_prod_ele_opt (см. листинг 3.53). Напоминаем вам, что процессор имеет всего шесть регистров, доступных для хранения временных данных, поскольку регистры %ebp и %esp не могут быть использованы для этой цели. Один из этих регистров должен быть использован для хранения результата команды перемножения. Следовательно, вы должны уменьшить число локальных переменных в цикле с шести (result, Aprt, B, nTjPk, n и cnt) до пяти.

УПРАЖНЕНИЕ 3.36 ◆◆

В ваши обязанности входит сопровождение большой программы на языке С, и вы столкнулись со следующим кодом:

```

1  typedef struct {
2      int left;
3      a_struct a[CNT];
4      int right;
5  } b_struct;
6
7  void test(int i, b_struct *bp)
8  {
9      int n = bp->left + bp->right;
10     a_struct *ap = &bp->a[i];
11     ap->x[ap->idx] = n;
12 }
```

К сожалению, файл .h, определяющий статическую константу CNT, и структура a_struct хранятся в файлах, к которым вы не имеете привилегированного доступа. В то же время вы имеете доступ к версии .o программного кода, над которым вы можете выполнить операцию обратного ассемблирования посредством программы objdump, получив дизассемблированный следующий программный код:

```

1  00000000 <test>:
2  0  55          push %ebp
3  1: 89 e5       mov %esp,%ebp
4  3: 53          push %ebx
5  4: 8b 45 08    mov 0x8(%ebp),%eax
6  7: 8b 4d 0c    mov 0xc(%ebp),%ecx
7  a: 8d 04 80    lea (%eax,%eax,4),%eax
8  d: 8d 44 81 04 lea 0x4(%ecx,%eax,4),%eax
9  11: 8b 10      mov (%eax),%edx
10 13: c1 e2 02   shl $0x2,%edx
```

```

11 16: 8b 99 b8 00 00 00      mov 0xb8(%ecx),%ebx
12 1c: 03 19                  add (%ecx),%ebx
13 1e: 89 5c 02 04              mov %ebx,0x4(%edx,%eax,1)
14 22: 5b                      pop %ebx
15 23: 89                      ec mov %ebp,%esp
16 25: 5d                      pop %ebp
17 26: c3                      ret

```

Воспользовавшись своими навыками специалиста по восстановлению программ по исходным текстам, определите следующие программные элементы:

- значение CNT;
- полное объявление структуры `a_struct`. При этом предполагается, что эта структура содержит только два поля — `idx` и `x`.

УПРАЖНЕНИЕ 3.37 ◆

Составьте функцию `good_echo`, которая считывает строку из устройства стандартного ввода и записывает ее на устройство стандартного вывода. Ваша программа должна работать со строкой произвольной длины. Вы можете воспользоваться библиотечной функцией `fgets`, но, прежде всего, вы должны убедиться в том, что ваша функция работает правильно, даже когда входная строка требует больше пространства памяти, чем вы отвели для своего буфера. Ваша программа должна также выявлять условия ошибки и сообщать о них, если таковые имеют место. При необходимости вам следует обращаться к определениям стандартных функций ввода-вывода, содержащихся в документах [32, 40].

УПРАЖНЕНИЕ 3.38 ◆◆◆

По условиям этой задачи вы должны смоделировать в вашей программе атаку злоумышленника с переполнением буфера. Как уже было отмечено выше, мы не можем мириться с использованием этой или других форм атак системы с целью осуществить несанкционированный доступ к системе, однако в процессе выполнения этого упражнения вы многое узнаете о программировании на машинном уровне.

В файле `bufbomb.c` определите следующую функцию:

```

1 int getbuf()
2 {
3     char buf[12];
4     gets(buf);
5     return 1;
6 }
7
8 void test()
9 {
10    int val;
11    printf("Type Hex string:");

```

```
12 val = getbuf();  
13 printf("getbuf returned 0x%x\n", val);  
14 }
```

Функция `getxs` (также и в `bufbomb.c`) подобна библиотечной функции `gets`, за исключением того, что она считывает символы, закодированные как пары шестнадцатеричных цифр. Например, чтобы передать ей строку

0123

пользователь должен ввести с клавиатуры строку

30 31 32 33

Эта функция игнорирует символы пробела. Напомним, что десятичная цифра x имеет в ASCII представление `0x3x`.

Обычно эта программа выполняется следующим образом:

```
unix> ./bufbomb  
Type Hex string: 30 31 32 33  
getbuf returned 0x1
```

Даже поверхностный анализ функции `getbuf` показывает, что она возвращает значение 1 всякий раз, когда ее вызывают. Отсюда следует, что вызов функции `getxs` бесполезен. Ваша задача заключается в том, чтобы функция `getbuf` возвращала значение -559038737 (`Oxdeadbeef`) по команде `test`, введя с клавиатуры соответствующую шестнадцатеричную строку в ответ на приглашение. Следующие предположения, возможно, помогут вам решить эту задачу.

- Воспользуйтесь программой `OBJDUMP` с целью построить дизассемблированную версию функции `bufbomb`. Внимательно изучите эту версию с тем, чтобы определить, как организован стековый фрейм функции `getbuf`, как в результате переполнения буфера поменяется сохраненное состояние программы.
- Осуществите прогон вашей программы под управлением отладчика `GDB`. Установите точку прерывания в функции `getbuf` и выполните ее до этой точки прерывания. Определите такие параметры, как значение в регистре `%ebp` и сохраненное значение любого состояния, которое будет затерто, когда вы вызовете переполнение буфера.
- Определение байтового кодирования последовательностей команд вручную обременительно и может служить причиной возникновения ошибок. Вы можете доверить эту работу инструментальным средствам, для чего вы создаете файл ассемблерных кодов и данные, которые хотите поместить в стек. Проведите ассемблирование этого файла с помощью компилятора `GCC`, а затем выполните его дизассемблирование с помощью программы `OBJDUMP`. Вы должны иметь возможность получить точную байтовую последовательность, которую вы вводите с клавиатуры по приглашению. Программа `OBJDUMP` генерирует некоторую довольно странную совокупность команд ассемблирования, когда она предпринимает попытку дизассемблирования данных из вашего файла, однако последовательность шестнадцатеричных байтов должна быть правильной.

Имейте в виду, что ваша атака в значительной степени зависит от специфики машины и компилятора. Возможно, вам придется изменить строку при работе на другой машине или с другой версией компилятора.

УПРАЖНЕНИЕ 3.39 ◆◆◆

Воспользуйтесь оператором `asm` при реализации функции, имеющей прототип:

```
void full_umul(unsigned x, unsigned y, unsigned dest[]);
```

Эта функция должна вычислять полное 64-разрядное произведение своих аргументов и сохранить результаты в массиве назначения, при этом в `dest[0]` сохраняются младшие 4 байта произведения, а в `dest[1]` сохраняются 4 старших байта.

УПРАЖНЕНИЕ 3.40 ◆◆◆

Команда `fscale` вычисляет функцию $x \cdot 2^{RTZ(y)}$ для значений с плавающей точкой x и y , при этом RTZ (round-toward-zero) означает округление положительных чисел в меньшую сторону и отрицательных чисел в большую сторону. Аргументы функции `fscale` берутся из регистрового стека с плавающей точкой, x в регистре `%st(0)` и y в регистре `%st(1)`. Она записывает вычисленное значение в регистр `%st(0)` без выталкивания из стека второго аргумента. Фактическая реализация этой команды работает как прибавление $RTZ(y)$ к показателю значения x .

Используя `asm`, реализуйте функцию со следующим прототипом:

```
double scale(double x, int n, double *dest);
```

которая вычисляет $x \cdot 2^n$, используя команду `fscale`, и сохраняет результат в ячейке, на которую ссылается указатель `dest`. Расширенный оператор `asm` не обеспечивает надежной поддержки архитектуры IA32 с плавающей точкой. Однако в этом случае вы можете получить доступ к аргументам из стека программы.

Решение упражнений

РЕШЕНИЕ УПРАЖНЕНИЯ 3.1

Это упражнение предоставляет вам возможность научиться работать с операндами различных видов.

Операнд	Значение	Комментарий
<code>%eax</code>	0x100	Регистр
0x104	0xAB	Абсолютный адрес
\$0x108	0x108	Непосредственный
(<code>%eax</code>)	0xFF	Адрес 0x100
4 (<code>%eax</code>)	0xAB	Адрес 0x104

(окончание)

Операнд	Значение	Комментарий
9(%eax,%edx)	0x11	Адрес 0x10C
260(%ecx,%edx)	0x13	Адрес 0x108
0xFF(,%ecx,4)	0xFF	Адрес 0x100
(%eax,%edx,4)	0x11	Адрес 0x10C

РЕШЕНИЕ УПРАЖНЕНИЯ 3.2

Обратное проектирование представляет собой хороший способ достичь понимания системы. В этом случае мы добиваемся результата, обратного действиям компилятора языка С, чтобы знать, что стало источником данного программного кода на языке ассемблера. Лучший способ выполнить "моделирование", начиная со значений *x*, *y* и *z* в ячейках, определяемых, соответственно, указателями *xp*, *yp* и *zp*. В этом случае мы получаем следующее поведение:

```

1  movl  8(%ebp),%edi      xp
2  movl  12(%ebp),%ebx     yp
3  movl  16(%ebp),%esi     zp
4  movl  (%edi),%eax      x
5  movl  (%ebx),%edx      y
6  movl  (%esi),%ecx      z
7  movl  %eax,(%ebx)       *yp = x
8  movl  %edx,(%esi)       *zp = y
9  movl  %ecx,(%edi)       *xp = z

```

На основании этого фрагмента мы можем получить следующий программный код на С:

```

1 void decode1(int *xp, int *yp, int *zp)
2 {
3     int tx = *xp;
4     int ty = *yp;
5     int tz = *zp;
6
7     *yp = tx;
8     *zp = ty;
9     *xp = tz;
10 }

```

РЕШЕНИЕ УПРАЖНЕНИЯ 3.3

Это упражнение показывает, насколько разносторонней является команда *leal*, оно позволяет вам получить навыки декодирования операндов форм. Обратите внимание

на то, что хотя формы операндов классифицируются как тип памяти на рис. 3.3, никакого доступа к памяти не происходит.

Выражение	Результат
leal 6(%eax), %edx	6+x
leal (%eax,%ecx), %edx	x+y
leal (%eax,%ecx,4), %edx	x+4y
leal 7(%eax,%eax,8), %edx	7+9x
leal 0xA(%eax,%ecx,4), %edx	10+4y
leal 9(%eax,%ecx,2), %edx	9+x+2y

РЕШЕНИЕ УПРАЖНЕНИЯ 3.4

Эта задача предоставляет вам шанс проверить ваше понимание операндов и арифметические команды.

Команда	Назначение	Значение
addl %ecx, (%eax)	0x100	0x100
subl %edx, 4(%eax)	0x104	0xA8
imull \$16, (%eax,%edx,4)	0x10C	0x110
incl 8(%eax)	0x108	0x14
decl %ecx	%ecx	0x0
subl %edx,%eax	%eax	0xFD

РЕШЕНИЕ УПРАЖНЕНИЯ 3.5

Это упражнение предоставляет вам возможность построить небольшой фрагмент программного кода на языке ассемблера. Код решения создан компилятором GCC. Загрузив параметр *n* в регистр %ecx, он затем использует байтовый регистр %cl для задания величины сдвига команды *sarl*:

```

1  movl 12(%ebp),%ecx    Получить n
2  movl 8(%ebp),%eax     Получить x
3  sall $2,%eax          x <= 2
4  sarl %cl,%eax         x >= n

```

РЕШЕНИЕ УПРАЖНЕНИЯ 3.6

Эта инструкция используется для установки регистра %edx в 0 с использованием того свойства, что $x \wedge x = 0$ для любого x . Она соответствует оператору *i = 0*.

Это пример идиомы языка ассемблера — фрагмент программного кода, который часто генерируется для специальных целей. Распознавание таких идиом представляет собой один из лучших способов приобретения навыков чтения ассемблерных кодов.

РЕШЕНИЕ УПРАЖНЕНИЯ 3.7

Этот пример требует, чтобы вы рассматривали возможность применения различных команд сравнения. Особое внимание следует обратить на тот факт, что при преобразовании одного из значений оператора сравнения в значения без знака, операция сравнение выполняется, как если бы обе сравниваемые величины были значениями без знака, что объясняется неявным приведением типов.

```

1  char ctest(int a, int b, int c)
2  {
3      char t1 =      a <          b;
4      char t2 =      b < (unsigned) a;
5      char t3 = (short) c >= (short) a;
6      char t4 = (char)  a != (char)  c;
7      char t5 =      c >          b;
8      char t6 =      a >          0;
9      return t1 + t2 + t3 + t4 + t5 + t6;
10 }
```

РЕШЕНИЕ УПРАЖНЕНИЯ 3.8

Это упражнение требует, чтобы вы провели подробный анализ дизассемблированного программного кода и задумались над тем, как кодируется место назначение оператора перехода. При этом вы сможете усовершенствовать свои навыки в шестнадцатеричной арифметике.

- Команда `jbe` в качестве своей цели имеет адрес `0x8048d1c + 0xda`. Как показывает дизассемблированный исходный код, таковым является `0x8048cf8`.

8048d1c:	76 da	<code>jbe 8048cf8</code>
8048d1e:	eb 24	<code>jmp 8048d44</code>

- Согласно комментарию, которым сопроводил свои коды дизассемблер, место назначения оператора перехода есть абсолютный адрес `0x8048d44`. Из байтовых кодов следует, что таковым должен быть адрес со смещением `0x54` байтов относительно команды `mov`. Вычитая один из другого, получаем адрес `0x8048cf0`, который подтверждается кодом дизассемблера:

8048ce0:	eb 54	<code>jmp 8048d44</code>
8048cf0:	c7 45 f8 10 00	<code>mov \$0x10,0xffffffff(%ebp)</code>

- Адрес назначения сдвинут на `000000cb` относительно ячейки `0x8048907` (адрес команды `por`). Сумма этих значений дает адрес `0x80489d2`.

8048902:	e9 cb 00 00 00	<code>jmp 80489d2</code>
8048907:	90	<code>por</code>

4. Операция непрямого перехода обозначается кодом ff 25. Адрес, по которому производится считывание, явно определяется в следующих 4 байтах. Поскольку в рассматриваемой машине реализован обратный порядок следования, имеем последовательность e0 a2 04 08.

```
80483f0: ff 25 e0 a2 04 jmp      *0x804a2e0
80483f5: 08
```

РЕШЕНИЕ УПРАЖНЕНИЯ 3.9

Полезные первоначальные действия по изучению программы на языке ассемблера заключаются в том, чтобы снабдить программу комментариями и промоделировать ее поток управления на языке С. Эта задача позволит вам получить навыки работы с примерами простых потоков управления. Она также даст вам возможность изучить способы реализации различных логических операций.

```
1. 1 void cond(int a, int *p)
2  {
3     if (*p == 0)
4         goto done;
5     if (a <= 0)
6         goto done;
7     *p += a;
8 done:
9 }
```

2. Первый условный переход представляет собой часть реализации выражения `&a`. Если проверка ненулевого значения `p` закончится неудачно, в рассматриваемом программном коде проверка условия `a > 0` опускается.

РЕШЕНИЕ УПРАЖНЕНИЯ 3.10

Программный код, генерируемый компилятором для циклов, может оказаться трудным для анализа, т. к. компилятор может выполнять множество операций оптимизации кодов, реализующих циклы, и при этом возникают сложности сопоставления программных переменных с регистрами. Мы начнем приобретать соответствующие навыки с изучения простых циклов.

1. Способ использования регистра может быть определен, если проанализировать, как производится выборка аргументов.

Использование регистров

Регистр	Переменная	Начальное значение
%esi	x	x
%ebx	y	y
%ecx	n	n

2. Та часть программного кода, которая представляет *body-оператор* (тело цикла), содержится в строках от 4 до 6 программы на языке С и в строках от 6 до 8 кодов на языке ассемблера. Порция кодов, реализующих *test-expr* (условие продолжения цикла), содержится в строке 7 программного кода на языке С. В кодах на языке ассемблера это условие реализовано командами, содержащимися в строках 9—14, а также условием перехода, содержащемся в строке 15.

3. Код, снабженный комментариями, принимает следующий вид:

```

1  movl  8(%ebp),%esi      Поместить x в %esi
2  movl  12(%ebp),%ebx      Поместить y в %ebx
3  movl  16(%ebp),%ecx      Поместить n в %ecx
4  .p2align 4,,7
5  .L6:                   loop:
6  imull %ecx,%ebx        y *= n
7  addl  %ecx,%esi        x += n
8  decl  %ecx              n--
9  testl %ecx,%ecx        Проверка значения n
10 setg %al                n > 0
11 cmpl %ecx,%ebx        Сравнение y:n
12 setl %dl                y < n
13 andl %edx,%eax         {n > 0} & {y < n}
14 testb $1,%al            Проверка значения наименьшего значащего разряда
15 jne .L6                 If != 0, goto loop

```

Обратите внимание на довольно странную реализацию условия продолжения цикла. Компилятор распознает, что два предиката ($n > 0$) и ($y < n$) могут принимать значения 0 или 1, и отсюда следует, что при вычислении условия перехода достаточно всего лишь проверить значение наименьшего значащего разряда результата операции AND над ними. Сам компилятор мог бы быть более благородным и воспользоваться командой *testb*, чтобы выполнить операцию AND.

РЕШЕНИЕ УПРАЖНЕНИЯ 3.11

Эта задача представляет собой еще одну возможность попрактиковаться в расшифровке кодов цикла. Компилятор языка С выполнил довольно интересные операции оптимизации.

1. Режим использования регистра может быть определен тем, как производится выборка аргументов и инициализация регистров.

Использование регистров

Регистр	Переменная	Начальное значение
%eax	a	a
%ebx	b	b
%ecx	x	0
%edx	result	a

2. Условие продолжения цикла встречается в строке 5 программного кода на языке C, а также в строке 10 и в условии перехода в строке 11 кода на языке ассемблера. Тело цикла размещается в строках 6—8 программных кодов на языке C и в строках 7—9 ассемблерного кода. Компилятор обнаружил, что начальная проверка цикла while всегда имеет результат true, поскольку переменная i инициализирована 0, который намного меньше, чем 256.

3. Программный код, снабженный комментариями, имеет вид:

```

1  movl  8(%ebp),%eax      Поместить a в регистр %eax
2  movl  12(%ebp),%ebx      Поместить b в регистр %ebx
3  xorl  %ecx,%ecx        i = 0
4  movl  %eax,%edx        result = a
5  .p2align 4,,7
6  .L5:                   loop:
7  addl  %eax,%edx        result += a
8  subl  %ebx,%eax        a -= b
9  addl  %ebx,%ecx        i += b
10  cmpl $255,%ecx       Сравнение i:255
11  jle .L5              If <= goto loop
12  movl  %edx,%eax       Использовать result как возвращаемое

```

4. Эквивалентный код операции goto принимает следующий вид:

```

1  int loop_while_goto(int a, int b)
2  {
3      int i = 0;
4      int result = a;
5      loop:
6          result += a;
7          a -= b;
8          i += b;
9          if (i <= 255)
10             goto loop;
11         return result;
12     }

```

РЕШЕНИЕ УПРАЖНЕНИЯ 3.12

Один из способов анализа ассемблерного кода заключается в том, чтобы обратить процесс компиляции и генерировать программный код на языке C, который казался бы "естественным" для программиста, работающего в языке C. Например, мы не намерены использовать операторы goto, поскольку они достаточно редко используются в программах на языке C. Скорее всего, мы также не будем употреблять оператор do-while. Это упражнение требует задуматься о трансляции циклов for. Оно также служит примером использования технологии оптимизации, известной как *вынесение части текста программы* (code motion), когда вычисления с целью оптимизации

программы выносятся из циклов, когда выясняется, что их результаты не изменяются в рамках цикла.

1. Мы можем убедиться в том, что программная переменная `result` обязательно должна находиться в регистре `%eax`. Она обязательно должна быть инициализирована значением 0 и оставаться в регистре `%eax` в конце цикла в качестве возвращаемого значения. Легко видеть, что переменная `i` находится в регистре `%edx`, т. к. этот регистр используется как база для проверки двух условий.
2. Команды в строках 2 и 4 устанавливают значение в регистре `%edx` равным $n-1$.
3. Проверки, реализованные в строках 5 и 12, требуют, чтобы значение переменной `i` было неотрицательным.
4. Команда в строке 4 уменьшает значение переменной `i`.
5. Команды 1, 6 и 7 запоминают произведение $x \cdot y$ в регистре `%eax`.
6. Ниже приводится исходный код:

```
1 int loop(int x, int y, int n)
2 {
3     int result = 0;
4     int i;
5     for (i = n-1; i >= 0; i = i-x) {
6         result += y * x;
7     }
8     return result;
9 }
```

РЕШЕНИЕ УПРАЖНЕНИЯ 3.13

Эта задача предоставляет вам возможность провести анализ потока управления оператора выбора. Чтобы ответить на вопросы, вам придется выбирать нужную информацию из нескольких мест ассемблерного кода.

1. В строке 2 ассемблерного кода производится сложение 2 и `x` с тем, чтобы нижняя граница диапазона случаев была равна 0. Это означает, что минимальная метка случая есть -2 .
2. Строки 3 и 4 осуществляют переход программы на случай по умолчанию, если приведенное значение случая больше 6. Отсюда следует, что максимальная метка случая есть $-2 + 6 = 4$.
3. Из таблицы переходов мы видим, что второе вхождение (метка случая -1) имеет то же назначение (`.L10`), что и команда перехода в строке 4, указывая на случай поведения по умолчанию. Следовательно, метка случая -1 отсутствует в теле оператора.
4. Из таблицы переходов мы видим, что пятое и шестое вхождение имеют одно и то же назначение. Это соответствует меткам случаев 2 и 3.

Принимая во внимание все эти соображения, мы приходим к двум заключениям:

- Метки случаев в теле оператора выбора принимают значения -2, 0, 1, 2, 3 и 4.
- Случай с назначением L8 имеют метки 2 и 3.

РЕШЕНИЕ УПРАЖНЕНИЯ 3.14

Это еще один пример идиом на языке ассемблера. Сначала он кажется довольно-таки странным: команда `call`, для которой не существует соответствующая ей команда `ret`. Затем, в конце концов, мы понимаем, что это вовсе не вызов процедуры.

1. В регистре `%eax` устанавливается адрес команды `popl`.
2. Это не есть истинная подпрограмма, т. к. поток управления сохраняет тот же порядок, что и команды, а адрес возврата выталкивается из стека.
3. В архитектуре IA32 это единственный способ поместить значение счетчика команд в целочисленный регистр.

РЕШЕНИЕ УПРАЖНЕНИЯ 3.15

Эта задача вносит конкретику в обсуждение соглашений об использовании регистров. В регистрах `%edi`, `%esi` и `%ebx` сохраняются данные вызываемой процедуры. Процедура должна сохранить их в стеке перед тем, как изменить их значения, и восстановить их перед возвращением в вызывающую процедуру. Три других регистра сохраняют информацию вызывающей процедуры. Они могут подвергаться изменениям без последствий для поведения вызывающей процедуры.

РЕШЕНИЕ УПРАЖНЕНИЯ 3.16

Способность определить, как функции используют стек, очень важна для понимания программного кода, построенного компилятором. Как показывает рассматриваемый пример, компилятор резервирует большое количество памяти, которая никогда не используется.

1. Мы начинаем с того, что в регистр `%esp` помещаем значение `0x800040`. Стока 2 уменьшает это значение на 4, в результате получаем `0x80003C`, и это становится новым значением регистра `%ebp`.
2. Мы можем видеть, как с помощью двух команд `leal` вычисляются аргументы, которые затем передаются команде `scanf`. Поскольку аргументы заталкиваются в обратном порядке, мы можем видеть, что переменная `x` смешена на `-4` относительно `%ebp`, а переменная `y` смешена на `-8`. Поэтому адресами являются `0x800038` и `0x800034`.
3. Начиная с исходного значения `0x800040`, строка 2 уменьшает указатель стека на 4. Стока 4 уменьшает его на 24, а строка 5 уменьшает его на 4. Три операции проталкивания уменьшают его на 12, при этом общее изменение достигает значения 44. Следовательно, после выполнения строки 10 содержимое регистра `%esp` равно `0x800014`.

4. Стековый фрейм имеет следующую структуру и содержимое:

0x80003C	0x800060	%ebp
0x800038	0x53	x
0x800034	0x46	y
0x800030		
0x80002C		
0x800028		
0x800024		
0x800020		
0x80001C	0x800038	
0x800018	0x800034	
0x800014	0x300070	%esp

5. Байтовые адреса от 0x800020 до 0x800033 не используются.

РЕШЕНИЕ УПРАЖНЕНИЯ 3.17

Это упражнение проверяет, насколько правильно вы понимаете такие вещи, как размерность данных и индексирование массивов. Обратите внимание на то обстоятельство, что указатель на любой тип данных требует 4 байтов памяти. Компилятор GCC отводит под каждое значение типа `long double` 12 байтов памяти, даже в тех случаях, когда фактический формат требует только 10 байтов.

Массив	Размер элемента	Общий размер	Начальный адрес	Элемент i
s	2	28	x_s	$x_s + 2i$
T	4	12	x_T	$x_T + 4i$
U	4	24	x_U	$x_U + 4i$
V	12	96	x_V	$x_V + 12i$
W	4	16	x_W	$x_W + 4i$

РЕШЕНИЕ УПРАЖНЕНИЯ 3.18

Эта задача представляет собой вариант задачи с целочисленным массивом e. Важно представлять себе различия между указателем и объектом, на который он указывает. Поскольку тип данных `short` требует двух байтов, все индексы подвергаются масштабированию с коэффициентом, равным двум. Вместо того, чтобы пользоваться, как раньше, командой `movl`, здесь мы прибегаем к помощи команды `movw`.

Выражение	Тип	Значение	Код на языке ассемблера
S+1	short *	$x_S + 2$	leal 2(%edx), %eax
S[3]	short	M[x _S + 6]	movw 6(%edx), %ax
&S[i]	short *	$x_S + 2i$	leal (%edx, %ecx, 2), %eax
S[4*i+1]	short	M[x _S + 8i+2]	movw 2(%edx, %ecx, 8), %ax
S+i-5	short *	$x_S + 2i - 10$	leal -10(%edx, %ecx, 2), %eax

РЕШЕНИЕ УПРАЖНЕНИЯ 3.19

Эта операция требует от вас правильного применения операций масштабирования при вычислении адресов и формулы индексирования для вычисления ведущего индекса строки. Первое действие заключается в том, чтобы снабдить комментарием ассемблерный код с тем, чтобы определить, как следует вычислять адресные ссылки:

```

1  movl  8(%ebp), %ecx          Получить i
2  movl  12(%ebp), %eax         Получить j
3  leal   0(%eax, 4), %ebx      4*j
4  leal   0(%ecx, 8), %edx      8*i
5  subl  %ecx, %edx             7*i
6  addl  %ebx, %eax             5*j
7  sall  $2, %eax               20*j
8  movl  mat2(%eax, %ecx, 4), %eax  mat2[(20*j) + 4*i]/4
9  addl  mat1(%ebx, %edx, 4), %eax  + mat1[(4*j) + 28*i]/4

```

Отсюда мы можем видеть, что ссылка на матрицу mat1 есть смещение $4(7i + j)$ в байтах, в то время как ссылка на матрицу mat2 есть смещение $4(5j + i)$ в байтах. Отсюда мы можем определить, что в матрице mat1 7 столбцов, в то время как в матрице mat2 5 столбцов. Следовательно, M = 5 и N = 7.

РЕШЕНИЕ УПРАЖНЕНИЯ 3.20

Это упражнение требует от вас хорошего знания ассемблерных кодов, чтобы понимать, как осуществляется его оптимизация. Это очень важный навык, способствующий улучшению рабочих характеристик программ. Внося соответствующие изменения в исходный код, вы можете достичь производительности, близкой к производительности машинного кода, построенного машиной.

Приводим оптимизированную версию программного кода на языке C:

```

1  /* Присвоить всем диагональным элементам значение val */
2  void fix_set_diag_opt(fixture A, int val)
3  {
4      int *Aptr = &A[0][0] + 255;
5      int cnt = N-1;
6      do {

```

```

7     *Aptr = val;
8     Aptr -= (N+1);
9     cnt--;
10    } while (cnt >= 0);
11 }

```

То, как он соотносится с ассемблерным кодом, можно видеть из следующих комментариев:

```

1 movl 12(%ebp),%edx      Получить val
2 movl 8(%ebp),%eax      Получить A
3 movl $15,%ecx          i = 0
4 addl $1020,%eax        Aptr = &A[0][0] + 1020/4
5 .p2align 4,,7
6 .L50:                  loop:
7 movl %edx,(%eax)        *Aptr = val
8 addl $-68,%eax          Aptr -= 68/4
9 decl %ecx               i--
10 jns .L50                if i >= 0 goto loop

```

Обратите внимание на то, что программа в ассемблерном коде начинает обработку с конца массива и продвигается к его началу. Она уменьшает значение указателя на 68 (= 17 × 4), поскольку элементы массива $A[i-1][i-1]$ и $A[i][i]$ отделены друг от друга на $N+1$ элементов.

РЕШЕНИЕ УПРАЖНЕНИЯ 3.21

Эта задача заставляет вас обратить особое внимание на топологию структуры и на коды, используемые для доступа к полям структуры. Объявление структуры представляет собой вариант примера, приведенного в тексте. Он показывает, что построение вложенных структур осуществляется путем встраивания внутренних структур во внешние структуры.

1. Топология структуры имеет следующий вид:

Смещение	0	4	8	12
Содержимое	p	s.x	s.y	next

2. Она использует 16 байтов.

3. Как всегда, мы начинаем с того, что снабжаем программный код комментариями:

```

1 movl 8(%ebp),%eax      Взять sp
2 movl 8(%eax),%edx      Взять sp->s.y
3 movl %edx,4(%eax)       Скопировать в sp->s.x
4 leal 4(%eax),%edx       Взять &(sp->s.x)
5 movl %edx,(%eax)        Скопировать в sp->p
6 movl %eax,12(%eax)       sp->next = p

```

Взяв за основу этот ассемблерный код, получим следующий код на языке С:

```
void sp_init(struct prob *sp)
{
    sp->s.x = sp->s.y;
    sp->p = &(sp->s.x);
    sp->next = sp;
}
```

РЕШЕНИЕ УПРАЖНЕНИЯ 3.22

Это довольно-таки сложная проблема. Для ее решения требуется особая сообразительность при решении задач обратного проектирования. Она четко показывает, что объединения представляют собой простейший способ связывания множество имен (и типов) с одной и той же ячейкой памяти.

1. Топология объединения показана в приводимой далее таблице. Как следует из этой таблицы, это объединение может иметь собственную интерпретацию e1 (наличие полей e1.p и e1.y), собственную интерпретацию e2 (наличие полей e2.x и e2.next).

Смещение	0	4
Содержимое	e1.p	e1.y
	e2.x	e2.next

2. Она использует 8 байтов.
3. Это объединение может иметь собственную интерпретацию некоторых инструкций.

1 movl 8(%ebp),%eax	Взять up
2 movl 4(%eax),%edx	up->e1.y (no) или up->e2.next
3 movl (%edx),%ecx	up->e2.next->e1.p или up->e2.next->e2.x (no)
4 movl (%eax),%eax	up->e1.p (no) или up->e2.x
5 movl (%ecx),%ecx	* (up->e2.next->e1.p)
6 subl %eax,%ecx	* (up->e2.next->e1.p) - up->e2.x
7 movl %ecx,4(%edx)	Сохранить в up->e2.next->e1.y

На основании этого кода мы можем составить следующую программу на языке С:

```
void proc (union ele *up)
{
    up->e2.next->e1.y = *(up->e2.next->e1.p) - up->e2.x;
}
```

РЕШЕНИЕ УПРАЖНЕНИЯ 3.23

Правильное представление топологии и выравнивания структур имеет очень важное значение для понимания того, сколько памяти требуется для размещения той или

иной структуры, а также того, какие коды, обеспечивающие доступ к структурам, генерирует компилятор. Эта проблема позволяет вам правильно разрабатывать все детали для тех или иных примеров структур.

1. struct P1 { int i; char c; int j; char d; };

i	c	j	d	Всего	Выравнивание
0	4	8	12	16	4

2. struct P2 { int i; char c; char d; int j; };

i	c	d	j	Всего	Выравнивание
0	4	5	8	12	4

3. struct P3 { short w[3]; char c[3] };

w	c	Всего	Выравнивание
0	6	10	2

4. struct P4 { short w[3]; char *c[3] };

w	c	Всего	Выравнивание
0	8	20	4

5. struct P5 { struct P1 a[2]; struct P2 *p };

a	p	Всего	Выравнивание
0	32	36	4

РЕШЕНИЕ УПРАЖНЕНИЯ 3.24

Эта задача охватывает широкий диапазон тем, таких как стековые фреймы, представления строк, коды ASCII и упорядочение байтов. Она демонстрирует опасности ссылок на ячейки памяти, выходящие за заданные пределы, и пагубные последствия переполнения буферов.

1. Стек в строке 7:

08 04 86 43	Адрес возврата
bf ff fe 94	Сохранен %ebp \leftarrow %ebp
	buf[4-7]
	buf[0-3]
00 00 00 01	Сохранен %esi
00 00 00 02	Сохранен %ebx

2. Стек после строки 10 (показаны только слова, которые подверглись изменениям).

08 04 86 00	Адрес возврата
31 30 39 38	Сохранен %ebp \leftarrow %ebp
37 36 35 34	buf[4-7]
33 32 31 30	buf[0-3]

- Программа делает попытку возврата по адресу 0x08048600. Младший байт был перезаписан завершающим символом Null.
- Сохраненное значение регистра %ebp было изменено на 0x31303938, сохраненная величина будет загружена в регистр, прежде чем состоится возврат getline. Другие сохраненные регистры не затрагиваются, поскольку они запоминаются в стеке с меньшими адресами, чем buf.
- Обращение к malloc должно иметь strlen(buf)+1 в качестве аргумента, оно должно также проверить, что возвращаемое значение не есть Null.

РЕШЕНИЕ УПРАЖНЕНИЯ 3.25

Эта задача дает вам возможность разработать рекурсивную процедуру, описанную в разд. 3.14.2. (рис. 3.16).

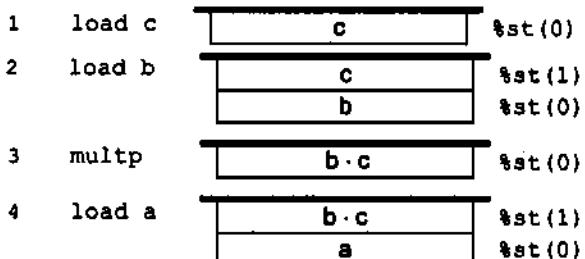


Рис. 3.16. Диаграмма решения упражнения 3.25 (см. продолжение)

5	addp	<table border="1"><tr><td>a + b · c</td></tr></table>	a + b · c	%st(0)			
a + b · c							
6	neg	<table border="1"><tr><td>- (a + b · c)</td></tr></table>	- (a + b · c)	%st(0)			
- (a + b · c)							
7	load c	<table border="1"><tr><td>- (a + b · c)</td></tr><tr><td>c</td></tr></table>	- (a + b · c)	c	%st(1) %st(0)		
- (a + b · c)							
c							
8	load b	<table border="1"><tr><td>- (a + b · c)</td></tr><tr><td>c</td></tr><tr><td>b</td></tr></table>	- (a + b · c)	c	b	%st(2) %st(1) %st(0)	
- (a + b · c)							
c							
b							
9	load a	<table border="1"><tr><td>- (a + b · c)</td></tr><tr><td>c</td></tr><tr><td>b</td></tr><tr><td>a</td></tr></table>	- (a + b · c)	c	b	a	%st(3) %st(2) %st(1) %st(0)
- (a + b · c)							
c							
b							
a							
10	multp	<table border="1"><tr><td>- (a + b · c)</td></tr><tr><td>c</td></tr><tr><td>a · b</td></tr></table>	- (a + b · c)	c	a · b	%st(2) %st(1) %st(0)	
- (a + b · c)							
c							
a · b							
11	divp	<table border="1"><tr><td>- (a + b · c)</td></tr><tr><td>a · b/c</td></tr></table>	- (a + b · c)	a · b/c	%st(1) %st(0)		
- (a + b · c)							
a · b/c							
12	multp	<table border="1"><tr><td>a · b/c - - (a + b · c)</td></tr></table>	a · b/c - - (a + b · c)	%st(0)			
a · b/c - - (a + b · c)							
13	storep x	<table border="1"><tr><td></td></tr></table>					

Рис. 3.16. Окончание

РЕШЕНИЕ УПРАЖНЕНИЯ 3.26

Приводимый далее программный код аналогичен коду, построенному компилятором для выбора одного из двух значений на основании исхода тестовой проверки (рис. 3.17).

Полученный результат в вершине стека есть x ? a : b.

1	test %eax, %eax	<table border="1"><tr><td>b</td></tr></table>	b	%st(1)
b				
2	jne L11	<table border="1"><tr><td>a</td></tr></table>	a	%st(0)
a				
3	fstp %st(0)	<table border="1"><tr><td>b</td></tr></table>	b	%st(0)
b				
4	jmp L9			
5	L11:			
6	fstp %st(1)	<table border="1"><tr><td>a</td></tr></table>	a	%st(0)
a				
7	L9:			

Рис. 3.17. Решения упражнения 3.26

РЕШЕНИЕ УПРАЖНЕНИЯ 3.27

Код программы, осуществляющей обработку значений с плавающей точкой, достаточно сложен, поскольку необходимо соблюдать требования различных соглашений, касающихся выталкивания operandов из стека, соблюдения порядка следования аргументов и т. п. Эта задача предоставляет вам возможность во всех подробностях изучить некоторые специальные случаи (рис. 3.18).

Этот код вычисляет выражение $x = a \cdot b - c / b$.

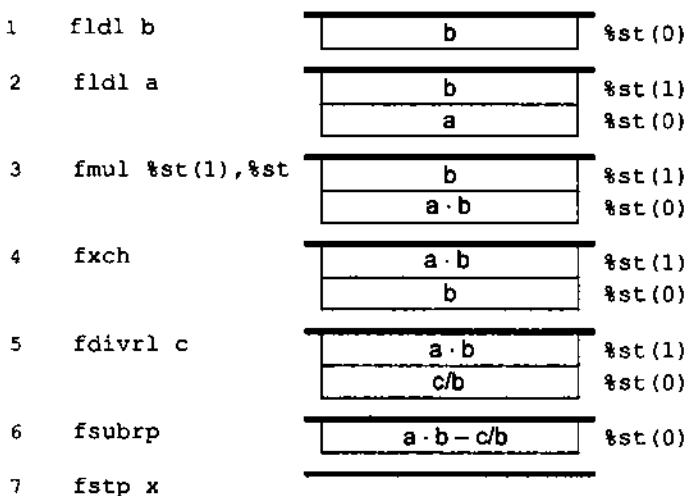


Рис. 3.18. Диаграмма решения упражнения 3.27

РЕШЕНИЕ УПРАЖНЕНИЯ 3.28

Эта задача требует от вас умения пользоваться operandами различных типов в программном коде, выполняющем обработку значений с плавающей точкой.

```
1  double funct2(int a, double x, float b, float i)
2  {
3      return a/(x+b) - (i+1);
4  }
```

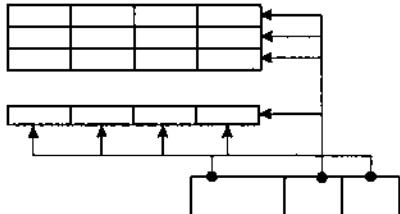
РЕШЕНИЕ УПРАЖНЕНИЯ 3.29

Вставьте между строками 4 и 5 следующий программный код:

```
1  cmpb $1,%ah
```

РЕШЕНИЕ УПРАЖНЕНИЯ 3.30

```
1 int ok_smul(int x, int y, int *dest)
2 {
3     long long prod = (long long) x * y;
4     int trunc = (int) prod;
5
6     *dest = trunc;
7     return (trunc == prod);
8 }
```



ГЛАВА 4

Архитектура процессора

- Архитектура системы команд Y86.
 - Логическое проектирование и язык управления аппаратными средствами HCL.
 - Последовательная реализация Y86.
 - Общие принципы конвейерной обработки.
 - Конвейерная реализация Y86.
 - Резюме.
-

Современные микропроцессоры входят в число наиболее сложных систем, когда-либо созданных человеком. Одна кремниевая микросхема размером с ноготь может содержать в себе полный высокопроизводительный процессор, обширную буферную память, а также логику, необходимую для взаимодействия с внешними устройствами. В том, что касается производительности, на сегодняшний день процессоры, выполненные в виде одной микросхемы, превращают в карликов суперкомпьютеры, занимавшие целые помещения и стоившие больше 10 млн долларов каких-то 20 лет назад. Даже процессоры, встроенные в приборы ежедневного пользования: мобильные телефоны, карманные записные книжки, игровые приставки и т. д. — оказываются намного более мощными, чем могли в свое время предположить разработчики компьютерной техники.

До сих пор авторы книги рассматривали компьютерные системы на уровне программ, написанных на машинном языке. Из предыдущих глав становится понятно, что процессор призван совершать некую последовательность команд, каждая из которых в свою очередь выполняет некоторую примитивную операцию, например, сложение двух чисел. Любая команда кодируется в двоичной форме в виде последовательности одного или более байтов. Команды, поддерживаемые конкретным процессором, и их кодирование на уровне байтов называются архитектурой системы команд (ISA, instruction-set architecture). Разные семейства процессоров, например Intel IA32, IBM/Motorola Power PC и Sun Microsystems SPARC, имеют разные ISA. Программа, скомпилированная для машины одного типа, не будет работать на машине другого типа. С другой стороны, в рамках каждого отдельного семейства существует

вует много различных моделей процессоров. Каждый производитель создает процессоры постоянно повышающейся мощности и сложности, однако разные модели остаются совместимыми на уровне ISA. Популярные семейства, например IA32, имеют процессоры, поставляемые многими производителями. Таким образом, ISA обеспечивает концептуальный уровень абстракции между создателями компиляторов, которым необходимо знать только разрешенные команды и их кодировки, и проектировщиками процессоров, которые должны создавать машины, способные выполнять эти команды.

В данной главе авторы вкратце рассматривают процесс проектирования физических процессоров. Здесь рассматривается способ выполнения системой аппаратных средств команд конкретной ISA. Такой подход поможет лучше понять принципы работы компьютеров, а также разобраться в технологических тонкостях, с которыми приходится сталкиваться производителям компьютерной техники. Одним из важных положений является то, что фактическое функционирование современного процессора может отличаться от расчетов, предполагаемых ISA. Может показаться, что модель ISA предполагает последовательное выполнение команд, где каждая подается и полностью выполняется до поступления следующей. Посредством одновременного выполнения различных частей многих команд процессор достигает большей производительности, нежели при выполнении одной команды за один раз. Для подтверждения того, что результаты расчетов процессора не расходятся с результатами при последовательном выполнении команд, задействованы специальные механизмы. Идея использования хитроумных способов для повышения производительности с одновременным поддержанием функциональности простой и более абстрактной модели хорошо известна в области вычислительной техники. В число примеров входит применение кэширования в браузерах и таких структур информации, как сбалансированные двоичные деревья и хэш-таблицы.

Шансы на то, что кому-то удастся создать свой собственный процессор, невероятно малы. Это задача экспертов и специалистов, работающих в сотне компаний по всему миру. Зачем же тогда изучать проектирование процессоров?

Это представляет интеллектуальный интерес. В изучении принципов функционирования тех или иных систем и приборов присутствует внутренняя ценность. При этом особенно интересно изучать "внутренности" системы, являющейся частью повседневной жизни специалистов и инженеров по вычислительной технике, но которая при этом остается для многих из них тайной за семью печатями. Проектирование процессора охватывает многие принципы хорошей и полезной инженерной практики. Это требует создания максимально простой структуры для решения сложных задач.

Понимание принципов работы процессора способствует представлению работы компьютерной системы в целом. В главе 6 будет рассматриваться система памяти и методологии, применяемые при создании образа памяти очень большого объема с минимальным временем доступа. Рассмотрение интерфейса "процессор-память" еще больше дополнит данное представление.

Несмотря на то, что проектировщиков процессоров очень мало, существует множество систем проектирования аппаратных средств, содержащих процессоры. Это положение уже стало общим местом по мере внедрения процессоров в реальные системы,

например автомобили и бытовую технику. Проектировщики встроенных систем должны понимать принципы работы процессора, потому что, как правило, такие системы проектируются и программируются на более низком уровне абстракции, нежели настольные рабочие станции.

Любой человек мог бы работать над проектированием процессора. Несмотря на небольшое количество компаний-производителей микропроцессоров, рабочие группы проектировщиков постоянно расширяются. Над основной частью проектирования процессора может работать до 800 человек, занимающихся различными его аспектами.

Данная глава начинается с определения простой системы команд, используемой в качестве рабочего примера для реализации процессора. Это называется системой команд Y86, потому что прообразом ее стала система команд IA32, в просторечии называемая X86. По сравнению с IA32, система команд Y86 имеет меньше типов данных, команд и режимов адресации. Здесь также более простые кодировки на уровне байта. Впрочем, Y86 — вполне совершенная система, позволяющая писать более простые программы с обработкой целочисленных данных. Проектирование процессора для реализации Y86 ставит перед разработчиками множество сложных задач.

Далее описываются некоторые основополагающие механизмы проектирования цифровых аппаратных средств. Авторы рассматривают базовые компоненты процессора, а также их взаимодействие и совместное функционирование. Данное представление строится на обсуждении булевой алгебры и операциях на уровне бита, описанных в главе 2. Здесь же вводится понятие языка HCL (Hardware Control Language) для описания блоков управления систем аппаратного обеспечения. Далее этот язык будет использоваться для описания проектов процессоров. С данным разделом стоит ознакомиться для понимания обозначений, приводимых в книге, даже несмотря на наличие у читателя определенной подготовки в области логического проектирования.

В качестве шаблона проектирования процессора представлен функционально корректный, но в определенной степени непрактичный процессор Y86 на основе последовательных операций. Этот процессор выполняет полную команду Y86 в каждом такте цикла. Такт должен быть синхронизирован для того, чтобы вся серия операций имела возможность завершения в рамках одного цикла. Создать такой процессор можно, однако его производительность будет на порядок ниже той, которой можно добиться для подобного устройства.

В результате последовательного проектирования применяется серия преобразований для создания конвейерного процессора. Он разбивает выполнение каждой команды на 5 шагов, каждый из которых управляет отдельной секцией. Команды проходят по конвейеру, одна команда в каждый такт цикла. В результате этого процессор может одновременно выполнять разные шаги пяти команд. Имея целью заставить такой процессор поддерживать последовательное поведение Y86, ISA требует обработки множества разнообразных условий прерывания, когда местоположение или операнды одной команды зависят от местоположения или операндов других команд, находящихся "на конвейере".

Авторы разработали разнообразные инструментальные средства для изучения проектирования процессоров и экспериментов. В их число входит ассемблер для Y86, мо-

делирующее устройство (имитатор) для запуска программ Y86 на компьютере пользователя, а также имитаторы для двух последовательных и одного конвейерного процессоров. Управляющая логика этих проектов представлена файлами в терминах HCL. Путем редактирования этих файлов и перекомпиляции имитатора можно изменить и расширить моделирование поведения. Здесь также представлены упражнения, связанные с реализацией новых команд и модификацией способа обработки команд машиной. Имеется код тестовой программы, предназначенный для оценки корректности вносимых изменений. Все эти упражнения имеют неоценимое значение для полного понимания изложенного материала, а также обеспечат формирование оценки множества вариантов проектирования, с которыми имеют дело разработчики процессоров.

4.1. Архитектура системы команд Y86

Как показано на рис. 4.1, каждая команда в программе Y86 может считывать и модифицировать некоторую часть состояния процессора. Это называется состоянием, видимым для программиста, где "программистом" в данном случае является некто, пишущий программы в компонующем автокоде, либо компилятор, генерирующий код на машинном уровне. При реализации процессора будет видно, что это состояние не будет необходимо представлять и организовывать именно так, как подразумевается ISA, пока есть уверенность в том, что программы на машинном уровне выполняются успешно для состояния, видимого программистом. Состояние для Y86 похоже на состояние для IA32. Существуют восемь *регистров команд*: %eax, %ecx, %edx, %ebx, %esi, %edi, %esp и %ebp. В каждом из них сохранено слово. Регистр %esp используется командами push, pop, call и return в качестве указателя стека. Другими словами, регистры не имеют ни фиксированных значений, ни величин. Имеются три однобайтовых *кода ситуаций*: ZF, SF и OF, в которых хранится информация о влиянии самой последней арифметической или логической команды. Счетчик команд (PC) содержит адрес выполняемой в данный момент команды. В принципе, память как таковая — это обширный массив байтов, содержащий как программу, так и данные.



Рис. 4.1. Состояние, видимое для программиста Y86

На рис. 4.2 кодировки команд варьируются от 1 до 6 байтов. Команда состоит из однобайтового спецификатора, возможно, из однобайтового спецификатора регистра и, вероятно, из четырехбайтового слова-константы. Поле fn обозначает определенную целочисленную операцию (op1) или определенное условие ветви (jxx). Все числовые значения даны в шестнадцатеричной системе.

Байты	0	1	2	3	4	5
nop	0	0				
halt	1	0				
rmovl rA, rB	2	0	rA	rB		
irmovl V, rB	3	0	s	rB	V	
rmovl rA, D(rB)	4	0	rA	rB		D
irmovl D(rB), rA	5	0	rA	rB		D
opl rA, rB	6	fn	rA	rB		
jxx Dest	7	fn			Dest	
call Dest	8	0			Dest	
ret	9	0				
pushl rA	A	0	rA	s		
popl rA	B	0	rA	s		

Рис. 4.2. Система команд Y86

Программы Y86 ссылаются на ячейки памяти, используя виртуальные адреса. Комбинация аппаратных средств и программного обеспечения операционной системы переводит их в фактические или физические адреса, указывающие на места фактического сохранения значений в памяти. Более подробно виртуальная память рассматривается в главе 10. На данный момент о системе виртуальной памяти можно думать как об обеспечении программ Y86 образом сплошного массива байтов.

На рис. 4.2 представлено краткое описание отдельных команд ISA в Y86. Данная система команд служит целью внедрения нашего процессора. По большому счету, система команд Y86 является подмножеством системы команд IA32. Она включает в себя только четырехбайтовые целочисленные операции; здесь меньше режимов адресации и меньший набор операций. Поскольку используются только четырехбайтовые данные, то они называются "словами". На рис. 4.2 показан код команд в левой части и байтовые кодировки — в правой. Ассемблерный код аналогичен коду программ IA32.

Далее описываются некоторые подробности различных команд Y86.

□ Команда IA32 movl делится на четыре разные команды: irmovl, rmovl, imovl и rmovl, явно указывая на форму источника и пункт назначения. Источник может быть непосредственный (i), регистр (r) или память (m). Он обозначается первым символом в имени команды. Пунктом назначения является либо регистр (r), либо

память (m). Он обозначается вторым символом в имени команды. Явное обозначение четырех типов переноса данных оказывается полезным при принятии решений об их реализации. Ссылки памяти на две команды движения памяти имеют простой базовый адрес и адрес смещения. При адресных расчетах не поддерживаются ни второй индексный регистр, ни какое-либо масштабирование значения регистра. Как и в случае с IA32, прямые переносы из одного местоположения памяти в другое не допускаются. Кроме этого, не допускается перенос в память непосредственно получаемых данных.

- Существуют четыре команды для целочисленных операций, показанных на рис. 4.2 в виде OPI: addl, subl, andl и xorl. Они работают только с данными регистров, тогда как IA32 допускает операции с данными памяти. Эти команды задают три кода условий: ZF, SF и OF (ноль, знак и переполнение).
- Семь команд перехода (показаны на рис. 4.2 как jxx): jmp, jle, jl, je, jne, jge и jg. Ветви выбираются в соответствии с типом ветви и настроек кодов условий. Условия ветвей — те же, что и в IA32 (см. рис. 3.11).
- Команда call проталкивает обратный адрес в стек и переходит в адрес назначения. Команда ret осуществляет возврат из такого вызова.
- Команды pushl и popl реализуют проталкивание (push) и выталкивание (pop) точно так же, как и в IA32.
- Команда halt останавливает выполнение любой команды. В IA32 имеется аналогичная команда, называемая hlt. Программным приложениям IA32 не разрешается пользоваться этой командой, поскольку она вызывает остановку работы всей системы. Команда используется в наших программах Y86 для остановки имитатора.

На рис. 4.2 также показана кодировка команд на байтовом уровне. Каждая команда требует от 1 до 6 байтов, в зависимости от того, какие требуются поля. Каждая команда имеет исходный байт, обозначающий тип команды. Этот байт делится на две части, по четыре бита каждая: старшая часть (код) и младшая часть (функция). На рис. 4.2 видно, что диапазон значений кода составляет от 0 до шестнадцатеричного B. Величины функции значимы только в случаях, когда группа взаимосвязанных команд пользуется общим кодом. Они представлены на рис. 4.3, где показаны специфические кодировки целочисленной операции и команды ветвления.

Целые операции	Ветви		
addl 6 0	jmp 7 0	jne 7 4	
subl 6 1	jle 7 1	jge 7 5	
andl 6 2	jl 7 2	jg 7 6	
xorl 6 3	je 7 3		

Рис. 4.3. Рабочие коды для системы команд Y86

Как показано на рис. 4.4, каждый из восьми регистров команд имеет относящийся к нему идентификатор (ID) в диапазоне от 0 до 7. Нумерация регистров в Y86 совпадает с нумерацией в IA32. Регистры команд хранятся в CPU в регистровом файле, не большой области оперативной памяти, где идентификаторы регистров выполняют роль адресов. Значение 8 используется в кодировках команд и при проектировании аппаратных средств, когда необходимо указать, что доступ к регистрам запрещен.

Длина некоторых команд составляет всего лишь один байт, однако у команд, требующих operandов, кодировки длиннее. Во-первых, здесь может присутствовать дополнительный байт спецификатора регистра, задающий один или два регистра. На рис. 4.2 поля этих регистров обозначены rA и rB . Как показывают версии ассемблерного кода команд, они могут задавать регистры, используемые для источников и приемников данных, а также базовый регистр, используемый при вычислении адреса, в зависимости от типа команды. Команды, не имеющие operandов регистров (переходы и `call`), не имеют байта спецификатора регистра. Команды, требующие одного операнда регистра (`irmovl`, `pushl` и `popl`), имеют спецификатор другого регистра, установленный в значение 8. При реализации процессора данное правило окажется полезным.

Рабочие коды на рис. 4.3 определяют конкретную целочисленную операцию или состояние перехода. Эти команды показаны на рис. 4.2 как `OPI` и `jXX`.

В табл. 4.1 каждый из восьми регистров команд имеет идентификатор (ID) в диапазоне от 0 до 7. Идентификатор 8 в поле регистра команды указывает на отсутствие операнда регистра.

Таблица 4.1. Идентификаторы регистров команд Y86

Номер	Название регистра
0	<code>%eax</code>
1	<code>%ecx</code>
2	<code>%edx</code>
3	<code>%ebx</code>
4	<code>%esp</code>
5	<code>%ebp</code>
6	<code>%esi</code>
7	<code>%edi</code>
8	Регистр отсутствует

Некоторые команды требуют дополнительного константного слова, состоящего из четырех байтов. Это слово может служить в качестве непосредственно получаемых данных для `irmovl`, смещения для адресных спецификаторов `jmpovl` и `tmovl`, а также приемником переходов и вызовов. Обратите внимание, что приемники переходов и

вызовов представлены в виде абсолютных адресов, а не как относительная адресация по счетчику команд, имеющая место в IA32. В процессорах используется относительная адресация по счетчику команд для обеспечения большей компактности кодировок команд перехода для того, чтобы код было возможно копировать из одной области памяти в другую без необходимости обновления всех целевых адресов перехода. Поскольку авторы заинтересованы в относительной простоте описания, то здесь используется абсолютная адресация. Как и в IA32, все целые числа обладают "остроконечной" кодировкой. Когда команда написана в так называемой разобранной форме, то эти байты появляются в обратном порядке.

Для примера сгенерируем побайтовую кодировку команды

```
irmovl %esp, 0x12345 (%edx)
```

в шестнадцатеричном виде. Из рис. 4.2 видно, что `irmovl` имеет начальный байт 40. Также видно, что исходный регистр `%esp` должен кодироваться в поле `rA`, а базовый регистр `%edx` должен кодироваться в поле `rB`. Используя номера регистров, показанные в табл. 4.1, получаем байт спецификатора регистра 42. Наконец, смещение кодируется в константном слове, состоящем из четырех байтов. Сначала заполним `0x12345` ведущими нулями до четырех байтов, в результате чего получим последовательность байтов `00 01 23 45`. В обратном порядке это записывается как `45 23 01 00`. После объединения получаем кодировку команды `404245230100`.

Одним из важных свойств любой системы команд является то, что кодировки байтов должны иметь уникальную интерпретацию. Произвольная последовательность байтов либо кодирует уникальную последовательность команд, либо является неразрешенной последовательностью байтов. Это свойство поддерживается и для Y86, потому что каждая команда обладает уникальной комбинацией кода и функции в своем начальном байте, и имея этот байт, можно определить длину и значения любых дополнительных байтов. Данное свойство обеспечивает выполнение процессором программы объектного кода, и значение последнего будет однозначным. Даже если код встроен в другие байты программы, последовательность команд легко определяется до тех пор, пока отсчет ведется с первого байта последовательности. С другой стороны, если начальная позиция последовательности кода неизвестна, тогда нельзя определить с уверенностью способ разбиения этой последовательности на отдельные команды. Это вызывает проблемы для дизассемблеров и других инструментальных средств, пытающихся выделить программы машинного уровня непосредственно из последовательностей байтов объектных кодов.

УПРАЖНЕНИЕ 4.1

Определите побайтовую кодировку следующей последовательности команд Y86. Стока ".pos 0x100" указывает на то, что начальный адрес объектного кода должен быть `0x100`.

```
.pos 0x100#Начало генерирования кода в адресе 0x100
irmovl $15, %ebx
irmovl %ebx, %esx
```

```

loop:
    movl %esx, -3(%ebx)
    addl %ebx, %esx
    jmloop

```

УПРАЖНЕНИЕ 4.2

Для каждой из перечисленных последовательностей байтов определите кодируемую ими последовательность команд Y86. Если в последовательности присутствует неправильный байт, укажите последовательность команд до этого места и местонахождение неправильного значения. Для каждой последовательности дан начальный адрес, двоеточие и сама последовательность байтов.

1. 0x100:3083fcfffff40630008000010
2. 0x200:a06880080200001030830a00000090
3. 0x300:50540700000000f0b018
4. 0x400:6113730004000010
5. 0x500:6362a080

Сравнение кодировки команд IA32 и Y86

По сравнению с кодированием команд, применяемым в IA32, кодирование Y86 намного проще, но и менее компактно. Поля регистров возникают только в фиксированных позициях во всех командах Y86, тогда как в разных командах IA32 они пакуются в разные позиции. Используется 4-битовое кодирование регистров, несмотря на то, что возможных регистров всего 8. В IA32 используется только 3 бита. Следовательно, IA32 может паковать стековую команду только в 1 байт с 5-битовым полем, указывающим тип команды, и оставшимися 3 битами для спецификатора регистра. IA32 может кодировать постоянные величины в 1, 2 или 4 байтах, тогда как Y86 всегда требует 4 байтов.

Системы команд RISC и CISC

Иногда IA32 называют архитектурой с "полным набором команд" (Complex Instruction Set Computer — CISC) и считают противоположностью ISA, которую классифицируют как архитектуру "с сокращенным набором команд" (Reduced Instruction Set Computers — RISC). Исторически машины CISC появились первыми, став потомками самых первых компьютеров. К началу 80-х годов системы команд для универсальных вычислительных машин и мини-ЭВМ стали активно расширяться по мере того, как проектировщики внедряли новые команды, поддерживающие задачи высокого уровня, такие как манипуляции с кольцевыми буферами, выполнение операций десятичной арифметики и вычисление многочленов. Первые микропроцессоры появились в начале 1970-х годов и имели ограниченные системы команд, потому что в то время технология интегральных схем накладывала серьезные ограничения на то, что вообще можно было реализовать в одной микросхеме. Микропроцессоры развивались очень быстро и к началу 80-х годов пошли по пути повышения сложности систем команд, заданному универсальными и мини-ЭВМ. Семейство 80×86 выбрало этот путь, в результате чего появилась IA32.

Эта система также продолжает развиваться с добавлением новых классов команд в поддержку обработки информации, требуемой мультимедийными приложениями.

Философия проектирования RISC также появилась в начале 80-х годов в качестве альтернативы описанным тенденциям. Под влиянием идей исследователя Джона Кокка группы экспертов по аппаратным средствам и компиляторам компании IBM обнаружила, что можно сгенерировать эффективный код для гораздо более простой формы системы (набора) команд. Фактически, многие команды высокого уровня, добавляемые к системам команд, было очень трудно генерировать с помощью компилятора, поэтому они использовались редко. Более простую систему команд можно было реализовать с гораздо меньшим количеством аппаратных средств и организовать ее в эффективную конвейерную структуру, похожую на ту, что описывается в настоящей главе далее. Эту идею IBM поставила на коммерческую основу только спустя много лет, когда были созданы ISA Power и PowerPC.

В дальнейшем концепция RISC развивалась профессорами Дэвидом Паттерсоном в университете Беркли и Джоном Хеннеси из Стенфордского университета. Именно Паттерсон присвоил название RISC новому классу машин, а CISC — существующему, поскольку раньше не было потребности в особом обозначении почти универсальной формы системы команд.

Сравнивая CISC с первоначальными системами команд RISC (табл. 4.2), обнаруживаем следующие общие характеристики:

Таблица 4.2. Характеристики сравнения

CISC	Ранняя RISC
Большое количество команд. Документация Intel, описывающая полную систему команд [19], насчитывает свыше 700 страниц	Намного меньшее количество команд. Обычно менее 100
Некоторые команды имеют продолжительное время выполнения. Сюда входят команды, копирующие целый блок из одной области памяти в другую, а также команды, копирующие многочисленные регистры в память и из памяти	Отсутствие команд с продолжительным временем выполнения. На некоторых ранних RISC-машинах не было даже команды умножения целых чисел, по причине чего для реализации функции умножения как последовательности приращений требовались компиляторы
Кодирование с переменной длиной слова. Команды IA32 могут варьироваться от 1 до 15 байтов	Кодирование с фиксированной длиной слова. Как правило, все команды кодировались в четырех битах
Многочисленные форматы для задания операндов. В IA32 спецификатор операнда памяти может иметь много разных комбинаций смещения, базового и индексного регистров и масштабного коэффициента	Простые форматы адресации. Обычно присутствует только базовая адресация и адресация смещения

Таблица 4.2 (окончание)

CISC	Ранняя RISC
Арифметические и логические операции могут применяться как к operandам ячеек памяти, так и регистров	Арифметические и логические операции используют только operandы регистров. Ссылка на ячейки памяти допускается только командами загрузки, считающими информацию из памяти в регистр, и командами сохранения, записывающими информацию из регистра в память. Это правило называется архитектурой load/store (загрузка/хранение)
Артефакты реализации скрыты от программ машинного уровня. ISA обеспечивает непосредственное отношение между программами и способом их выполнения	Артефакты реализации видны для программ машинного уровня. На некоторых RISC-машинах запрещены определенные последовательности команд, и на них имеются переходы, не имеющие силы до выполнения следующей команды. Компилятор оптимизирует производительность в рамках указанных ограничений
Коды условий. Специальные флагги устанавливаются как побочный эффект команд, после чего используются для проверки условий ветвления	Отсутствие кодов условий. Вместо этого для оценки условий используются явные команды тестирования, которые сохраняют результат тестирования в обычном регистре
Связывание процедур со стековой организацией. Стек используется для аргументов процедур и обратных адресов	Связывание процедур с регистровой организацией. Регистры используются для аргументов процедур и обратных адресов. Поэтому некоторые процедуры могут избегать любых ссылок на память. Обычно процессор имеет гораздо больше регистров (до 32)

Система команд Y86 включает в себя атрибуты наборов команд как RISC, так и CISC. От CISC получены коды условий, команды с переменной длиной слова и связывание процедур со стековой организацией. От RISC взята архитектура load/store и регулярное кодирование. Y86 можно рассматривать как систему команд CISC (IA32), упрощенную использованием некоторых принципов RISC.

Разногласия RISC и CISC

В 1980-х годах прошлого века в кругах специалистов по разработке компьютерной архитектуры не прекращалась жаркая полемика относительно преимуществ наборов команд RISC и CISC. Сторонники RISC заявляли, что объединением упрощенного проектирования систем команд, передовой технологии компилирования и реализации конвейерного процессора можно значительно повысить вычислитель-

ную мощность определенного набора аппаратных средств. Приверженцы CISC возражали, что для решения отдельно взятой задачи требуется меньше команд CISC, и поэтому совокупная производительность машин этого типа выше.

Крупные компании, в число которых входят Sun Microsystems (SPARC), IBM, Motorola (PowerPC) и Digital Equipment Corporation (Alpha), внедрили линии производства процессоров RISC.

В начале 90-х годов дебаты поутихли после того, как стало понятно, что ни RISC, ни CISC в чистом виде не превосходили проекты, объединяющие лучшие идеи обоих типов архитектуры. Появились машины RISC, где было представлено больше команд, на выполнение многих из которых требовалась многочисленные циклы. Сегодня в распоряжении RISC-машин имеются сотни команд, что уже едва ли можно связать с самим названием архитектуры "сокращенным набором команд". Идея выставления артефактов реализации на обозрение программ машинного уровня оказалась недальновидной. По мере разработки новых моделей процессоров с использованием более совершенных структур аппаратных средств, многие из этих артефактов оказались попросту ненужными, что, впрочем, не помешало им остаться частью системы команд. Ядром архитектуры RISC по-прежнему осталась система команд, хорошо подходящая для выполнения на конвейерной машине.

В CISC-машинах, появившихся позднее, также используются преимущества высокопроизводительных конвейерных структур. В разд. 5.7 будет рассматриваться выборка команд CISC и их динамический перевод в последовательность упрощенных операций, подобных имеющимся в архитектуре RISC. Например, команда, добавляющая регистр в память, переводится в три операции: одну — для считывания начального значения памяти, другую — для выполнения сложения и третью — для записи суммы в память. Поскольку динамический перевод может выполняться гораздо раньше фактического выполнения команды, процессор способен поддерживать очень высокую скорость выполнения.

Помимо технологических тонкостей, коммерческие вопросы также сыграли немаловажную роль при определении успешности представления на рынок различных систем команд. Путем поддержания совместимости с существующими моделями процессоров Intel и IA32 удалось довольно безболезненно перейти от одного их поколения к другому. По мере развития технологии интегральных схем Intel и другие производители процессора IA32 смогли преодолеть неэффективность первоначального проекта систем команд 8086 путем применения методик RISC с целью обеспечения производительности, достигаемой лучшими RISC-машинами. В области же разработки настольных рабочих станций и ноутбуков господство IA32 можно считать безраздельным.

Процессоры RISC пользуются огромной популярностью на рынке встроенных процессоров, применяемых во множестве устройств: в мобильных телефонах, тормозных системах автомобилей, средствах выхода в Internet и т. д. Во всех этих областях сокращение стоимости и экономия мощности более важны, чем поддержание обратной совместимости. Исходя из количества проданных процессоров, можно сказать, что у этого рынка весьма успешное будущее.

В листингах 4.1—4.3 приведены компонующие автокоды IA32 и Y86 для функции суммирования в C:

Листинг 4.1 Программный код на C

```
int Sum (int *Start, int Count)
{
int sum = 0;
while (Count) {
sum += *Start;
Start++;
Count--;
}
return sum;
}
```

Листинг 4.2 Код x86

```
1 Sum:
2 pushl %ebp
3 movl %esp, %ebp
4 movl 8(%ebp), %ecx cx = Start
5 movl 12(%ebp), %edx edx = Count
6 xorl %eax, %eax sum = 0
7 testl %edx, %edx
8 je .L34
9 .L35:
10 addl (%ecx), %eax add *Начать суммирование
11 addl $4, %ecx Start++
12 decl %edx Count--
13 jnz .L35 Остановиться при 0
14 .L34:
15 movl %ebp, %esp
16 popl %ebp
17 ret
```

Листинг 4.3 Код x86

```
int Sum (int *Start, int Count)
1 Sum:
2 pushl %ebp
3 rrmovl %esp, %ebp
4 mrmovl 8(%ebp), %ecx ecx = Start
5 mrmovl 12(%ebp), %edx edx = Count
6 xorl %eax, %eax sum = 0
7 andl %edx, %edx
8 je End
```

```

9 Loop:
10 irmovl (%ecx), %esi get * Start
11 addl %esi, %eax прибавить к сумме
12 irmovl $4, %ebx
13 addl %ebx, %ecx Start++
14 irmovl $ - 1, %ebx
15 addl %ebx, %edx Count--
16 jneLoop Остановиться при 0
17 End:
18 rrmovl %ebp, %esp
19 popl %ebp
20 ret

```

Функция Sum высчитывает сумму целочисленного массива. Код Y86 отличается от IA32 в основном тем, что в нем могут потребоваться многочисленные команды для выполнения того, что в IA32 выполняется с помощью одной команды.

Код IA32 генерирован GCC — компилятором C. В принципе, код Y86 — точно такой же, за исключением того, что он иногда требует двух команд для завершения того, что в IA32 выполняется с помощью одной. Однако если программа была написана с использованием индексирования массива, тогда преобразование в код Y86 было бы более сложным, поскольку он не имеет режимов масштабной адресации.

В листинге 4.4 показан пример полного файла программы, написанной в коде Y86. В программе содержатся как данные, так и команды. Директивы указывают на место размещения кода или данных, а также способ выравнивания. Данная программа задает такие положения, как размещение стека, инициализация данных, инициализация программы и останов выполнения программы.

Листинг 4.4. Программа в коде X86

```

1 # Выполнение начинается с адреса 0
2 .pos 0
3 init:irmovl Stack, %esp# Установка указателя стека
4 irmovl Stack, %ebp# Установка базового указателя
5 jmp Main# Выполнение основной программы
6
7 # Массив из четырех элементов
8 .align 4
9 array.long 0xd
10 .long 0xc0
11 .long 0xb00
12 .long 0xa000
13
14 Main:irmovl $4, %eax
15 pushl %eax# Проталкивание 4

```

```

16 irmovl array, %edx
17 pushl %edx# Проталкивание массива
18 call Sum# Сумма (массив, 4)
19 halt
20
21 # int Sum (int *Start, int, Count)
22 Sum:pushl %ebp
23 rrmovl %esp, %ebp
24 mrmovl 8 (%ebp), %ecx# ecx = Start
25 mrmovl 12 (%ebp), %edx# edx = Count
26 irmovl $0, %eax# sum = 0
27 andl %edx, %edx
28 je End
29 Loop:mrmovl (%ecx), %esi# get * Start
30 addl %esi, %eax# прибавить к сумме
31 irmovl $4, %ebx#
32 addl %ebx, %ecx# Start++
33 irmovl $ - 1, %ebx#
34 addl %ebx, %edx# Count--
35 jneLoop# Остановиться при 0
36 End:popl %ebp
37 ret
38 .pos 0x100
39 Stack:# Здесь проходит стек

```

Здесь слова, начинающиеся с символа точки, являются директивами ассемблера, информирующими его о необходимости настройки адреса, в котором необходимо скомпилировать код, либо вставить слова или данные. Директива .pos 0 (строка 2) указывает на то, что ассемблер должен начать генерирование кода, начиная с адреса 0. Это — начальная точка всех программ Y86. Следующие две команды (строки 3 и 4) инициализируют стек и указатели фрейма. Можно заметить, что метка Stack объявлена в конце программы (строка 39) для указания на адрес 0x100, используя директиву .pos (строка 38). Таким образом, данный стек будет начинаться с этого адреса и уменьшаться.

В строках с 8 по 12 программы объявляется массив из четырех слов со значениями 0xd, 0xc0, 0xb00 и 0xa000. Метка array обозначает начало массива и выровнена по четырехбайтовой границе (используется директива .align). В строках с 14 по 19 показана "основная" процедура, которая вызывает функцию Sum в массив из четырех слов, после чего останавливается.

Данный пример доказывает, что написание программы в Y86 требует от программиста выполнения задач, которые обычно поручаются компилятору, редактору связей и системе поддержки выполнения. К счастью, это делается только в небольших программах, для которых достаточно простых механизмов.

В листинге 4.5 показан результат компоновки кода листинга 4.4 ассемблером, называемым YAS. Для удобства прочтения выходные данные представлены в формате

ASCII. В строках файла компоновки, содержащего команды или данные, объектный код содержит адрес, за которым следуют значения от 1 до 6 байтов.

Листинг 4.5. Ассемблерный код

```

# Выполнение начинается с адреса 0
.pos 0
init:irmovl Stack, %esp# Установка указателя стека
irmovl Stack, %ebp# Установка базового указателя
jmp Main# Выполнение основной программы

# Массив из четырех элементов
.align 4
array.long 0xd
.long 0xc0
.long 0xb00
.long 0xa000

0x014:
0x014: 0d000000
0x018: c0000000
0x01c: 000b0000
0c020: 00a00000

0x024: 308004000000
0x02a: a008
0x02c: 308214000000
0x032: a028
0x34: 803a000000
0x39: 10

# int Sum (int *Start, int, Count)
Sum:pushl %ebp
rmovl %esp, %ebp
rmovl 8 (%ebp), %ecx# ecx = Start
rmovl 12 (%ebp), %edx# edx = Count
irmovl $0, %eax# sum = 0
andl %edx, %edx
je End
Loop:rmovl (%ecx), %esi# get * Start
addl %esi, %eax# прибавить к сумме
irmovl $4, %ebx#
addl %ebx, %ecx# Start++
irmovl $ - 1, %ebx#
addl %ebx, %edx# Count--
jneLoop# Остановиться при 0
End:popl %ebp
ret
.pos 0x100
Stack:# Здесь проходит стек
0x100:

```

Осуществлена реализация имитатора набора команд, называемого YIS. При выполнении на выборочном объектном коде генерируются следующие выходные данные (листиг 4.6):

Листинг 4.6. Выходные данные

```
Остановка через 46 шагов на PC = 0x3a. Исключение "HLT", CC Z=1 S=0 O=0
Изменения в регистрах:
%eax:0x0000000000x0000abcd
%ecx:0x0000000000x00000024
%ebx:0x0000000000xfffffff
%esp:0x0000000000x000000f8
%ebp:0x0000000000x00000100
%esi:0x0000000000x0000a000
Изменения в памяти:
0x00f0:0x0000000000x00000100
0x00f4:0x0000000000x00000039
0x00f8:0x0000000000x00000014
0x00fc:0x0000000000x00000004
```

Данный имитатор распечатывает в регистрах или в памяти только слова, изменяющиеся во время моделирования. Первоначальные значения (здесь они представлены нулями) показаны в левой части, а окончательные — справа. По этим выходным данным видно, что регистр %eax содержит 0xabcd — сумму массива из четырех элементов, переданную в подпрограмму rSum. Кроме этого, видно, что использовался стек, начинающийся с адреса 0x100 и далее уменьшающийся, что вызвало изменения памяти в адресах с 0xf0 по 0xfc.

УПРАЖНЕНИЕ 4.3

Напишите код Y86 для реализации рекурсивной суммы-функции rSum, основываясь на следующем коде C:

```
int rSum (int *Start, int Count)
{
if (Count <= 0)
return 0;
return *Start + rSum (Start+1, Count-1);
}
```

Возможно, будет проще скомпилировать код C на машине IA32, а потом перевести команды в Y86.

УПРАЖНЕНИЕ 4.4

Команда pushl уменьшает значение указателя стека на 4 и записывает значение регистра в память. Не совсем понятно, что процессор должен делать с командой pushl

`%esp`, поскольку заполняемый регистр изменяется этой же командой. Здесь возможно применение двух условий: заполнение первоначального значения `%esp` или заполнение уменьшенного значения `%esp`.

Попробуем решить эту задачу выполнением того же, что сделал бы процессор IA32. Можно попытаться ознакомиться с описанием этой команды в документации Intel, однако будет проще провести эксперимент на реальной машине. Компилятор C не сгенерирует эту команду корректно, поэтому необходимо использовать компонующий код, генерируемый вручную. Как описано в разд. 3.15, лучшим способом вставки небольших частей компонующего кода в программу C будет использование функции `asm` GCC. Далее приведена тестовая программа. Вместо того чтобы пытаться проще объявление `asm`, проще всего будет прочитать компонующий код в предшествующем ему комментарии.

```
int pushtest ()
{
int rval;
/* Вставить следующий компонующий код:
movl %esp, %eax# Сохранить указатель стека
pushl %esp# Протолкнуть указатель стека
popl %edx# Вытолкнуть указатель стека
subl %edx, %eax# 0 или 4
movl %eax, rval# Задать как возвращаемое значение
*/
asm ("movl %%esp, %%eax; pushl %%esp; popl %%edx;
subl %%edx, %%eax; movl %%eax, %0"
: "=r" (rval)
: /* Входных данных нет*/
: "%edx", "%eax");
return rval
}
```

При проведении экспериментов обнаружено, что функция `pushtest` возвращает 0. Что этим подразумевается в отношении поведения команды `pushl %esp` в IA32?

УПРАЖНЕНИЕ 4.5

Подобная же двусмысленность имеет место для команды `popl %esp`. Можно привести `%esp` к значению, считанному из памяти, или к приращенному указателю стека. Как и в упр. 4.4, проведем эксперимент для определения того, как эту команду обработает машина IA32, после чего спроектируем собственную машину Y86 для следования тому же условию.

```
int poptest (int tval)
{
int rval;
/* Вставить следующий компонующий код:
Pushl tval# Сохранить tval на стеке
```

```

movl %esp, %edx# Сохранить указатель стека
popl %esp# Вытолкнуть на указатель стека
movl %esp, rval# Задать вытолкнутое значение как возвращаемое
movl %edx, %esp# Восстановить первоначальный указатель стека
*/
asm ("pushl $1; movl %esp, %edx; popl %%esp;
      movl %%esp, %0; movl %%edx, %%esp"
      : "=r" (rval)
      : "r" (tval)
      : "%edx");
      return rval
)

```

Данная функция всегда возвращает *tval* — значение, переданное как аргумент. Что этим подразумевается в отношении поведения команды *popl %esp*? Какая другая команда Y86 будет демонстрировать точно такое же поведение?

4.2. Логическое проектирование и язык управления аппаратными средствами HCL

При проектировании аппаратных средств для вычисления функций и хранения битов в элементах памяти разных типов применяются электронные схемы. Современная элементно-конструктивная база представляет различные значения битов в форме повышенного или пониженного напряжения в сигнальных шинах. По существующей технологии логическое значение 1 представлено высоким напряжением порядка 1 В, а логическое значение 0 — низким напряжением в 0 В. Для реализации любой цифровой системы необходимы три основных компонента: комбинаторная логика для вычисления функций битов, элементы памяти для хранения битов и сигналы синхронизации для регулирования обновления элементов памяти.

В настоящем разделе представлено краткое описание этих компонентов. Также авторы вводят понятие языка управления аппаратными средствами (*Hardware Control Language*, HCL), используемого для описания управляющей логики различных проектов процессоров. Здесь HCL представлен неформально; подробные материалы помещены в *приложении I*.

Современное логическое проектирование

Когда-то проектировщики аппаратных средств производили расчеты логических схем, рисуя всевозможные диаграммы (сначала карандашом на бумаге, а позже на терминалах компьютерной графики). Сегодня большинство проектов представлены на языке описания программных средств (*Hardware Description Language*, HDL) — текстовой записи, похожей на язык программирования, которая вместе различных шагов программ описывает структуры аппаратного обеспечения. В число наиболее широко используемых языков входит Verilog с синтаксисом, по-

хожим на C, и VHDL с синтаксисом, похожим на синтаксис языка Ada. Изначально эти языки разрабатывались для выражения имитационных моделей цифровых цепей. В середине 80-х годов исследователи разработали программы логического синтеза, способные создавать эффективные логические описания по описаниям HDL. На данном этапе существует много коммерческих программ синтеза, и для генерирования цифровых цепей их использование стало основополагающей методикой. Такой переход от разработки схем вручную к синтезу можно сравнить с переходом от написания программ в компонующем коде к написанию их на языке высокого уровня, когда машинный код генерируется компилятором.

4.2.1. Логические шлюзы

Логическими шлюзами называются базовые вычислительные элементы цифровых цепей. Они генерируют выходные данные, эквивалентные некоторой булевой функции битовых значений на входе. На рис. 4.4 показаны стандартные символы, используемые для булевых функций AND, OR и NOT. Выражения HCL показаны под шлюзами для булевых операций. По схеме понятно, что здесь принят синтаксис для логических операторов в C (см. разд. 2.1.9): "&&" — AND, "| |" — OR и "!" — NOT. Они используются вместо операторов C на уровне бита: &, | и ~, потому что логические шлюзы имеют дело с одноразрядными величинами, а не с целыми словами.

Логические шлюзы всегда активны. При изменении входящих в шлюз данных через какое-то минимальное время изменяются и выходные данные.

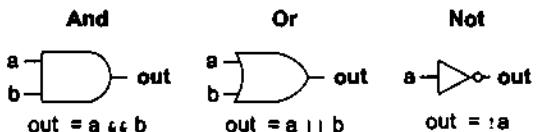


Рис. 4.4. Типы логических шлюзов

Каждый шлюз генерирует выходные данные, эквивалентные некоторой булевой функции входящих данных.

4.2.2. Комбинационные схемы и булевы выражения HCL

При объединении нескольких логических шлюзов в сеть можно построить вычислительные блоки, известные как *комбинационные ряды*. При создании рядов следует учитывать два ограничения:

- Два или более логических шлюза нельзя соединять между собой, иначе они могут попытаться сменить направление шины на противоположное, что может вызвать возникновение некорректного напряжения или сбоя в цепи.
- Цепь должна быть *ацикличной*. При этом бесконечные циклы невозможны.

На рис. 4.5 показан пример простой комбинационной схемы, которая может оказаться довольно полезной. Она имеет два входа: a и b. Она создает единый вывод eq та-

кой, что он будет равен 1, если либо a и b равны единице (что определяется верхним шлюзом AND), либо они равны нулю (что определяется нижним шлюзом AND). На языке HCL функцию этой сети можно записать так:

```
bool eq = (a && b) || (!a && !b);
```

Данный код просто определяет сигнал на уровне бита eq (обозначенный типом данных `bool`) как функцию значений a и b на входе. Как показывает пример, HCL используется синтаксис, подобный синтаксису в C, где символ равенства ассоциирует название сигнала с выражением. Впрочем, в отличие от C, это не рассматривается как выполнение расчетов и присвоение результата некоторой ячейке памяти. Это — всего лишь способ присвоения выражению имени.

Выход будет равен единице, когда оба значения на входе равны нулю или единице.

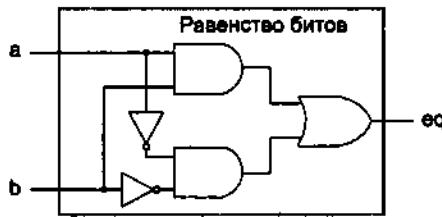


Рис. 4.5. Комбинационная схема для проверки равенства битов

УПРАЖНЕНИЕ 4.6

Напишите выражение HCL для сигнала xor , эквивалентного операции EXCLUSIVE-OR входных значений a и b . Каково соотношение между сигналами xor и описанным выше eq ?

На рис. 4.6 приведен другой пример простой, но полезной комбинационной схемы под названием *мультиплексор*. Он выбирает значение из набора разных сигналов данных, в зависимости от значения управляющего входного сигнала. В этом однобитовом мультиплексоре двумя сигналами данных являются входящие биты a и b , а управляющим сигналом является входной бит s . Вывод будет равен следующей величине:

- a — при s равном единице;
- b — при s равном нулю.

В данной цепи можно заметить, что два шлюза AND определяют, передавать ли соответствующие им входные данные на шлюз OR. Верхний шлюз AND передает сигнал b , когда s равен нулю (поскольку другие входные данные на шлюзе $!s$), а нижний шлюз AND передает сигнал a , когда s равен единице. Опять же, для выходного сигнала можно написать выражение на HCL, используя те же операции, что и комбинационной схеме:

```
bool out = (s && a) || (!s && b);
```

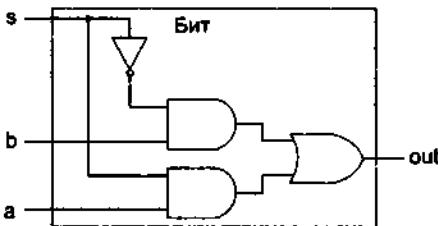


Рис. 4.6. Цепь однобитового мультиплексора

Написанные здесь выражения HCL демонстрируют четкую параллель между цепями комбинаторной логики и логическими выражениями в С. Для вычисления функций входных данных и те, и другие применяют булевы операции. Стоит отметить несколько различий между этими двумя способами выражения расчетов.

- Поскольку комбинационный ряд состоит из серии логических шлюзов, он обладает свойством непрерывной реакции выходных данных на изменения во входных данных. При изменении определенных входных данных по истечении минимального количества времени выходные данные изменяются соответствующим образом. В отличие от этого, выражение С рассчитывается только при его появлении во время выполнения программы.
- Логические выражения в С допускают в качестве аргументов произвольные целые числа, рассматривая 0 как "Ложно", а все, кроме нуля — как "Истинно". Логические же выводы оперируют только с битовыми значениями 0 и 1.
- Логические выражения в С можно рассчитывать и частично. Если выход операции AND и OR можно определить по расчету только первого аргумента, тогда второй аргумент не рассчитывается. Например, в выражении С

`(a && !a) && func (b, c)`

функция `func` вызываться не будет, потому что выражение `(a && !a)` равно 0. Комбинаторная логика, наоборот, не обладает никакими правилами частичных расчетов. Выводы просто реагируют на изменения входных данных.

4.2.3. Комбинационные схемы на уровне слова и целочисленные выражения в HCL

При построении больших цепей логических шлюзов можно создавать комбинационные ряды, способные рассчитывать намного более сложные функции. Обычно проектируются цепи, оперирующие с информационными словами. Это — группы сигналов на уровне бита, представляющие целое число или определенную контрольную комбинацию. Например, рассматриваемые здесь проекты процессора будут содержать большое количество слов с длинами в диапазоне от 4 до 32 битов, представляющие целые числа, адреса, коды команды и идентификаторы регистров.

Для выполнения расчетов на уровне слова комбинационные схемы строятся с использованием логических шлюзов для вычисления отдельных битов слова на выводе,

исходя из отдельных битов слова на входе. Например, на рис. 4.7 показана комбинационная схема проверки того, равны ли 32-битовые слова A и B. То есть, выходной сигнал будет равен 1, если, и только если каждый бит A является равным соответствующему биту B. Эта цепь реализована с использованием 32 однобитовых схем равенства, представленных на рис. 4.5. Выходные данные этих однобитовых цепей объединены с выводом AND для формирования вывода цепи.

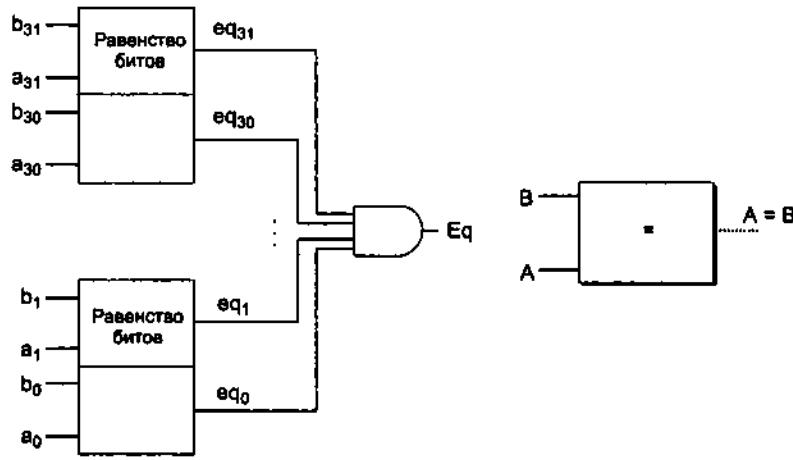


Рис. 4.7. Контрольная схема равенства на уровне слова

В HCL любой сигнал на уровне слова объявляется как `int` без указания длины слова. Это делается исключительно для простоты. В полнофункциональном языке описания программных средств можно объявить, что каждое слово имеет определенное количество битов. HCL позволяет сравнивать слова на предмет их равенства, поэтому функциональность цепи, показанной на рис. 4.7, можно описать на уровне слов, как

`bool Eq = (A == B);`

где аргументы A и B имеют тип `int`. Обратите внимание, что здесь используются те же соглашения о синтаксисе, что и в C, где знак равенства обозначает присвоение (распределение), тогда как `==` обозначает оператор равенства.

Как показано в правой части рис. 4.7, схемы на уровне слова используют полужирные линии для представления системы проводов (шин), передающих отдельные биты слова, а окончательные булевые сигналы в виде линейного пунктира.

УПРАЖНЕНИЕ 4.7

Предположим, что необходимо реализовать схему равенства на уровне слова, используя EXCLUSIVE-OR из упр. 4.6, а не на уровне бита. Спроектируйте такую схему для 32-битового слова, состоящего из схем EXCLUSIVE-OR 32-битового уровня и двух дополнительных логических шлюзов.

На рис. 4.8 представлена схема мультиплексора на уровне слова. Она генерирует 32-битовое слово Out, равное одному из двух входных слов a или b, в зависимости от бита s, управляющего входом. Данная схема состоит из 32 одинаковых подсхем, каждая из которых имеет структуру побитового мультиплексора, изображенного на рис. 4.6. Вместо точного воспроизведения побитового мультиплексора 32 раза версия на уровне слова сокращает число инверторов (обратных преобразователей) однократным созданием !s и подстановкой его в позиции каждого бита.

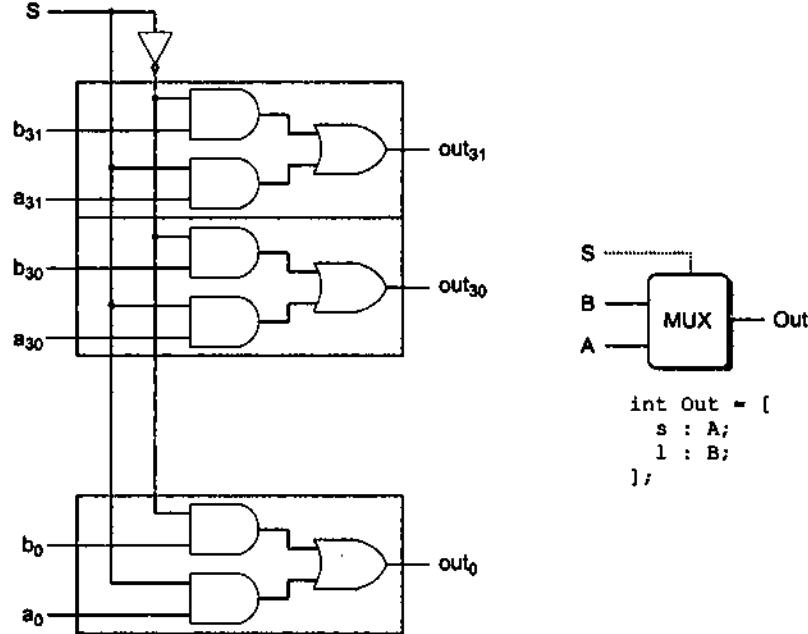


Рис. 4.8. Схема мультиплексора на уровне слова

Выходные данные будут равны входному слову A, когда управляющий сигнал равен единице; в противном случае он будет равен B. Мультиплексоры описываются в HCL с использованием выражений для каждого конкретного случая.

В проектах процессоров, представленных в данной книге, будут использованы многие формы. Они позволяют выбрать слово из множества источников, в зависимости от некоторого количества управляющих условий. Функции мультиплексора описаны в HCL в виде *case-ветвления*. Оно имеет общую форму, представленную как

```
[  
select1:expr1  
select2:expr2  
:  
selectk:exprk  
]
```

Данное выражение содержит набор случаев, где каждый случай і состоит из булевого выражения `select`, указывающего на то, когда должен быть выбран этот случай, и целочисленного выражения `expr{i}`, указывающего на значение результата.

В отличие от оператора выбора С, здесь не требуется, чтобы выбираемые выражения были взаимно исключающими. Логически подбираемые выражения оцениваются последовательно, когда выбирается случай для первого из них, дающего в результате единицу. Например, мультиплексор на уровне слова, показанный на рис. 4.8, можно описать на HCL как:

```
int Out = [
s: A;
1: B;
];
```

В данном примере второе выбираемое выражение — просто единица, указывающая на то, что выбран должен быть именно этот случай, если не был выбран предыдущий. Так в HCL задается случай по умолчанию, и фактически все case-ветвления заканчиваются так.

Допущение неоднозначных выбираемых выражений упрощает читаемость кода HCL. Реальный мультиплексор аппаратных средств должен иметь взаимоисключающие сигналы, управляющие тем, какое слово на входе должно передаваться на вывод, такие как сигналы `s` и `!s`, показанные на рис. 4.8. Для перевода выражения случая HCL в аппаратные средства программе логического синтеза потребуется проанализировать набор выбираемых выражений и разрешить все возможные противоречия (конфликты) путем выбора только совпадающих случаев.

Выбираемые выражения могут быть произвольными булевыми выражениями, и может быть произвольное число случаев. Это позволяет описывать блоки, где имеется много вариантов входных сигналов со сложными критериями выбора. Например, рассмотрим следующую диаграмму мультиплексора с четырьмя выводами (рис. 4.9):

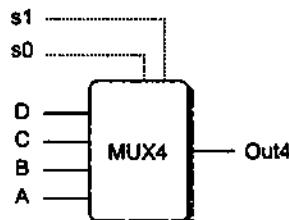


Рис. 4.9. Диаграмма мультиплексора

Данная цепь делает выбор из четырех входных слов: А, В, С и D, исходя из управляющих сигналов `s1` и `s0` и рассматривая их как двухбитовые двоичные числа. Это можно выразить на языке HCL, используя булевые выражения для описания различных комбинаций битов в листинге 4.7:

Листинг 4.7. Комбинации управляющих битов

```
int Out4 = [
!s1 && !s0: A:#00
!s1: B:#01
!s0: C:#10
1: D:#11
];
```

Комментарии справа (любой текст после символа # до конца строки называется комментарием) указывают на то, какая комбинация s1 и s0 вызовет выбор данного случая. Обратите внимание на то, что выбираемые выражения иногда могут упрощаться, поскольку выбирается только первый совпадающий случай. Например, второе выражение можно записать как !s1, а не в более сложной форме !s1 && s0, потому что единственная альтернатива, когда s1 равно 0, представлена в выбираемом выражении.

И наконец, приведем последний пример. Предположим, что нужно спроектировать логическую цепь, находящую минимальное значение в наборе слов A, B и C. В виде диаграммы это выглядит следующим образом (рис. 4.10):

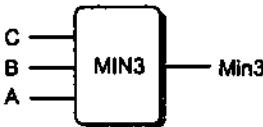


Рис. 4.10. Диаграмма логической цепи

На языке HCL это можно выразить так (листинг 4.8):

Листинг 4.8. Минимальное значение

```
int Min3 = [
A <= B && A <= C:A;
B <= A && B <= C:B;
1 :C;
];
```

УПРАЖНЕНИЕ 4.8

Напишите HCL-код для входящих слов A, B и C, описывающий медиану этих трех значений. Выходной сигнал должен быть равен слову, лежащему между минимальной и максимальной величиной этих трех входных значений.

Цепи комбинаторной логики можно создавать для выполнения множества операций различных типов с данными на уровне слова. Подробное описание этих проектов не

входит в цели книги. Одна важная комбинаторная цепь, известная как арифметико-логическое устройство (АЛУ), представлена на абстрактном уровне на рис. 4.11. Здесь три входных значения: два — данные A и B, а третье — управляющее. В зависимости от настроек управляющего входного значения, цепь будет выполнять различные арифметические или логические операции с входными данными. Обратите внимание, что четыре операции, изображенные для этого АЛУ на диаграмме, соответствуют четырем различным целочисленным операциям, поддерживаемым системой команд Y86, а управляющие значения соответствуют кодам функций для этих команд (рис. 4.3). Также отметьте упорядочивание операндов для вычитания, где входное значение A вычитается из входного значения B. Такое упорядочивание выбрано для будущего упорядочивания аргументов команды `subl`.

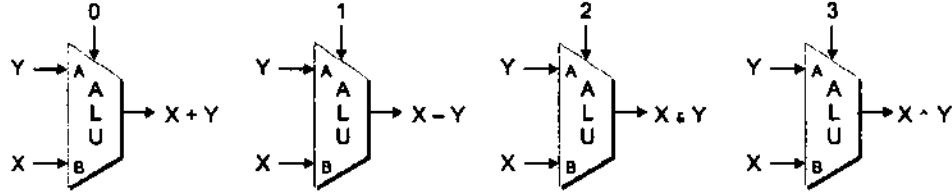


Рис. 4.11. Арифметико-логическое устройство (АЛУ)

4.2.4. Принадлежность множеству

При проектировании процессора будет обнаруживаться много примеров, когда потребуется сравнить один сигнал с некоторым количеством возможных совпадающих сигналов, например, для проверки совпадения какого-либо обрабатываемого кода команды с категорией кодов команды. В качестве простого примера предположим, что необходимо сгенерировать сигналы s_1 и s_0 для мультиплексора с четырьмя выводами, показанного на рис. 4.8, путем выбора старших и младших битов для ниже-приведенного кода двухбитового сигнала (рис. 4.12):

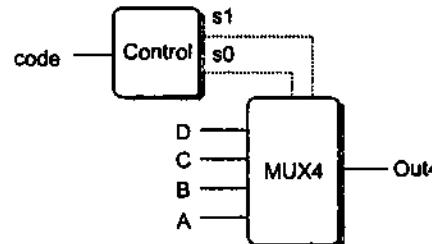


Рис. 4.12. Диаграмма логической цепи

В данной цепи двухбитовый сигнал будет управлять выбором из четырех слов данных A, B, C и D. Генерирование сигналов s_1 и s_0 можно выразить с использованием тестов равенства, основанных на возможных значениях code:

```
bool s1 = code == 2 || code == 3;
bool s0 = code == 1 || code == 3;
```

Можно написать более сокращенное выражение того свойства, что `s1` равно единице, когда `code` принадлежит множеству {2, 3}, а `s0` равно единице, когда `code` принадлежит множеству {1, 3}:

```
bool s1 = code in { 2, 3 };
bool s0 = code in { 1, 3 };
```

Общая форма теста принадлежности множеству выражается формулой:

 $iexpr \in \{iexpr_1, iexpr_2, \dots, iexpr_k\}$

где тестируемое значение `iexpr` и кандидаты на совпадение с `iexpr1` по `iexprk` являются целочисленными выражениями.

4.2.5. Память и синхронизация

По своей природе комбинационные цепи не предназначены для хранения информации. Они просто реагируют на входные сигналы, генерируя выходные сигналы, равные некоторой функции на входе. Для создания схем *последовательного действия*, т. е. системы, имеющей состояние и выполняющей в этом состоянии расчеты, необходимо представить устройства, хранящие информацию, выраженную в битах. Здесь рассматривается два класса запоминающих устройств:

- Синхронизированные регистры (или просто регистры) сохраняют отдельные биты или слова. Синхронизирующий сигнал управляет загрузкой регистра значением на входе.
- В памяти оперативного доступа (или просто памяти) хранятся слова; для определения слова для считывания или записи используются адреса. В число примеров оперативной памяти входит система виртуальной памяти процессора, где комбинация аппаратных и программных средств обеспечивает процессор возможностью доступа к любому слову в рамках определенного обширного адресного пространства, и регистровый файл, где роль адресов выполняют идентификаторы регистров. В процессоре IA32 или Y86 в регистровых файлах содержится восемь программных регистров (`%eax`, `%ecx` и т. д.).

При рассмотрении аппаратных средств относительно программирования на машинном языке становится понятно, что слово "регистр" обозначает в некоторой степени разные вещи. В аппаратном обеспечении регистр напрямую связан со всей цепью каналами ввода и вывода. В программировании на машинном уровне регистры представляют небольшой набор адресуемых слов в центральном процессоре, где адреса состоят из идентификаторов регистров. Обычно эти слова хранятся в регистровом файле, хотя далее в книге описывается, что аппаратные средства иногда могут передавать слово непосредственно из одной команды в другую, во избежание задержки первой записи и последующего считывания регистрового файла. Во избежание двусмысленности, два класса регистров будут называться *аппаратными регистрами* и *регистрами команд* соответственно.

На рис. 4.13 более подробно представлен аппаратный регистр и его функционирование. Большую часть времени данный регистр остается в фиксированном состоянии (изображено как x), генерируя выходные данные, равные текущему состоянию. Сигналы передаются через предшествующую регистру комбинаторную логику, создавая новое значение для входа регистра (изображено как y), однако выходное значение регистра остается фиксированным до тех пор, пока сохраняется низкая синхронизация. При повышении синхронизации сигналы на входе загружаются в регистр по мере перехода в другое состояние (y), превращаясь в новые выходные данные регистра до следующего возрастающего фронта синхроимпульса.

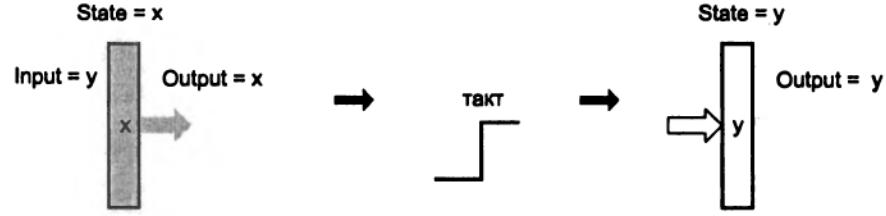


Рис. 4.13. Функционирование регистра

Ключевой точкой является то, что регистры играют роль барьеров между комбинаторной логикой в разных частях цепи. Значения только передаются через вход регистра на выход, как только каждый цикл синхронизации находится на возрастающем фронте синхросигнала.

На рис. 4.14 показан типичный регистровый файл:



Рис. 4.14. Регистровый файл

Данный регистровый файл имеет два *порта считывания*, обозначенных как А и В, и один *порт записи*, обозначенный литерой W. Подобного рода оперативная память обеспечивает множественное считывание и запись операций, выполняемых одновременно. В показанном на рис. 4.14 регистровом файле цепь может считывать значения двух регистров команд и обновлять состояние третьего. Каждый порт имеет адресный вход, указывающий на то, какой должен быть выбран регистр команд, а также выход или вход данных, задающий значение этого регистра команд. Адреса являются идентификаторами регистров, использующих кодирование, показанное в табл. 4.1. Эти два порта считывания имеют адресные входы srcA и srcB (источник А и источник В) и выходные данные valA и valB (значение А и значение В). Порт записи имеет адресный вход dstW (место назначения W) и входные данные valW (значение W).

Несмотря на то, что регистровый файл не является логической цепью (поскольку в нем присутствует внутреннее запоминающее устройство), считывание слов в нем происходит так же, как действует блок комбинаторной логики, имеющий адреса в качестве входных данных, и информацию — в качестве выходных данных. Когда srcA или srcB настроен на какой-либо идентификатор регистра, тогда по истечении некоторого времени значение, сохраненное в соответствующем регистре команд, появится в valA или valB. Например, установка srcA равным 3 вызовет считывание значения регистра команд `verb`, и это значение появится на выходе valA.

Запись слов в регистровый файл регулируется сигналом синхронизации способом, похожим на загрузку значений в синхронизированный регистр. С каждым возрастанием синхронизации значение входного valW записывается в регистр команд, указанный идентификатором регистров на входном dstW. Когда dstW установлено равным особому значению идентификатора 8, тогда никакой регистр команд не записывается.

4.3. Последовательные реализации Y86

Теперь имеются компоненты, необходимые для реализации процессора Y86. В качестве первого шага опишем процессор, называемый SEQ (последовательный, sequential). На каждом цикле синхронизации SEQ выполняет все шаги, необходимые для обработки полной команды. Однако этот период цикла синхронизации довольно продолжительный, поэтому синхронизирующая частота будет неприемлемо низкой. Данной целью разработки SEQ является обеспечение первого шага к конечной цели реализации эффективного конвейерного процессора.

4.3.1. Поэтапная организация процессора

Вообще говоря, обработка команды включает в себя несколько операций. В данном случае они организованы в определенную последовательность шагов, направленную на то, чтобы заставить все команды придерживаться универсальной последовательности, несмотря на то, что команды очень разнятся по выполняемым операциям. Подробная обработка на каждом этапе зависит от каждой конкретно выполняемой команды. Создание такой схемы позволит спроектировать процессор, наиболее оптимально и эффективно использующий возможности аппаратных средств. Далее приведено неформальное описание этих этапов и выполняемых в их рамках операций.

Выборка

На стадии выборки байтычитываются в команду из памяти с использованием счетчика команд (PC) в качестве адреса памяти. Он извлекает из команды четырехбитовые порции ее байта-спецификатора, называемые icode (код команды) и ifun (функция команды). Возможно, что осуществляется выборка байта-спецификатора регистра с получением одного или более спецификаторов операнда регистра гA и гB. Также возможно, что осуществляется выборка четырехбайтового константного слова

`valC`. Просчитывается `valP` в качестве адреса инструкции, следующей за текущей в последовательном порядке. То есть, `valP` равно значению РС плюс длине выбиряемой команды.

Декодирование

Этап декодирования считывает из регистрового файла до двух операндов, выдавая значения `valA` и/или `valB`. Обычно считаются регистры, обозначенные регистровыми полями `gA` и `tB`, однако для некоторых команд считывается регистр `%esp`.

Выполнение

На этапе выполнения арифметико-логическое устройство выполняет операцию, обозначенную командой (в соответствии со значением `ifun`), высчитывает исполнительный адрес ссылки на ячейку памяти либо увеличивает/уменьшает указатель стека. Результирующее значение называется здесь `valE`. Возможно, что заданы коды условий. Для команды перехода на данном этапе осуществляется проверка кодов условий и условий ветвления (задаваемых `ifun`) для проверки необходимости ветвления.

Память

На этапе памяти данные могут записываться в память или считываться из нее. Данное значение здесь называется `valM`.

Обратная запись

На данном этапе осуществляется запись до двух результатов в регистровый файл.

Обновление РС

РС настраивается на адрес следующей команды.

Циклы работы процессора непрерывны; всякий раз выполняются описанные этапы. Он останавливается только при столкновении с командой `halt` или состоянием ошибки. Рассматриваемыми здесь состояниями ошибки являются недействительные адреса памяти (программа или данные) и недействительные команды.

По предыдущему описанию видно, что для выполнения одной-единственной команды требуется на редкость большой объем обработки. Выполнить нужно не только очередную операцию команды, но и просчитать адреса, обновить указатели стека и определить адрес следующей команды. К счастью, общий поток может быть сходным для каждой команды. При проектировании аппаратных средств важно пользоваться очень простой и универсальной структурой, поскольку их общее количество желательно свести к минимуму и полностью отобразить на двумерной поверхности встроенной монтажной микросхемы. Один из способов минимизации сложности — сделать так, чтобы разными командами совместно использовалось как можно больше аппаратных средств. Например, каждый из рассматриваемых проектов процессора содержит одно арифметико-логическое устройство, используемое по-разному, в за-

вистимости от типа выполняемой команды. Затраты на дублирование логических блоков аппаратного обеспечения намного выше затрат на получение многочисленных копий кода программных средств. Также намного сложнее иметь дело с особыми случаями и отличительными особенностями аппаратных средств, нежели программных.

Задачей на данном этапе является организация вычислительного процесса, необходимого для того, чтобы каждая команда соответствовала общей структуре. Для иллюстрации обработки различных команд Y86 будет использован код, показанный в табл. 4.3. В табл. 4.4—4.7 описан переход команд Y86 от одного этапа к другому. Эти таблицы заслуживают тщательного изучения. Они представлены в форме, обеспечивающей непосредственное отображение на аппаратные средства. Каждая строка таблиц описывает присваивание тому или иному сигналу или сохраненному состоянию (обозначенное операцией присваивания \leftarrow). Прочтение следует осуществлять в виде последовательности сверху вниз. Когда впоследствии расчеты будут отображаться на аппаратное обеспечение, тогда станет понятно, что вычисления не нужно выполнять в строгом последовательном порядке.

В табл. 4.3 показана обработка, необходимая для типов команд op1 (целочисленные и логические операции), rmovl (обмен данными между регистрами) и irmovl (мгновенный обмен данными между регистрами). Для начала рассмотрим целочисленные операции. На рис. 4.2 виден тщательный подбор кодировки команд так, что четыре целочисленные операции (addl, subl, andl и xorl) имеют одинаковое значение icode. Их обработку можно осуществить путем применения одинаковой последовательности шагов, за исключением того, что вычисление АЛУ должно выполняться в соответствии с конкретной операцией команды, закодированной в ifun.

Обработка команды целочисленной операции следует за общей комбинацией, представленной в начале раздела. На этапе выборки константного слова не требуется, поэтому valP рассчитывается как PC + 2. На этапе декодированиячитываются оба операнда. Они передаются на АЛУ на этапе выполнения вместе со спецификатором функции ifun так, что valE превращается в результат команды.

Таблица 4.3. Типовая последовательность команды Y86

1	0x000: 308209000000	irmovl \$9, %edx	
2	0x006: 308315000000	irmovl \$21, %ebx	
3	0x00c: 6123	subl %edx, %ebx	# Вычитание
4	0x00e: 308480000000	irmovl \$128, %esp	# Упражнение 4.9
5	0x014: 404364000000	rmovl %esp, 100(%ebx)	# Сохранение
6	0x01a: a028	pushl %edx	# Проталкивание
7	0x01c: b008	popl %eax	# Упражнение 4.10
8	0x01e: 7328000000	je done	# Не принимается
9	0x023: 8029000000	call proc	# Упражнение 4.13

Таблица 4.3 (окончание)

10	0x028:	done:	
11	0x028: 10	halt	
12	0x029:	proc:	
13	0x029: 90	Ret	# Возврат

Прослеживается обработка данных команд на разных этапах.

Эти команды в табл. 4.4 рассчитывают значение и сохраняют результат в регистре. Обозначение i code: i fun указывает два компонента байта команды, а $rA:rB$ — два компонента байта спецификатора регистра. Обозначение $M_1[x]$ указывает на доступ (для считывания или записи) одного байта в ячейке памяти x , а $M_4[x]$ — на доступ к четырем байтам.

Таблица 4.4. Расчеты последовательной реализации команд Y86

Этап	$OP1\ rA, rB$	$rrmovl\ rA, rB$	$irmovl\ V, rB$
Выборка	i code: i fun $\leftarrow M_1[PC]$ $rA: rB \leftarrow M_1[PC + 1]$ $valP \leftarrow PC + 2$	i code: i fun $\leftarrow M_1[PC]$ $rA: rB \leftarrow M_1[PC + 1]$ $valP \leftarrow PC + 2$	i code: i fun $\leftarrow M_1[PC]$ $rA: rB \leftarrow M_1[PC + 1]$ $valC \leftarrow M_4[PC + 2]$ $valP \leftarrow PC + 6$
Декодирование	$valA \leftarrow R[rA]$ $valB \leftarrow R[rB]$	$valA \leftarrow R[rA]$	
Выполнение	$valE \leftarrow valB\ OP\ valA$ Set CC	$valE \leftarrow 0 + valA$	$valE \leftarrow 0 + valC$
Память			
Обратная запись	$R[rB] \leftarrow valE$	$R[rB] \leftarrow valE$	$R[rB] \leftarrow valE$
Обновление PC	$PC \leftarrow valP$	$PC \leftarrow valP$	$PC \leftarrow valP$

Данный расчет показан в виде выражения $valB\ OP\ valA$, где OP указывает на операцию, обозначенную i fun. Обратите внимание на упорядочивание двух аргументов: этот порядок соответствует условию Y86 (и IA32). Например, предполагается, что команда `subl %eax, %edx` рассчитывает значение $R[%edx] - R[%eax]$. На этапе памяти для этих команд ничего не происходит, но $valE$ записывается в регистр rB на этапе обратной записи, а PC устанавливается на $valP$ для завершения выполнения команды (табл. 4.5).

Таблица 4.5. Расчеты последовательной реализации команд Y86 чтения и записи

Этап	<code>rrmovl rA, D, (rB)</code>	<code>rmovl D, (rB), rA</code>
Выборка	$\text{icode: ifun} \leftarrow M_1[\text{PC}]$ $rA: rB \leftarrow M_1[\text{PC} + 1]$ $\text{valC} \leftarrow M_4[\text{PC} + 2]$ $\text{valP} \leftarrow \text{PC} + 6$	$\text{icode: ifun} \leftarrow M_1[\text{PC}]$ $rA: rB \leftarrow M_1[\text{PC} + 1]$ $\text{valC} \leftarrow M_4[\text{PC} + 2]$ $\text{valP} \leftarrow \text{PC} + 6$
Декодирование	$\text{valA} \leftarrow R[rA]$ $\text{valB} \leftarrow R[rB]$	$\text{valB} \leftarrow R[rB]$
Выполнение	$\text{valE} \leftarrow \text{valB} + \text{valC}$	$\text{valE} \leftarrow \text{valB} + \text{valC}$
Память	$M_4[\text{valE}] \leftarrow \text{valA}$	$\text{valM} \leftarrow M_4[\text{valE}]$
Обратная запись		$R[rA] \leftarrow \text{valM}$
Обновление PC	$\text{PC} \leftarrow \text{valP}$	$\text{PC} \leftarrow \text{valP}$

Команды в табл. 4.6 описывают операции со стеком.

Таблица 4.6. Расчеты последовательной реализации команд Y86 работы со стеком

Этап	<code>pushl rA</code>	<code>popl rA</code>
Выборка	$\text{icode: ifun} \leftarrow M_1[\text{PC}]$ $rA: rB \leftarrow M_1[\text{PC} + 1]$ $\text{valP} \leftarrow \text{PC} + 2$	$\text{icode: ifun} \leftarrow M_1[\text{PC}]$ $rA: rB \leftarrow M_1[\text{PC} + 1]$ $\text{valP} \leftarrow \text{PC} + 2$
Декодирование	$\text{valA} \leftarrow R[rA]$ $\text{valB} \leftarrow R[\%esp]$	$\text{valA} \leftarrow R[\%esp]$ $\text{valB} \leftarrow R[\%esp]$
Выполнение	$\text{valE} \leftarrow \text{valB} + (-4)$	$\text{valE} \leftarrow \text{valB} + 4$
Память	$M_4[\text{valE}] \leftarrow \text{valA}$	$\text{valM} \leftarrow M_4[\text{valA}]$
Обратная запись	$R[\%esp] \leftarrow \text{valE}$	$R[\%esp] \leftarrow \text{valE}$ $R[rA] \leftarrow \text{valM}$
Обновление PC	$\text{PC} \leftarrow \text{valP}$	$\text{PC} \leftarrow \text{valP}$

Прослеживание выполнения команды `subl`

В качестве примера проследим обработку команды `subl` в 3 строке объектного кода, показанного на рис. 4.15. Видно, что предыдущие две команды инициализируют регистры `%edx` и `%ebx` до 9 и 21 соответственно. Также видно, что данная команда расположена в адресе 0x00c и состоит из двух байтов со значениями 0x61 и 0x23. Этапы выполняются, как показано в таблице ниже, в которой приведено правило обработки команды `OP1` (см. табл. 4.3) в левой части и расчеты этой команды — в правой.

Расчеты последовательной реализации команд `Y86`: `jxx`, `call` и `ret` приведены в табл. 4.7.

Таблица 4.7. Расчеты последовательной реализации команд `Y86` передачи управления

Этап	<code>jxx Dest</code>	<code>call Dest</code>	<code>ret</code>
Выборка	$\text{icode: ifun} \leftarrow M_1[\text{PC}]$ $\text{valC} \leftarrow M_4[\text{PC} + 1]$ $\text{valP} \leftarrow \text{PC} + 5$	$\text{icode: ifun} \leftarrow M_1[\text{PC}]$ $\text{valC} \leftarrow M_4[\text{PC} + 1]$ $\text{valP} \leftarrow \text{PC} + 5$	$\text{icode: ifun} \leftarrow M_1[\text{PC}]$ $\text{valP} \leftarrow \text{PC} + 1$
Декодирование		$\text{valB} \leftarrow R[\%esp]$	$\text{valA} \leftarrow R[\%esp]$ $\text{valB} \leftarrow R[\%esp]$
Выполнение	$Bch \leftarrow \text{Cond(CC, ifun)}$	$\text{valE} \leftarrow \text{valB} + (-4)$	$\text{valE} \leftarrow \text{valB} + 4$
Память		$M_4[\text{valE}] \leftarrow \text{valP}$	$\text{valM} \leftarrow M_4[\text{valA}]$
Обратная запись		$R[\%esp] \leftarrow \text{valE}$	$R[\%esp] \leftarrow \text{valE}$
Обновление PC	$\text{PC} \leftarrow Bch ? \text{valC} : \text{valP}$	$\text{PC} \leftarrow \text{valC}$	$\text{PC} \leftarrow \text{valM}$

Табл. 4.8 показывает, что достигнут желаемый эффект установки регистра `%ebx` равным 12, установки всех трех кодов условий равными 0 и увеличения PC на 2.

Таблица 4.8. Расчеты последовательной реализации команд `Y86` особых случаев

Этап	Общий	Специфический
	<code>OP1 rA, rB</code>	<code>subl %edx, %ebx</code>
Выборка	$\text{icode: ifun} \leftarrow M_1[\text{PC}]$ $rA: rB \leftarrow M_1[\text{PC} + 1]$ $\text{valP} \leftarrow \text{PC} + 2$	$\text{icode: ifun} \leftarrow M_1[0x00c] = 6:1$ $rA: rB \leftarrow M_1[0x00d] = 2:3$ $\text{valP} \leftarrow 0x00c + 2 = 0x00e$

Таблица 4.8 (окончание)

Этап	Общий	Специфический
	OPL rA, rB	subl %edx, %ebx
Декодирование	valA $\leftarrow R[rA]$ valB $\leftarrow R[%esp]$	valA $\leftarrow R[%edx] = 9$ valB $\leftarrow R[%ebx] = 21$
Выполнение	valE $\leftarrow valB \text{ OP } valA$ Set CC	valE $\leftarrow 21 - 9 = 12$ ZF $\leftarrow 0$, SF $\leftarrow 0$, OF $\leftarrow 0$
Память		
Обратная запись	R[rB] $\leftarrow valE$	R[%ebx] $\leftarrow valE = 12$
Обновление PC	PC $\leftarrow valP$	PC $\leftarrow valP = 0x00e$

Выполнение команды `irmovl` в большей степени начинается как арифметическая операция. Однако производить выборку операнда второго регистра не нужно. Вместо этого вход второго АЛУ устанавливается равным 0 и прибавляется к первому, что дает $valE = valA$, после чего это значение записывается в регистровый файл. Аналогичная обработка имеет место и для `irmovl`, за исключением того, что для входа первого АЛУ используется постоянное значение $valC$. Кроме этого, для `irmovl` счетчик команд необходимо увеличить на 6, из-за длинного формата команды. Ни одна из этих команд не меняет коды условий.

УПРАЖНЕНИЕ 4.9

Заполните правый столбец нижеприведенной таблицы для описания обработки команды `irmovl` в строке 4 табл. 4.3:

Этап	Общий	Специфический
	irmovl V, rB	irmovl \$128, %esp
Выборка	icode: ifun $\leftarrow M_1[PC]$ rA: rB $\leftarrow M_1[PC + 1]$ valC $\leftarrow M_4[PC + 2]$ valP $\leftarrow PC + 6$	
Декодирование		
Выполнение	valE $\leftarrow 0 + valC$	
Память		

(окончание)

Этап	Общий	Специфический
	imovl V, rB	imovl \$128, %esp
Обратная запись	R[rB] \leftarrow valE	
Обновление PC	PC \leftarrow valP	

Как выполнение этой команды изменяет регистры и PC?

В табл. 4.5 показана обработка, необходимая для записи в память и считывания команд `imovl` и `imovl`. Поток — тот же, что и раньше, но здесь для прибавления `valC` к `valB` используется АЛУ (арифметико-логическое устройство), что дает исполнительный адрес (сумму смещения и значения базового регистра) операции памяти. На этапе памяти либо записывается значение регистра `valA`, либо считывается значение `valM`.

Выполнение команды `imovl`

Проследим обработку команды `imovl` в строке 5 табл. 4.3. Видно, что предыдущая команда инициализировала регистр `%esp` до 128, тогда как `%ebx` по-прежнему остается 12, как просчитано командой `subl` (строка 3). Также видно, что данная команда расположена в адресе 0x014 и состоит из 6 байтов. Первые два имеют значения 0x40 и 0x43, а оставшиеся четыре являются обратной байтовой версией числа 0x00000064 (десятичное 100). Этапы начинаются, как описано в табл. 4.9.

Таблица 4.9. Расчеты последовательной реализации команды `imovl`

Этап	Общий	Специфический
	<code>imovl rA, (D) rB</code>	<code>imovl %esp, 100(%ebx)</code>
Выборка	$iode: ifun \leftarrow M_1[PC]$ $rA: rB \leftarrow M_1[PC + 1]$ $valC \leftarrow M_4[PC + 2]$ $valP \leftarrow PC + 6$	$iode: ifun \leftarrow M_1[0x014] = 4:0$ $rA: rB \leftarrow M_1[0x015] = 4:3$ $valC \leftarrow M_4[0x016] = 100$ $valP \leftarrow 0x014 + 6 = 0x01a$
Декодирование	$valA \leftarrow R[rA]$ $valB \leftarrow R[rB]$	$valA \leftarrow R[%esp] = 128$ $valB \leftarrow R[%ebx] = 12$
Выполнение	$valE \leftarrow valB + valC$	$valE \leftarrow 12 + 100 = 112$
Память	$M_4[valE] \leftarrow valA$	$M_4[112] \leftarrow 128$

Таблица 4.9 (окончание)

Этап	Общий	Специфический
	гппmovl rA, (D) rB	гппmovl %esp, 100(%ebx)
Обратная запись		
Обновление PC	PC \leftarrow valP	PC \leftarrow 0x01a

В табл. 4.6 показаны этапы, необходимые для обработки команд pushl и popl. Для реализации — это наиболее сложные команды Y86, потому что они включают в себя как доступ к памяти, так и приращение или убавление указателя стека. Несмотря на то, что эти две команды имеют сходные потоки, они характеризуются важными различиями.

Команда pushl во многом начинается так же, как и предыдущие команды, однако на этапе декодирования в качестве идентификатора операнда второго регистра используется %esp, что присваивает указателю стека значение valB. На этапе выполнения для уменьшения значения указателя стека на 4 применяется АЛУ. Это уменьшенное значение используется для записи адреса ячейки памяти и сохраняется назад в %esp на этапе обратной записи. Пользуясь valE в качестве адреса для операции записи, разработчики придерживаются условия Y86 (и IA32) о том, что команда pushl должна уменьшать значение указателя стека до записи, несмотря на то, что фактического обновления указателя стека не происходит до завершения операции памяти.

Выполнение команды pushl

Проследим обработку команды pushl в строке 6 табл. 4.3. В этой точке в регистре %edx имеем 9, а в регистре %esp — 128. Также видно, что данная команда расположена в адресе 0x01a и состоит из двух байтов со значениями 0xa0 и 0x28. Этапы начинаются, как описано в табл. 4.10.

Таблица 4.10. Расчеты последовательной реализации команды pushl

Этап	Общий	Специфический
	pushl rA	pushl %edx
Выборка	icode: ifun \leftarrow M ₁ [PC] rA: rB \leftarrow M ₁ [PC + 1] valP \leftarrow PC + 2	icode: ifun \leftarrow M ₁ [0x01a] = a:0 rA: rB \leftarrow M ₁ [0x01b] = 2:8 valP \leftarrow 0x01a + 2 = 0x01c
Декодирование	valA \leftarrow R[rA] valB \leftarrow R[%esp]	valA \leftarrow R[%edx] = 9 valB \leftarrow R[%esp] = 128

Таблица 4.10 (окончание)

Этап	Общий	Специфический
	pushl rA	pushl %edx
Выполнение	$\text{valE} \leftarrow \text{valB} + (-4)$	$\text{valE} \leftarrow 128 + (-4) = 124$
Память	$M_4[\text{valE}] \leftarrow \text{valA}$	$M_4[124] \leftarrow 9$
Обратная запись	$R[\%esp] \leftarrow \text{valE}$	$R[\%esp] \leftarrow 124$
Обновление PC	$\text{PC} \leftarrow \text{valP}$	$\text{PC} \leftarrow 0x01c$

Табл. 4.10 показывает, что команда имеет эффект задания $\%esp$ равным 124 с записью 9 в адрес 124 и приращения PC на 2.

Команда `popl` начинается во многом так же, как и `pushl`, за исключением того, что на этапе декодирования считаются две копии указателя стека. Это абсолютно излишне, однако видно, что если указатель стека имеет значения valA и valB , то последующий поток становится более похожим на потоки других команд, что обуславливает общее единообразие проектирования. На этапе выполнения для увеличения указателя стека на 4 применяется АЛУ, однако увеличенное значение используется в качестве адреса операции памяти. На этапе обратной записи обновляется как регистр указателя стека с приращенным указателем стека, так и регистр rA со значением, считанным из памяти. Использование не увеличенного указателя стека в качестве адреса считывания из памяти поддерживает условие Y86 (и IA32) о том, что команда `popl` сначала должна считать информацию из памяти, а уже потом выполнить приращение указателя стека.

УПРАЖНЕНИЕ 4.10

Заполните правый столбец таблицы для описания обработки команды `popl` в строке 7 табл. 4.3:

Этап	Общий	Специфический
	<code>popl rA</code>	<code>popl %eax</code>
Выборка	$i\text{code}: i\text{fun} \leftarrow M_1[\text{PC}]$ $rA: rB \leftarrow M_1[\text{PC} + 1]$ $\text{valP} \leftarrow \text{PC} + 2$	
Декодирование	$\text{valA} \leftarrow R[\%esp]$ $\text{valB} \leftarrow R[\%esp]$	

(окончание)

Этап	Общий	Специфический
	popl rA	popl %eax
Выполнение	valE \leftarrow valB + 4	
Память	valM \leftarrow M ₄ [valA]	
Обратная запись	R[%esp] \leftarrow valE R[rA] \leftarrow valM	
Обновление PC	PC \leftarrow valP	

Какое воздействие выполнение данной команды оказывает на регистры и PC?

УПРАЖНЕНИЕ 4.11

Каково будет влияние команды `pushl %esp`, в соответствии с этапами, приведенными в табл. 4.6? Соответствует ли это желаемому поведению для Y86, как определено в упр. 4.4?

УПРАЖНЕНИЕ 4.12

Предположим, что две записи в регистр на этапе обратной записи для `popl` происходят в порядке, показанном в табл. 4.6. Каков эффект выполнения `popl %esp`? Соответствует ли это желаемому поведению для Y86, как определено в упр. 4.5?

В табл. 4.7 показана обработка трех команд передачи управления: разные команды перехода, `call` и `ret`. Понятно, что эти команды можно реализовать в том же потоке, что и предыдущие.

Так же, как и в случае с целочисленными операциями, переходы различаются только анализом, стоит ли перейти к ветвлению. Команда перехода начинается с этапов выборки и декодирования во многом так же, как и предыдущие команды, за исключением того, что она не требует байта спецификатора регистра. На этапе выполнения проверяются коды условия и условие перехода для определения того, принимать или нет ветвление, с выдачей однобитового сигнала Bch. На этапе обновления PC сигнал тестируется, и PC устанавливается равным valC (цель перехода), если сигнал равен единице, и valP (адрес следующей команды), если значение равно нулю. Представление $x : a:b$ подобно условному выражению в C: оно дает a , когда x не равно нулю, и b , когда x равно нулю.

Выполнение команды `je`

Проследим обработку команды `je` в строке 8 табл. 4.3. Все коды условий установлены в 0 командой `subl` (строка 3), поэтому ветвления не будет. Данная команда распо-

ложена в адресе 0x01e и состоит из пяти байтов. Первый имеет значение 0x73, а оставшиеся четыре являются обратной байтовой версией числа 0x00000028 — целевым переходом. Этапы начинаются, как описано в табл. 4.11.

Таблица 4.11. Расчеты последовательной реализации команды je

Этап	Общий	Специфический
	jxx Dest	je 0x28
Выборка	$\text{icode: ifun} \leftarrow M_1[\text{PC}]$ $\text{valC} \leftarrow M_4[\text{PC} + 1]$ $\text{valP} \leftarrow \text{PC} + 5$	$\text{icode: ifun} \leftarrow M_1[0x01e] = 7:3$ $\text{valC} \leftarrow M_4[0x01f] = 0x028$ $\text{valP} \leftarrow 0x01e + 5 = 0x023$
Декодирование		
Выполнение	$Bch \leftarrow \text{Cond}(\text{CC}, \text{ifun})$	$Bch \leftarrow \text{Cond}((0, 0, 0), 3) = 0$
Память		
Обратная запись		
Обновление PC	$\text{PC} \leftarrow Bch ? \text{valC} : \text{valP}$	$\text{PC} \leftarrow 0 ? 0x028 : 0x023 = 0x023$

Табл. 4.11 показывает, что команда имеет приращения PC на 5 значений.

Команды `call` и `ret` имеют некоторое сходство с командами `pushl` и `popl`, за исключением того, что заполняются и выталкиваются значения счетчика команд. Командой `call` заполняется значение `valP` — адрес команды, следующей за командой `call`. На этапе обновления PC последнее устанавливается равным `valC` — назначение вызова. Командой `ret` на этапе обновления PC назначается `valM` — значение, выталкиваемое из стека.

УПРАЖНЕНИЕ 4.13

Заполните правый столбец таблицы для описания обработки команды `call` в строке 9 табл. 4.3:

Этап	Общий	Специфический
	call Dest	call 0x029
Выборка	$\text{icode: ifun} \leftarrow M_1[\text{PC}]$ $\text{valC} \leftarrow M_4[\text{PC} + 1]$ $\text{valP} \leftarrow \text{PC} + 5$	

(окончание)

Этап	Общий	Специфический
	call Dest	call 0x029
Декодирование	$valB \leftarrow R[\%esp]$	
Выполнение	$valE \leftarrow valB + (-4)$	
Память	$M_4[valE] \leftarrow valP$	
Обратная запись	$R[\%esp] \leftarrow valE$	
Обновление PC	$PC \leftarrow valC$	

Какое воздействие выполнение данной команды оказывает на регистры, PC и память?

Выполнение команды ret

Проследим обработку команды *ret* в строке 13 табл. 4.3. Адрес команды 0x029, закодированный в одном байте 0x90. Предыдущая команда *call* устанавливает *esp* равным 124 и сохраняет возвратный адрес 0x028 в адрес памяти 124. Этапы начинаются, как описано в табл. 4.12.

Таблица 4.12. Расчеты последовательной реализации команды ret

Этап	Общий	Специфический
	ret	ret
Выборка	$icode: ifun \leftarrow M_1[PC]$ $valP \leftarrow PC + 1$	$icode: ifun \leftarrow M_1[0x029] = 9:0$ $valP \leftarrow 0x029 + 1 = 0x02a$
Декодирование	$valA \leftarrow R[\%esp]$ $valB \leftarrow R[\%esp]$	$valA \leftarrow R[\%esp] = 124$ $valB \leftarrow R[\%esp] = 124$
Выполнение	$valE \leftarrow valB + 4$	$valE \leftarrow 124 + 4 = 128$
Память	$valM \leftarrow M_4[valA]$	$valM \leftarrow M_4[124] = 0x028$
Обратная запись	$R[\%esp] \leftarrow valE$	$R[\%esp] \leftarrow 128$
Обновление PC	$PC \leftarrow valM$	$PC \leftarrow 0x028$

Табл. 4.12 показывает, что команда настраивает РС на 0x028 — адрес команды halt. Также %esp устанавливается в 128.

Таким образом, создана универсальная структура, управляющая разными типами команд Y86. Несмотря на то, что все команды значительно отличаются по своему поведению, обработку можно организовать в шесть этапов. Теперь задачей является создание проекта аппаратных средств, в котором эти этапы будут реализованы и объединены.

4.3.2. Структура аппаратных средств SEQ

Вычисления, необходимые для реализации всех команд Y86, можно структурировать в серию из шести основных этапов: выборка, декодирование, выполнение, память, обратная запись и обновление РС. На рис. 4.15 абстрактно показана структура аппаратных средств, с помощью которой эти вычисления можно произвести. Счетчик команд сохраняется в регистре, изображенном в нижнем левом углу (обозначение — РС). Затем информация передается по кабелям (обозначены сгруппированными в виде широкой полосы серого цвета) сначала вверх, а затем по часовой стрелке. Обработка осуществляется *аппаратными устройствами*, относящимися к разным этапам. Пути обратной связи, возвращающиеся вниз справа, содержат обновленные значения для их записи в регистровый файл и обновленный счетчик команд. В данной диаграмме не приводятся некоторые мелкие блоки комбинаторной логики (а также вся управляющая логика), необходимые для функционирования различных аппаратных устройств и передачи соответствующих значений в устройства. Эти подробности будут добавлены позднее. Такой метод изображения работы процессоров с потоком, направленным снизу вверх, не совсем обычен. Причина подобного условия будет объяснена позже, при проектировании конвейерных процессоров.

Выборка

Используя в качестве адреса регистр счетчика команд, ячейка для хранения команды считывает байты команды. Вычисляется valP — приращенный счетчик команд.

Декодирование

Регистровый файл имеет два порта считывания — А и В, через которые одновременно считаются значения регистра valA и valB.

Выполнение

На этапе выполнения для различных целей используется арифметико-логическое устройство (АЛУ), в зависимости от типа команды. Для целочисленных операций выполняется заданная операция. Для других команд АЛУ служит сумматором для вычисления приращенного или уменьшенного указателя стека, для вычисления эффективного адреса либо просто для передачи данных входа на выход путем добавления 0.

Регистр кода условия (CC) содержит три бита кода условия. Новые значения кодов условия рассчитываются АЛУ. При выполнении команды перехода сигнал ветвления *Branch* вычисляется на основе кодов условия и типа перехода.

Память

При выполнении команды памяти ячейка для хранения данных считывает или записывает слово. Команда или данные имеют доступ к одним и тем же ячейкам памяти, однако пользуются ими с разными целями.

Обратная запись

Регистровый файл имеет два порта записи. Порт Е используется для записи значений, вычисляемых АЛУ, тогда какпорт М применяется для записи значений, считываемых из памяти данных.

Информация, обрабатываемая во время выполнения команды, следует потоком по часовой стрелке, начинаясь с команды выборки со счетчиком команд (PC), показанным в нижнем левом углу рис. 4.15.

На рис. 4.16 представлена более подробная схема аппаратных средств, необходимая для реализации SEQ (хотя всех подробностей не увидеть до изучения отдельных этапов). Набор аппаратных устройств — такой же, что и предыдущий, однако теперь соединения (кабели) показаны явно. На данной диаграмме, как и на других, используются следующие правила изображения:

- Аппаратные устройства изображены в светлых рамках. Сюда входят разные типы памяти, АЛУ и т. д. Для реализации процессора будет использован тот же базовый набор устройств. Они фигурируют как "черные ящики" и подробно здесь не рассматриваются.
- Управляющие логические блоки изображены в серых рамках с закругленными углами. Эти блоки служат для выбора из множества источников сигналов либо для вычисления некой булевой функции. Эти блоки в книге рассматриваются очень подробно; сюда же входит разработка описаний HCL.
- Названия кабелей обозначены в белых рамках с закругленными углами. Это всего лишь ярлыки кабелей, не являющиеся никакими элементами аппаратных средств.
- Каналы передачи данных глобальной сети изображены в виде тонких линий. Каждая из этих линий фактически представляет собой пучок из 32 параллельно соединенных проводов для передачи слова из одной части аппаратных устройств в другую.
- Каналы передачи данных на уровне одного байта. В каждой из этих строк на самом деле представлен пучок из 4—8 проводов, в зависимости от того, какой тип значений должен быть передан по этим проводам.
- Каналы передачи однобитовых данных изображены в виде пунктирных линий. Этим представлены контрольные значения, передаваемые между устройствами и блоками микросхемы.

Все вычисления, показанные в табл. 4.4—4.7, обладают тем свойством, что в каждой строке представлено либо вычисление удельной величины, такой как, например, valP, либо активизация некоторого аппаратного устройства, например памяти. Эти вычисления и действия приведены во втором столбце табл. 4.13. Помимо уже описанных

сигналов, в данный список входят четыре сигнала идентификаторов регистров: srcA — источник valA, srcB — источник valB, dstE — регистр, в который записывается valE, и dstM — регистр, в который записывается valM.

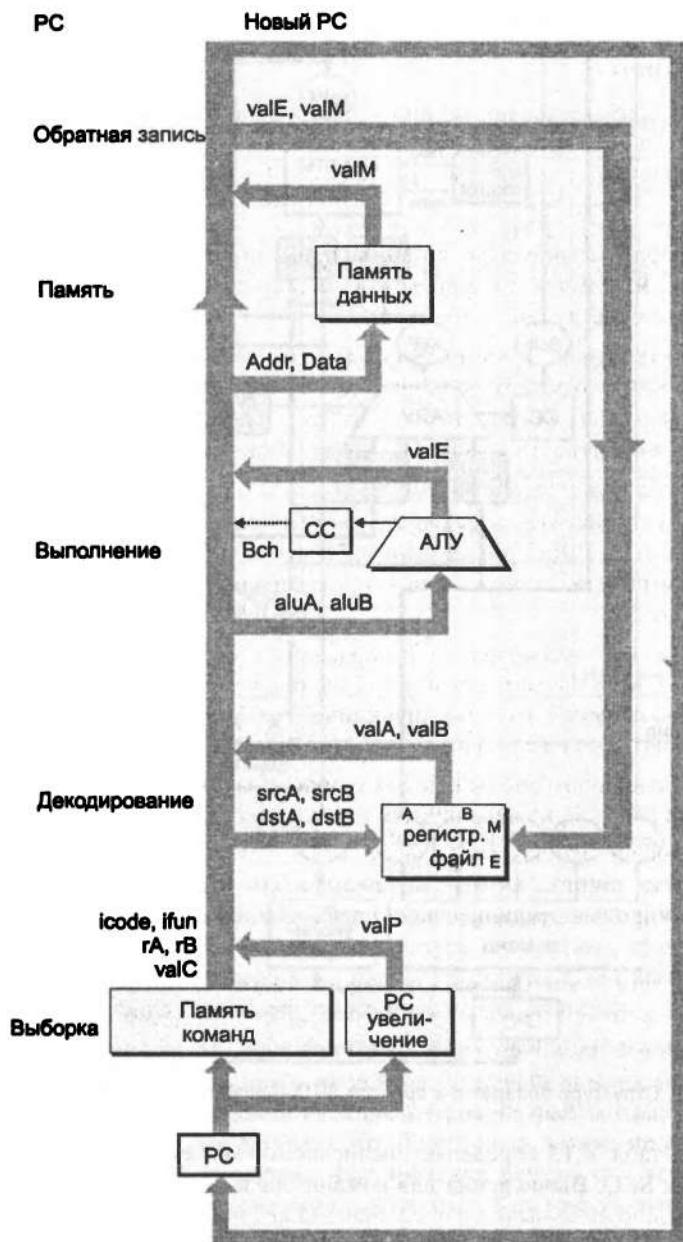


Рис. 4.15. Абстрактное представление SEQ, последовательная реализация

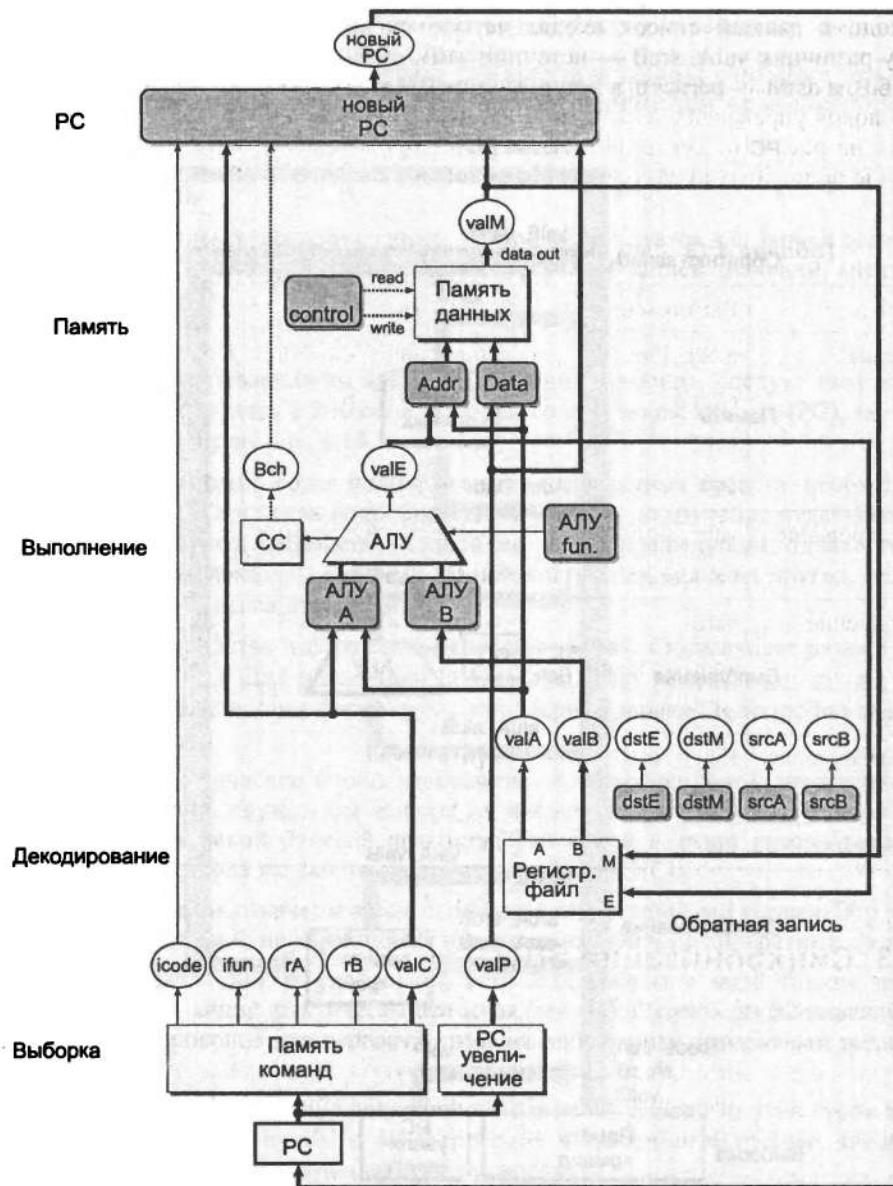


Рис. 4.16. Структура аппаратных средств SEQ — последовательная реализация

Второй столбец табл. 4.13 определяет вычисляемое значение или выполняемую операцию на этапах SEQ. Вычисления для команд **OP1** и **memovl** показаны в виде примеров вычислений.

В двух правых столбцах показаны расчеты для команд **OP1** и **memovl** с целью иллюстрации вычисляемых значений. Для отображения вычислений в аппаратные средства

необходимо реализовать управляющую логику, которая будет переносить данные между различными аппаратными устройствами и производить с ними операции так, что обозначенные операции будут выполняться для каждого типа команд. Такова цель блоков управляющей логики, показанных в виде серых рамок с закругленными углами на рис. 4.16. Теперь в задачу входит прохождение через отдельные этапы и создание подробных проектов для этих блоков.

Таблица 4.13. Идентификация вычислений в последовательной реализации

Этап	Вычисление	OP1 rA, rB	mrnmov1 D (rB), rA
Выборка	icode: ifun rA, rB valC valP	icode : ifun $\leftarrow M_1 [PC]$ rA : rB $\leftarrow M_1 [PC + 1]$ valP $\leftarrow PC + 2$	icode : ifun $\leftarrow M_1 [PC]$ rA : rB $\leftarrow M_1 [PC + 1]$ valC $\leftarrow M_4 [PC + 2]$ valP $\leftarrow PC + 6$
Декодирование	valA, srcA valB, srcB	valA $\leftarrow R[rA]$ valB $\leftarrow R[rB]$	valB $\leftarrow R[rB]$
Выполнение	valE коды условий	valE $\leftarrow valB \text{ OP } valA$ Set CC	valE $\leftarrow valB + valC$
Память	считывание/запись		valM $\leftarrow M_4 [vale]$
Обратная запись	E port, dstE M port, dstM	R[rB] $\leftarrow valE$	R[rA] $\leftarrow valM$
Обновление PC	PC	PC $\leftarrow valP$	PC $\leftarrow valP$

4.3.3. Синхронизация SEQ

При представлении таблиц 4.4—4.7 отмечалось, что их следует читать так, словно они написаны в программном обозначении с присвоениями, выполняемыми последовательно сверху вниз. С другой стороны, структура аппаратных средств, показанная на рис. 4.16, функционирует совершенно иначе. Рассмотрим, как аппаратные средства могут реализовать типы поведения, описанные в этих таблицах.

Представляемая реализация SEQ состоит из комбинаторной логики и двух форм устройств памяти: синхронизированных регистров (счетчика команд и регистра кодов условия) и устройств оперативной памяти (регистрового файла, памяти для хранения команд и памяти для хранения данных). Комбинаторная логика не требует установления последовательности или управления: значения передаются через сеть логических шлюзов всякий раз при смене входных данных. Как уже указывалось, считывание можно просматривать из оперативной памяти в виде комбинаторной логики, когда выходное слово генерируется из адресного входа. Поскольку память для хра-

нения команд используется только для считывания команд, это устройство можно рассматривать как комбинаторную логику.

Осталось всего четыре аппаратных устройства, требующих явного управления установлением их последовательности: счетчик команд, регистр кодов условий, память для хранения данных и регистровый файл. Управление ими осуществляется через один синхронизирующий сигнал, запускающий процесс загрузки новых значений в регистры и запись этих значений в устройства оперативной памяти. Счетчик команд загружается каждый цикл синхронизации с новым адресом команды. Регистр кодов условий загружается только при выполнении команды целочисленной операции. Память для хранения данных записывается только при выполнении команд `jmpv1`, `push1` или `call`. Два порта записи регистра файла обеспечивают обновление двух регистров команд на каждом цикле, однако в качестве адреса порта можно использовать особый идентификатор регистра 8 для обозначения того, что для данного порта никакой записи производиться не должно.

Такая синхронизация регистров и устройств памяти — все, что требуется для управления установлением последовательности операций процессора. Аппаратные средства добиваются такого же эффекта, что и при последовательном выполнении присвоений, показанных в таблицах 4.4—4.7, несмотря на то, что все обновления состояний происходят фактически одновременно и только при повышении синхронизации для начала следующего цикла. Подобная эквивалентность поддерживается благодаря природе системы команд Y86, а также потому, что вычисления организованы так, что проект подчиняется следующим правилам:

- Для процессора нет необходимости выполнять обратное считывание состояния, обновленного командой, для завершения обработки данной команды.
- Соблюдение данного принципа является основополагающим условием для успеха реализации.

В качестве иллюстрации предположим, что реализована команда `push1` первым уменьшением `%esp` на 4 и последующим использованием обновленного значения `%esp` в качестве адреса операции записи. Такой подход нарушит принцип, сформулированный ранее. Для выполнения операции памяти потребуется считывание из регистра файла обновленного значения указателя стека. Вместо этого, при авторской реализации (см. табл. 4.6) уменьшенное значение указателя стека генерируется как сигнал `valE`, после чего этот сигнал используется как данные для записи в регистр, так и как адрес для записи в память. В результате, это значение может осуществлять одновременную запись в регистр и в память с возрастанием синхронизации для начала очередного цикла.

Другая иллюстрация данного принципа: видно, что одни команды (целочисленные операции) задают коды условий, а другие (команды перехода) считывают эти коды условий, однако ни одна команда не должна задавать и считывать коды условий. Даже если коды условий не заданы до возрастания синхронизации для начала очередного цикла, они будут обновляться до того, как какая-либо команда попытается их считать.

На рис. 4.17 показано, как аппаратные средства SEQ обрабатывают команды в строках 3 и 4 в листинге 4.9 с адресами команд, перечисленными в левой части:

Листинг 4.9. Обработка команд

```

1 0x000:irmovl $0x100, %ebx # %ebx <- - 0x100
2 0x006:irmovl $0x200, %edx # %edx <- - 0x200
3 0x00c:addl %edx,      %ebx # %ebx <- - 0x300 CC <- - 000
4 0x00e:je dest          # Не выбирается
5 0x013:rmovl %ebx, 0 (%edx) # M [0x200] <- - 0x300
6 0x019:    dest: halt

```

Каждая из диаграмм рис. 4.17, помеченная от 1 до 4, показывает четыре элемента состояния, а также комбинаторную логику и связи между элементами состояния. Комбинаторная логика показана как заключенная в оболочку вокруг регистра кодов условий, потому что некоторая ее часть (например, АЛУ) генерирует входные данные в регистр кодов условий, тогда как другие компоненты (например, вычисление ветвления и логика выбора РС) имеют регистр кодов условий в качестве входных данных. Регистровый файл и память для хранения данных показаны как имеющие раздельные соединения для считывания и записи, поскольку операции считывания передаются через эти устройства так, как будто они являются комбинаторной логикой, а операции записи управляются генератором синхронизирующих импульсов.

Предполагается, что процесс обработки начинается с кодов условий, перечисленных в порядке ZF, SF и OF, установленных равными 100. В начале цикла синхронизации 3 (точка 1 на рис. 4.17) элементы состояния удерживают его как обновленное второй командой irmovl (строка 2 листинга 4.10). Комбинаторная логика обозначена белым цветом, указывающим на то, что у нее еще не было времени отреагировать на измененное состояние. Цикл синхронизации начинается с адреса 0x00c, загружаемого в счетчик команд. Этим активизируется команда addl (строка 3 листинга 4.10), которая должна быть выполнена и обработана. Значения проходят через комбинаторную логику, включая считывание данных из устройств оперативной памяти. К концу цикла (точка 2 на рис. 4.17) комбинаторная логика создает новые значения (000) для кодов условий и обновляет регистр команд %ebx и новое значение (0x00e) для счетчика команд. В данной точке комбинаторная логика обновляется в соответствии с командой addl, однако состояние по-прежнему сохраняет значения, заданные второй командой irmovl.

Листинг 4.10. Выполнение двух циклов

```

Цикл 1 0x000:irmovl $0x100, %ebx # %ebx <- - 0x100
Цикл 2 0x006:irmovl $0x200, %edx # %edx <- - 0x200
Цикл 3 0x00c:addl %edx,      %ebx # %ebx <- - 0x300 CC <- - 000
Цикл 4 0x00e:je dest          # Не выбирается
Цикл 5 0x013:rmovl %ebx, 0 (%edx) # M [0x200] <- - 0x300

```

Каждый цикл начинается с элементов состояния (счетчик команд, регистр кодов условий, регистровый файл и память для хранения данных), настроенных в соответст-

вии с предыдущей командой. Сигналы проходят через комбинаторную логику, создавая новые значения для элементов состояния. Эти значения загружаются в элементы состояния для начала нового цикла.

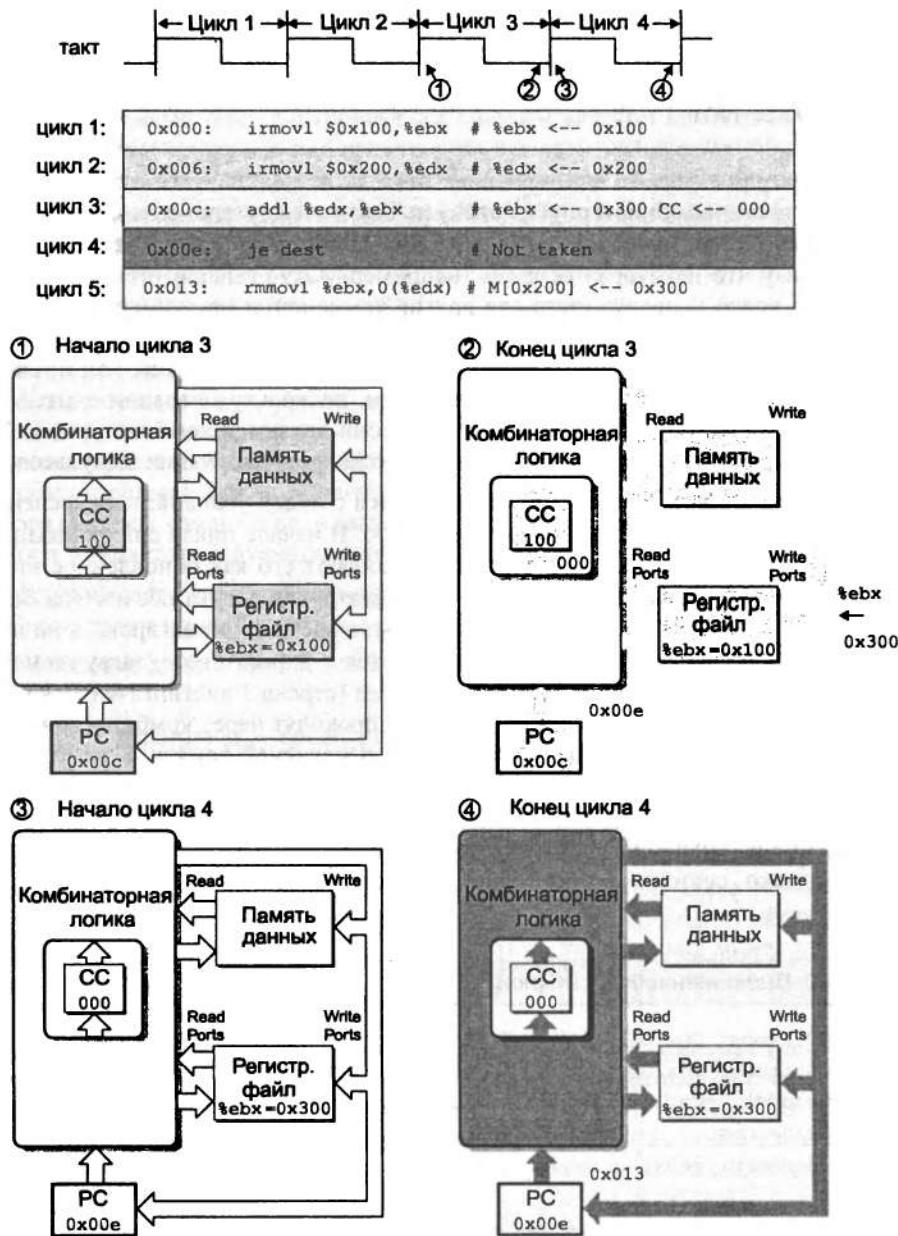


Рис. 4.17. Прослеживание двух циклов выполнения SEQ

По мере возрастания генератора синхронизации для начала цикла 4 (точка 3 на рис. 4.17) происходит обновление счетчика команд, регистрового файла и регистра кодов условий, однако комбинаторная логика еще не отреагировала на эти изменения, поэтому она обозначена белым цветом. В этом цикле выбирается и выполняется команда `je` (строка 4 в листинге 4.10). Поскольку код условий `ZF` равен нулю, ветвь не выбирается. К концу цикла (точка 4 на рис. 4.17) для счетчика команд генерируется новое значение `0x00e`. Комбинаторная логика обновляется в соответствии с командой `je` (обозначена темно-серым цветом на рис. 4.17), однако состояние по-прежнему поддерживает значения, заданные командой `addl` до начала следующего цикла.

Как иллюстрирует данный пример, использования тактового генератора для управления обновлением элементов состояния в комбинации с передачей значений через комбинаторную логику достаточно для управления вычислениями, выполняемыми для каждой команды в данной реализации SEQ. Всякий раз при передаче синхронизации с нижней позиции в верхнюю процессор начинает выполнение новой команды.

4.3.4. Реализация этапов SEQ

В данном разделе создаются описания HCL для блоков управляющей логики, необходимых для реализации SEQ. Полное описание HCL для SEQ представлено в *приложении I*. Здесь показываются примеры некоторых блоков; другие же представлены в виде упражнений. Их рекомендуется тщательно прорабатывать с целью проверки понимания отношения между этими блоками и вычислительными требованиями различных команд.

Не рассматриваемая здесь часть HCL-описания SEQ является определением различных целочисленных и булевых сигналов, которые можно использовать в качестве аргументов для операций HCL. В нее входят имена различных сигналов аппаратного обеспечения, а также постоянные величины для кодов разных команд, имена регистров и операций АЛУ. Используемые константы представлены в табл. 4.14. Условно говоря, для постоянных величин используются имена, написанные прописными литерами.

Таблица 4.14. Постоянные значения, используемые в описании HCL

Имя	Значение (шестн.)	Значение
INOP	0	Код команды <code>nop</code>
IHALT	1	Код команды <code>halt</code>
IRRMOVL	2	Код команды <code>rrmovl</code>
IIRMOVL	3	Код команды <code>irmovl</code>
IRMMOVL	4	Код команды <code>irmovl</code>
IMRMOVL	5	Код команды <code>imovl</code>

Таблица 4.14 (окончание)

Имя	Значение (шестн.)	Значение
IOPL	6	Код команд целочисленных операций
IJXX	7	Код команд перехода
ICALL	8	Код команды call
IRET	9	Код команды ret
IPUSHL	a	Код команды pushl
IPOPL	b	Код команды popl
RESP	6	Идентификатор регистра для %esp
RNONE	8	Указывается на отсутствие доступа к регистровому файлу
ALUADD	0	Функция для операции сложения

В добавление к командам, показанным в табл. 4.4—4.7, сюда включена обработка команд `nop` и `halt`. Обе эти команды попросту проходят через все этапы без особой обработки, за исключением приращения РС на 1. Авторы не вдаются в подробности того, как команда `halt` фактически останавливает работу процессора; просто предполагается, что процессор останавливается при появлении значения 1 для `icode`.

Этап выборки

Как показано на рис. 4.18, в этап выборки входит аппаратное устройство памяти для хранения команд. За один раз это устройство считывает шесть байтов из памяти, используя РС в качестве адреса первого байта (байт 0). Этот байт рассматривается как байт команды и делится (устройством под названием *Split* означает деление) на два четырехбитовых множества `icode` и `ifun`. Исходя из значения `icode`, можно вычислить три однобитовых сигнала (обозначены пунктирными линиями на рис. 4.18):

- `instr_valid` — соответствует ли этот байт действующей команде Y86. Этот сигнал используется для выявления недействующей команды;
- `need_regs` — включает ли эта команда в себя байт спецификатора регистра;
- `need_valC` — включает ли это команда в себя константное слово.

Для примера, HCL-описание `need_REGS` просто определяет, является ли значение `icode` одной из команд, имеющих байт спецификатора регистра:

```
bool need_REGS =
icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,
IIRMOVL, IRMMOVL, IMRMOVL };
```

УПРАЖНЕНИЕ 4.14

Напишите код HCL для сигнала `need_valC` в реализации SEQ.

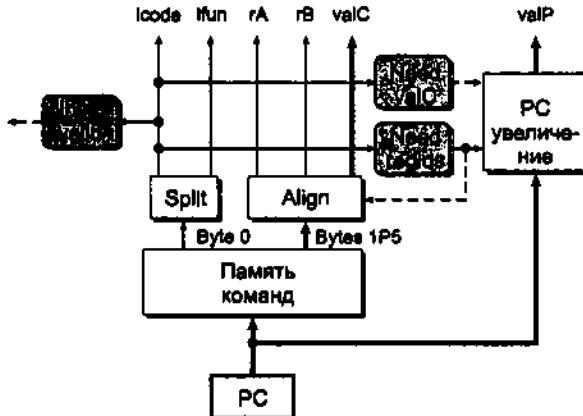


Рис. 4.18. Этап выборки SEQ

Как показано на рис. 4.18, оставшиеся пять байтов, считанные из памяти для хранения команд, кодируют некоторую комбинацию байта спецификатора регистра и константного слова. Эти байты обрабатываются аппаратным устройством с ярлыком Align (выравнивание) в поля регистров и константного слова. Когда вычисленный сигнал *need_regs* равен единице, тогда байт 1 делится на спецификаторы регистра *rA* и *rB*. В противном случае, эти два поля настраиваются на 8 (RNONE), указывая на то, что для данной команды регистры не заданы.

Вспомните также (см. рис. 4.2), что для любой команды, имеющей только один регистровый операнд, другое поле байта спецификатора регистра составляет 8 (RNONE). Следовательно, можно предположить, что сигналы *rA* и *rB* либо кодируют регистры, к которым необходимо осуществить доступ, либо указывают на то, что доступ к регистру не требуется. Устройство, обозначенное Align, также генерирует константное слово *valC*. Это будут либо байты 1—4, либо байты 2—5, в зависимости от значения сигнала *need_regs*.

Аппаратное устройство приращения PC (счетчик команд) генерирует сигнал *valP*, исходя из текущего значения PC, и два сигнала *need_regs* и *need_valC*. Для значения PC *p*, значения *need_regs* *r* и значения *need_valC* *i* инкремент генерирует значение *p + r + 4i*.

Этапы декодирования и обратной записи

На рис. 4.19 представлена подробная схема логики, реализующей этапы декодирования и обратной записи в SEQ. Эти два этапа объединены, потому что оба имеют доступ к регистровому файлу.

Поля команд декодированы для генерирования идентификаторов регистров для четырех адресов (двух для считывания и двух — для записи), используемых регистровым файлом. Считываемые из регистрового файла значения становятся сигналами *valA* и *valB*. Два значения обратной записи *valE* и *valM* выполняют роль данных для записи.

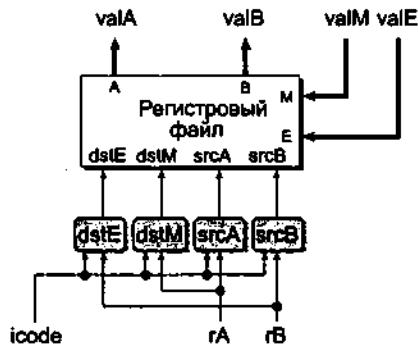


Рис. 4.19. Этапы декодирования и обратной записи SEQ

Регистровый файл имеет четыре порта. Он поддерживает до двух одновременных операций считывания (на портах А и В) и двух одновременных операций записи (на портах Е и М). Каждый порт имеет адресное соединение и соединение данных, где первое — идентификатор регистра, а второе — набор из 32 кабелей, служащих либо словом выхода (для порта считывания), либо словом входа (для порта записи) регистра файла. Два порта считывания имеют адресные входы srcA и srcB, а два порта записи — адресные входы dstA и dstB. Специальный идентификатор 8 (RNONE) адресного порта указывает на то, что ни к одному регистру не должен осуществляться доступ.

Четыре блока в нижней части рис. 4.19 генерируют четыре идентификатора разных регистров для регистрационного файла, исходя из кода команды iCode и регистрационных спецификаторов rA и rB. Регистровый идентификатор scrA указывает на регистр, который должен быть прочитан для генерирования valA. Нужное значение зависит от типа команды, как показано в первой строке этапа декодирования табл. 4.4—4.7. Объединение всех этих записей в единое вычисление дает следующее HCL-описание scrA (следует помнить, что RESP является идентификатором регистра %esp):

```
int scrA = [
  icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL } : rA;
  icode in { IPOPL, IRET } : RESP;
  1 : RNONE; # Регистра не требует
];
```

УПРАЖНЕНИЕ 4.15

Регистровый сигнал srcB указывает на регистр, который должен считываться, для генерирования сигнала valB. Нужное значение показано в виде второго шага этапа декодирования табл. 4.4—4.7. Напишите HCL-код для srcB.

Регистровый идентификатор dstE указывает выходной регистр для порта записи Е, где хранится вычисленное значение valE. Это показано в табл. 4.4—4.7 в виде первого шага этапа обратной записи. Объединение выходных регистров для всех разных команд дает следующее HCL-описание dstE:

```

int dstE = [
icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL } : rB;
icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
1 : RNONE; # Регистра не требует
];

```

УПРАЖНЕНИЕ 4.16

Регистровый идентификатор dstM указывает выходной регистр для порта записи M, где хранится считанное из памяти значение valM. Это показано в табл. 4.4—4.7 в виде второго шага этапа обратной записи. Напишите HCL-код для dstM.

УПРАЖНЕНИЕ 4.17

Одновременно оба порта записи регистра файла используются только командой popl. Для команды popl %esp тот же адрес будет использоваться для портов записи E и M, но с другими данными. Для устранения этого конфликта между двумя портами записи необходимо задать приоритет для того, чтобы при их попытке записи одного и того же регистра в один цикл запись осуществлялась бы только из порта высшего приоритета. Какому из двух портов должен быть отдан приоритет для реализации нужного поведения, как определено в упр. 4.5?

Этап выполнения

Этап выполнения включает в себя арифметико-логическое устройство (АЛУ). Последнее выполняет операцию ADD, SUBTRACT, AND или EXCLUSIVE-OR на входах aluA и aluB, исходя из настройки сигнала alufun. Эти данные и управляемые сигналы генерируются тремя управляющими блоками, как показано на рис. 4.20. Выход АЛУ становится сигналом valE.

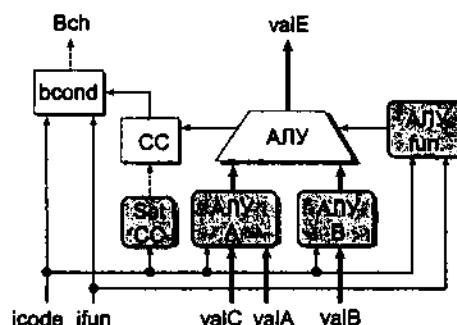


Рис. 4.20. Этап выполнения SEQ

В табл. 4.4—4.7 вычисление АЛУ для каждой команды показано в виде первого шага этапа выполнения. Операнды перечислены с aluB, за которым следует aluA для того, чтобы команда sub1 вычитала valA из valB. Можно видеть, что значение aluA может быть valA, valC, -4 или +4, в зависимости от типа команды. Таким образом, поведе-

ние управляющего блока, генерирующего aluA, можно выразить следующим образом:

```
int aluA = [
icode in { IRRMOVL, IOPL } : valA;
icode in { IRRMOVL, IRMMOVL, IMRMOVL } : valC;
icode in { ICALL, IPUSHL } : ~4;
icode in { IRET, IPOPL } : 4;
# Другие команды не требуют АЛУ
];
```

УПРАЖНЕНИЕ 4.18

Исходя из первого операнда первого шага этапа выполнения, показанного в табл. 4.4—4.7, напишите HCL-описание для сигнала aluB в SEQ.

АЛУ либо выполняет операцию для команды целочисленной операции, либо выступает в роли сумматора. Регистры кодов условий настраиваются в соответствии со значением АЛУ. Значения кодов условий тестируются для определения необходимости ветвления.

При взгляде на операции АЛУ на этапе выполнения видно, что оно используется в основном в качестве сумматора. Однако для команд ОР1 необходимо, чтобы оно использовало операцию, закодированную в поле ifun команды. Следовательно, HCL-описание для управления АЛУ можно записать так:

```
int alufun = [
icode == IOPL : ifun;
1 : ALUADD
];
```

Этап выполнения также включает в себя регистр кодов условий. АЛУ генерирует три сигнала, на которых основаны коды условий — ноль, знак и переполнение — всякий раз при срабатывании АЛУ. Однако коды условий необходимо задавать только при выполнении команды ОР1. Следовательно, генерируется сигнал set_cc, управляющий необходимостью обновления (или не обновления) регистра кодов условий:

```
bool set_cc = icode in { IOPL};
```

Аппаратное устройство, обозначенное как bcond, определяет, должна или не должна команда вызывать переход (выбранная ветвь) или продолжать выполнение очередной команды (не выбранной), создавая управляющий сигнал Bch. Данная команда должна вызывать переход, только если это команда перехода (icode равен 1xxx), а комбинация значений кодов условий и типа перехода (закодированного в ifun) указывает на выбранную ветвь (см. рис. 3.11). Подробное проектирование данного устройства не рассматривается.

Этап памяти

Задачей этапа памяти является считывание или запись данных программы. Как показано на рис. 4.21, два управляющих блока генерируют значения для адреса памяти и

данных ввода в память (для операций записи). Два других блока генерируют управляющие сигналы, указывающие на выполнение операции считывания или операции записи. При выполнении операции считывания память для хранения данных генерирует значение valM.

Операция памяти, необходимая для каждого типа команды, показана на этапе памяти (см. табл. 4.4—4.7). Обратите внимание, что адрес для операций записи и считывания памяти — всегда valE или valA. Этот блок можно описать на HCL следующим образом:

```
int mem_addr = [
  icode in { IRRMOVL, IPUSHL, ICALL, IMRMOVL } : valE;
  icode in { IPOPL, IRET } : valA;
  icode in { ICALL, IPUSHL } : -4;
  icode in { IRET, IPOPL } : 4;
  # Другие команды не требуют адреса
];
```

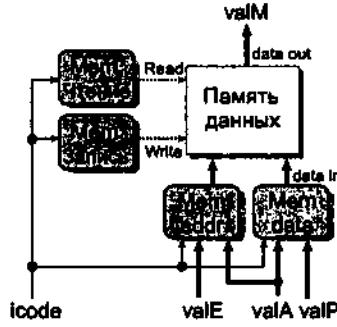


Рис. 4.21. Этап памяти SEQ

Память для хранения данных может либо записывать, либо считывать значения памяти. Считываемое из памяти значение формирует сигнал valM.

УПРАЖНЕНИЕ 4.19

При рассмотрении операций памяти для разных команд, показанных в табл. 4.4—4.7, можно видеть, что данные для записи в память всегда valA или valP. Напишите HCL-код для сигнала mem_data в SEQ.

Задавать управляющий сигнал mem_read только для команд, которыечитывают данные из памяти, необходимо так, как выражено следующим HCL-кодом:

```
bool mem_read = icode in { IMRMOVL, IPOPL, IRET };
```

УПРАЖНЕНИЕ 4.20

Необходимо задать управляющий сигнал mem_write только для команд, которые записывают данные в память. Напишите HCL-код для сигнала mem_write в SEQ.

Этап обновления PC

Последний этап в SEQ генерирует новое значение счетчика команд (рис. 4.22). Как показывают последние шаги табл. 4.4—4.7, новый счетчик команд будет valC, valM или valP, в зависимости от типа команды и от того, будет ли выбрана ветвь или нет. Этую выборку на HCL можно выразить следующим образом:

```
int new_pc = [
# Вызов. Использовать константу команды
icode == ICALL : valC;
# Выбранная ветвь. Использовать константу инструкции
icode == IJXX && Bch : valC;
# Завершение команды RET. Использовать значение из стека
icode == IRET : valM;
# По умолчанию: Использование приращенного PC
1 : valP;
];
```

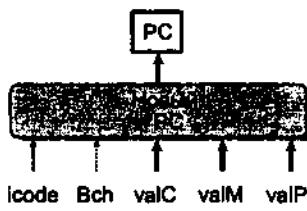


Рис. 4.22. Этап обновления PC SEQ

Исследование SEQ

Таким образом, мы полностью рассмотрели проектирование процессора Y86. Отмечалось, что путем организации шагов, необходимых для выполнения каждой команды, можно реализовать процессор целиком с помощью небольшого числа разных аппаратных устройств с одним генератором синхронизирующих импульсов для управления последовательностью вычислений. Затем управляющая логика должна передавать сигналы между этими устройствами и генерировать необходимые управляющие сигналы на основании типов команд и условий ветвления.

Единственной проблемой с SEQ является то, что он работает слишком медленно. Генератор должен работать достаточно медленно, чтобы сигналы могли передаваться через все этапы в рамках одного цикла. В качестве примера рассмотрим обработку команды ret. Начиная с обновленного счетчика команд в начале цикла синхронизации, команда должна считываться из памяти команд, указатель стека — из регистрационного файла, АЛУ должен уменьшать значение указателя стека, а возвратный адрес должен считываться из памяти для определения следующего значения счетчика команд. Все эти операции должны завершаться к концу цикла синхронизации.

При таком стиле реализации аппаратные устройства используются не в достаточно полной мере, поскольку каждое из них активно только для определенной доли всего

цикла синхронизации. Далее в книге представлена конвейерная обработка, с помощью которой достигается большая производительность.

4.3.5. Реконфигурация этапов вычисления

В качестве переходного шага к конвейерному проектированию осуществляется реконфигурация порядка шести этапов так, что этап счетчика команд оказывается в начале цикла синхронизации, а не в конце, что дает проект процессора под названием SEQ+, потому что базовый процессор расширяется. Все эти операции могут показаться довольно странными, потому что определение нового значения PC может потребовать тестирования условия ветвления на этапе выполнения (для команды условного перехода) или считывания возвращаемого значения на этапе памяти (для команды `ret`).

На рис. 4.23 показано, что этап PC можно переместить так, что его логика будет активной в начале цикла синхронизации, путем вычисления значения PC для текущей команды. Затем значение PC поступает на этап выборки, и оставшаяся часть обработки продолжается в прежнем режиме. Комбинаторная логика создает все сигналы, необходимые для вычисления нового значения PC к концу цикла синхронизации. Эти значения сохраняются в наборе регистров, показанных на схеме в рамке pState ("предыдущее состояние"). Теперь задачей этапа PC является выбор значения PC для текущей команды, а не вычисление обновленного PC для следующей команды.

На рис. 4.24 показана более подробная схема аппаратных средств SEQ+. Видно, что здесь представлены абсолютно те же аппаратные устройства и блоки управления, что и в SEQ (см. рис. 4.16), однако здесь логика PC сдвинута вниз. Результаты из предыдущей команды хранятся в регистрах, изображенных в самом низу схемы, обозначенных своими значениями с литерой "p" (previous, предыдущий).

Данная структура помогает перейти к конвейерной реализации.

Единственным изменением в управляющей логике является реконфигурация вычисления PC с использованием значений предыдущего состояния. На следующих диаграммах (рис. 4.25) показаны блоки вычисления PC для SEQ и SEQ+.

Видно, что единственным различием между двумя блоками является сдвиг регистров, в которых сохранено состояние процессора со времени после вычисления PC до времени до его вычисления. Это — пример общего преобразования, называемого *восстановлением синхронизации цепи*. Восстановление синхронизации изменяет представление состояния системы без изменения ее логического поведения. Оно часто используется для поддержания равновесия задержек между разными компонентами системы. HCL-описание вычислений PC принимает следующую форму:

```
int pc = [
# Вызов. Использовать константу команды
pIcode == ICALL : pValC;
# Выбранная ветвь. Использовать константу инструкции
pIcode == IJXX && pBch : pValC;
# Завершение команды RET. Использовать значение из стека
pIcode == IRET : pValM;
```

По умолчанию: Использование приращенного РС

1 : pValP;

};

Полное HCL-описание SEQ+ представлено в *приложении 1*.

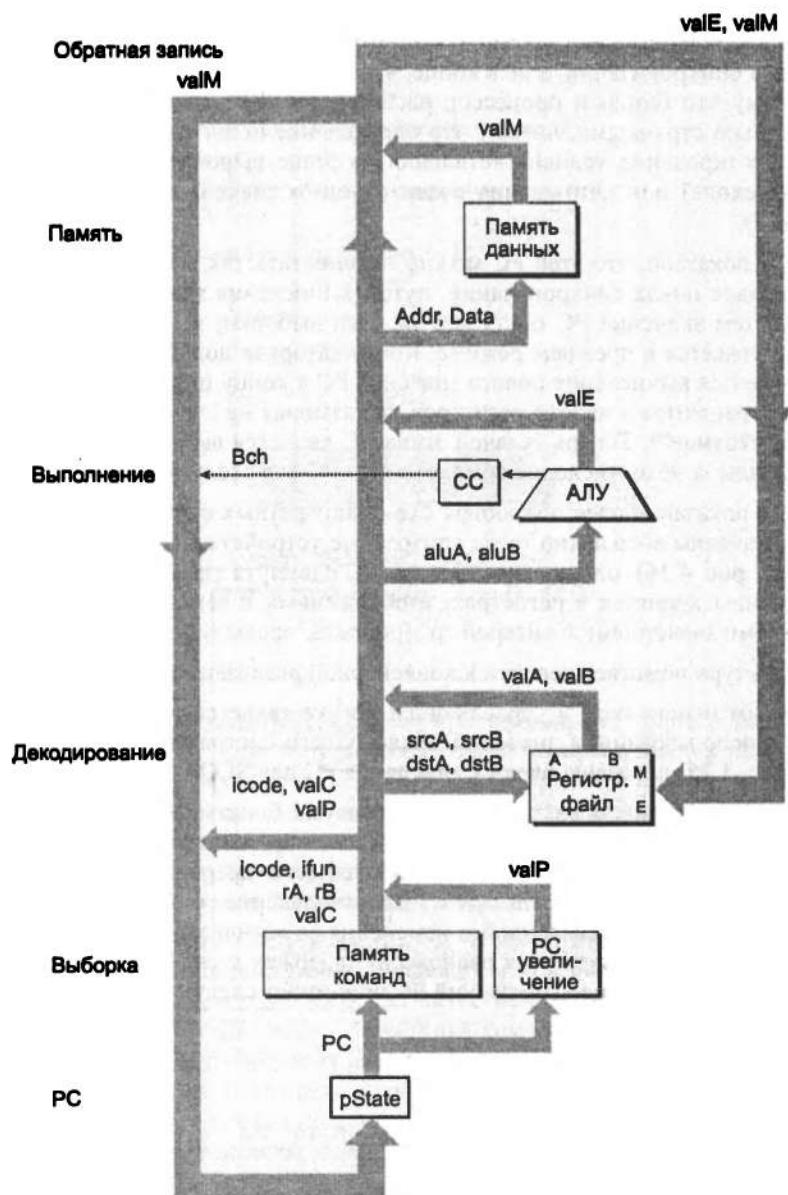


Рис. 4.23. Абстрактное представление SEQ+

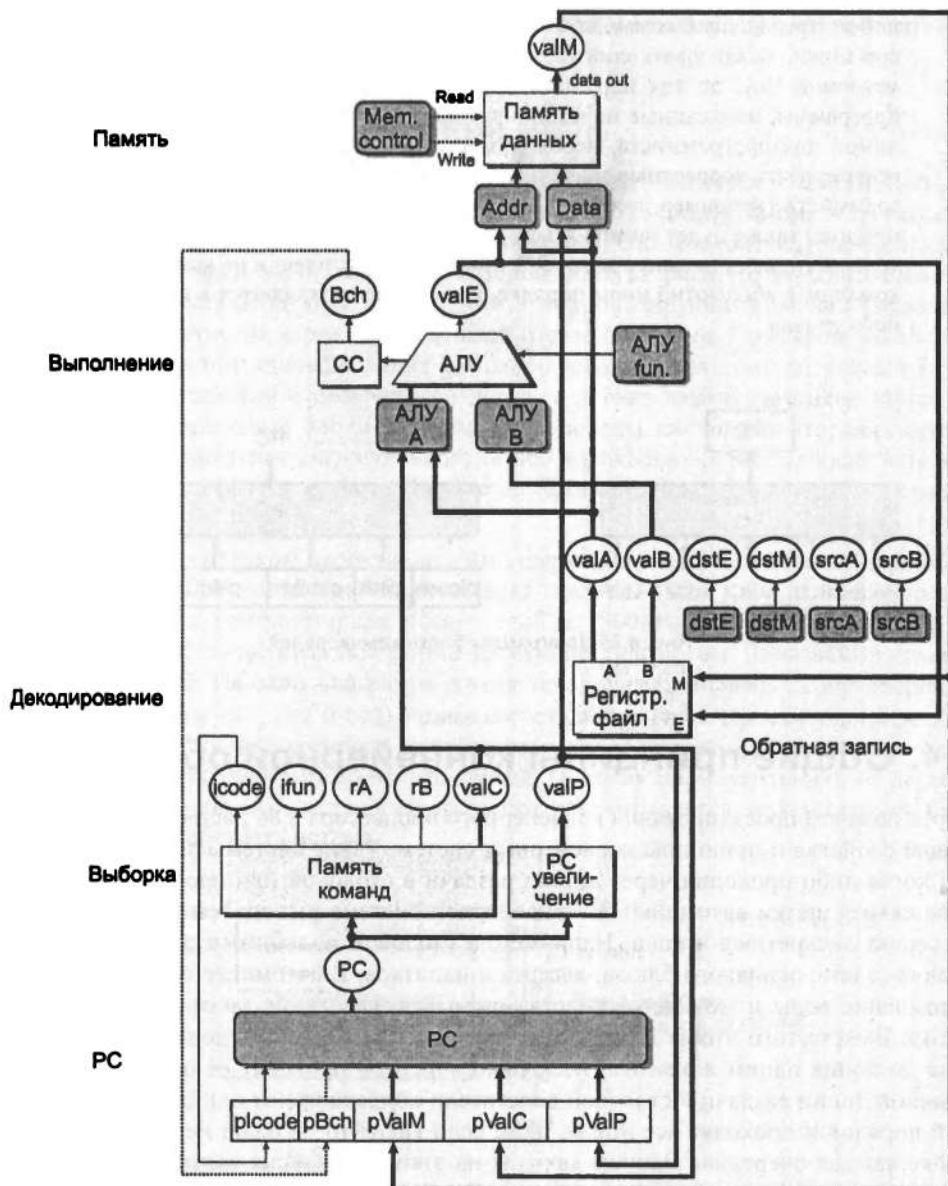


Рис. 4.24. Структура аппаратных средств SEQ+

Где находится PC в SEQ+?

Одной из любопытных особенностей SEQ+ является то, что в нем не существует аппаратного регистра, сохраняющего счетчик программ. Вместо этого PC вычисляется динамически, на основании информации о некотором состоянии, сохраненном

ной из предыдущей команды. Это небольшая иллюстрация того факта, что процессор можно реализовать способом, отличным от концептуальной модели, подразумеваемой ISA, до тех пор, пока процессор корректно выполняет произвольные программы, написанные на машинном языке. Кодировать состояние в форме, видимой для программиста, необходимости нет до тех пор, пока процессор может генерировать корректные значения для любой части состояния, видимого для программиста (например, счетчика команд). При конвейерном проектировании этот принцип также будет применяться. Методики разработки с изменением последовательности, как описано в разд. 5.7, применяют эту идею в ее крайности, выполняя команды в абсолютно ином порядке, нежели они появляются в программе машинного уровня.

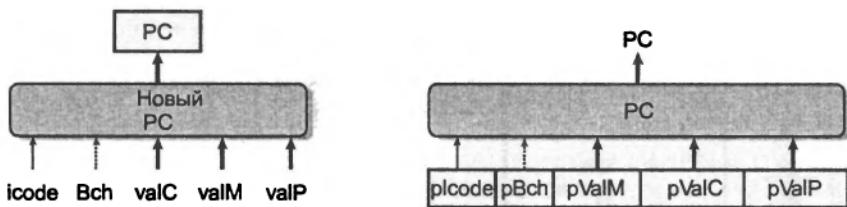


Рис. 4.25. Диаграммы блоков вычислений

4.4. Общие принципы конвейерной обработки

Перед началом проектирования конвейерного процессора Y86 рассмотрим некоторые общие свойства и принципы конвейерных систем. Такие системы знакомы каждому, кто когда-либо проходил через линию раздачи в столовой или проезжал на автомобиле сквозь щетки автомойки. В конвейерной системе выполняемая задача делится на серию дискретных этапов. Например, в столовой подобными этапами являются подача салата, основного блюда, десерта и напитков. В автомойке в их число входит распыление воды и моющего средства, помывка, нанесение защитного покрытия и сушка. Вместо того чтобы дожидаться прохождения всей последовательности от начала до конца одним клиентом, несколько клиентов проходят ее одновременно. На обычной линии раздачи в столовой посетители поддерживают одинаковый конвейерный порядок и проходят все этапы, даже если какие-то из блюд не нужны. На автомойке каждая очередная машина заходит на этап распыления воды, как только предыдущая перемещается на этап помывки. Вообще говоря, автомобили должны перемещаться в системе с одинаковой скоростью, во избежание столкновения.

Основной особенностью конвейерной обработки является повышение с ее помощью пропускной способности системы, т. е. числа клиентов, обслуживаемых за единицу времени, но она же может немного увеличить время ожидания, необходимое на обслуживание одного клиента. Например, посетитель столовой, которому нужен только салат, мог бы пройти очень быстро, задержавшись только у стойки с салатами. Если

же он попытается сделать то же самое в конвейерной системе, то другим посетителям это может не понравиться.

4.4.1. Конвейеры вычислений

Если говорить о конвейерах вычислений, то "клиентами" являются команды, и этапы выполняют некоторую часть собственно выполнения команды. На рис. 4.26 показан пример простой системы аппаратного обеспечения. Она состоит из определенной логики, выполняющей вычисления, за которой следует регистр, предназначенный для хранения результатов этих вычислений. Синхронизирующий сигнал управляет загрузкой этого регистра через определенный отрезок времени. Примером такой системы является декодер проигрывателя компакт-дисков. Входящими сигналами являются биты, считываемые с поверхности диска, а логика декодирует их в звуковые сигналы. Вычислительные блоки на схеме реализованы как комбинаторная логика, что означает прохождение сигналов через серию логических шлюзов, после чего через некоторый промежуток времени на выходе оказывается некоторая функция входного сигнала.

В современном логическом проектировании задержки цепи измеряются в пикосекундах. Пикосекунда это одна триллионная доля секунды (10^{-12}). В данном примере предположим, что комбинаторная логика требует 300 пс, а загрузка регистра — 20 пс. На рис. 4.26 также показана форма временной диаграммы, называемой *конвейерной диаграммой*. На этой диаграмме время течет слева направо. Серия операций (здесь это операции OP1, OP2 и OP3) записывается сверху вниз. Прямоугольники обозначают время, в течение которого выполняются эти операции. В этой системе одна операция должна закончиться до начала другой. Поэтому прямоугольники не пересекаются вертикально. По следующей формуле рассчитывается максимальная скорость, с которой работает система.

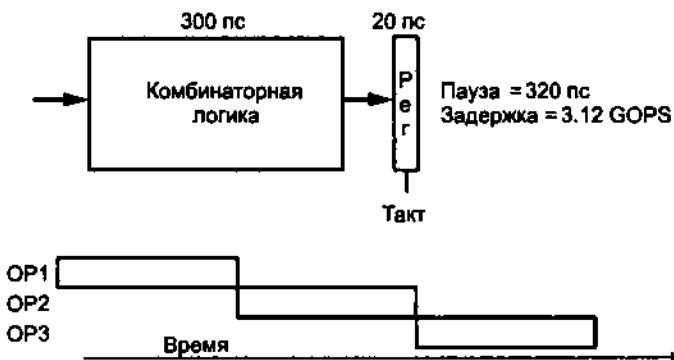


Рис. 4.26. Система аппаратных средств не конвейерного типа

На каждый цикл из 320 пс (рис. 4.26) система тратит 300 пс для расчета функции комбинаторной логики и 20 пс на сохранение результатов в выходном регистре.

Пропускная способность выражена в гигаоперациях в секунду (GOPS) или млрд операций в секунду. Общее время, необходимое на выполнение одной операции от начала и до конца, называется латентностью (или временем задержки). В данной системе время задержки составляет 320 пс, что эквивалентно пропускной способности.

Предположим, что вычисления, выполняемые системой, можно разделить на три этапа — А, В и С, каждый из которых требует 100 пс (рис. 4.27). Затем между этими этапами можно разместить конвейерные регистры так, что каждая операция будет проходить через систему за три шага, требующих три полных цикла синхронизации от начала и до конца.

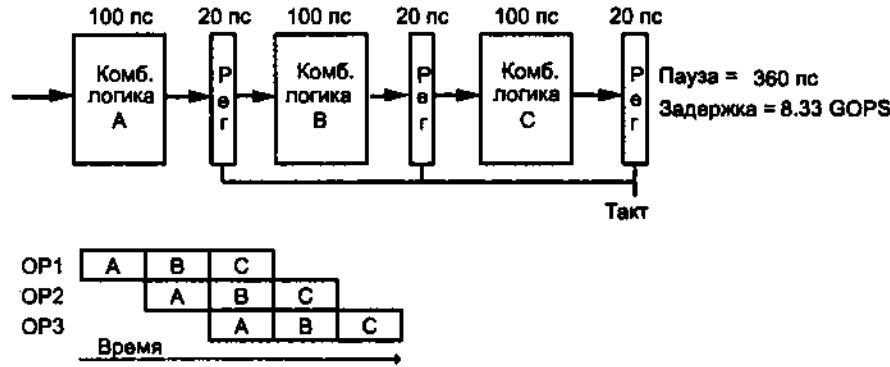


Рис. 4.27. Трехступенчатая конвейерная система аппаратных средств

На рис. 4.27 вычисление делится на три этапа: А, В и С. На каждый цикл из 120 пс каждая операция проходит один этап.

Как показано на конвейерной диаграмме 4.27, OP2 может войти в этап А только при перемещении OP1 из этапа А в В. В устойчивом состоянии все три этапа будут активными, когда одна операция выходит из системы, а другая входит в нее на каждом цикле синхронизации. Это видно на третьем цикле конвейерной диаграммы, где OP1 находится на этапе С, OP2 — на этапе В, а OP3 — на этапе А. В этой системе циклы оборачиваются каждые $100 + 20 = 120$ пс, обеспечивая пропускную способность порядка 8.33 GOPS. Поскольку обработка одной операции требует 3 цикла синхронизации, время задержки данного конвейера составляет $3 \times 120 = 360$ пс. Пропускная способность системы повышается на коэффициент $8.33/3.12 = 2.67$ за счет добавления некоторых аппаратных средств и небольшого увеличения времени задержки ($360/320 = 1.12$). Увеличение времени задержки происходит из-за дополнительного времени на добавленные конвейерные регистры.

4.4.2. Подробное описание конвейерной операции

Чтобы лучше разобраться в принципе конвейерной обработки, рассмотрим немного подробнее синхронизацию и функционирование конвейерных вычислений. На

рис. 4.28 показана конвейерная диаграмма для уже рассмотренного трехступенчатого конвейера (см. рис. 4.27). Переход операций с одного конвейерного этапа на другой управляется синхронизирующим сигналом, как показано над конвейерной диаграммой. Каждые 120 пс этот сигнал возрастает с 0 до 1, инициируя очередное множество расчетов конвейерных этапов.

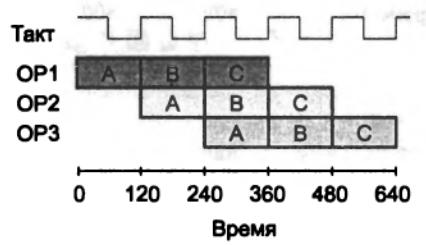


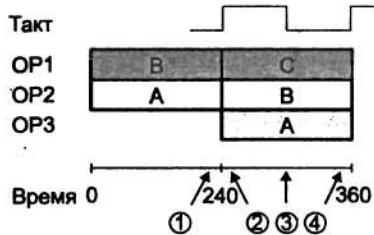
Рис. 4.28. Синхронизация трехступенчатого конвейера

Возрастающий край синхронизирующего сигнала управляет переходом операций с одного конвейерного этапа на другой.

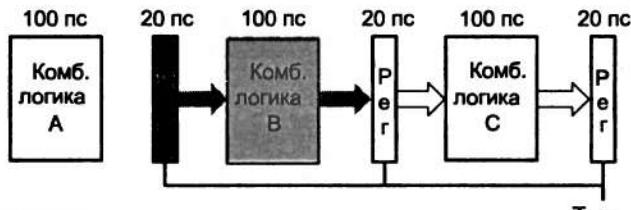
Рис. 4.29 прослеживает деятельность цепи между показателями времени с 240 до 360, когда операция OP1 проходит через этап C, операция OP2 — через этап B, а операция OP3 — через этап A. Непосредственно перед возрастанием синхронизации в значении времени 240 (точка 1) значения, вычисленные на этапе A для операции OP2, достигли входа первого конвейерного регистра, однако его состояние и выход остаются вычисляемыми на этапе A для операции OP1. Значения, вычисленные на этапе B для операции OP1, достигли входа второго конвейерного регистра. С возрастанием синхронизации эти входы загружаются в конвейерные регистры, превращаясь в выходные данные регистров (точка 2). Помимо этого, вход в этап A настраивается на вычисление операции OP3. Затем сигналы передаются через комбинаторную логику разных этапов (точка 3). Как предполагают кривые границы на диаграмме в точке 3, сигналы могут передаваться через разные секции с разными скоростями. Перед значением времени 360 результирующие значения достигают входа в конвейерные регистры (точка 4). При возрастании синхронизации в значении времени 360 каждая из операций пройдет через один конвейерный этап.

Из такого подробного представления конвейерной операции видно, что замедление операции синхронизации не изменит поведения конвейера. Сигналы передаются на входы конвейерных регистров, однако состояние регистров остается неизменным до возрастания синхронизации. С другой стороны, слишком быстрая синхронизация может оказать разрушительный эффект: у значений не будет достаточно времени на прохождение через комбинаторную логику, и поэтому входы регистров при возрастании синхронизации еще не будут действительными.

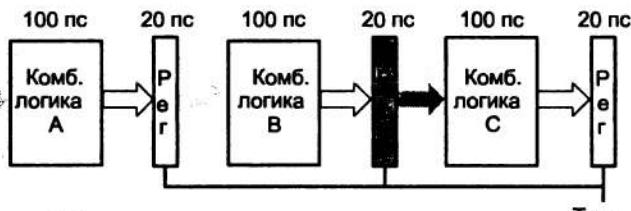
Как и при обсуждении синхронизации для процессора SEQ (см. разд. 4.3.3), видно, что простого механизма размещения синхронизированных регистров между блоками комбинаторной логики достаточно для управления потоком операций в конвейере. По мере того, как синхронизация попеременно возрастает и убывает, различные операции проходят этапы конвейера, не накладываясь одна на другую.



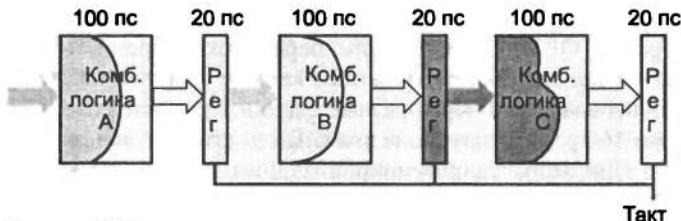
① Время = 239



② Время = 241



③ Время = 300



④ Время = 359

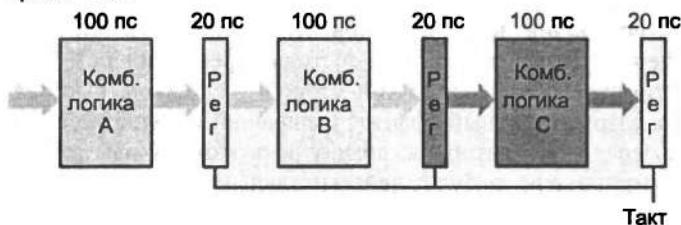


Рис. 4.29. Один цикл синхронизации конвейерной операции

4.4.3. Ограничения конвейерной обработки

Пример на рис. 4.27 показывает идеальную конвейерную систему, в которой вычисления можно разделить на три независимых этапа, каждый из которых требует одну треть времени, необходимого исходной логике. Но, к сожалению, здесь имеют место и другие факторы, часто снижающие эффективность конвейерной обработки.

Неравномерное разбиение

На рис. 4.30 показана система, в которой все вычисления делятся на три этапа, как ранее, однако задержки перехода от одного этапа к другому составляют диапазон от 50 до 150 пс. Сумма всех задержек на всех этапах по-прежнему составляет 300 пс. Впрочем, скорость, с которой можно работать с генератором синхронизирующих импульсов, ограничена задержкой самого медленного этапа. Как показывает конвейерная диаграмма на этой схеме, этап А остается в режиме ожидания (изображен в белой рамке) в течение 100 пс каждого цикла синхронизации, а этап С — в течение 50 пс. Постоянно активным будет только этап В. Цикл синхронизации необходимо строить на $150 + 20 = 170$ пс, что дает пропускную способность в 5.88 GOPS. Кроме этого, время ожидания возрастает до 510 пс из-за пониженной тактовой частоты.

Создание разбиения вычислительных операций системы на серию этапов, имеющих однородные задержки, может стать основной трудностью разработчиков аппаратных средств. Часто некоторые аппаратные устройства в процессоре (например, АЛУ и устройства памяти) нельзя разделить на несколько устройств с более коротким периодом задержки. Это затрудняет создание сбалансированных этапов. При проектировании конвейерного процессора Y86 авторы не будут вдаваться в такого рода подробности, однако имеет смысл оценить важность оптимизации синхронизации при проектировании фактической системы.

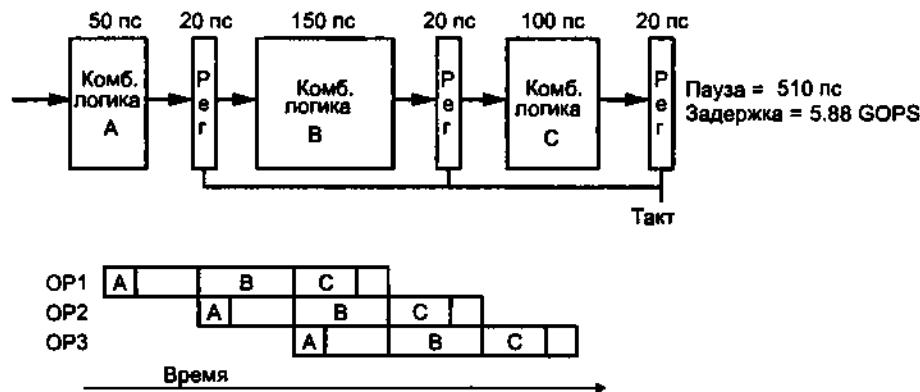


Рис. 4.30. Ограничения конвейерной обработки из-за неравномерных задержек выполнения этапов

Пропускная способность системы ограничена скоростью наиболее медленно выполняемого этапа.

УПРАЖНЕНИЕ 4.21

Предположим, что производится анализ комбинаторной логики, показанной на рис. 4.26, и выясняется, что ее можно разделить на последовательность из шести блоков (A—F), имеющих времена задержки 80, 30, 60, 50, 70 и 10 пс, соответственно, изображаемых следующим образом (рис. 4.31).

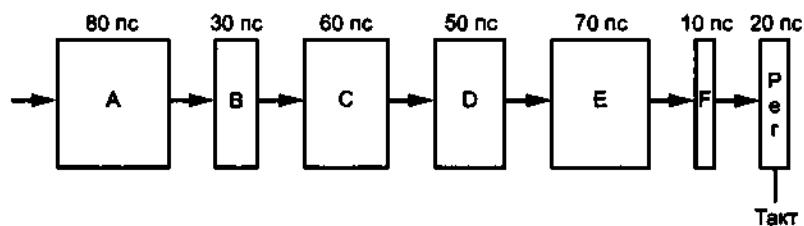


Рис. 4.31. Последовательность блоков

Можно создать конвейерные версии данного проекта путем вставки конвейерных регистров между парами этих блоков. Возникают различные комбинации глубины конвейера (количество этапов) и максимальной пропускной способности, в зависимости от того, где вставляются конвейерные регистры. Предположим, что конвейерный регистр имеет задержку 20 пс.

1. Вставка одного регистра дает двухступенчатый конвейер. Куда необходимо вставить регистр для максимального увеличения пропускной способности? Каковы будут значения пропускной способности и времени задержки?
2. Куда необходимо вставить два регистра для максимального увеличения пропускной способности трехступенчатого конвейера? Каковы будут значения пропускной способности и времени задержки?
3. Куда необходимо вставить три регистра для максимального увеличения пропускной способности четырехступенчатого конвейера? Каковы будут значения пропускной способности и времени задержки?
4. Каково должно быть минимальное число этапов для получения проекта с максимально достижимой пропускной способностью? Опишите этот проект, его пропускную способность и время задержки.

Уменьшение возвратов глубокой конвейерной обработки

На рис. 4.32 показано еще одно ограничение конвейерной обработки. В данном примере вычисление разделено на шесть этапов, каждый из которых требует 50 пс. Вставка конвейерного регистра между каждой парой этапов дает шестиступенчатый конвейер. Минимальный тактовый интервал для данной системы $50 + 20 = 70$ пс, что обеспечивает пропускную способность 14.29 GOPS. Следовательно, с удвоением числа конвейерных этапов производительность увеличивается на коэффициент $14.29/8.33 = 1.71$. Даже если сократить в два раза время, необходимое для срабатывания каждого вычислительного блока, удвоения пропускной способности не произойдет из-за задержки проходов через конвейерные регистры. Эта задержка становится

фактором, ограничивающим пропускную способность конвейера. В данном новом проекте такая задержка расходует 28.6% общего тактового интервала.

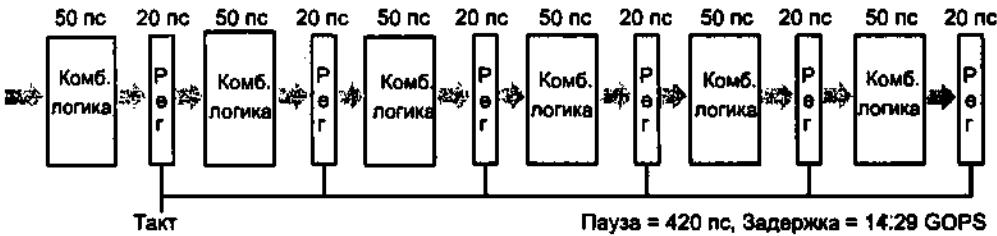


Рис. 4.32. Ограничения конвейерной обработки из-за служебных сигналов

В современных процессорах используются очень глубокие конвейерные системы (до 15 и более этапов) с целью максимизации тактовой частоты процессора. Архитекторы процессоров делят выполнение команд на большое количество очень простых шагов для того, чтобы задержка каждого этапа была предельно малой. Для этого системные проектировщики тщательно продумывают разработку конвейерных регистров. Разработчики микросхем также должны быть предельно внимательными при проектировании сети распределения тактовых сигналов с тем, чтобы их изменения происходили точно в одно и то же время во всех частях микросхемы. Все эти факторы являются определяющими при создании высокоскоростных микропроцессоров.

УПРАЖНЕНИЕ 4.22

Предположим, что можно взять систему, показанную на рис. 4.26, и разделить ее на произвольное количество конвейерных этапов с одинаковым временем задержки. Каков будет максимальный предел пропускной способности, если задержки конвейерных регистров составляют 20 пс?

4.4.4. Конвейеризация системы с обратной связью

До сих пор рассматривались только системы, в которых объекты, проходящие по конвейеру, будь то автомобили, люди или команды, абсолютно независимы друг от друга. Однако для системы, выполняющей машинные программы, такой как IA32 или Y86, существуют потенциальные зависимости между последовательными командами. Рассмотрим, к примеру, листинг 4.11 команд Y86.

Листинг 4.11. Последовательные команды

```
1 irmovl $50, %eax
2 addl %eax, %ebx
3 mrmovl 100(%ebx), %edx
```

В данной трехступенчатой последовательности существует зависимость по данным между каждой последовательной парой команд. Команда `imovl` (строка 1) сохраняет свой результат в `%eax`, который затем должен был считываться командой `addl` (строка 2); последняя сохраняет свой результат в `%ebx`, который должен считываться командой `irmovl` (строка 3).

Другой источник последовательных зависимостей возникает из-за управляющей логики команд. Рассмотрим следующий листинг 4.12 команд Y86.

Листинг 4.12. Управляющая зависимость

```

1 loop:
2 subl $edx, %ebx
3 jne targ
4 irmovl $10, %edx
5 jmp loop
6 targ:
7 halt

```

Команда `jne` (строка 3) создает *управляющую зависимость*, поскольку выход проверки условия определяет, какая команда будет выполнена следующей: `irmovl` (строка 4) или `halt` (строка 7). В рассматриваемом проекте для SEQ эти зависимости управлялись цепями обратной связи, показанными в правой части рис. 4.15. Данная обратная связь передает обновленные значения регистров вниз в регистровый файл, а новое значение счетчика команд — в регистр PC.

При переходе от системы без конвейера с обратной связью (A) к конвейерной (C) осуществляются изменения поведения вычислений, как можно видеть на конвейерных диаграммах (B и D).

На рис. 4.33 показаны опасности введения конвейера в систему, содержащую цепи обратной связи. В первоначальной системе результат каждой операции передается на следующую операцию. Это проиллюстрировано на конвейерной диаграмме (B), где результат OP1 становится входом в OP2 и т. д. Если попытаться осуществить преобразование в трехступенчатый конвейер (C), то придется изменить поведение системы. Как показано на конвейерной диаграмме (C), результат OP1 становится входом в OP4. При попытке ускорить работу системы с помощью конвейера произведена смена поведения системы.

При введении конвейера в процессор Y86 необходимо учитывать воздействие обратных связей. Понятно, что изменять поведение системы, как показано на рис. 4.33, было бы неприемлемо. Необходимо каким-то образом обрабатывать данные и управляющие зависимости между командами так, чтобы результирующее поведение соответствовало модели, определенной ISA.

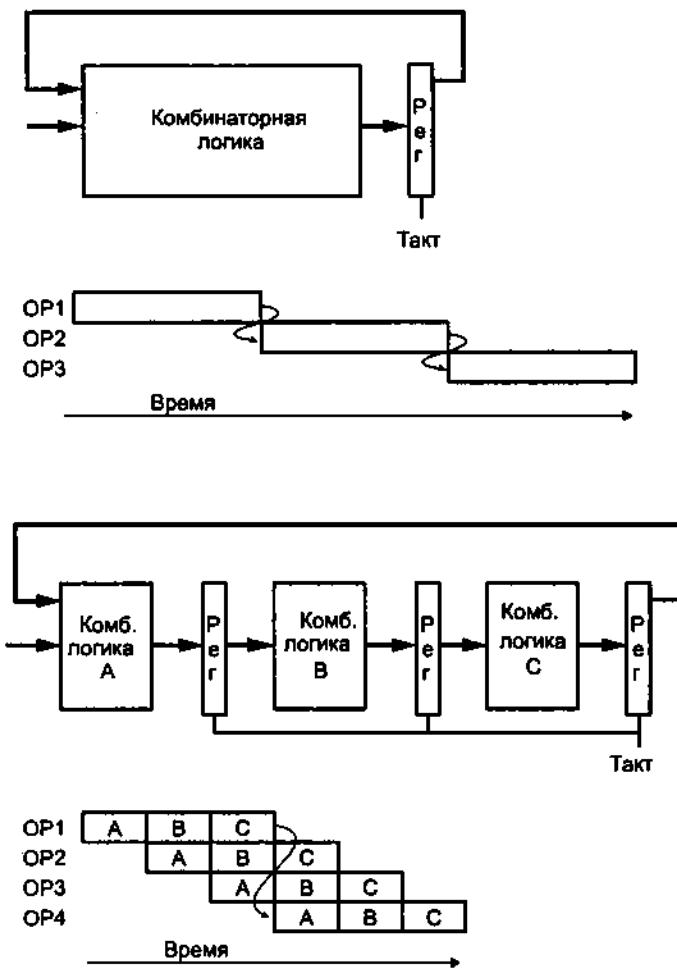


Рис. 4.33. Ограничения конвейера из-за логических зависимостей

4.5. Конвейерные реализации Y86

Наконец, все готово для решения основной задачи данной главы: проектирования конвейерного процессора Y86. В качестве базы возьмем SEQ+ и добавим между этапами конвейерные регистры. При первой попытке различные данные и управляющие зависимости корректно не обрабатываются. Однако после некоторых модификаций получаем эффективный конвейерный процессор, реализующий Y86 ISA.

4.5.1. Вставка конвейерных регистров

При первой попытке создания конвейерного процессора Y86 осуществляется вставка конвейерных регистров между этапами SEQ+ и определенным образом реконфигурируются сигналы, в результате чего получается процессор PIPE-, где минус в имени обозначает то, что данный процессор имеет меньшую производительность, нежели процессор, являющийся конечной целью данного проектирования. Абстрактная структура PIPE- показана на рис. 4.34. Конвейерные регистры обозначены на схеме в виде прямоугольников. В каждом из этих регистров содержатся множественные байты и слова (рассматриваются далее). Обратите внимание, что используется абсолютно тот же набор аппаратных устройств, что и в двух последовательных разработках, описанных ранее: SEQ (рис. 4.15) и SEQ+ (рис. 4.23).

Конвейерные регистры обозначены следующим образом:

- F — содержит прогнозируемое значение счетчика команд (далее описан кратко).
- D — расположен между этапами выборки и декодирования, содержит информацию о самых последних выбранных командах для обработки на этапе декодирования.
- E — расположен между этапами декодирования и выполнения. Здесь содержится информация о самых последних декодированных командах для обработки на этапе выполнения.
- M — расположен между этапами выполнения и памяти. Здесь содержится информация о самых последних выполненных командах для обработки на этапе памяти. Здесь также хранится информация об условиях ветвления и точках ветвления для обработки условных переходов.
- W — расположен между этапами памяти и цепями обратной связи, передающими вычисленные результаты в регистровый файл для записи и возвратный адрес в логику выборки РС при завершении команды ret.

Вставкой конвейерных регистров в SEQ+ (см. рис. 4.23) создается пятиступенчатый конвейер. Существует несколько недостатков этой версии, которые будут рассмотрены позже.

На рис. 4.35 показано, как последовательность следующего кода пройдет через пятиступенчатый конвейер, где комментарии обозначают команды, как I1 – I5 для ссылки:

```

1 irmovl $1, %eax #I1
2 irmovl $2, %escx #I2
3 irmovl $3, %edx #I3
4 irmovl $4, %ebx #I4
5 halt      #I5

```

В правой части схемы показана конвейерная диаграмма для данной последовательности команд. Как и на диаграммах для простых конвейерных вычислительных устройств, описанных в разд. 4.4, показано прохождение каждой команды через этапы, где время увеличивается слева направо. Цифры сверху обозначают циклы синхронизации различных этапов. Например, в цикле 1 выбирается команда I1, после чего она

проходит через конвейерные этапы; результаты записываются в регистровый файл по окончании этапа 5. Команда 12 выбирается в цикле 2, и ее результат записывается по окончании цикла 6 и т. д. В нижней части расшифровано показан конвейер цикла 5. В этой точке на каждом конвейерном этапе имеется инструкция.

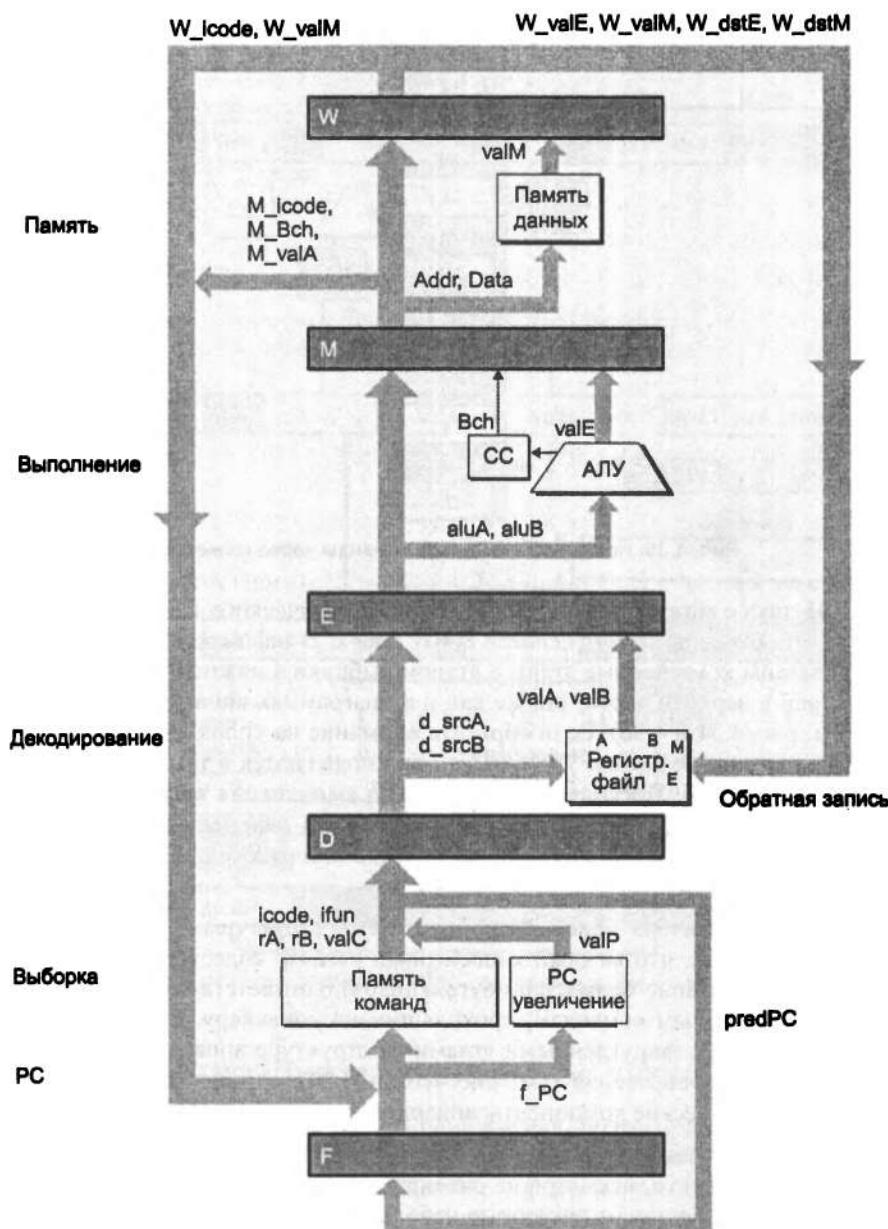


Рис. 4.34. Абстрактное представление PIPE-, первоначальная конвейерная реализация

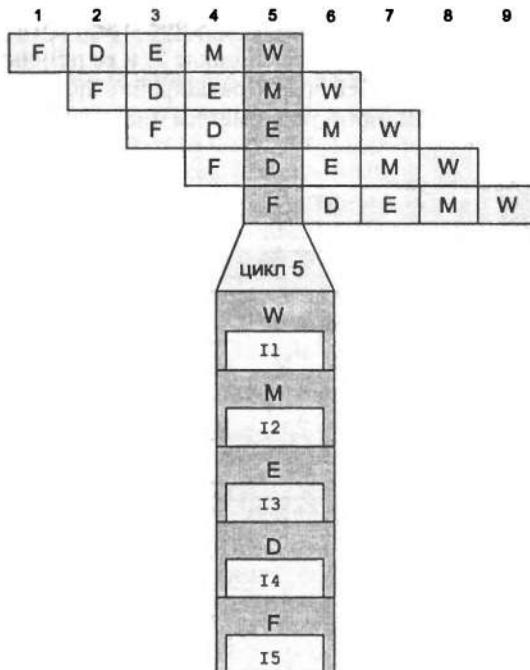


Рис. 4.35. Пример прохождения команды через конвейер

По рис. 4.35 также можно подтвердить правило графического изображения процессоров так, что команды перемещаются снизу вверх. В расширенном представлении цикла 5 показаны конвейерные этапы с этапом выборки в нижней части и этапом обратной записи в верхней части, так же как и в диаграммах конвейерных аппаратных средств (см. рис. 4.34 и 4.36). Если обратить внимание на упорядочивание команд на конвейерных этапах, то можно увидеть, что они появляются в том же порядке, что и в листинге программы. Поскольку типичный ход выполнения программы осуществляется в листинге сверху вниз, этот порядок сохранен перемещением конвейерного потока снизу вверх. Это условие особенно полезно при работе с имитаторами, сопутствующими данному тексту.

На рис. 4.36 представлена более подробная схема структуры аппаратных средств PIPE-. Можно видеть, что каждый конвейерный регистр содержит многочисленные поля (изображены в виде белых прямоугольников), соответствующие сигналам, относящимся к различным командам, проходящим по конвейеру. В отличие от ярлыков, изображенных с закругленными углами на структуре аппаратных средств двух последовательных процессоров (см. рис. 4.16 и 4.24), эти белые прямоугольники представляют фактические компоненты аппаратных средств.

При сравнении абстрактной структуры SEQ+ (см. рис. 4.23) со структурой PIPE- (см. рис. 4.34) видно, что, несмотря на очевидное сходство общих потоков, проходящих через этапы, существуют некоторые небольшие различия. Они будут рассмотрены перед началом подробной реализации.

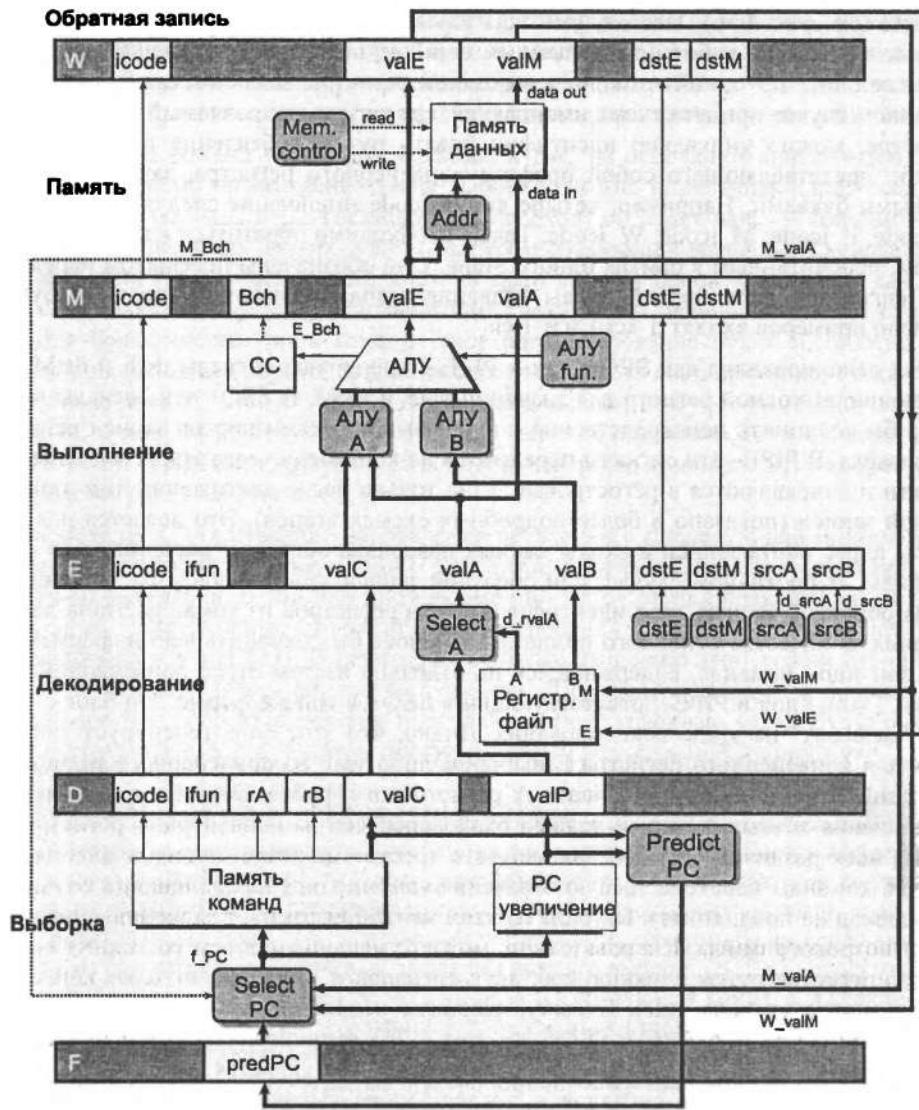


Рис. 4.36. Структура аппаратных средств PIPE-; первоначальная конвейерная реализация

4.5.2. Реконфигурация и смена меток сигналов

SEQ+ за один раз обрабатывает только один сигнал, поэтому для сигналов существуют уникальные значения valC, scrA и valE. В данном конвейерном проекте будут иметь место версии этих значений, связанные с разными командами, проходящими через систему. Например, в подробной структуре PIPE- имеется четыре белых прямогоугольника с ярлыками "icode", содержащие сигналы icode для четырех разных

команд (см. рис. 4.36). Необходимо тщательно следить за использованием нужной версии сигнала во избежание серьезных ошибок, например сохранения результата, вычисленного для одной команды в выходном регистре, заданном другой командой. В данном случае принята схема именования, где сигнал, сохраняемый в конвейерном регистре, можно уникально идентифицировать путем добавления префикса к его имени, представляющего собой префикс конвейерного регистра, записанного прописными буквами. Например, четыре копии `icode` именованы следующим образом: `D_icode`, `E_icode`, `M_icode`, `W_icode`. Также необходимо обратиться к некоторым сигналам, просчитанным в рамках одного этапа. Они обозначены префиксом перед именем сигнала в виде первой буквы названия этапа, написанной строчной буквой. В число примеров входят `d_scrA` и `e_Bch`.

Этапы декодирования как SEQ+, так и PIPE— генерируют сигналы `dstE` и `dstM`, указывающие выходной регистр для значений `valE` и `valM`. В SEQ+ эти сигналы можно было бы соединить непосредственно с входными адресами портов записи регистрационного файла. В PIPE— эти сигналы передаются по конвейеру через этапы выполнения и памяти и направляются в регистровый файл только после достижения ими этапа обратной записи (показано в более подробных схемах этапов). Это делается для того, чтобы адрес порта записи и входы данных наверняка обладали значениями из одной команды. В противном случае, при обратной записи записывались бы значения для этапа обратной записи, но с идентификаторами регистров из команды этапа декодирования. В качестве основного принципа хотелось бы сохранить всю информацию о той или иной команде, содержащейся на отдельно взятом этапе конвейерной обработки. Один блок в PIPE—, отсутствующий в SEQ+ в той же форме, это блок с ярлыком "Select A" на этапе декодирования. Видно, что этот блок генерирует значение `valA` для конвейерного регистра Е выбором либо `valP` из конвейерного регистра D, либо значения, считанного с порта А регистрационного файла. Этот блок включен для уменьшения объема, который должен быть перенесен на конвейерные регистры Е и M. Из всех различных команд только `call` требует на этапе памяти значения `valP`. Только команды перехода требуют значения `valP` на этапе выполнения (в случае, если переход не предпринят). Ни одна из этих команд не требует значения, считанного из регистрационного файла. Следовательно, можно уменьшить объем состояния конвейерного регистра путем слияния этих двух сигналов и их передачи через конвейер в виде единого сигнала `valA`. Этим устраняется необходимость в блоке с ярлыком "Data" в SEQ (см. рис. 4.26) и SEQ+ (см. рис. 4.24), выполнявшим аналогичную цель. При проектировании программных средств обычным является тщательная идентификация использования сигналов с последующим уменьшением числа состояний регистров и записью путем слияния представленных здесь сигналов.

4.5.3. Прогнозирование следующего значения РС

Для более тщательной обработки зависимостей управления в проекте PIPE— выполнены некоторые измерения. Целью конвейерного проектирования является выдача новой команды на каждом цикле синхронизации с тем, чтобы новая команда вступала в этап выполнения и полностью завершалась. Достижение этой цели обеспечит пропускную способность одной команды за цикл. Для этого необходимо определить местонахождение следующей команды сразу после выборки текущей. К сожалению,

если выбранная команда является условным ветвлением, то неизвестно, должно ли выбираться ветвление через несколько циклов после того, как команда пройдет этап выполнения. Подобным же образом, если выбранной командой является `ret`, то нельзя определить обратное местоположение, пока команда не пройдет через этап памяти.

За исключением команд условного перехода и `ret`, на основании информации, вычисленной во время прохождения этапа выборки, можно определить адрес следующей команды. Для `call` и `jmp` (безусловный переход) он будет `valC` — константным словом в команде, а для других `valP` — адресом следующей команды. Следовательно, в большинстве случаев можно достичь цели выдачи новой команды в каждом цикле синхронизации путем прогнозирования следующего значения счетчика команд (PC). Для большинства типов команд такое прогнозирование будет абсолютно надежным. Для условных переходов можно спрогнозировать либо выбор перехода (т. е. новым значением PC станет `valC`), либо переход выбран не будет (тогда новым значением PC станет `valP`). В любом из этих случаев, так или иначе, прогноз некорректен, т. е. были выбраны и частично выполнены неверные команды. Этот вопрос еще будет рассматриваться в разд. 4.5.9.

Данная методика "угадывания" направления ветвления с последующей инициализацией выборки команд в соответствии с результатами угадывания называется *прогнозированием ветвления*. В той или иной форме оно используется практически во всех процессорах. Эффективные стратегии прогнозирования выбора ветвей еще необходимо подробно изучать [31]. В некоторых системах решению этой задачи отводится достаточно большой объем аппаратных средств. В рассматриваемом же проекте будет применяться простая стратегия прогнозирования: условные ветви будут выбираться всегда, поэтому результатом прогноза (новым значением PC) станет `valC`.

Другие стратегии прогнозирования ветвей

В данном проекте используется стратегия прогнозирования "выбирать всегда". Исследования показывают, что доля успешных попыток этой стратегии составляет порядка 60% [31]. В противоположность ей, доля успешных попыток стратегии "никогда не выбирать" (NT) составляет порядка 40%. Немного более сложная стратегия под названием "обратный выбор без прямого выбора" (BTFNT) прогнозирует ветви на нижние адреса, после чего выбирается следующая команда, а верхние адреса не выбираются. Доля успешных попыток этой стратегии составляет порядка 65%. Подобный прогресс происходит из того факта, что петли (циклы) закрываются обратными ветвями, и циклы обычно выполняются многократно. Передние ветви используются для условных операций, которые выбираются с меньшей вероятностью. В упр. 4.39 и 4.40 конвейерный процессор Y86 можно модифицировать для реализации стратегий прогнозирования ветвей NT и BTFNT.

Воздействие от неудачного прогнозирования ветвей на производительность программы рассматривается в разд. 5.1.2 в контексте оптимизации программы.

Остается прогнозирование нового значения PC в результате выполнения команды `ret`. В отличие от условных переходов, здесь имеется практически бесконечное множество возможных результатов, поскольку обратный адрес будет любым словом, находящимся в верхней части стека. В рассматриваемом проекте попыток прогнози-

рования какого бы то ни было значения обратного адреса делать не будет. Вместо этого будет осуществлена простая задержка обработки любых команд до прохождения командой `ret` этапа обратной записи. Мы вернемся к данной части реализации в разд. 4.5.9.

Прогнозирование возвратного адреса со стеком

В большинстве программ обратные адреса прогнозировать очень просто, поскольку вызовы процедур и возврата имеют место в согласованных парах. Большую часть времени вызова процедуры она возвращается в команду, следующую за вызовом. Это свойство применяется в высокопроизводительных процессорах путем включения аппаратного стека в устройство выборки команды, содержащее обратный адрес, сгенерированный командами вызова процедуры. Всякий раз при выполнении команды вызова процедуры ее обратный адрес выталкивается в стек. Когда обратная команда выбрана, самое верхнее значение выталкивается из этого стека и используется в качестве спрогнозированного обратного адреса. Как и при прогнозировании ветви, здесь должен быть механизм восстановления, на случай, если прогноз некорректен, поскольку бывает так, что вызовы и возврата не совпадают. Вообще говоря, данный прогноз весьма надежен. Такой аппаратный стек не является частью состояния, видимого для программиста.

Этап выборки PIPE-, показанный в нижней части рис. 4.36, отвечает как за прогнозирование следующего значения PC, так и за выбор фактического PC для выборки команды. Видно, что блок с ярлыком "Predict PC" может выбрать либо valP (вычисленное приращением PC), либо valC из выбранной команды. Данное значение хранится в конвейерном регистре F, как спрогнозированное значение счетчика команд. Блок с ярлыком "Select PC" похож на блок с ярлыком "PC" на этапе выбора PC SEQ+ (см. рис. 4.24). Он выбирает одно из трех значений, которые выполняют роль адресов для команды памяти: спрогнозированный PC, значение valP для команды невыбранной ветви, достигающей конвейерного регистра M (в регистре M_valA), либо значение обратного адреса, когда команда `ret` достигает конвейерного регистра W (сохранено в W_valM).

Мы вернемся к рассмотрению команд перехода и возврата по завершении логики управления конвейером в разд. 4.5.9.

4.5.4. Риски конвейерной обработки

Структура PIPE- является хорошим началом создания конвейерного процессора Y86. Однако, если вспомнить разд. 4.4.4, введение конвейерной обработки в систему с обратной связью может привести к проблемам, когда между последовательными командами обнаружатся зависимости. Этот вопрос необходимо прояснить до полного завершения данного проекта. Подобного рода зависимости принимают две формы:

- зависимость по данным, когда результаты, вычисленные одной командой, используются в качестве данных для следующей команды;
- зависимость управления, когда одна команда определяет местоположение следующей команды, например, при выполнении команд перехода, вызова или возврата.

Когда такие зависимости имеют потенциал ошибочных вычислений конвейером, они называются *рисками*. Как и зависимости, риски можно классифицировать либо как *риски по данным*, либо как *риски управления*. В этом разделе рассматриваются только риски по данным. Риски управления будут рассмотрены в качестве части общего управления конвейером (см. разд. 4.5.9).

На рис. 4.37 показана обработка последовательности команд, называемой процессором PIPE- как prog1. Этот код загружает в регистры команд %edx и %eax значения 10 и 3 соответственно, выполняет три команды *nop*, после чего добавляет регистр %edx к %eax. Внимание сосредотачивается на потенциальных данных по рискам, являющихся результатом зависимостей по данным между двумя командами *irmovl* и командой *addl*. В правой части схемы показана конвейерная диаграмма последовательности команд. Конвейерные этапы для циклов 6 и 7 на конвейерной диаграмме выделены отдельно. Ниже показано расширенное представление процедуры обратной записи в цикле 6 и процесс декодирования в цикле 7. После начала цикла 7 обе команды *irmovl* проходят через этап обратной записи, и получается, что регистровый файл содержит значения %edx и %eax. По мере того как команда *addl* проходит этап декодирования в цикле 7, она соответствующим образом будет считывать корректные значения исходных операндов. Зависимости по данным между двумя командами *irmovl* и командой *addl* в этом примере не создали рисков по данным.

```
# prog1
0x000: irmovl $10,%edx
0x006: irmovl $3,%eax
0x00c: nop
0x00d: nop
0x00e: nop
0x00f: addl %edx,%eax
0x011: halt
```

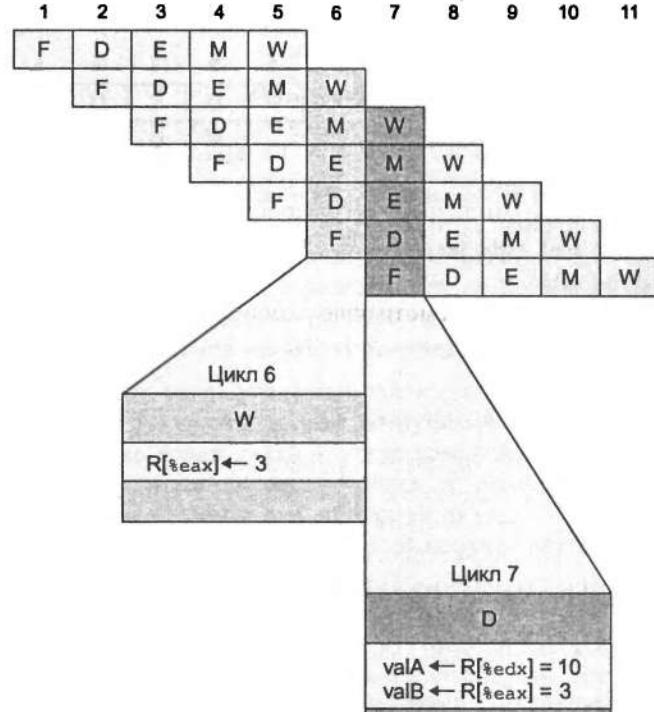


Рис. 4.37. Конвейерное выполнение prog1 без специального конвейерного управления

Читатели уже отметили, что `prog1` будет проходить по конвейеру и получать корректные результаты, потому что три команды `por` создают задержку между командами с зависимостями по данным. Посмотрим, что произойдет, если убрать эти пор-команды. На рис. 4.38 показан конвейерный поток программы под названием `prog2`, содержащей две команды `por` между двумя командами `irmovl`, что создает значения для регистров `%edx` и `%eax`, а также команду `addl`, имеющую эти два регистра в качестве операндов. В этом случае критический шаг имеет место в цикле 6, когда команда `addl` считывает свои операнды из регистрового файла. Расширенное представление конвейерной деятельности во время этого цикла показано в нижней части схемы. Первая команда `irmovl` прошла этап обратной записи, поэтому регистр команд `%edx` в регистровом файле оказывается обновленным. Вторая команда `irmovl` находится в данном цикле на этапе обратной записи, поэтому запись в регистр команд `%eax` происходит только с началом цикла 7, по мере возрастания синхронизации (тактовой частоты). В результате для регистра `%eax` будет считано некорректное значение (здесь предполагается, что изначально значения всех регистров равны 0), поскольку нерешенной записи для этого регистра еще не появилось. Понятно, что созданный конвейер придется соответствующим образом адаптировать для корректного разрешения данного риска.

По окончании цикла 6 записи в регистр команд `%eax` не происходит (рис. 4.38), что обеспечивает команде `addl` некорректное значение для данного регистра на этапе декодирования.

```
# prog2
0x000: irmovl $10,%edx
0x006: irmovl $3,%eax
0x00c: por
0x00d: por
0x00e: addl %edx,%eax
0x010: halt
```

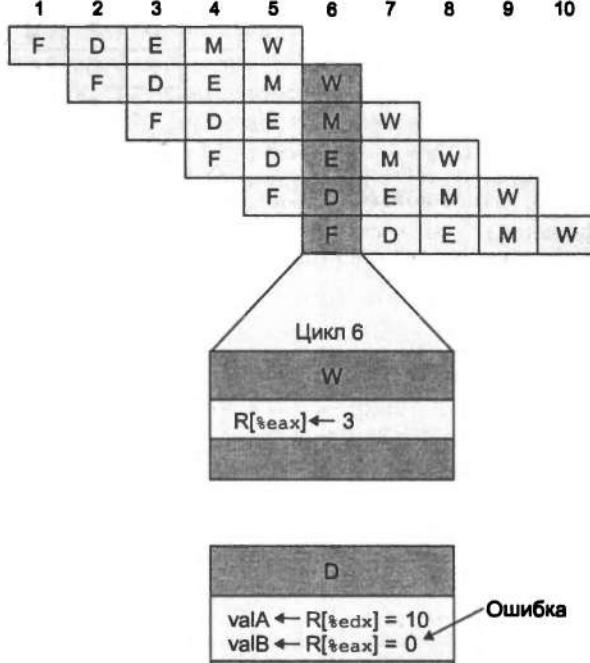


Рис. 4.38. Конвейерное выполнение `prog2` без специального конвейерного управления

prog3

```

0x000: irmovl $10,%edx
0x006: irmovl $3,%eax
0x00c: nop
0x00d: addl %edx,%eax
0x00f: halt

```

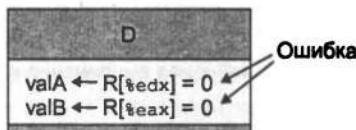
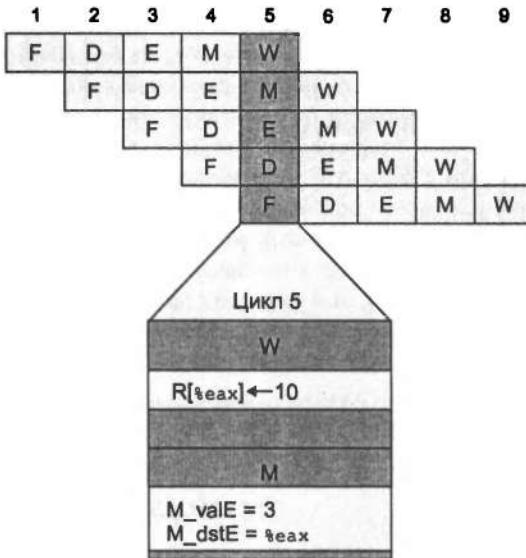


Рис. 4.39. Конвейерное выполнение prog3 без специального конвейерного управления

На рис. 4.39 показано, что происходит, когда между командами `irmovl` и `addl` имеется только одна команда `nop`, что создает программу `prog3`. Теперь необходимо изучить поведение конвейера во время цикла 5, когда команда `addl` проходит этап декодирования. К сожалению, незавершенная запись в регистр `%edx` по-прежнему находится на этапе обратной записи, а в регистр `%eax` — на этапе памяти. Следовательно, команда `addl` получит некорректные значения для обоих операндов.

На рис. 4.40 показано, что происходит, когда удаляются все команды `nop` из промежутка между командами `irmovl` и `addl`, что создает программу `prog4`. Теперь необходимо изучить поведение конвейера во время цикла 4, когда команда `addl` проходит этап декодирования. К сожалению, незавершенная запись в регистр `%edx` по-прежнему находится на этапе памяти, а новое значение в регистр `%eax` — только вычисляется на этапе выполнения. Следовательно, команда `addl` получит некорректные значения для обоих операндов.

Эти примеры иллюстрируют то, что риски по данным могут возникнуть для команды, когда один из ее операндов обновляется любыми из трех предшествующих команд. Эти риски возникают оттого, что разрабатываемый конвейерный процессор считывает операнды для команды из регистрового файла на этапе декодирования, но не записывает результаты для инструкции в регистровый файл до завершения очередных трех циклов, т. е. до того, как команда пройдет этап обратной записи.

```
# prog4
0x000: irmovl $10,%edx
0x006: irmovl $3,%eax
0x00c: addl %edx,%eax
0x00e: halt
```

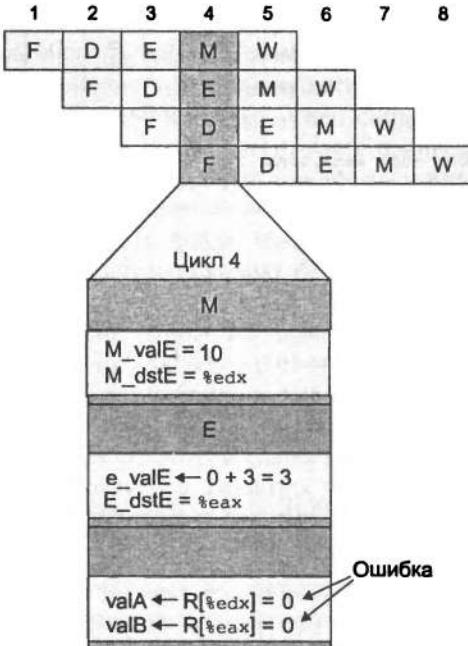


Рис. 4.40. Конвейерное выполнение prog4 без специального конвейерного управления

Перечисление классов рисков по данным

Потенциальные риски возникают тогда, когда одна команда обновляет некую часть состояния программы, которая (часть) будет считываться более поздней командой. В состояние программы входят регистры команд, коды состояний, память и счетчик программ. Рассмотрим возможности рисков для каждого из перечисленных состояний.

- Регистры команд. Этим рискам определение уже дано. Они возникают из-за того, что регистровый файл считывается в одном состоянии, а записывается — в другом, что приводит к возможным ненамеренным взаимодействиям между разными командами.
- Коды условий. Они записываются (целочисленными операциями) и считаются (условными переходами) на этапе выполнения. К тому времени, как условный переход проходит данный этап, любые из предшествующих целочисленных операций этот этап уже завершают. Здесь не возникает никаких рисков.
- Счетчик команд. Расхождения между обновлением и считыванием значений счетчика программ обуславливают риски. Рисков не возникает, когда логика на этапе выборки правильно прогнозирует новое значение счетчика программ до выборки следующей команды. Неправильно спрогнозированные ветви и команды *ret* требуют особой обработки, которая рассматривается в разд. 4.5.9.

□ Память. Процессы записи и считывания памяти для хранения данных осуществляются на этапе памяти. К тому времени, как память считывания команды достигает этого этапа, любая память записи любых предыдущих команд уже это сделает. С другой стороны, на этапе памяти может иметь место взаимное влияние данных записи команд и считывание команд на этапе выборки, поскольку память команд и данных делят единое адресное пространство. Это может происходить только с программами, содержащими *самоизменяющийся код*, когда команды записываются в область памяти, из которой они потом выбираются. Некоторые системы включают в себя сложные механизмы выявления и избегания подобных рисков, в других же программах простодается указание о недопустимости самоизменяющегося кода. Для простоты предполагается, что программы не должны модифицироваться самостоятельно.

Данный анализ показывает, что обрабатывать необходимо только риски регистровых данных и риски управления.

4.5.5. Как избежать рисков по данным с помощью останова

Одной из наиболее распространенных методик избегания рисков является так называемый *останов*, когда процессор удерживает в конвейере одну или несколько команд до тех пор, пока условие риска не исчезнет. Процессор может избежать рисков по данным путем удерживания команды на этапе декодирования до тех пор, пока один из ее исходных операндов будет генерироваться некоторой командой на более позднем этапе прохождения конвейера. Эта методика показана в виде схемы на рис. 4.41 (prog2), 4.42 (prog3) и 4.43 (prog4). Когда команда addl находится на этапе декодирования, управляющая логика конвейера выявляет, что как минимум одна из команд на этапе выполнения, памяти или обратной записи будет обновлять либо регистр %edx, либо регистр %eax. Вместо того чтобы предоставлять команде addl возможность прохождения этапа с некорректными результатами, команда останавливается и удерживается на этапе декодирования для одного (prog2), двух (prog3) или даже трех (prog4) дополнительных циклов. Для всех трех программ команда addl, наконец, получает корректные значения для двух ее исходных операндов в цикле 7, после чего проходит вниз по конвейеру.

На рис. 4.41 после декодирования команды addl в цикле 6 логика управления остановами выявляет риск по данным, благодаря задержке процесса записи в регистр %eax на этапе обратной записи. Она вставляет так называемый *кружок* (*bubble*) в этап выполнения и повторяет декодирование команды addl в цикле 7. По существу, машина осуществила динамическую вставку команды пор, что обеспечило поток, сходный с потоком для prog1 (см. рис. 4.37).

На рис. 4.42 после декодирования команды addl в цикле 5 логика управления остановами выявляет риски по данным для обоих исходных регистров. Она вставляет кружок в этап выполнения и повторяет декодирование команды addl в цикле 6. Затем выявляется риск для регистра %eax, на этап выполнения вставляется второй кружок, и повторяется декодирование команды addl в цикле 7. По существу, машина осуществ-

вила динамическую вставку двух команд пор, что обеспечило поток, сходный с потоком для prog1 (см. рис. 4.37).

```
# prog2
0x000: irmovl $10,%edx
0x006: irmovl $3,%eax
0x00c: nop
0x00d: nop
    bubble
0x00e: addl %edx,%eax
0x010: halt
```

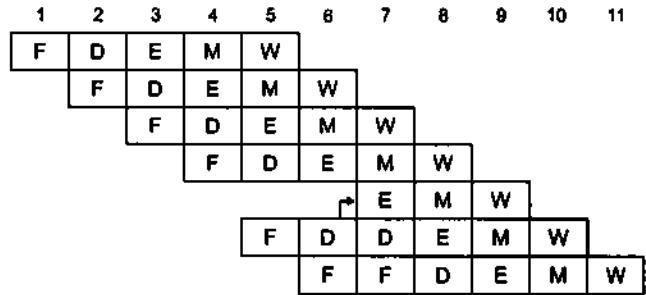


Рис. 4.41. Конвейерное выполнение prog2 с использованием остановов

```
# prog3
0x000: irmovl $10,%edx
0x006: irmovl $3,%eax
0x00c: nop
    bubble
    bubble
0x00d: addl %edx,%eax
0x00f: halt
```

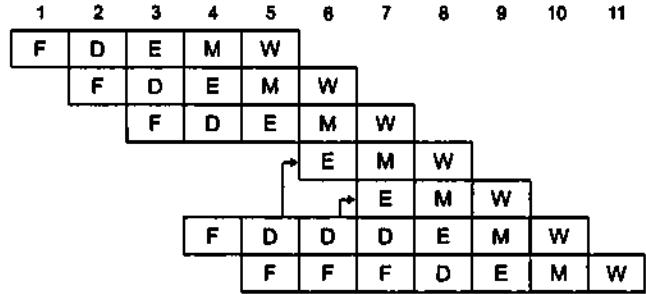


Рис. 4.42. Конвейерное выполнение prog3 с использованием остановов

На рис. 4.43 после декодирования команды addl в цикле 4 логика управления остановами выявляет риски по данным для обоих исходных регистров. Она вставляет кружок в этап выполнения и повторяет декодирование команды addl в цикле 5. Затем выявляются риски для обоих исходных регистров, на этап выполнения вставляется кружок, и повторяется декодирование команды addl в цикле 6. Также выявляется риск для исходного регистра %eax, на этап выполнения вставляется кружок, и повторяется декодирование команды addl в цикле 7. По существу, машина осуществила динамическую вставку трех команд пор, что обеспечило поток, сходный с потоком для prog1 (см. рис. 4.37).

При удерживании команды addl на этапе декодирования также необходимо удерживать команду halt, следующую за ней на этапе выборки. Это можно сделать удерживанием фиксированного значения счетчика программ так, чтобы команда halt выбиралась многократно до завершения останова.

Остановы включают в себя удержание одной группы команд на своих этапах, в то время как другие команды продолжают прохождение по конвейеру. В рассматриваемом примере команда addl удерживается на этапе декодирования, а halt — на этапе выборки для одного из трех циклов; в это время две команды irmovl и команды пор

(в случаях с prog2 и prog3) продолжают прохождение этапов выполнения, памяти и обратной записи. Что же следует делать на этапах, на которых бы обычно обрабатывалась команда add1? На этапе выполнения вставляется кружок всякий раз при удержании команды на этапе декодирования. Кружок похож на динамически генерированную команду por, он не вызывает изменений регистров, памяти или кодов условий. Они изображены в виде белых рамок на конвейерных диаграммах 4.41—4.43. На этих схемах одна из рамок, помеченных литерой D для команды add1, соединена стрелками с рамкой, помеченной литерой E для одного из конвейерных кружков. Стрелки указывают на то, что на этапе выполнения вместо команды add1, которая обычно проходила бы с этапа декодирования на этап выполнения, вставляется кружок. В разд. 4.5.9 рассматриваются подробные механизмы остановки конвейера и вставки кружков.

```
# prog4
0x000: irmovl $10,%edx
0x006: irmovl $3,%eax
        bubble
        bubble
        bubble
0x00c:  addl %edx,%eax
0x00e:  halt
```

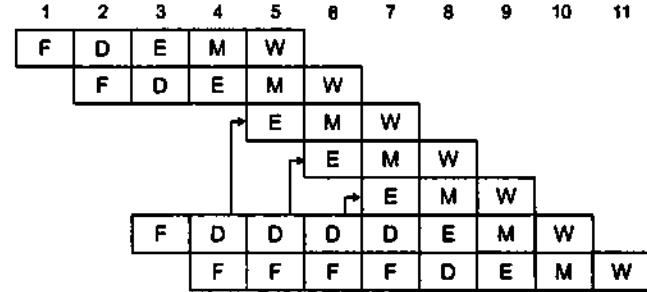


Рис. 4.43. Конвейерное выполнение prog4 с использованием остановов

Использованием остановок при обработке рисков по данным эффективно выполняются программы prog2, prog3 и prog4 путем динамического генерирования конвейерного потока, показанного для prog1 (рис. 4.37). Вставка одного кружка для prog2, двух — для prog3 и трех — для prog4 оказывает тот же эффект, что и наличие трех команд por между второй командой irmovl и командой add1. Реализация этого механизма достаточно проста, однако конечная производительность оказывается не очень высокой. Существует много случаев, когда одна команда обновляет значение регистра, а команда, следующая непосредственно за ней, использует тот же регистр. При этом конвейер останавливается на время до трех циклов, что заметно снижает общую пропускную способность.

4.5.6. Как избежать рисков по данным с помощью продвижения

В данном проекте PIPE — исходные операндычитываются из регистрового файла на этапе декодирования, однако на этапе обратной записи для одного из этих исходных регистров может существовать приостановленная запись. Вместо остановки до завершения процесса записи она может просто проходить значение, которое должно записываться в конвейерный регистр E в виде исходного операнда. На рис. 4.44 показана эта стратегия с расширенным представлением конвейерной диаграммы цикла б

для prog2. Логика этапа декодирования выявляет, что регистр %eax является исходным для операнда valB, а также то, что существует приостановленная запись в %eax в порту записи Е. Следовательно, можно избежать останова простым использованием слова данных, передаваемого на порт Е (сигнал W_valE) в виде значения для операнда valB. Такая методика передачи результирующего значения непосредственно с одного конвейерного этапа на более ранний этап называется *прдвижением данных* (или просто *прдвижением*). Оно позволяет командам prog2 проходить конвейер без остановов.

```
# prog2
0x000: irmovl $10,%edx
0x006: irmovl $3,%eax
0x00c: nop
0x00d: nop
0x00e: addl %edx,%eax
0x010: halt
```

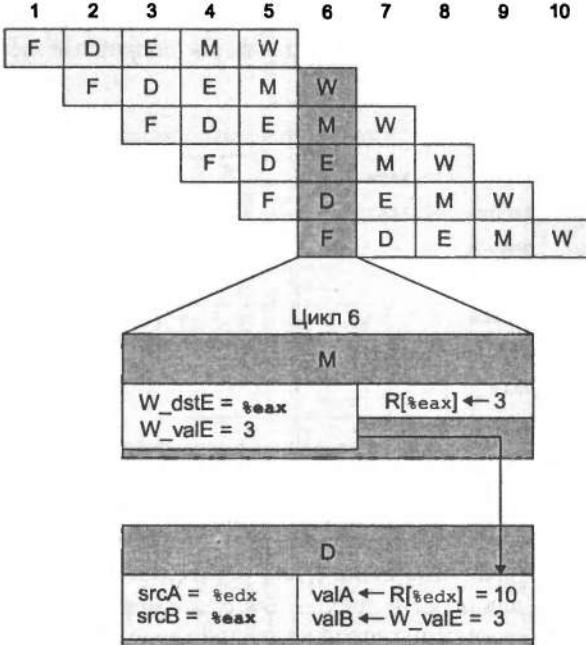


Рис. 4.44. Конвейерное выполнение prog2 с использованием продвижения

На рис. 4.45 показано, что продвижение данных также можно использовать при наличии приостановленной записи в какой-либо регистр этапа памяти для того, чтобы избежать необходимости останова программы prog3. В цикле 5 логика этапа декодирования выявляет приостановленную запись в регистр %edx на порту Е на этапе обратной записи, а также приостановленную запись в регистр %eax, на пути на порт Е, но по-прежнему находящуюся в этапе памяти. Вместо останова до завершения процессов записи можно использовать значение на этапе обратной записи (сигнал W_valE) для операнда valA и значение на этапе памяти (сигнал M_valE) для операнда valB.

Для применения продвижения данных в полной мере можно также передавать вновь вычисленные значения из этапа выполнения на этап декодирования, избегая необходимости останова программы prog4, как показано на рис. 4.46. В цикле 4 логика этапа

prog3

```

0x000: irmovl $10,%edx
0x006: irmovl $3,%eax
0x00c: nop
0x00d: addl %edx,%eax
0x00f: halt

```

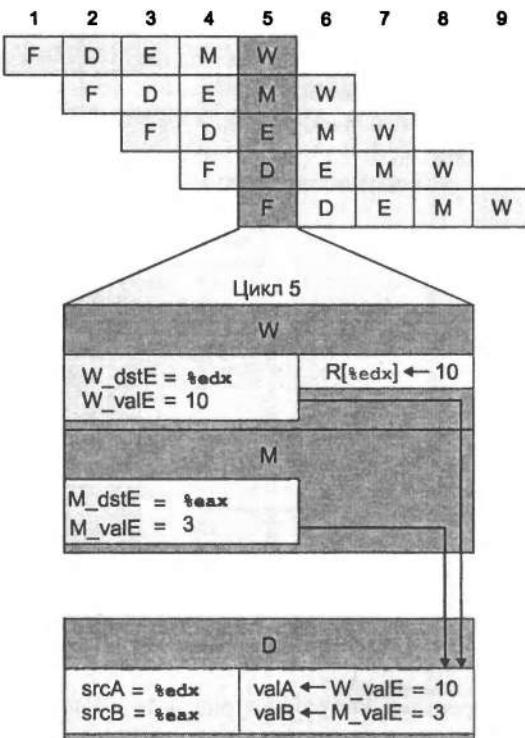


Рис. 4.45. Конвейерное выполнение prog3 с использованием продвижения

декодирования выявляет наличие приостановленной записи в регистр %eax на этапе памяти, также то, что значение, вычисляемое АЛУ на этапе выполнения, будет позднее записано в регистр %eax. Логика может использовать это значение на этапе памяти (сигнал M_valE) для операнда $valA$. Она также может использовать выход АЛУ (сигнал e_valE) для операнда $valB$. Обратите внимание на то, что использование выхода АЛУ не вызывает никаких проблем синхронизации. Этапу декодирования потребуется только сгенерировать сигналы $valA$ и $valB$ к концу цикла синхронизации с тем, чтобы конвейерный регистр Е можно было загрузить результатами с этапа декодирования по мере возрастания синхронизации для начала следующего цикла. До этой точки выход АЛУ будет действителен.

Виды использования продвижения, показанные в программах, включают в себя продвижение значений, сгенерированных АЛУ и предназначенных для передачи на порт записи Е. Продвижение также можно использовать со значениями, считанными из памяти и предназначенными для передачи на порт записи М. С этапа памяти можно направить значение, только что считанное из памяти для хранения данных (сигнал m_valM). С этапа обратной записи приостановленную запись можно направить на порт М (сигнал W_valM). При этом получается пять различных источников продвижения (e_valE , m_valM , M_valE , W_valM и W_valE) и два различных назначения продвижения ($valA$ и $valB$).

```
# prog4
0x000: irmovl $10,%edx
0x006: irmovl $3,%eax
0x00c: addl %edx,%eax
0x00e: halt
```

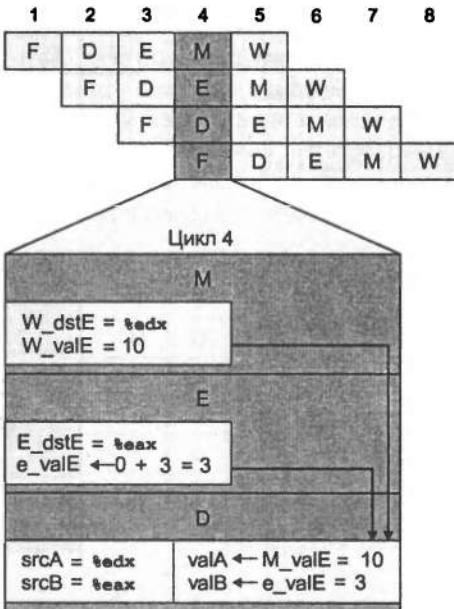


Рис. 4.46. Конвейерное выполнение prog4 с использованием продвижения

На расширенных диаграммах рис. 4.44—4.46 показано, как логика этапа декодирования может определить то, какое значение следует использовать: из регистрового файла или продвинутый. Идентификатор выходного регистра связан с каждым значением, записываемым обратно в регистровый файл. Управляющая логика может сравнить эти идентификаторы с идентификаторами исходного регистра $srcA$ и $srcB$ для выявления случая продвижения. Может получиться так, что идентификаторы регистра универсального назначения совпадут с одним из исходных регистров. Для обработки таких случаев необходимо установить приоритет между различными источниками продвижения. Этот вопрос обсуждается при рассмотрении подробного проектирования логики продвижения.

На рис. 4.47 показана абстрактная структура PIPE, расширение PIPE-, которая может управлять рисками по данным путем продвижения. Видно, что дополнительные цепи обратной связи (выделены темно-серым цветом) добавлены от источников продвижения вниз на этап декодирования. Эти обходные пути входят в блок, обозначенный "Block" на этапе декодирования.

Дополнительные обходные пути (обозначены темно-серым цветом) обеспечивают продвижение результатов из трех предшествующих команд. Это позволяет управлять большей частью форм рисков по данным без останова конвейера.

Данный блок генерирует исходные операнды $valA$ и $valB$, используя либо значения, считанные из регистрового файла, либо одно из продвинутых значений.

На рис. 4.48 представлена более подробная схема аппаратной структуры PIPE. При ее сравнении со структурой PIPE- (см. рис. 4.36) видно, что значения из пяти источни-

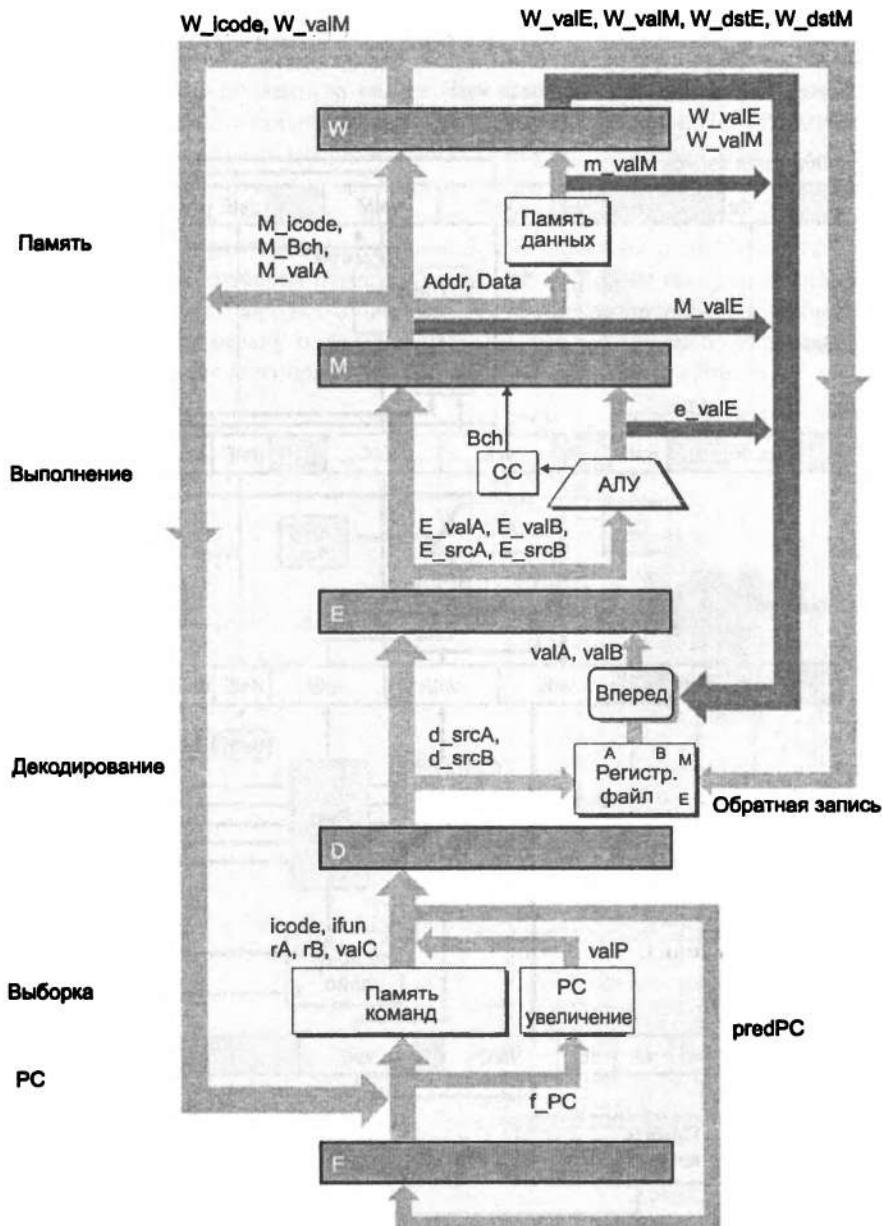


Рис. 4.47. Абстрактное представление PIPE: окончательная конвейерная реализация

ков продвижения возвращаются в два блока, помеченные как "Sel+Fwd A" и "Fwd B" на этапе декодирования. Блок, помеченный как "Sel+Fwd A" выполняет объединенную роль блока, помеченного как "Select A" в PIPE-, с логикой продвижения. Это позволяет значению valA конвейерного регистра M быть либо приращенным значе-

нием valP счетчика команд, значением, считанным с порта A регистра файла, либо одним из продвинутых значений. Блок, помеченный "Fwd B", реализует логику продвижения для исходного операнда valB.

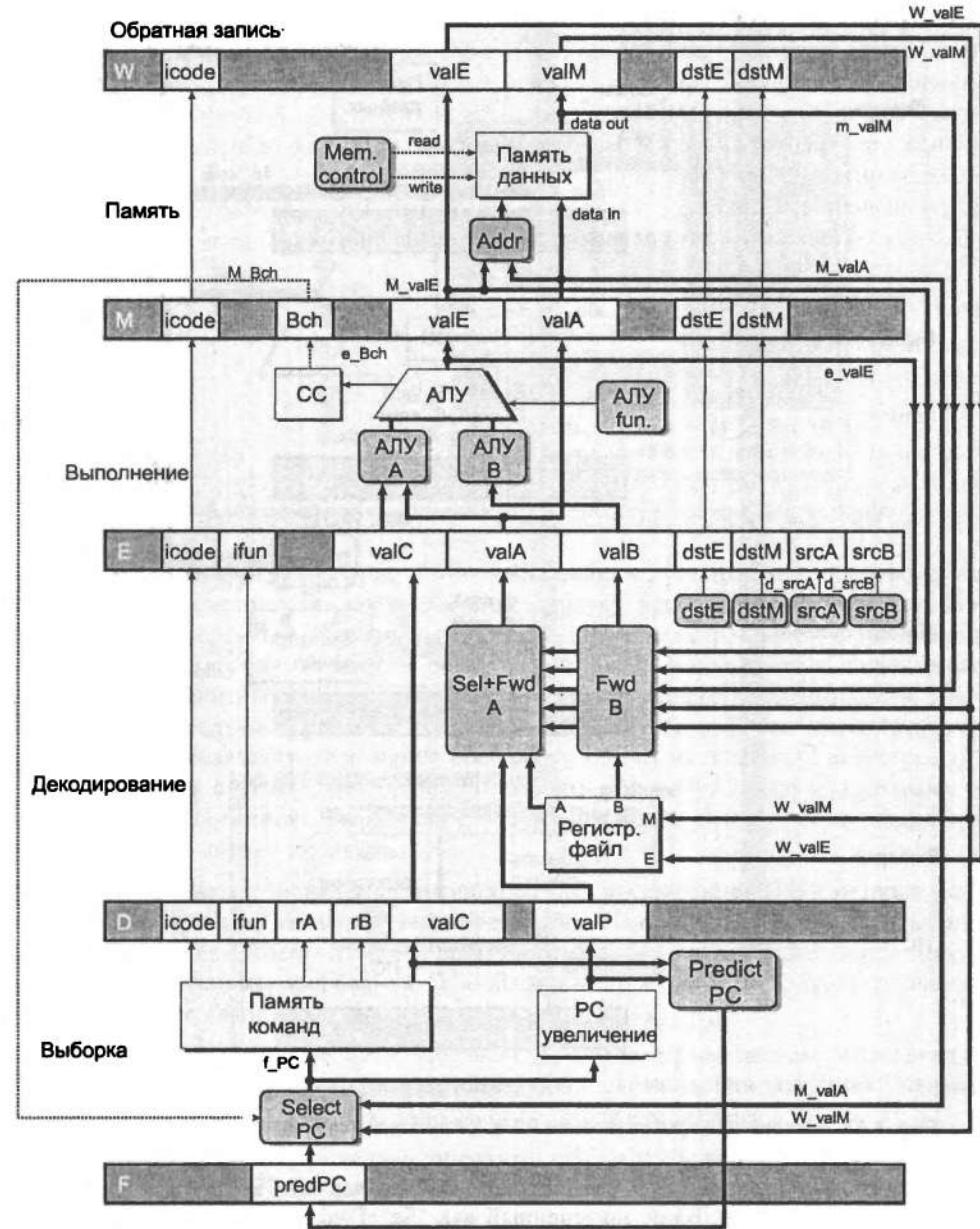


Рис. 4.48. Структура аппаратных средств PIPE; окончательная конвейерная реализация

4.5.7. Риски по данным load/use

Одним классом рисков по данным нельзя управлять чистым продвижением, потому что считывание из памяти происходит ближе к концу конвейера. На рис. 4.49 показан пример риска по загрузке и использованию (load/use), где одна команда (`irmovl` в адресе 0x018) считывает значение из памяти для регистра `%eax`, тогда как следующей команде (`addl` в адресе 0x01e) это значение необходимо в качестве исходного операнда. Расширенные представления циклов 7 и 8 показаны в нижней части схемы. Команда `addl` требует значения регистра в цикле 7, но оно не генерируется командой `irmovl` до цикла 8. Для "передвижения" из `irmovl` в `addl` потребовалось бы вовремя вернуть значение! Поскольку очевидно, что это невозможно, необходимо найти какой-то другой механизм для обработки такой формы рисков по данным.

```
# prog5
0x000: irmovl $128,%edx
0x006: irmovl $3,%ecx
0x00c: rmovl %ecx, 0(%edx)
0x012: irmovl $10,%ebx
0x018: irmovl 0(%edx),%eax # Load %eax
0x01e: addl %ebx,%eax # Use %eax
0x020: halt
```

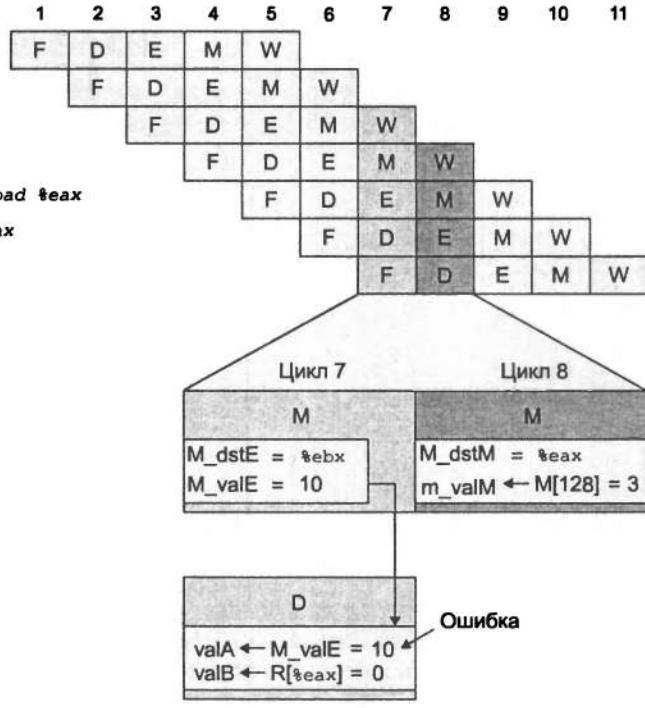


Рис. 4.49. Пример риска по данным load/use

Обратите внимание, что значение для регистра `%ebx`, сгенерированное командой `irmovl` в адресе 0x00c, можно передвинуть из этапа памяти в команду `addl` на этап ее декодирования в цикле 7.

Как показано на рис. 4.50, риска по данным load/use можно избежать использованием комбинации остановов и передвижения. По мере прохождения командой `irmovl` эта- па выполнения управляющая логика конвейера выявляет, что команда на этапе декодирования (`addl`) требует результата, считанного из памяти. Логика приостанавливает

ет выполнение команды на этапе декодирования для одного цикла, что вызывает вставку кружка в этап выполнения. Как показано на расширенном представлении цикла 8, значение, считываемое из памяти, может быть впоследствии перемещено с этапа памяти в команду addl на этапе декодирования. Значение для регистра %edx также направляется с этапа обратной связи на этап памяти. Как показано на конвейерной диаграмме стрелкой от рамки, помеченной литерой D в цикле 7 к рамке, помеченной литерой E в цикле 8, вставленный кружок замещает команду addl, которая бы в противном случае продолжала прохождение по конвейеру.

```
# prog5
0x000: irmovl $128,%edx
0x006: irmovl $3,%ecx
0x00c: rmovl %ecx, 0(%edx)
0x012: irmovl $10,%ebx
0x018: rmovl 0(%edx),%eax # Load %eax
      bubble
0x01e: addl %ebx,%eax # Use %eax
0x020: halt
```

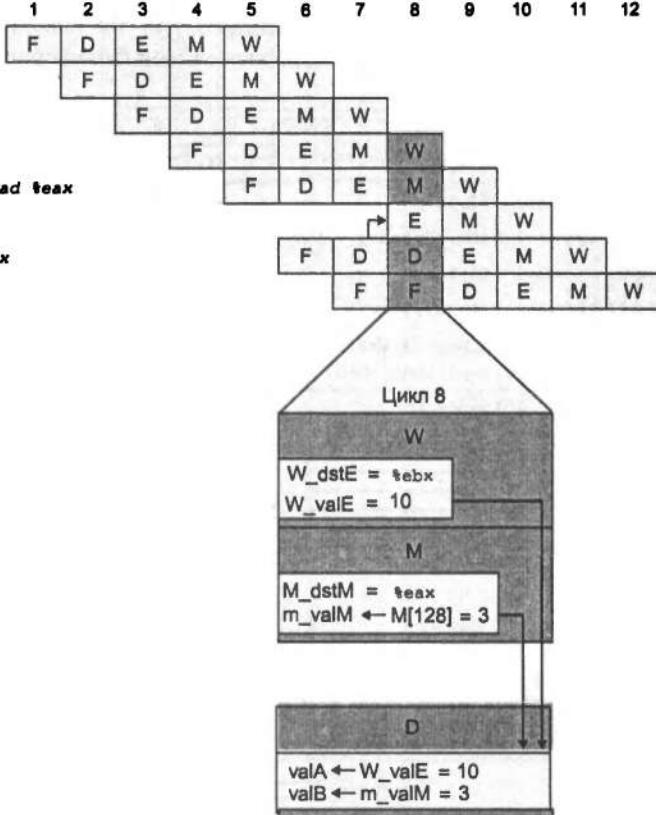


Рис. 4.50. Обработка риска путем останова

Подобное использование остановов для управления риска load/use называется **блокировкой загрузки**. Блокировок загрузки в комбинации с передвижением достаточно для обработки всех возможных форм рисков по данным. Поскольку пропускную способность конвейера снижают только блокировки загрузки, к достижению цели получения необходимой пропускной способности можно приблизиться выдачей одной новой команды на каждом цикле синхронизации.

4.5.8. Реализации этапов PIPE

Теперь мы имеем созданную общую структуру PIPE конвейерного процессора Y86 с передвижением. Здесь применяется тот же набор аппаратных устройств, что и в последовательных проектах, с добавлением конвейерных регистров, некоторых реконфигурированных логических блоков и дополнительной управляющей логики конвейера. В данном разделе рассматривается проектирование различных логических блоков; проектирование управляющей логики перенесено в следующий раздел. Многие логические блоки имеют аналоги в SEQ и SEQ+, за исключением того, что необходимо выбирать нужные версии различных сигналов из конвейерных регистров (записанные с названием конвейерного регистра, прописными буквами в виде префикса) или из вычисления этапов (записанные с первым символом названия этапа, строчными буквами в виде префикса).

Для примера сравним код HCL для логики, генерирующей в SEQ сигнал scrA, с соответствующим кодом в PIPE (листинг 4.13).

Листинг 4.13. Код сигнала

```
# Код из SEQ
int scrA = [
icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL } : rA;
icode in { IPOPL, IRET } : RESP;
1 : RNONE; # Регистра не требуют
];

# Код из PIPE
int new_E_scrA = [
D_icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL } : D_rA;
D_icode in { IPOPL, IRET } : RESP;
1 : RNONE; # Регистра не требуют
];
```

Они различаются только префиксом `D_`, добавленным к сигналам PIPE для указания того, что они поступают из конвейерного регистра `D`. Во избежание повторений, код HCL для блоков здесь не приводится, потому что он отличается от кодов SEQ только префиксами к названиям. Впрочем, для ссылки полный код HCL для PIPE приведен в *приложении 1*.

Выбор PC и этап выборки

На рис. 4.51 показано подробное представление логики этапа выборки PIPE. Как уже обсуждалось выше, этот этап также должен выбрать текущее значение для счетчика команд и спрогнозировать следующее значение PC. Аппаратные устройства для считывания команды из памяти и для извлечения различных полей команды — те же, что и для SEQ (см. описание этапа выборки в разд. 4.3.4).

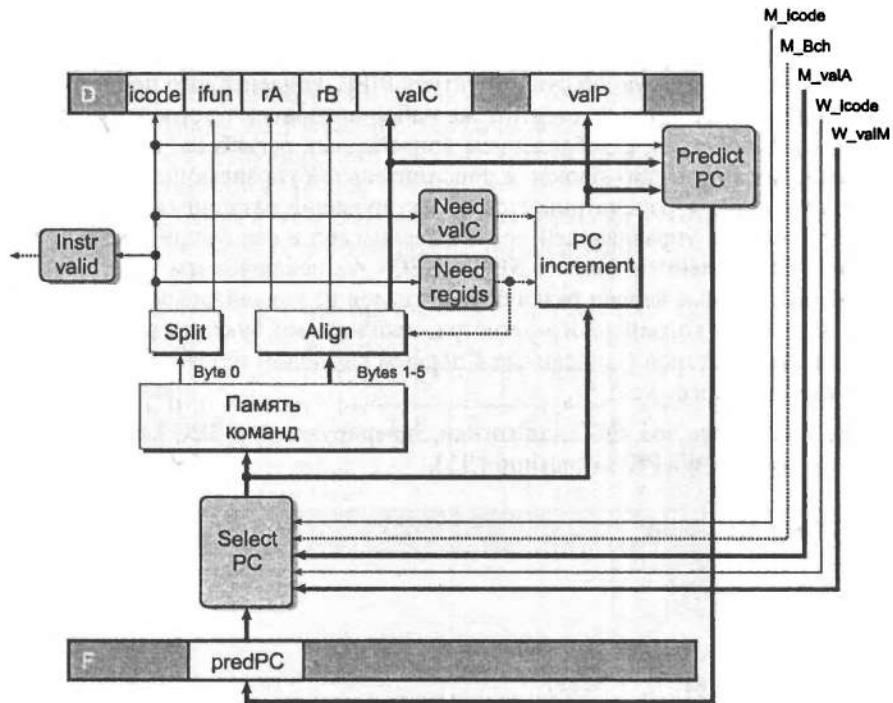


Рис. 4.51. Выбор счетчика команд PIPE и логика выборки

Логика выбора PC делает выбор из трех источников счетчика команд. Как только неправильно спрогнозированная ветвь входит в этап памяти, значение valP для данной команды (указывающее адрес следующей команды) считывается из конвейерного регистра M (сигнал M_valA). Когда команда ret входит на этап обратной записи (листинг 4.14), возвратный адрес считывается из конвейерного регистра W (сигнал W_valM). Все другие случаи используют спрогнозированное значение счетчика команд, сохраненное в конвейерном регистре F (сигнал F_predPC).

Листинг 4.14. Прогноз значения счетчика

```
int f_pc = [
# Неправильно спрогнозированная ветвь. Выборка при приращенном PC
M_icode == IJXX && !M_Bch : M_valA;
# Завершение команды RET
W_icode == IRET : W_valM;
# По умолчанию: Использовать спрогнозированное значение PC
1 : F_predPC;
];
```

Логика выбора PC выбирает valC для выбранной команды, когда она либо вызов, либо переход, и valP — в противном случае:

```
int new_F_predPC = [
f_icode in { IJXX, ICALL } : f_valC;
1 : f_valP;
```

Логические блоки, обозначенные как "Instr valid", "Need regids" и "Need valC", — те же, что и для SEQ, с именованными соответствующим образом исходными сигналами.

Этапы декодирования и обратной записи

На рис. 4.52 дано подробное представление логики декодирования и обратной записи PIPE. Блоки, обозначенные как "dstE", "dstM", "scrA" и "scrB", очень похожи в реализации SEQ. Обратите внимание на то, что идентификаторы регистра, переданные на порты записи, поступают с этапа обратной записи (сигналы W_{dstE} и W_{dstM}), а не с этапа декодирования. Это происходит потому, что запись должна осуществляться на выходные регистры, обозначенные командой на этапе обратной записи.

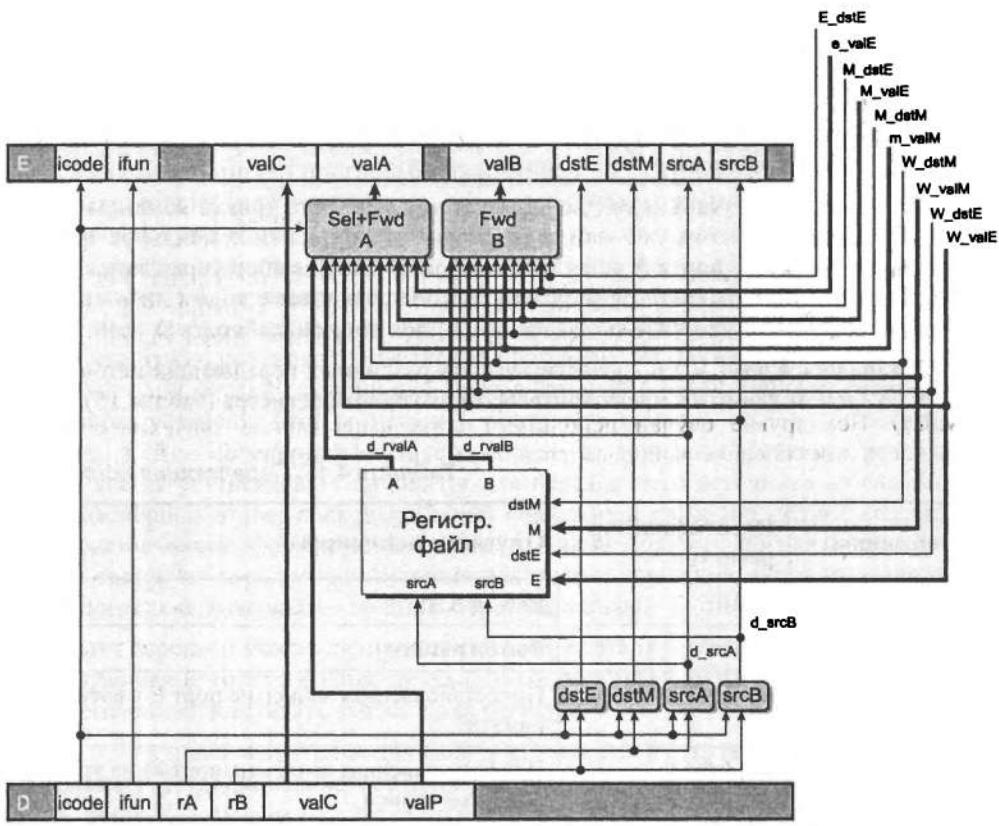


Рис. 4.52. Логика этапов декодирования и обратной связи PIPE

На рис. 4.52 ни одна из команд не требует valP и значения, считанного с порта регистра A, поэтому их можно объединить для формирования сигнала valA для последующих этапов. Блок, обозначенный как "Sel+Fwd A", выполняет эту задачу, а также реализует передающую логику для исходного операнда valA. Блок, обозначенный как "Fwd B", реализует передающую логику для исходного операнда valB. Местоположения записи в регистры обозначены сигналами dstA и dstB из этапа обратной связи, а не из этапа декодирования, поскольку записываются результаты команды, находящейся в данный момент на этапе обратной записи.

УПРАЖНЕНИЕ 4.23

Блок, обозначенный на этапе декодирования как "dstE", генерирует одноименный сигнал dstE на основании полей команды, выбранной в конвейерном регистре D. В HCL-описании PIPE результирующий сигнал называется new_E_dstE. Напишите код для этого сигнала, исходя из HCL-описания SEQ-сигнала dstE (см. описание этапа декодирования в разд. 4.3.4).

Большая часть всей сложности данного этапа связана с передающей (продвигающей) логикой. Как уже упоминалось, блок, обозначенный как "Sel_Fwd A", выполняет две роли. Он объединяет сигнал valP с сигналом valA последующих этапов для уменьшения объема состояния в конвейерном регистре. Он также реализует передающую логику исходного операнда valA.

Объединение сигналов valA и valP использует тот факт, что только команды call и перехода требуют значения valP на последующих этапах, и эти команды не требуют значения, считанного с порта A регистрационного файла. Такой выбор управляет сигналом icode для данного этапа. Когда сигнал D_icode совпадает с кодом либо команды call, либо JXX, тогда в качестве выхода данный блок должен выбирать D_valP.

Как упоминалось в разд. 4.5.6, существуют пять различных передающих источников, каждый со словом данных и идентификатором выходного регистра (табл. 4.15).

Таблица 4.15. Передающие источники

Слово данных	Идентификатор регистра	Описание источника
e_valE	E_dstE	Выход ALU
m_valM	M_dstM	Выход памяти
M_valE	M_dstE	Приостановленная запись на порт E на этапе памяти
W_valM	W_dstM	Приостановленная запись на порт M на этапе обратной записи
W_valE	W_dstE	Приостановленная запись на порт E на этапе обратной записи

Если ни одно из условий передвижения не выполняется, тогда блок должен выбрать `d_rvalA` — значение, считанное с регистрового порта А в виде выхода.

Объединив все полученное, имеем HCL-описание нового значения `valA` для конвейерного регистра Е в листинге 4.15.

Листинг 4.15. Новое значение конвейерного регистра

```
int new_E_valA = {
D_icode in { ICALL, IJXX } : D_valP; # Использовать приращенный РС
d_scrA == E_dstE:e_valE; # Передвинуть valE из этапа выполнения
d_scrA == M_dstM:m_valM; # Передвинуть valM из этапа памяти
d_scrA == M_dstE:M_valE; # Передвинуть valE из этапа памяти
d_scrA == W_dstM:W_valM; # Передвинуть valM из этапа обратной записи
d_scrA == W_dstE:W_valE; # Передвинуть valE из этапа обратной записи
    1:d_rvalA; # Использовать значение, считанное из регистрового файла
};
```

Приоритет, устанавливаемый для пяти передающих источников в HCL-коде листинга, очень важен. Этот приоритет определяется в HCL-коде порядком тестирования пяти идентификаторов выходного регистра. При выборе иного порядка, нежели указанный, для некоторых программ поведение конвейера будет некорректным. На рис. 4.53 показан пример программы, требующей корректной установки приоритета среди передающих источников на этапах выполнения и заломинания. В данной программе первые две команды записывают в регистр `%edx`, а третья использует этот регистр в качестве исходного операнда. Когда команда `imovl` достигает этапа декодирования в цикле 4, передающая логика должна выбрать одно из двух значений, предназначенных для ее исходного регистра. Какое же значение выбрать? Для установки приоритета нужно рассмотреть поведение программы на машинном языке, когда она выполняет одну команду за один раз. Первая команда `iimovl` установит регистр `%edx` в 10, вторая установит этот регистр в 3, после чего команда `imovl` считает из `%edx` значение 3. Для моделирования такого поведения данная конвейерная реализация должна всегда устанавливать приоритет для передающего источника на самом раннем конвейерном этапе, поскольку в нем содержится самая последняя команда из последовательности, формирующей регистр. Таким образом, логика листинга 4.15 сначала тестирует передающий источник на этапе выполнения, затем на этапе памяти, и в последнюю очередь — на этапе обратной записи.

Приоритет передачи между двумя источниками на этапах памяти или обратной связи имеет отношение только к команде `popl %esp`, поскольку только эта команда может одновременно осуществлять запись в два регистра.

На рис. 4.53 в цикле 4 значения для `%edx` доступны как из этапа выполнения, так и из этапа памяти. Передающая логика должна выбрать одно из значений на этапе выполнения, поскольку последний представляет самое последнее сгенерированное значение для данного регистра.

```
# prog6
0x000: irmovl $10,%edx
0x006: irmovl $3,%edx
0x00c: rrmovl %edx,%eax
0x00e: halt
```

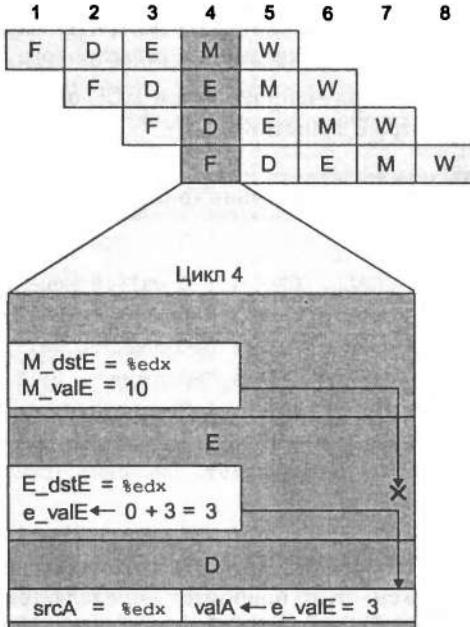


Рис. 4.53. Демонстрация приоритета передвижения

УПРАЖНЕНИЕ 4.24

Предположим, что порядок для третьего и четвертого случаев (два передающих источника из этапа памяти) в HCL-коде для new_E_valA поменялись местами. Опишите результирующее поведение команды rrmovl (строка 5) для следующей программы:

```
1 irmovl $5, %edx
2 irmovl $0x100, %esp
3 rmmovl %edx, 0 (%esp)
4 popl %esp
5 rrmovl %esp, %eax
```

УПРАЖНЕНИЕ 4.25

Предположим, что порядок для пятого и шестого случаев (два передающих источника из этапа обратной записи) в HCL-коде для new_E_valA поменялись местами. Напишите программу Y86, которая бы выполнялась некорректно. Опишите, как произойдет ошибка, и ее влияние на поведение программы.

УПРАЖНЕНИЕ 4.26

Напишите HCL-код для сигнала new_E_valB с предоставлением значения для исходного операнда valB, переданного в конвейерный регистр E.

Этап выполнения

На рис. 4.54 показана логика этапа выполнения для PIPE. Аппаратные устройства и логические блоки — те же, что и для SEQ, с соответствующим переименованием сигналов. Видно, что сигналы e_valE и E_dstE направлены на этап декодирования.

Данная часть проекта очень похожа на логику в реализации SEQ.

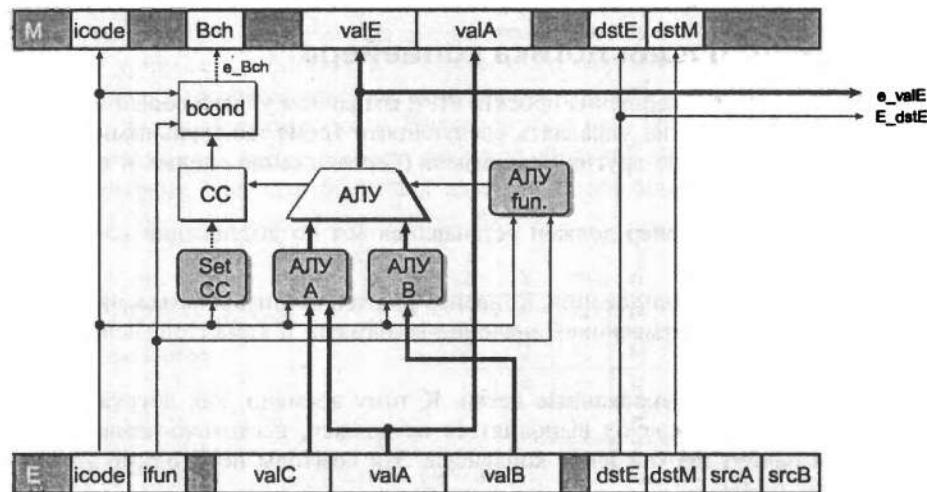


Рис. 4.54. Логика этапа выполнения PIPE

Этап памяти

На рис. 4.55 показана логика этапа памяти для PIPE. Если сравнивать ее с этапом памяти для SEQ (см. рис. 4.21), можно заметить, что блок, обозначенный "Data" в

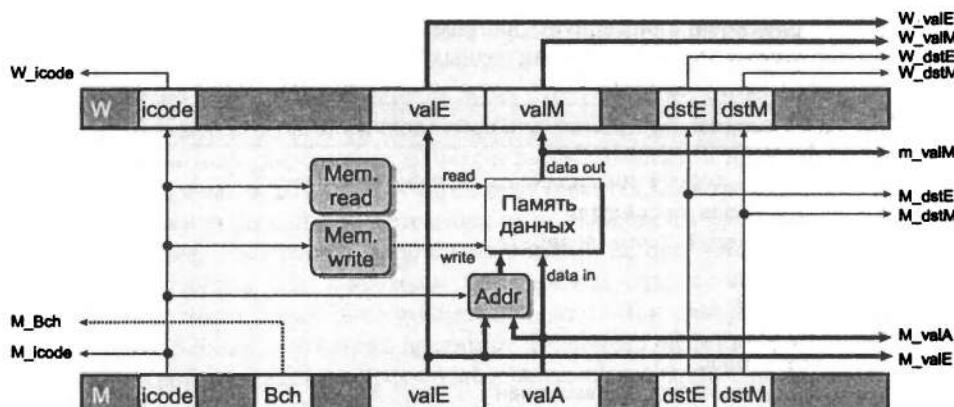


Рис. 4.55. Логика этапа памяти PIPE

SEQ, отсутствует в PIPE. Этот блок служил для выбора значений valP источников данных (для команд call) и valA, однако на этапе декодирования этот выбор не осуществляется блоком, обозначенным "Sel+Fwd A". Все остальные блоки на данном этапе, как и в SEQ, с соответствующим переименованием сигналов. На данной схеме также видно, что многие из значений в конвейерных регистрах, а также M и W, передаются в другие части цепи как часть передающей логики и управляющей логики конвейера.

4.5.9. Управляющая логика конвейера

Теперь все готово для завершения проекта PIPE созданием управляющей логики конвейера. Эта логика должна управлять следующими тремя контрольными случаями, для которых недостаточно других механизмов (передвижение данных и прогнозирование ветвей):

- Обработка ret. Конвейер должен останавливаться по достижении командой ret этапа обратной связи.
- Риски загрузки и использования. Конвейер должен останавливаться на один цикл между командой,читывающей значение из памяти, и командой, использующей это значение.
- Некорректно спрогнозированные ветви. К тому времени, как логика ветвления обнаруживает, что переход выполняться не должен, несколько команд на цели ветвления начнут прохождение конвейера. Эти команды необходимо удалить из конвейера.

Сначала будут выполнены все необходимые действия для отдельного случая, после чего разработана управляющая логика для каждого из них.

Многие из сигналов из конвейерных регистров M и W (рис. 4.55) передаются на более ранние этапы для получения результатов обратной записи, адресов команд и передвигнутых результатов.

Желательная обработка особых контрольных случаев

Рассмотрим следующую примерную программу для команды ret. Эта программа показана в листинге 4.16, но с адресами разных команд для ссылок в левой части.

Листинг 4.16. Примерная программа для команды возврата

```

0x000:irmovl Stack, %esp # Инициализация указателя стека
0x006:call proc# Вызов процедуры
0x00b:irmovl $10, %edx# Точка возврата
0x011:halt
0x020: .pos 0x20
0x020: proc:
0x020:     ret# proc
0x021: rrmovl %edx, %ebx# Не выполнено
0x030: .pos 0x30
0x030: Stack# Стек:Указатель стека

```

На рис. 4.56 показан желательный способ обработки команды `ret`. Как и на конвейерных диаграммах, на схеме показана конвейерная деятельность со значением времени, возрастающим в правую сторону. Команды перечисляются не в том порядке, в каком они появляются в программе, поскольку в программе задействована управляющая логика, где команды выполняются не в линейной последовательности. Обратите внимание на адреса команды, чтобы видеть, в каком месте различные команды входят в программу.

На схеме показано, что команда `ret` выбирается во время цикла 3 и начинает прохождение конвейера, достигая в 7 цикле этапа обратной записи. При прохождении командой этапов декодирования, выполнения и памяти конвейер не может выполнять никакой полезной деятельности. Вместо этого в него необходимо вставить три кружка. Как только команда `ret` достигнет этапа обратной записи, логика выбора РС установит счетчик команд в обратный адрес и, таким образом, этап выборки выберет команду `irmovl` в точке возврата адрес 0x00b).

```
#prog7
0x000: irmovl Stack,%edx
0x006: call proc
0x020: ret
    bubble
    bubble
    bubble
0x00b: irmovl $10,%edx # Return point
```

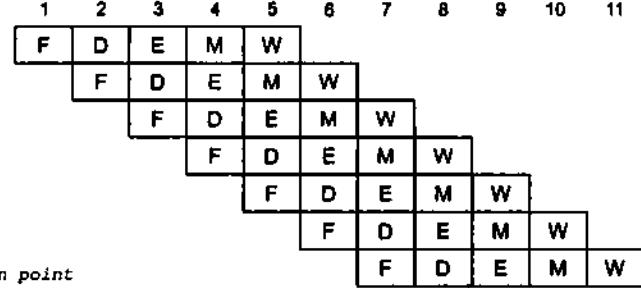


Рис. 4.56. Упрощенное представление обработки команды возврата

На рис. 4.57 показана фактическая обработка команды `ret` для кода в листинге 4.16. Основным наблюдением здесь является то, что на этапе выборки данного конвейера нет возможности вставить кружок. На каждом цикле этап выборки считывает из памяти команд определенную команду. Если рассмотреть HCL-код реализации логики прогнозирования РС, представленный в разд. 4.5.8, то можно увидеть, что для команды `ret` новым прогнозируемым значением РС будет `valP` — адрес следующей команды. В данной программе значением будет 0x021 — адрес команды `rrmovl`, следующей за `ret`. Для данного примера такой прогноз некорректен, как и не будет он корректным для большинства случаев, однако в рассматриваемом проекте задача корректного прогнозирования возвратных адресов не ставится. Для трех циклов синхронизации этап выборки приостанавливается; при этом выбирается команда `rrmovl`, но замещается кружком. Этот процесс проиллюстрирован на рис. 4.62 в виде трех выборок со стрелкой, ведущей вниз к кружкам, проходящим через оставшиеся этапы конвейера. Наконец, в цикле 7 выбирается команда `irmovl`. При сравнении рис. 4.57 с рис. 4.56 видно, что данная реализация достигает желаемого эффекта, но с особой выборкой некорректной команды для трех последовательных циклов.

Что касается риска загрузки и использования, то желаемая конвейерная операция уже описывалась в разд. 4.5.7 и проиллюстрирована примером на рис. 4.50. Данные из

памяти считывают только команды `irmovl` и `popl`. Когда любая из этих команд находится на этапе выполнения, а команда, требующая выходного регистра, находится на этапе декодирования, тогда на этом этапе необходимо удержать вторую команду и вставить кружок в этап выполнения на следующем цикле. После этого проводящая логика устранит риск по данным. Конвейер может удерживать команду на этапе декодирования путем удержания конвейерного регистра D в фиксированном состоянии. При этом конвейерный регистр F также должен находиться в фиксированном состоянии, так что следующая команда будет выбрана повторно. Вообще говоря, реализация такого конвейерного потока требует выявления условия риска, поддерживания конвейерных регистров F и D в фиксированном состоянии и вставки кружка на этапе выполнения.

```
# prog7
0x000: irmovl Stack, %edx
0x006: call proc
0x020: ret
0x021: rrmovl %edx, %ebx # Not executed
        bubble
0x021: rrmovl %edx, %ebx # Not executed
        bubble
0x021: rrmovl %edx, %ebx # Not executed
        bubble
0x00b: irmovl $10, %edx # Return point
```

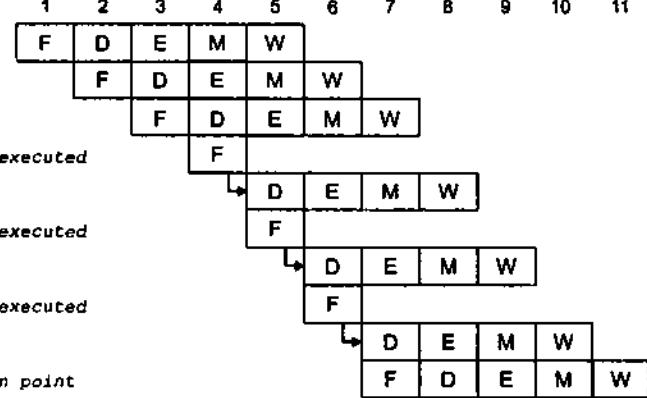


Рис. 4.57. Фактическая обработка команды возврата

Этап выборки многократно выбирает команду `rrmovl`, следующую за командой `ret`, однако затем управляющая логика конвейера вставляет кружок в этап декодирования вместо допуска начала выполнения команды `rrmovl`. Результатирующее поведение эквивалентно представленному на рис. 4.56.

Конвейер прогнозирует выбор ветвей (рис. 4.58) и начинает выбирать команды на цели перехода. Две команды выбираются до того, как некорректное прогнозирование будет выявлено в цикле 4, когда команда перехода проходит этап выполнения. В цикле 5 конвейер отменяет две целевые команды вставкой кружков в этапы выполнения и декодирования и выбирает команду, следующую за переходом.

Для обработки некорректно спрогнозированной ветви рассмотрим следующую программу, представленную в листинге 4.17, с адресами команд показанных для ссылки в левой части.

Листинг 4.17. Некорректно спрогнозированная ветвь

```
0x00:xorl %eax, %eax
0x02:jne target# Не выбирается
```

```

0x007:irmovl $1, %eax# Передача управления вниз
0x00d:halt
0x00e: target:
0x00e:irmovl $2, %edx# Цель
0x014:irmovl $3, %ebx# Цель +1
0x01a:halt

```

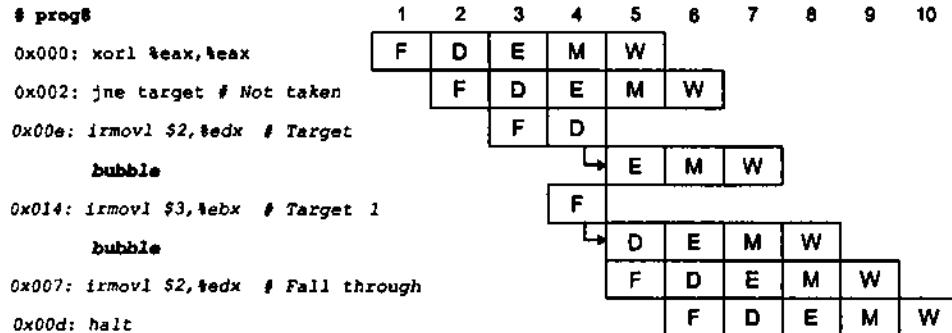


Рис. 4.58. Обработка некорректно спрогнозированных команд ветвления

На рис. 4.58 представлена обработка этих команд. Как и раньше, команды перечислены в порядке их поступления на конвейер, а не в порядке, в котором они появляются в программе. Поскольку команда перехода прогнозируется как предпринятая, команда в цели перехода выбирается в цикле 3, а следующая за ней команда выбирается в цикле 4. К тому времени, как логика ветвления выявит, что переход не должен выполняться во время цикла 4, выбираются две команды, выполнение которых уже должно быть остановлено. К счастью, ни одна из этих команд не вызывает изменений в состоянии, видимом для программиста. Это происходит только по достижении командой этапа выполнения, где может быть вызвано изменение кодов условий. Две неправильно выбранные команды можно просто отменить (иногда этот процесс называется *сплющиванием команды*) путем вставки кружков в команды декодирования и выполнения в следующем цикле с одновременной выборкой команды, следующей за командой перехода. Тогда две неправильно выбранные команды просто исчезнут из конвейера.

Выявление особых условий управления

В табл. 4.16 подводится итог условиям, требующим особого управления конвейером. Здесь представлены HCL-выражения, описывающие условия, при которых возникают три особых случая. Эти выражения реализуются простыми блоками комбинаторной логики, которые должны генерировать свои результаты до конца цикла синхронизации для того, чтобы управлять действиями конвейерных регистров по мере возрастания синхронизации для начала следующего цикла. Во время цикла синхронизации конвейерные регистры D, E и M удерживают состояния команд, находящихся на конвейерных этапах декодирования, выполнения и памяти соответственно. По мере

приближения к концу цикла синхронизации сигналы d_srcA и d_srcB будут настроены на регистровые идентификаторы исходных operandов для команд на этапе декодирования. Выявление команды `ret` по мере прохождения ею конвейера заключается в проверке кодов команд на этапах декодирования, выполнения и памяти. Выявление рисков `load/use` заключается в проверке типа команды (`memovl` или `popl`) на этапе выполнения и сравнения ее выходного регистра с исходными регистрами команды на этапе декодирования. Управляющая логика конвейера должна выявлять некорректно спрогнозированную ветвь во время нахождения команды перехода на этапе выполнения для задания условий, необходимых для восстановления прогноза по мере вхождения команды на этап памяти. При появлении на этапе выполнения команды перехода сигнал `e_Bch` указывает на то, предпринимать переход или нет.

Таблица 4.16. Выявление условий для управляющей логики конвейера

Условие	Триггер
Обработка <code>ret</code>	$IRET \in \{D_icode, E_icode, M_icode\}$
Риски <code>load/use</code>	$E_icode \in \{IMRMOVL, IPOPL\} \&& E_dstM \in \{d_srcA, d_srcB\}$
Некорректно спрогнозированная ветвь	$E_icode = IJXX \&& ! e_Bch$

Три различных условия требуют изменения конвейерного потока либо с помощью останова конвейера, либо путем отмены частично выполненных команд.

Механизмы управления конвейером

На рис. 4.59 показаны механизмы нижнего уровня, позволяющие управляющей логике конвейера удерживать команду в конвейерном регистре или вставить в конвейер кружок. В этих механизмах задействованы маленькие расширения базового синхронизированного регистра, описанного в разд. 4.2.5. Предположим, что каждый конвейерный регистр имеет два управляющих входа: останов и кружок. Настройки этих сигналов определяют то, как обновляется конвейерный регистр при возрастании синхронизации. При нормальном функционировании оба этих входа установлены в 0, что вызывает регистр загружать входные данные при его новом состоянии. Когда сигнал останова (обозначен "Stall") установлен в 1, обновление состояния отключено. Регистр остается в предыдущем состоянии. Это делает возможным удержание команды на некотором конвейерном этапе. Когда кружок ("Bubble") установлен в 1, состояние регистра будет иметь **конфигурацию восстановления**, обеспечивая состояние, эквивалентное состоянию команды `por`. Особая комбинация единиц и нулей для конфигурации восстановления конвейерного регистра зависит от набора полей в конвейерном регистре. Например, для вставки кружка в конвейерный регистр D необходимо, чтобы поле `icode` имело постоянную величину `TNOR` (см. табл. 4.13). Для вставки кружка в конвейерный регистр E поле `icode` должно быть `TNOR`, а поля `dstE`, `dstM`, `srcA` и `srcB` — константой `RNONE`. Определение конфигурации восстановления — это одна из задач для проектировщика аппаратных средств, которую следует решить при

проектировании конвейерного регистра. Подробно данная тема здесь не рассматривается; установка обоих сигналов в 1 будет истолковываться как ошибка.

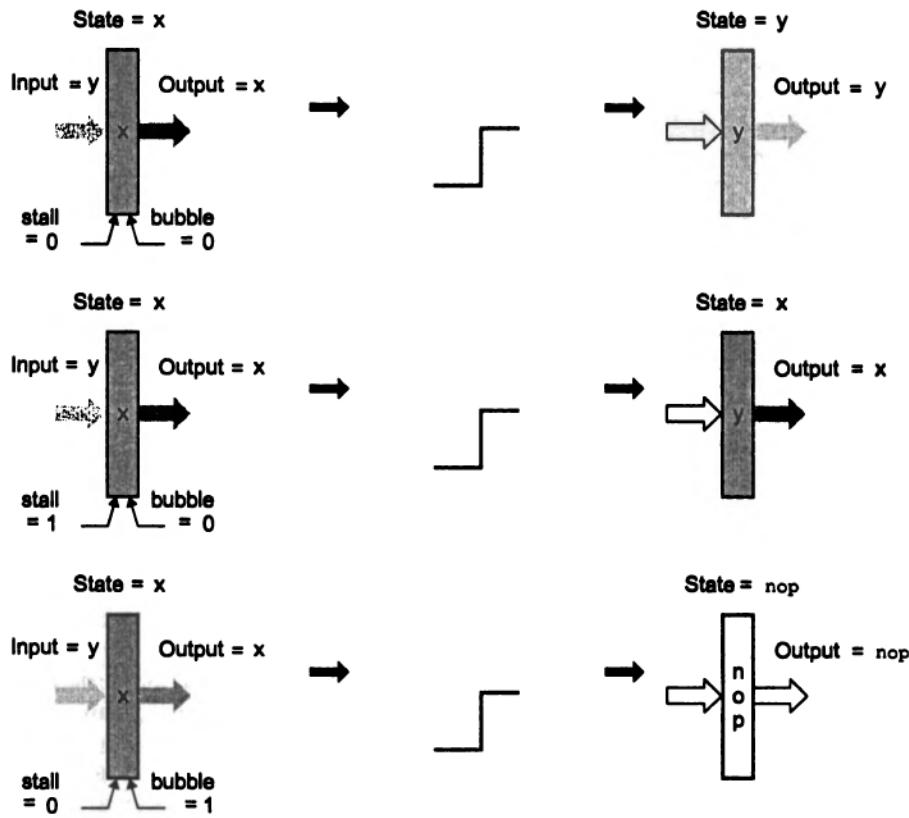


Рис. 4.59. Дополнительные операции конвейерного регистра

На рис. 4.59 при нормальных условиях (А) состояние и выход регистра установлены на значение на входе при возрастании синхронизации. При работе в режиме останова (В) состояние остается фиксированным на предыдущем значении. При работе в режиме кружка (С) состояние замещается (перезаписывается) состоянием операции *nop*.

Табл. 4.17 показывает действия, которые должны выполнять различные конвейерные этапы для каждого из трех особых условий. Каждое включает в себя некоторую комбинацию операций при нормальных условиях, при условии останова и кружка для конвейерных регистров. Разные условия требуют изменения конвейерного потока либо остановом конвейера, либо отменой частично выполненных команд.

В том, что касается синхронизации, управляющие сигналы *stall* и *bubble* для конвейерных регистров генерируются блоками комбинаторной логики. Эти значения долж-

ны быть действительными по мере возрастания синхронизации, заставляя каждый конвейерный регистр загружаться, останавливаться или вводить кружок с началом каждого нового цикла. С подобными мелкими расширениями проектов конвейерных регистров можно реализовать полный конвейер, включая все его управление, с использованием базовых блоков комбинаторной логики, синхронизированных регистров и оперативной памяти.

Таблица 4.17. Действия управляющей логики конвейера

Условие	Конвейерный регистр				
	F	D	E	M	W
Обработка ret	Останов	Кружок	Норм. условия	Норм. условия	Норм. условия
Риски load/use	Останов	Останов	Кружок	Норм. условия	Норм. условия
Некорректно спрогнозированная ветвь	Норм. условия	Кружок	Кружок	Норм. условия	Норм. условия

Комбинации управляющих условий

До сих пор при обсуждении особых условий управления конвейером предполагалось, что во время прохождения любого одного цикла синхронизации может возникнуть только один особый случай. Общей ошибкой при проектировании системы является невозможность управлять случаями, когда множественные особые условия возникают одновременно. Проанализируем некоторые из них. На рис. 4.60 в виде схемы показаны конвейерные состояния, вызывающие особые управляющие условия. На этих диаграммах показаны блоки этапов декодирования, выполнения и памяти. Заштрихованные рамки обозначают те или иные ограничения, которые должны быть соблюдены для возникновения условия. Риск load/use требует, чтобы команда на этапе выполнения считывала значение из памяти в регистр, и чтобы команда на этапе декодирования имела этот регистр в качестве исходного операнда. Некорректно спрогнозированная ветвь требует, чтобы команда на этапе выполнения имела команду перехода. Существует три возможных случая для ret: данная команда может находиться на этапе декодирования, выполнения или памяти. По мере прохождения командой ret конвейера, более ранние его этапы будут иметь кружки.

На указанных диаграммах видно, что большинство управляющих условий исключают друг друга. Например, невозможно одновременно иметь риск load/use и некорректно спрогнозированную ветвь, поскольку первый требует команды загрузки (`l1fmovl` или `popl`) на этапе выполнения, а вторая требует перехода. Подобным же образом, вторая и третья комбинации `ret` не могут происходить одновременно с риском load/use или с некорректно спрогнозированной ветвью. Одновременно могут возникать только две комбинации, указанные стрелками.

В комбинацию А входит невыбранная команда перехода на этапе выполнения и команда `ret` на этапе декодирования. Настройка этой комбинации требует, чтобы команда `ret` была в цели невыбранной ветви. Управляющая логика конвейера должна определить, что ветвь была выполнена, и, следовательно, отменить команду `ret`.

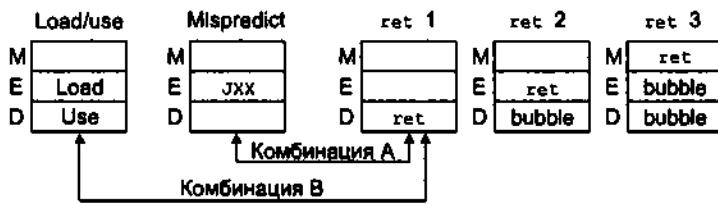


Рис. 4.60. Конвейерные состояния для особых управляющих условий

УПРАЖНЕНИЕ 4.27

Напишите программу Y86 на ассемблерном языке, вызывающую возникновение комбинации А и определяющую правильность ее обработки управляющей логикой.

При объединении управляющих действий для условий комбинации А получаем следующие управляющие действия для конвейера, предположив, что нормальный случай подменяется кружком или остановом (табл. 4.18).

Таблица 4.18. Условия комбинации А

Условие	Конвейерный регистр				
	F	D	E	M	W
Обработка <code>ret</code>	Останов	Кружок	Норм. условия	Норм. условия	Норм. условия
Некорректно спрогнозированная ветвь	Норм. условия	Кружок	Кружок	Норм. условия	Норм. условия
Комбинация	Останов	Кружок	Кружок	Норм. условия	Норм. условия

То есть, данная комбинация будет обрабатываться как некорректно спрогнозированная ветвь, но с остановом на этапе выборки. К счастью, в следующем цикле логика выбора РС выберет адрес команды, следующей за переходом, а не спрогнозированный счетчик команд, поэтому не имеет значения, что произойдет с конвейерным регистром F. Отсюда делается вывод, что конвейер обработает данную комбинацию корректно.

В комбинацию В входит риск `load/use`, где команда загрузки устанавливает регистр `%esp`, после чего команда `ret` использует этот регистр в качестве исходного операнда,

поскольку она должна вытолкнуть обратный адрес из стека. Управляющая логика конвейера должна удержать команду `ret` на этапе декодирования.

УПРАЖНЕНИЕ 4.28

Напишите программу Y86 на ассемблерном языке, вызывающую возникновение комбинации В и завершающуюся с командой `halt`, если конвейер работает корректно.

При объединении управляющих действий для условий комбинации В получаем следующие управляющие действия для конвейера (табл. 4.19).

Таблица 4.19. Условия комбинации В

Условие	Конвейерный регистр				
	F	D	E	M	W
Обработка <code>ret</code>	Останов	Кружок	Норм. условия	Норм. условия	Норм. условия
Риски <code>load/use</code>	Останов	Останов	Кружок	Норм. условия	Норм. условия
Комбинация	Останов	Кружок+ останов	Кружок	Норм. условия	Норм. условия
Исправлено	Останов	Останов	Кружок	Норм. условия	Норм. условия

При срабатывании обоих наборов действий управляющая логика попытается приостановить команду `ret`, во избежание риска `load/use`, и вставить в этап декодирования кружок, из-за команды `ret`. Понятно, что проектировщику не нужно, чтобы конвейер выполнял оба этих набора действий. Вместо этого необходимо, чтобы выполнялись только действия для риска `load/use`. Действия по обработке команды `ret` должны быть отложены на один цикл.

Данный анализ показывает, что комбинация В требует особой обработки. Фактически, первоначальная реализация управляющей логики PIPE обрабатывала эту комбинацию некорректно. Несмотря на то, что проект прошел множество имитационных проверок, в нем обнаружилась совсем небольшая ошибка, выявляемая только с помощью описанного анализа. При выполнении программы с комбинацией В управляющая логика устанавливала бы сигналы `bubble` и `stall` для конвейерного регистра D в 1. Этот пример доказывает важность систематического анализа. Указанная ошибка вряд ли была обнаружена простым запуском обычных программ. Если бы она не была исправлена, то конвейер бы неверно реализовывал соответствия поведению ISA.

Реализация управляющей логики

На рис. 4.61 показана полная структура управляющей логики конвейера. На основании сигналов из конвейерных регистров и конвейерных этапов управляющая логика генерирует управляющие сигналы *stall* и *bubble* для конвейерных регистров. Можно объединить условия табл. 4.16 с действиями, представленными в табл. 4.17, для создания HCL-описаний различных управляющих конвейером сигналов.

Регистр F должен быть остановлен либо для риска загрузки и использования, либо для команды *ret*:

```
bool F_stall =
    # Условия для риска загрузки/использования
    E_icode in { IMRMOVL, IPOPL } &&
        E_dstM in { d_srcA, d_srcB } ||
    # Останов на этапе выборки во время прохождения командой ret конвейера
    IRET in { D_icode, E_icode, M_icode };
```

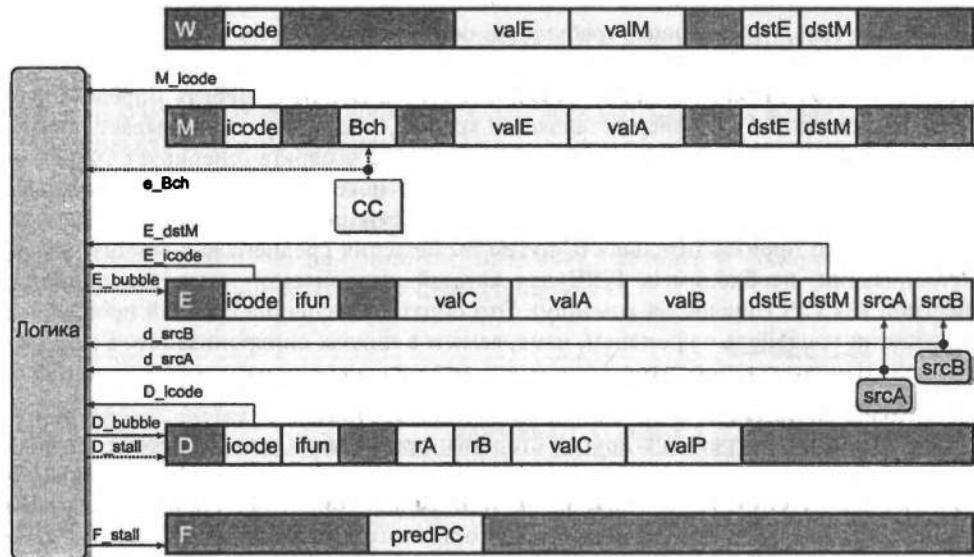


Рис. 4.61. Управляющая логика конвейера PIPE

Данная логика замещает обычный проход команд по конвейеру для обработки особых условий, таких как возвраты процедур, некорректно спрогнозированные ветви и риски load/use.

УПРАЖНЕНИЕ 4.29

Напишите HCL-код для сигнала *D_stall* в реализации PIPE.

Конвейерный регистр D должен быть установлен на кружок для некорректно спрогнозированной ветви или команды *ret*. Однако, как показывает анализ в предыдущем

разделе, регистр не должен вставлять кружок при наличии риска load/use в комбинации с командой ret:

```
bool D_bubble =
# Некорректно спрогнозированная ветвь
(E_icode == IJX && !e_Bch) || 
# Останов на этапе выборки во время прохождения командой ret конвейера
IRET in { D_icode, E_icode, M_icode };
```

УПРАЖНЕНИЕ 4.30

Напишите HCL-код для сигнала E_bubble в реализации PIPE.

Сюда входят все значения особых управляющих сигналов конвейера. В полном HCL-коде для PIPE все прочие управляющие сигналы установлены в 0.

4.5.10. Анализ эффективности

Можно отметить, что условия, требующие особых действий со стороны управляющей логики конвейера, не дают ему достигнуть цели выдачи новой команды на каждом цикле синхронизации. Такую неэффективность можно измерить определением частоты, с которой в конвейер вставляется кружок, поскольку это вызывает появление неиспользованных конвейерных циклов. Команда возврата генерирует три кружка, риск загрузки и использования — один, а некорректно спрогнозированная ветвь — два. Можно выразить в количественном отношении воздействие этих проблем на общую производительность путем вычисления среднего количества циклов синхронизации, необходимых PIPE для каждой выполняемой команды, единицы, известной как CPI (циклов на команду). Это обратная величина средней пропускной способности конвейера с временем, измеряемым в циклах синхронизации, а не пикосекундах. Данная единица измерения очень полезна при оценке архитектурной производительности проекта.

На CPI можно посмотреть и с другой стороны: представить, что процессор запущен на некоей эталонной тестовой программе, и обратить внимание на функционирование этапа выполнения. На каждом цикле этапа выполнения будет обрабатывать либо команду, которая продолжит прохождения оставшихся этапов до завершения, либо кружок, вставленный вследствие возникновения одного из трех особых случаев. Если этап обрабатывает общее количество C_s команд и C_b — кружков, тогда процессору потребуется порядка $C_s + C_b$ полных циклов синхронизации для выполнения C_s команд. Здесь употреблено слово "порядка", потому что игнорируются циклы, необходимые для запуска команд, проходящих через конвейер. CPI для данной тестовой программы можно вычислить следующим образом.

То есть, CPI равно 1.0 плюс штрафной элемент C_b/C_s , указывающий на среднее количество кружков, вставленных на одну выполненную команду. Поскольку вставку кружка могут вызвать только три разных типа команд, этот дополнительный элемент можно разбить на три компонента:

$$CPI = 1.0 + lp + mp + rp,$$

где:

- lp (load penalty, загрузить штраф) — средняя частота, с которой вставляются кружки во время останова из-за рисков загрузки/использования;
- mp (mispredicted branch penalty, штраф за некорректно спрогнозированную ветвь) — средняя частота, с которой вставляются кружки, при отмене команды из-за некорректно спрогнозированных ветвей;
- rp (return penalty, возврат штрафа) — средняя частота, с которой вставляются кружки, при останове команд `ret`.

Каждый из этих штрафов указывает общее количество кружков, вставленных по объявленной причине (некоторая часть C_b), разделенное на общее число выполненных команд (C_t).

Для оценки каждого из этих штрафов необходимо знать частоту происхождения соответствующих команд (загрузку, условную ветвь и возврат) и для каждой команды частоту возникновения особенного условия. Для расчетов CPI выберем следующий набор частот (они сравнимы с измерениями, приведенными в [31] и [33]):

- На команды загрузки (`lftmovl` и `popl`) приходится 25% всех выполняемых команд. 20% из них вызывают риски загрузки/использования.
- На долю условных ветвей приходится 20% всех выполняемых команд. 60% из них выбираются и 40% не выбираются.
- На команды возврата приходится 2% всех исполняемых команд.

Таким образом, каждый из штрафов можно рассматривать как произведение частоты типа команды, частоты, при которой возникает условие, и количества кружков, вставляемых при возникновении условия (табл. 4.20):

Таблица 4.20. Штрафы

Причина	Название	Частота команды	Частота условия	Кружки	Произведение
Загрузка/использование	lp	0.25	0.20	1	0.05
Некорректный прогноз	mp	0.20	0.40	2	0.16
Возврат	rp	0.02	1.00	3	0.06
Общий штраф					0.27

Сумма трех штрафов составляет 0.27, что дает CPI величину 1.27.

Нашей целью было создание проекта конвейера, который бы мог выдавать одну команду на цикл с результатирующим CPI 1.0. Данная цель достигнута не в полной мере, однако общая производительность по-прежнему остается достаточно высокой. Также понятно, что любая попытка в еще большей степени уменьшить CPI должна сосредоточиться на некорректно спрогнозированных ветвях. Они составляют 0.16 от общего штрафа в 0.27, потому что условные ветви распространены очень широко,

стратегия прогнозов часто себя не оправдывает, и для каждой ошибки прогнозов отменяются две команды.

УПРАЖНЕНИЕ 4.31

Предположим, что используется стратегия прогнозирования ветви "обратный выбор без прямого выбора" (BTFNT), достигающая доли успешных попыток в 65% (см. разд. 4.5.3). Каково будет влияние на CPI, если предположить, что на все другие частоты влияния не оказывается?

4.5.11. Незаконченная работа

Теперь создана структура для конвейерного микропроцессора PIPE, спроектированы блоки управляющей логики и реализована управляющая логика процессора для обработки особых случаев, где обычного конвейерного потока недостаточно. Однако PIPE по-прежнему недостает нескольких ключевых особенностей, которые будут необходимы при реальном проектировании микропроцессора. Некоторые из них рассматриваются далее вместе с тем, что необходимо для их добавления.

Обработка исключительных ситуаций

Когда программа машинного уровня сталкивается с состоянием ошибки, например, с недействительным кодом команды, с командой или адресом данных, находящимися вне адресного пространства, тогда в процессе выполнения программы возникает сбой, называемый *исключительной ситуацией*. Поведение последней напоминает вызов процедур, активизирующей *обработчик исключительных ситуаций*, — процедуру, входящую в операционную систему. Более подробно обработка исключительных ситуаций рассматривается в главе 8. Выполнение команды `halt` также должно запускать исключительную ситуацию. Обработка исключительных ситуаций является частью архитектуры системы команд для процессора. Вообще говоря, возникновение исключительной ситуации должно останавливать процессор в точке непосредственно перед командой ее вызывающей, или сразу за ней, в зависимости от типа исключительной ситуации. Очевидно, что все команды до точки возникновения исключительной ситуации должны быть выполнены, но ни одна из команд, следующих сразу за этой точкой, не должна никак влиять на состояние, видимое для программиста.

В конвейерной системе обработка исключительных ситуаций имеет определенные тонкости. Во-первых, исключительные ситуации могут одновременно запускаться многоадресными командами. Например, в ходе одного цикла конвейерного функционирования можно организовать такой порядок:

- память для хранения команд будет сообщать на этапе выборки адрес команды, выходящий за установленные рамки;
- память для хранения данных будет сообщать на этапе памяти адрес данных, выходящий за установленные рамки;
- управляющая логика будет сообщать недействительный код для команды на этапе декодирования.

Теперь необходимо определить, о какой из этих исключительных ситуаций процессор должен сообщить операционной системе. Основным правилом является установка приоритета самой дальней в конвейере исключительной ситуации, активизированной командой. В приведенном примере такой ситуацией будет адрес, выходящий за пределы адресного пространства, на который попытается выйти команда на этапе памяти. Что касается программы машинного уровня, команда на этапе памяти, по идее, должна выполниться до запуска команд на этапах декодирования или выборки, и, следовательно, только об этой исключительной ситуации должно быть сообщено операционной системе.

Вторая особенность возникает, когда команда выбирается, начинает выполняться, вызывает исключительную ситуацию и впоследствии отменяется из-за некорректно спрогнозированной ветви. В листинге 4.18 показан пример такой программы в форме объектного кода:

Листинг 4.18. Исключительная ситуация

```
0x000: 6300|xorl %eax, %eax
0x002: 740e000000|jne Target# Не выбирается
0x007: 308001000000|imovl $1, %eax# Передача управления вниз
0x00d: 10|halt
0x00e: |    Target:
0x00e: ff|.byte 0xFF# Недействительный код команды
```

В данной программе конвейер прогнозирует то, что ветвь выбираться не должна, поэтому он выберет и попытается использовать байт со значением 0xFF в качестве команды (генерированный в компонующем автокоде с использованием директивы .byte). Поэтому этап декодирования выявит исключительную ситуацию недействительной команды. Позже конвейер обнаружит, что ветвь отклоняется, поэтому команда в адресе 0x00e не должна даже выбираться. Управляющая логика конвейера отменит эту команду, однако в задачу входит необходимость избежать возникновения исключительной ситуации.

Третья тонкость возникает из-за того, что конвейерный процессор обновляет различные части состояния системы на разных этапах. Команда, следующая за командой, вызывающей исключительную ситуацию, может изменить определенную часть состояния до завершения исключающей команды. Например, рассмотрим следующую кодовую последовательность (листинг 4.19), в которой предполагается, что пользовательские программы не получают доступ к адресам, превышающим 0xc0000000 (как в случае для текущих версий Linux, см. главу 10).

Листинг 4.19. Ограничение по адресам

```
1 imovl $0, %esp# Установка указателя стека на 0
2 pushl %eax# Попытка записи в 0xfffffffffc
3 addl %esp, %eax# Установка кодов условий
```

Команда `pushl` вызывает исключительную ситуацию, потому что уменьшение значения указателя стека заключает его в оболочку `0xffffffffc`. Эта исключительная ситуация обнаруживается на этапе памяти. В том же цикле команда `addl` находится на этапе выполнения и вызывает установку новых значений кодов условий. Этим нарушается требование о том, чтобы ни одна команда, следующая за точкой возникновения исключительной ситуации, не оказывала никакого влияния на состояние системы.

Вообще говоря, можно корректно выбирать различные исключительные ситуации и избегать последних для команд из-за некорректно спрогнозированных ветвей, путем внедрения логики обработки исключительных ситуаций в структуру конвейера. В каждый конвейерный регистр добавляется специальное поле `exc`, обеспечивая команде на этом этапе исключительное состояние. Если команда генерирует исключительную ситуацию на некотором этапе обработки, тогда задается поле состояния для указания на природу исключительной ситуации. Исключительное состояние передается по конвейеру с остальной информацией для данной команды до достижения им этапа обратной записи. В этой точке управляющая логика конвейера выявляет возникновение исключительной ситуации и инициирует выборку кода для обработчика исключительных ситуаций.

Во избежание обновления видимого для программиста состояния командами, следующими за точкой исключительной ситуации, необходимо модифицировать управляющую логику конвейера для блокирования какого бы то ни было обновления регистра кода условия или памяти для хранения данных, когда команда на этапе памяти или обратной записи вызывает исключительную ситуацию. В листинге 4.18 управляющая логика обнаруживает, что команда `pushl` на этапе памяти вызвала исключительную ситуацию, и, следовательно, обновление регистра кода условия командой `addl` будет блокировано. (В имитаторе для PIPE будут показаны реализации методик обработки исключительных ситуаций в конвейерном процессоре.)

Рассмотрим, как данный метод обработки исключительных ситуаций применяется к описанным тонкостям. При возникновении исключительной ситуации на одном или нескольких этапах конвейера информация просто сохраняется в полях исключительного состояния конвейерных регистров. Это событие не оказывает никакого влияния на поток команд в конвейере до тех пор, пока исключающая команда не достигает последнего этапа конвейера, за исключением блокирования любых обновлений видимого для программиста состояния (регистра кода условия или памяти) более поздними командами в конвейере. Поскольку команды достигают этапа обратной записи в том же порядке, в каком они выполнялись бы в процессоре без конвейера, то можно гарантированно предположить, что первая команда, столкнувшаяся с исключительной ситуацией, будет первой командой, запускающей передачу управления обработчику исключительных ситуаций. Если какая-то команда выбирается, но позже отменяется, то любая информация об исключительном состоянии этой команды также отменяется. Ни одна команда, следующая за командой, вызывающей исключительную ситуацию, не может изменить состояние, видимое программисту. Простое правило переноса исключительного состояния вместе со всей информацией о команде по конвейеру формирует простой и надежный механизм обработки исключительных ситуаций.

Многократные команды

Все команды в системе команд Y86 включают в себя такие простые операции, как, например, добавление чисел. Они могут обрабатываться в рамках одного цикла синхронизации на этапе выполнения. В более полной системе команд будет необходима реализация команд, требующих более сложных операций, например умножения и деления целых чисел, а также операций с плавающей точкой. В процессоре со средней производительностью, таком как PIPE, обычное время выполнения этих операций составляет 3–4 цикла для сложения чисел с плавающей точкой, до 32 циклов для деления целых чисел. Для реализации этих команд требуются как дополнительные аппаратные средства для выполнения вычислений, так и механизм координации обработки этих команд оставшимися компонентами конвейера.

Одним из простых подходов к реализации многократных команд является расширение возможностей логики этапа выполнения для таких арифметических единиц, как целые числа и числа с плавающей точкой. Команда остается на этапе выполнения столько циклов синхронизации, сколько ей потребуется, что вызовет останов этапов выборки и декодирования. Реализовать этот подход просто, однако конечная производительность оказывается не слишком высокой.

Более высокой производительности можно добиться обработкой сложных операций специальными аппаратными устройствами, работающими независимо от основного конвейера. Как правило, для выполнения умножения и деления целых чисел и выполнения операций с числами с плавающей точкой используется по одному функциональному устройству. Когда команда входит в этап декодирования, она может быть выдана на специальное устройство. Пока устройство выполняет операцию, конвейер продолжает обработку других команд. Обычно устройство для обработки операций с плавающей точкой само по себе является конвейерным, поэтому операции можно выполнять параллельно в основном конвейере и в разных устройствах.

Операции разных устройств должны быть синхронизированы, во избежание некорректного поведения. Например, при наличии зависимостей по данным между разными операциями, обрабатываемыми разными устройствами, управляющей логике может потребоваться приостановить одну часть системы до полного завершения операции, обрабатываемой какой-либо другой частью системы. Часто используются различные формы передвижения для передачи результатов из одной части системы в другие, как происходило между различными этапами PIPE. Общее проектирование становится более сложным, нежели с PIPE, однако для того, чтобы сделать поведение соответствующим последовательной ISA-модели, можно использовать те же методики остановки, передвижения и управления конвейером.

Сопряжение с системой памяти

В представлении PIPE предполагалось, что и устройство выборки команды, и память для хранения данных могут считывать любую ячейку памяти или осуществлять в нее запись за один цикл синхронизации. При этом игнорировались возможные риски, вызываемые самоизменяющимся кодом, когда одна команда осуществляет запись в область памяти, из которой выбираются последующие команды. Более того, обраще-

ние к ячейкам памяти осуществляется в соответствии с их виртуальными адресами, требующими преобразования в физические адреса до фактического выполнения операций считывания или записи. Понятно, что осуществить всю указанную обработку за один цикл невозможно. Более того, значения памяти, к которым осуществляется доступ, могут храниться на диске, что требует миллионов циклов синхронизации для считывания в память процессора.

В главах 6 и 10 будет обсуждаться вопрос о том, что в системе памяти процессора для управления системой виртуальной памяти используется комбинация блоков аппаратной памяти и программных средств операционной системы. Система памяти организована в виде иерархии быстродействующих, но менее крупных блоков памяти, содержащих подмножество памяти, поддерживаемое менее быстродействующими и более крупными блоками памяти. На ближайшем к процессору уровне блоки кэш-памяти обеспечивают быстрый доступ к ячейкам памяти, на которые делается больше всего ссылок. В типичном процессоре имеются две кэш-памяти первого уровня: одна — для считывания команд, а другая — для считывания и записи данных. Другой тип кэш-памяти, известный как буфер быстрого преобразования адреса (TLB), обеспечивает быстрое преобразование виртуальных адресов в физические. С помощью комбинации TLB и блоков кэш-памяти можно большую часть времени считывать команды и считывать или записывать данные за один цикл синхронизации. Таким образом, упрощенное представление ссылок на ячейки памяти процессора является вполне приемлемым.

Несмотря на то, что в блоках кэш-памяти содержатся ячейки, на которые делается больше всего ссылок, случается, что возникает так называемый промах кэша, когда делается ссылка на ячейку, отсутствующую в нем. В лучшем случае отсутствующие данные можно получить из кэш более высокого уровня или из основной памяти процессора, что требует от 3 до 20 циклов синхронизации. На это время конвейер просто останавливается, удерживая команду на этапе выборки или памяти до тех пор, пока кэш не выполнит операцию считывания или записи. В том, что касается рассматриваемого конвейерного проекта, это может быть реализовано добавлением в управляющую логику конвейера большего количества условий останова. Промах кэша и последующая синхронизация с конвейером полностью управляется аппаратными средствами с привязкой необходимого времени к небольшому числу циклов синхронизации.

В некоторых случаях ячейка памяти, на которую делается ссылка, фактически сохраняется в памяти на диске. В этом случае аппаратные средства сигнализируют об исключительной ситуации ошибки страницы диска. Как и другие исключительные ситуации, она заставляет процессор активизировать код обработчика исключительных ситуаций операционной системы. Этот код запускает перенос с диска в основную память. По завершении этой операции операционная система вернется к первоначальной программе, где команда, вызывающая ошибку страницы диска, будет выполнена повторно. На этот раз ссылка на ячейку памяти будет успешной, хотя она и может вызвать промах кэша. Запуск аппаратными средствами рутинной процедуры операционной системы, которая затем возвращает управление аппаратным устройствам, позволяет аппаратным и программным средствам взаимодействовать при обра-

ботке ошибок страницы диска. Поскольку доступ к диску может потребовать миллионов циклов синхронизации, несколько тысяч циклов обработки, выполняемых обработчиком ошибки страниц диска операционной системы, не сильно влияют на производительность процессора.

С точки зрения процессора, комбинация остановов для обработки кратковременных промахов кэша и обработки исключительных ситуаций для устранения долговременных ошибок страниц диска заботится о непредсказуемости времени доступа к памяти из-за структуры ее иерархии.

Разработка современного процессора

Пятиэтапный конвейер, подобный рассмотренному процессору PIPE, представил проект современного процессора в середине 80-х годов. Процессор-прототип RISC, разработанный исследовательской группой Паттерсона (Patterson) в Беркли, заложил основу для процессора SPARC, разработанного компанией Sun Microsystems в 1987 г. Процессор, созданный группой исследователей Хеннеси (Hennessy) в Стэнфорде, был представлен на рынок MIPS Technologies (компания, основателем которой был Хеннеси) в 1986 г. И в той, и в другой моделях использовались пятиэтапные конвейеры. В процессоре Intel i486 также используется пятиэтапный конвейер, но с иным разделением ответственности между этапами; здесь имеется два этапа декодирования и комбинированный этап execute/store (выполнения, памяти).

Такие конвейерные проекты ограничены пропускной способностью максимум в одну команду на цикл синхронизации. Единица измерения CPI, описанная в разд. 4.5.10, никогда не может быть меньше 1.0. Разные этапы могут обрабатывать за раз только одну команду. Более поздние модели процессоров поддерживают суперскалярную операцию, т. е. они достигают CPI менее чем 1.0 путем выборки, декодирования и выполнения множественных команд параллельно. По мере того, как суперскалярные процессоры стали все более распространенными, приемлемое значение производительности сдвинулось от CPI к его противоположности: среднему числу команд, выполняемых за один цикл синхронизации, или IPC (команд на цикл). В суперскалярных процессорах IPC может превосходить 1.0. В самых передовых проектах используется методика, известная как *выполнение с изменением последовательности* для параллельного выполнения команд, возможно, в абсолютно ином порядке, нежели они появляются в программе с сохранением общего поведения, лежащего в основе последовательной модели ISA. Данная форма выполнения рассматривается в главе 5 как один из вопросов дискуссии об оптимизации программ.

Впрочем, конвейерные процессоры не являются просто историческими артефактами. Большинство продаваемых в мире процессоров используется во встроенных системах, управляющих функциями автомобилей, в бытовых товарах и в других устройствах, где сам процессор невидим для пользователя. В таких агрегатах простота конвейерного процессора, рассмотренного в данной главе, снижает его стоимость и потребляемую мощность, по сравнению с процессорами более высокой производительности.

4.6. Резюме

Из материала данной главы становится понятно, что архитектура системы команд (ISA) обеспечивает универсальность, что касается набора команд и их кодировок, и методы реализации процессора. ISA дает наглядное представление выполнения программы, когда одна команда полностью завершается до начала следующей.

Описание системы команд Y86 началось с команд IA32 и значительного упрощения типов данных, адресных режимов и кодировок команд. Полученная в результате ISA относится как к системе команд RISC, так и CISC. Затем обработка, необходимая для различных команд, была организована в серию из шести этапов, где операции на каждом из них варьировались, в зависимости от выполняемой команды. Из этого был создан процессор SEQ, в котором каждый тактовый цикл состоит из разбиения на шесть этапов и прохождения командой каждого из них. Путем изменения порядка этапов получен проект SEQ+, в котором первый этап определяет значение счетчика программ, используемое для вызова этой команды.

Конвейерный режим обработки повышает производительность системы, когда разные этапы выполняются параллельно. В любой отдельно взятый момент времени ступени конвейера обрабатывают разные операции. Необходимо обеспечить для пользователя эмуляцию последовательного выполнения программы. Затем введено понятие конвейера путем добавления конвейерных регистров в SEQ+ и перегруппировки циклов для создания конвейера PIPE-. Производительность конвейера заметно повышается добавлением логики передвижения, ускоряющей отправку результата от одной команды к другой. Несколько особых случаев требуют дополнительной управляющей логики конвейера для приостановки или отмены некоторых конвейерных этапов.

В данной главе пройдено несколько уроков относительно устройства процессора:

- Управление сложностью является высшим приоритетом. Основной задачей является достижение оптимального использования ресурсов аппаратных средств, для получения максимальной производительности с минимальными затратами. Задача решена созданием очень простой и универсальной структуры обработки всех существующих типов команд. С помощью этой структуры аппаратные устройства можно разделять в рамках логики для обработки различных типов команд.
- Необходимости в прямой реализации ISA нет. Прямая реализация ISA обозначала бы исключительно последовательный проект. Для достижения более высокой производительности необходимо использовать способность аппаратных средств к одновременному выполнению большого количества операций. Это привело к применению конвейерного проектирования. Путем тщательной разработки и анализа можно избегать различных конвейерных рисков так, что общий эффект от запуска программы в точности будет совпадать с тем, что можно получить с помощью модели ISA.
- Разработчики аппаратных средств должны быть очень требовательными. Если микросхема уже изготовлена, то исправить какие бы то ни было ошибки практи-

чески невозможно. Очень важно, чтобы процесс проектирования был безошибочным с самого начала. Это подразумевает подробнейший анализ различных типов команд и комбинаций, даже тех, которые, на первый взгляд, бессмысленны (например, выталкивание на указатель стека). Все проекты необходимо тщательно тестировать с помощью моделей тестирования систем. При разработке управляющей логики для PIPE в рассмотренном проекте была допущена небольшая ошибка, обнаруженная только после тщательного и систематического анализа управляющих комбинаций.

4.6.1. Имитаторы Y86

В материалы лабораторных работ по данной главе включают имитаторы для процессоров SEQ, SEQ+ и PIPE. Каждый имитатор представлен в двух версиях:

- Версия GUI (графический пользовательский интерфейс) отображает память, программный код и состояние процессора в графических окнах. При этом очень удобно наблюдать за потоком команд в процессоре. Панель управления позволяет интерактивно перезагружать имитатор, запускать его или действовать по шагам. Эти версии требуют языка подготовки сценариев Tcl и графической библиотеки Tk.
- Текстовая версия запускает тот же имитатор, но информация отображается только при распечатке на терминал. Данная версия не представляет особой ценности при отладке, однако она позволяет осуществлять автоматическое тестирование процессора, и ее можно запускать на системе, не поддерживающей Tcl/Tk.

Управляющая логика для имитаторов генерируется преобразованием HCL-объявленных логических блоков в код C. Затем этот код компилируется и связывается с оставшимся кодом моделирования. Такая комбинация позволяет тестировать варианты первоначальных проектов, используя имитаторы. Также доступны сценарии проверок, в ходе которых полностью тестируются различные команды и различные возможности рисков.

Библиографические примечания

Для интересующихся подробностями логического проектирования стандартным вводным материалом является учебник по логическому проектированию Каца (Katz) [39], в котором сделан особый упор на использование языков описания аппаратных средств.

В учебнике Хеннеси и Паттерсона, посвященном компьютерной архитектуре [33], подробно изложены вопросы проектирования процессора, включая как простые конвейеры, наподобие описанного здесь, так и более совершенные процессоры, выполняющие параллельно большее количество команд. Шривер (Shriver) и Смит (Smith) [69] дают очень подробное описание совместимого с Intel процессора IA32, изготовленного AMD.

Задачи для домашнего решения

УПРАЖНЕНИЕ 4.32 +

В примере программы Y86, например, функции `Sum`, показанной в листинге 4.3, встречается много ситуаций (например, строки 12 и 13, а также 14 и 15), когда в регистр нужно добавить постоянную величину. Это требует первого использования команды `imovl` для установки регистра на константу, затем команды `addl` для добавления этого значения в выходной регистр. Предположим, что нужно добавить новую команду `iaddl` со следующим форматом:

байт	0	1	2	3	4	5
<code>iaddl V, rB</code>	<code>C</code>	<code>0</code>	<code>B</code>	<code>rB</code>		<code>V</code>

Эта команда добавляет постоянную величину `V` в регистр `rB`. Опишите вычисления, выполненные для реализации этой команды. В качестве руководства используйте вычисления для `imovl` и `ORI` (см. табл. 4.4).

УПРАЖНЕНИЕ 4.33 +

Как описано в разд. 3.7.2, для подготовки стека к возврату можно использовать команду IA32 `leave`. Это эквивалентно следующей кодовой последовательности Y86:

<code>1 imovl %ebp %esp</code>	Установка указателя стека на начало фрейма
<code>2 popl %ebp</code>	Восстановление <code>%ebp</code> и установка стека <code>ptr</code> в конец

Предположим, что данная команда добавлена в систему команд Y86, используя следующую кодировку (рис. 4.62):

байт	0	1	2	3	4	5
<code>leave</code>	<code>D</code>	<code>0</code>				

Рис. 4.62. Кодировка команды

Опишите вычисления, выполненные для реализации данной команды. В качестве руководства используйте вычисления для `popl` (см. табл. 4.6).

УПРАЖНЕНИЕ 4.34 ++

Файл `seq_full.hcl` содержит HCL-описание для `SEQ` вместе с объявлением константы `IIADDL`, имеющей шестнадцатеричное значение `C` — код команды для `iaddl`. Измените HCL-описания блоков управляющей логики для реализации команды `iaddl`, как описано в упр. 4.32.

УПРАЖНЕНИЕ 4.35 ++

Файл `seq_full.hcl` также содержит объявление константы `ILEAVE`, имеющей шестнадцатеричное значение `D`, код команды для `leave`, а также объявление константы `REBP`,

имеющей значение 7 — идентификатор регистра для `%ebp`. Измените HCL-описания блоков управляющей логики для реализации команды `leave`, как описано в упр. 4.33.

УПРАЖНЕНИЕ 4.36 ***

Предположим, что необходимо создать недорогой конвейерный процессор на основе структуры, разработанной для PIPE—(см. рис. 4.34 и 4.36) без параллельных ветвей. Этот проект будет обрабатывать все зависимости по данным посредством останова до тех пор, пока команда, генерирующая необходимое значение, не пройдет этап обратной записи.

Файл `pipe_stall.hcl` содержит модифицированную версию кода HCL для PIPE, в котором обходная логика отключена. То есть сигналы `e_valA` и `e_valB` просто объявлены следующим образом:

```
## НЕ ИЗМЕНЯТЬ СЛЕДУЮЩИЙ КОД.
## Без продвижения. valA — либо valP, либо значение из регистрового файла

int new_E_valA = [
D_icode in { ICALL, IJXX } : D_valP; # Использовать приращенный PC
1 : d_rvalA; # Использовать значение, считанное из регистрового файла
];

## Без продвижения. valB — значение из регистрового файла
int new_E_valB = D_rvalB;
```

Модифицируйте управляющую логику конвейера в конце этого файла так, чтобы она корректно устранила все возможные риски управления и риски по данным. В качестве части попытки проектирования необходимо проанализировать различные комбинации управляющих случаев, как было сделано при проектировании управляющей логики конвейера PIPE. Здесь может возникнуть множество различных комбинаций, поскольку очень много условий требуют останова конвейера. Убедитесь, что управляющая логика обрабатывает каждую комбинацию корректно.

УПРАЖНЕНИЕ 4.37 **

Файл `pipe_full.hcl` содержит копию описания HCL PIPE вместе с объявлением постоянной величины `IADDL`. Модифицируйте данный файл для реализации команды `iaddl`, как описано в упр. 4.32.

УПРАЖНЕНИЕ 4.38 ***

Файл `pipe_full.hcl` также содержит объявления постоянных величин `ILEAVE` и `REBP`. Модифицируйте данный файл для реализации команды `leave`, как описано в упр. 4.33. Указания по способу генерирования имитатора для решения и его тестирования см. в лабораторных материалах.

УПРАЖНЕНИЕ 4.39 ***

Файл `pipe_nt.hcl` содержит копию HCL-кода для PIPE плюс объявление постоянной величины `J_YES` со значением 0, код режима работы для безусловной команды пере-

хода. Измените логику ветви так, чтобы она прогнозировала условные переходы как невыбранные, во время прогнозирования безусловных переходов, и `call` — как выбранные. Возникнет необходимость изобрести способ получения `valC` — адрес цели перехода к конвейерному регистру `M` — для восстановления из некорректно спрогнозированных ветвей.

УПРАЖНЕНИЕ 4.40 ***

Файл `pipe_nt.hcl` содержит копию HCL-кода для PIPE плюс объявление постоянной величины `J_YES` со значением 0, код режима работы для безусловной команды перехода. Измените логику ветви так, чтобы она прогнозировала условные переходы как выбранные, когда `valC < valP` (обратная ветвь), и как невыбранные, когда `valC ≥ valP` (прямая ветвь). Продолжайте прогнозировать безусловные переходы и `call`, как выбранные. Возникнет необходимость изобрести способ получения `valC` и `valP` для конвейерного регистра `M` для восстановления из некорректно спрогнозированных ветвей.

УПРАЖНЕНИЕ 4.41 ***

В рассмотренном проекте PIPE останов генерируется всякий раз, когда одна команда выполняет загрузку, считывая значение из памяти в регистр, а следующая команда требует данный регистр в качестве исходного операнда. Когда источник используется на этапе выполнения, тогда останов — единственный способ избежать риска.

На случай, когда во второй команде сохраняется исходный operand для памяти, например, с командами `lpmovl` или `pushl`, тогда останов не обязателен. Рассмотрим следующие примеры кодов (листинг 4.20).

Листинг 4.20. Примеры загрузки

```

1 lpmovl 0(%ecx), %edx# Загрузка 1
2 pushl%edx# Сохранение 1
3 pop
4 popl %edx# Загрузка 2
5 lpmovl%eax, 0 (%edx) # Сохранение 2

```

В строках 1 и 2 команда `lpmovl` считывает значение из памяти в `%edx`, после чего команда `pushl` проталкивает это значение на стек. Рассмотренный проект PIPE остановит команду `pushl`, во избежание риска загрузки/использования. Впрочем, обратите внимание, что команда `pushl` не требует значения `%edx` до тех пор, пока она не достигнет этапа памяти. Можно добавить дополнительный обходной путь, как показано на рис. 4.63, для передвижения выхода памяти (сигнал `m_valM`) в поле `valA` в конвейерном регистре `M`. На следующем цикле синхронизации это передвинутое значение можно записывать в память. Данная методика называется *передвижением загружаемых данных* (или *передвижением загрузки*).

Обратите внимание, что во втором случае (строки 4 и 5 листинга 4.20) в кодовой последовательности нельзя использовать передвижение загрузки. Значение, загружен-

ное командой `popl`, используется как часть вычисления адреса следующей командой, и это значение необходимо, скорее, на этапе выполнения, а не на этапе памяти.

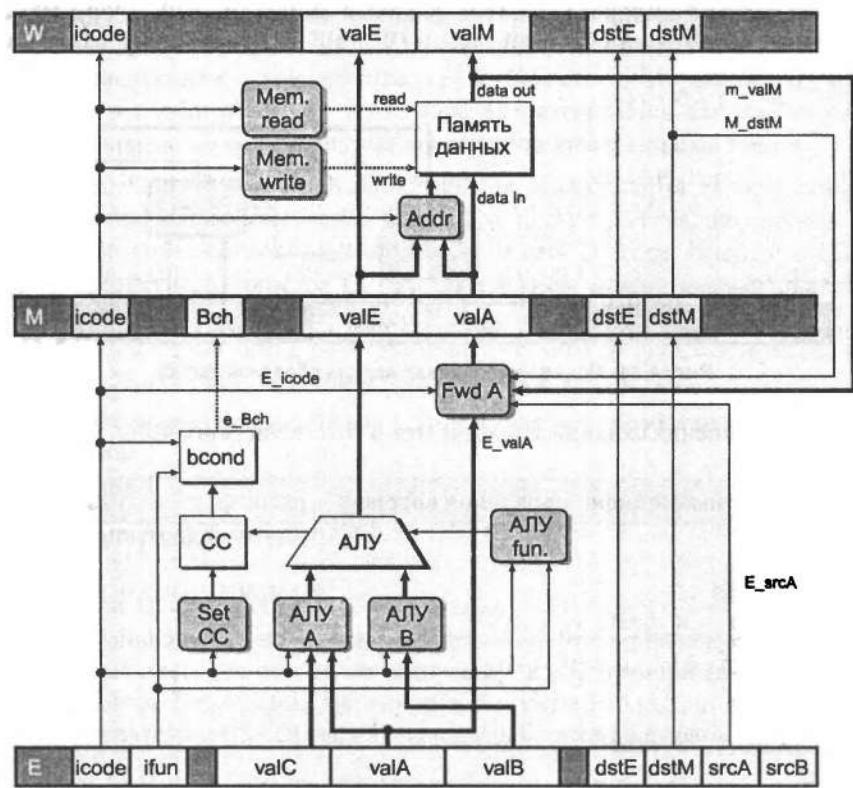


Рис. 4.63. Этапы выполнения и памяти передвижения загрузки

- Напишите формулу, описывающую условие выявления риска загрузки/использования, подобную формуле, представленной табл. 4.16, исключая то, что она не будет вызывать останов в случаях, когда можно использовать передвижение загрузки.
- Файл `pipe_1f.hcl` содержит модифицированную версию управляющей логики для PIPE. В нем содержится определение сигнала `new_M_valA` для реализации блока, обозначенного "Fwd A" и показанного на рис. 4.63. Условия риска `load/use` в управляющей логике конвейера, также содержащиеся в данном файле, установлены в 0, поэтому управляющая логика конвейера не обнаружит никаких форм рисков по загрузке/использованию. Модифицируйте данное HCL-описание для реализации продвижения загрузки.

УПРАЖНЕНИЕ 4.42 ◆◆◆

Представленный конвейерный проект в определенной степени нереалистичен в том, что для регистрового файла имеется два порта записи, однако только команда `popl`

требует двух одновременных записей в регистровый файл. Поэтому другие команды могут использовать один порт записи, разделяя его для записи valE и valM. На рис. 4.64 приведена модифицированная версия логики обратной записи, где происходит слияние идентификаторов регистров обратной записи (W_{dstE} и W_{dstM}) в один сигнал w_{dstE} , а значений обратной записи (W_{valE} и W_{valM}) — в единый сигнал w_{valE} .

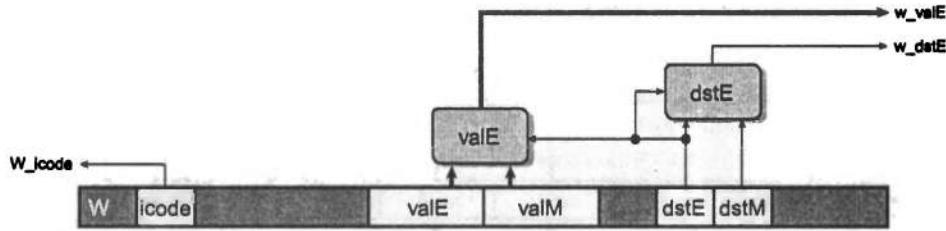


Рис. 4.64. Модифицированная версия обратной записи

Логика для выполнения слияний записывается в HCL в листинге 4.21:

Листинг 4.21. Логика модифицированной версии

```
int w_dstE = [
## запись из valM
W_dstM != RNONE:    W_dstM;
1 : W_dstE;
];
int w_valE = [
W_dstM != RNONE:    W_valM;
1 : W_valE;
];
```

Управление этими мультиплексорами определяется $dstE$; когда оно указывает на наличие определенного регистра, тогда выбирается значение для порта E, в противном случае выбирается значение для порта M.

В имитационной модели можно блокироватьпорт регистра M, как показано в следующем HCL-коде:

```
int w_dstM = RNONE;
int W_valM = 0;
```

Очередной сложностью является создание способа управления $pop1$. Один из них — использование управляющей логики для динамической обработки команды $pop1$ га так, чтобы она оказывала тот же эффект, что и последовательность из двух команд

```
iaddl $4, %esp
mrmlv -4 (%esp), rA
```

(Описание команды `iaddl` представлено в упр. 4.32.) Обратите внимание на упорядочение этих двух команд для того, чтобы `popl %esp` работали корректно. Это можно сделать с помощью управляющей логики, которая на этапе декодирования будет обрабатывать `popl` так же, как бы обрабатывалась приведенная выше команда `iaddl`, за исключением того, что следующий предсказанный счетчик команд будет идентичен текущему. В следующем цикле команда `popl` выбирается повторно, но код команды преобразуется в особое значение `IPOP2`. Оно рассматривается как особая команда, демонстрирующая то же поведение, что приведенная выше команда `irmovl`.

Файл `pipe-lw.hcl` содержит модифицированную логику порта записи, описанную выше. В нем содержится объявление константы `IPOP2` с шестнадцатеричным значением `E`. Здесь же содержится определение сигнала `new_D_icode`, генерирующего поле `icode` для конвейерного регистра `D`. Это определение можно модифицировать для вставки командного кода `IPOP2` при второй выборке команды `popl`. Файл `HCL` также содержит объявление сигнала `f_pc` — значение счетчика команд, сгенерированное на этапе выборки блоком, обозначенным "Select PC" (см. рис. 4.51).

Модифицируйте управляющую логику в этом файле для обработки команд `popl` описанным способом.

Решение упражнений

РЕШЕНИЕ УПРАЖНЕНИЯ 4.1

Кодирование команд вручную — занятие достаточно утомительное, однако это укрепляет понимание идеи, что ассемблер превращает компонующий автокод в последовательности байтов. В выходных данных из ассемблера `Y86` каждая строка указывает адрес и последовательность байтов, начинающуюся с этого адреса.

```
10x100:1 .pos 0x100# Начало генерирования кода в адресе 0x100
20x100:30830f0000000|irmovl $15, %ebx
30x106: 2031|rmovl %ebx, %ecx
40x108:1 loop
50x108: 4013fdfffffd|rmovl %ecx, -3 (%ebx)
60x10e: 6031|iaddl%ebx, %ecx
70x110: 700801000001jmp loop
```

Следует отметить несколько особенностей данной кодировки:

- Десятичное 15 (строка 2) имеет шестнадцатеричное представление `0x0000000f`. Запись этих байтов в обратном порядке дает `0f 00 00 00`.
- Десятичное 3 (строка 5) имеет шестнадцатеричное представление `0xfffffff7`. Запись этих байтов в обратном порядке дает `fd ff ff ff`.
- Код начинается в адресе `0x100`. Первая команда требует 6 байтов, вторая — 2. Следовательно, целью петли будет `0x00000108`. Запись этих байтов в обратном порядке дает `08 01 00 00`.

РЕШЕНИЕ УПРАЖНЕНИЯ 4.2

Декодирование последовательности байтов вручную помогает понять задачу, которую предстоит решить процессору. Он должен считать последовательность байтов и определить, какие команды должны быть выполнены. Далее представлен компонуемый код, используемый для генерирования каждой последовательности байтов. Слева от кода указан адрес и последовательность байтов для каждой команды.

Некоторые операции с перемещением непосредственно получаемых данных и адресов.

```
0x100: 3083fcfffff|irmovl $ -4, %ebx
0x106: 406300080000|rmovl %esi, 0x800 (%ebx)
0x10c: 10|halt
```

Код, включающий обращение к функции:

```
0x200: a068|pushl %esi
0x202: 8008020000|call proc
0x207: 10|halt
0x208: 1      proc:
0x208: 3083a0000000|irmovl $10, %ebx
0x20e: 90|ret
```

Код, содержащий байт спецификатора 0xf0 запрещенной команды:

```
0x300: 505407000000|rmovl 7 (%esp), %ebp
0x306: 00|nop
0x307: f0|.byte 0xf0 # код запрещенной команды
0x308: b018|      popl %ecx
```

Код, содержащий операцию перехода:

```
0x400: l      loop:
0x400: 6113|subl %ecx, %ebx
0x402: 7300040000|    je loop
0x407: 10|      halt
```

Код, содержащий в команде pushl недействительный байт:

```
0x500: 6362|      xorl %esi, %edx
0x502: a0|.byte 0xa0 # код команды pushl
0x503: 801     .byte 0x80 # недействительный байт регистра
```

РЕШЕНИЕ УПРАЖНЕНИЯ 4.3

Как предполагается в задаче, код, сгенерированный GCC, адаптирован для машины IA32:

```
# int Sum (int *Start, int Count)
rSumpushl %ebp
```

```
rrmovl %esp, %ebp
irmovl $20, %eax
subl %eax, %esp
pushl %ebx
mrnmovl 8 (%ebp), %ebx
mrnmovl 12 (%ebp), %eax
andl %eax, %eax
jle L38
irmovl $-8, %edx
addl %edx, %esp
irmovl $-1, %edx
addl %edx, %esp
pushl %eax
irmovl $4, %edx
rrmovl %ebx, %eax
addl %edx, %eax
pushl %eax
call rSum
mrnmovl (%ebx), %edx
addl %edx, %eax
jmp L39
L38:xorl %eax, %eax
L39:mrnmovl -24 (%ebp), %ebx
rrmovl %ebp, %esp
popl %ebp
ret
```

РЕШЕНИЕ УПРАЖНЕНИЯ 4.4

Несмотря на то, что для данной конкретной команды трудно представить практическое применение, при проектировании системы она важна для того, чтобы избежать двусмысленностей спецификации. Необходимо определить обоснованное соглашение для поведения команды и убедиться, что каждая реализация придерживается этого соглашения.

Команда `subl` в данном тесте сравнивает начальное значение `%esp` со значением, выталкиваемым в стек. Факт того, что результат вычитания равен нулю, подразумевает выталкивание старого значения `%esp`.

РЕШЕНИЕ УПРАЖНЕНИЯ 4.5

Еще сложнее представить необходимость выталкивания в указатель стека, хотя по-прежнему нужно определиться с соглашением и следовать ему. Эта кодовая последовательность проталкивает `tval` на стек, выталкивает в `%esp` и возвращает продвинутое значение. Поскольку результат равен `tval`, можно сделать вывод, что `popl %esp` должна устанавливать указатель стека на значение, считанное из памяти. Поэтому данная команда эквивалентна команде `mrnmovl 0 (%esp), %esp`.

РЕШЕНИЕ УПРАЖНЕНИЯ 4.6

Функция EXCLUSIVE-OR требует, чтобы два бита имели противоположные значения:

```
bool eq = (!a && b) || (a && !b);
```

Вообще говоря, сигналы *eq* и *xor* будут взаимно дополняющими, т. е. один будет равен единице, а другой — нулю.

РЕШЕНИЕ УПРАЖНЕНИЯ 4.7

Выходные данные цепей EXCLUSIVE-OR будут дополнениями значений равенства битов. С помощью законов Де Моргана (см. табл. 2.7) можно реализовать AND, используя OR и NOT, что дает следующую цепь (рис. 4.65):

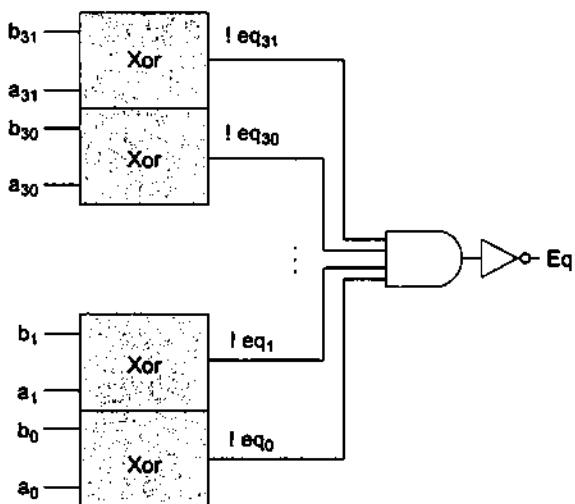


Рис. 4.65. Схема выходных данных

РЕШЕНИЕ УПРАЖНЕНИЯ 4.8

Данный проект является простым вариантом проекта для нахождения минимум трех входных значений:

```
int Med3 = [
A <= B && B <= C : B;
B <= A && A <= C : A;
: C;
];
```

РЕШЕНИЕ УПРАЖНЕНИЯ 4.9

Данные упражнения помогают конкретизировать вычисления этапов. Из объектного кода видно, что эта команда расположена в адресе 0x00e. Она состоит из шести бай-

тов, первые два из которых — 0x30 и 0x84. Последние четыре байта являются обратной байтовой версией 0x00000080 (десятичное 128).

Этап	Общий	Специфический
	irmovl V, rB	irmovl \$128, %esp
Выборка	$\text{icode: ifun} \leftarrow M_1[\text{PC}]$ $\text{rA: rB} \leftarrow M_1[\text{PC} + 1]$ $\text{valC} \leftarrow M_4[\text{PC} + 2]$ $\text{valP} \leftarrow \text{PC} + 6$	$\text{icode: ifun} \leftarrow M_1[0x00e] = 3:0$ $\text{rA: rB} \leftarrow M_1[0x00f] = 8:4$ $\text{valC} \leftarrow M_4[0x010] = 128$ $\text{valP} \leftarrow 0x00e + 6 = 0x014$
Декодирование		
Выполнение	$\text{valE} \leftarrow 0 + \text{valC}$	$\text{valE} \leftarrow 0 + 128 = 128$
Память		
Обратная запись	$R[rB] \leftarrow \text{valE}$	$R[%esp] \leftarrow \text{valE} = 128$
Обновление PC	$\text{PC} \leftarrow \text{valP}$	$\text{PC} \leftarrow \text{valP} = 0x014$

Эта команда устанавливает регистр %esp в 128 и увеличивает счетчик команд на 6.

РЕШЕНИЕ УПРАЖНЕНИЯ 4.10

Видно, что команда расположена в адресе 0x01c и состоит из двух байтов со значениями 0xb0 и 0x08. Регистр %esp был установлен в 124 командой pushl (строка 6), которая в этой ячейке памяти также сохраняла 9.

Этап	Основной	Специфический
	popl, rA	popl %eax
Выборка	$\text{icode: ifun} \leftarrow M_1[\text{PC}]$ $\text{rA: rB} \leftarrow M_1[\text{PC} + 1]$ $\text{valP} \leftarrow \text{PC} + 2$	$\text{icode: ifun} \leftarrow M_1[0x01c] = b:0$ $\text{rA: rB} \leftarrow M_1[0x01d] = 0:8$ $\text{valP} \leftarrow 0x01c + 2 = 0x01e$
Декодирование	$\text{valA} \leftarrow R[%esp]$ $\text{valB} \leftarrow R[%esp]$	$\text{valA} \leftarrow R[%esp] = 124$ $\text{valB} \leftarrow R[%esp] = 124$
Выполнение	$\text{valE} \leftarrow \text{valB} + 4$	$\text{valE} \leftarrow 124 + 4 = 128$
Память	$\text{valM} \leftarrow M_4[\text{valA}]$	$\text{valM} \leftarrow M_4[124] = 9$

(окончание)

Этап	Основной	Специфический
	popl, rA	popl %eax
Обратная запись	R[%esp] ← valE R[rA] ← valM	R[%esp] ← 128 R[%eax] ← 9
Обновление PC	PC ← valP	PC ← 0x01e

Эта команда устанавливает регистр %eax в 9, %esp — в 128 и увеличивает счетчик команд на 2.

РЕШЕНИЕ УПРАЖНЕНИЯ 4.11

Прослеживая шаги, показанные в табл. 4.6, где rA равно %esp, видно, что на этапе памяти команда будет сохранять в памяти valA — первоначальное значение указателя стека, так же, как и для IA32.

РЕШЕНИЕ УПРАЖНЕНИЯ 4.12

Прослеживая шаги, показанные в табл. 4.6, где rA равно %esp, видно, что обе операции обратной записи будут обновлять %esp. Поскольку операция, записывающая valM, происходит последней, чистым эффектом от выполнения команды будет запись значения, считанного из памяти в %esp, т. е. так же, как и для IA32.

РЕШЕНИЕ УПРАЖНЕНИЯ 4.13

Видно, что данная команда расположена в адресе 0x23 и состоит из 5 байтов. Первый байт имеет значение 0x80; последние четыре — обратная байтовая версия 0x00000029 — цель вызова. Команда popl (строка 7) установила указатель стека на 128.

Этап	Общий	Специфический
	call Dest	call 0x029
Выборка	icode: ifun ← M ₁ [PC] valC ← M ₄ [PC + 1] valP ← PC + 5	icode: ifun ← M ₁ [0x023] = 8:0 valC ← M ₄ [0x024] = 0x029 valP ← 0x023 + 5 = 0x028
Декодирование	valB ← R [%esp]	valB ← R [%esp] = 128
Выполнение	valE ← valB + -4	valE ← 128 + -4 = 124

(окончание)

Этап	Общий	Специфический
	call Dest	call 0x029
Память	$M_4[valE] \leftarrow valP$	$M_4[124] \leftarrow 0x028$
Обратная запись	$R[\%esp] \leftarrow valE$	$R[\%esp] \leftarrow 124$
Обновление PC	$PC \leftarrow valC$	$PC \leftarrow 0x029$

Воздействие данной команды: установка %esp в 124, сохранение 0x028 (обратный адрес) в данном адресе ячейки памяти и установка счетчика команд в 0x029 (цель вызова).

РЕШЕНИЕ УПРАЖНЕНИЯ 4.14

Весь HCL-код в этой и других упражнениях является прямым, однако попытка сгенерировать его самостоятельно поможет задуматься о различных командах и о способах их обработки. В данной задаче можно просто рассмотреть систему команд Y86 (рис. 4.2) и определить, какая из них имеет поле константы.

```
bool need_valC =
icode in { IIRMOVL, IRMOVL, IMRMOVL, IJXX, ICALL };
```

РЕШЕНИЕ УПРАЖНЕНИЯ 4.15

Данный код похож на код для srcA.

```
int srcB = [
icode in { IOPL, IRMMOVL, IMRMOVL } : rB;
icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
1 : RNONE; # Необходимости в регистре нет
];
```

РЕШЕНИЕ УПРАЖНЕНИЯ 4.16

Данный код похож на код для dstE.

```
int dstM = [
icode in { IMRMOVL, IPOPL } : rA;
1 : RNONE; # Необходимости в регистре нет
];
```

РЕШЕНИЕ УПРАЖНЕНИЯ 4.17

По упр. 4.12 видно, что необходимо осуществить запись через порт M для установки приоритета над записью через порт E для сохранения значения, считанного из памяти в регистр %esp.

РЕШЕНИЕ УПРАЖНЕНИЯ 4.18

Данный код похож на код для aluA.

```
int aluB = [
icode in { IRMMOVL, IMRMOVL, IOPL, ICALL,
IPUSHL, IRET, IPOPL } : valB;
icode in { IRRMOVL, IIRMOVL } : 0;
# Другие команды не требуют АЛУ
];
```

РЕШЕНИЕ УПРАЖНЕНИЯ 4.19

Данный код похож на код для mem_addr.

```
int mem_data = [
# Значение из регистра
icode in { IRMMOVL, IPUSHL } : valA;
# Возврат PC
icode == ICALL : valP;
# По умолчанию: Запись не осуществляется
];
```

РЕШЕНИЕ УПРАЖНЕНИЯ 4.20

Данный код похож на код для mem_read.

```
Bool mem_write = icode in { IRMMOVL, IPUSHL, ICALL };
```

РЕШЕНИЕ УПРАЖНЕНИЯ 4.21

Данная задача представляет собой интересное упражнение на поиск оптимального равновесия в наборе разделов. Она обеспечивает несколько возможностей вычисления пропускной способности и периодов ожидания в конвейерах.

Для двухэтапного конвейера лучшим разделением будет размещение блоков A, B и C на первом этапе, а блоков D, E и F — на втором. Время задержки на первом этапе — 170 пс; общее время цикла составляет $170 + 20 = 190$ пс. Следовательно, пропускная способность составляет 5.26 GOPS, а время ожидания — 380 пс.

- Для конвейера с тремя этапами блоки A и B должны размещаться на первом этапе, блоки C и D — на втором, а E и F — на третьем. Первые два этапа имеют задержку в 110 пс, что образует общее время цикла в 130 пс и пропускную способность в 7.69 GOPS. Время ожидания — 390 пс.
- Для конвейера с четырьмя этапами блок A должен размещаться на первом этапе, блоки B и C — на втором, D — на третьем, а E и F — на четвертом. Второй этап требует 90 пс, что дает общее время цикла в 110 пс, и пропускную способность в 9.09 GOPS. Время ожидания — 440 пс.
- Оптимальным будет проект, состоящий из пяти этапов, где каждый блок находится на отдельном этапе, кроме блоков E и F, находящихся на пятом этапе. Время

цикла: $80 + 20 = 100$ пс; пропускная способность — 10.00 GOPS, а время ожидания — порядка 500 пс. Добавление этапов ничего не изменит, потому что конвейер не будет выполнять один цикл быстрее, чем за 100 пс.

РЕШЕНИЕ УПРАЖНЕНИЯ 4.22

При ограничении можно получить конвейер, где каждый вычислительный блок имеет задержку в ϵ пикосекунд. Тактовый интервал составит $\epsilon + 20$ пикосекунд, что даст пропускную способность в $1000 / (\epsilon + 20)$. По мере произвольного возрастания количества этапов ϵ будет стремиться к нулю, что даст пропускную способность в 50.00 GOPS.

РЕШЕНИЕ УПРАЖНЕНИЯ 4.23

В этот код входят префиксы к названиям сигналов в коде для SEQ с D_.

```
int new_E_dstE = [
D_icode in { IRRMOVL, IIRMOVL, IOPL } : D_rB;
D_icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
1 : RNONE; # Необходимы в регистре нет
];
```

РЕШЕНИЕ УПРАЖНЕНИЯ 4.24

Команда `rrmovl` (строка 5) остановится на один цикл из-за риска load/use, вызванного командой `popl` (строка 4). На входе на этап декодирования команда `popl` будет находиться на этапе памяти, что обеспечит равенство M_{dstE} и M_{dstM} для `%esp`. Если два этих случая поменять местами, тогда обратная запись из M_{valE} будет пользоваться приоритетом, что вызовет передачу приращенного указателя стека в качестве аргумента в команду `rrmovl`. При этом возникнет расхождение с соглашением об обработке `popl %esp`, определенным в упр. 4.5.

РЕШЕНИЕ УПРАЖНЕНИЯ 4.25

Упражнение позволяет получить опыт решения одной из важнейших задач проектирования процессора: создания тестовых программ. Вообще говоря, следует иметь тестовые программы, проверяющие возможности возникновения всех рисков и генерирующие некорректные результаты в случае, если какая-то из зависимостей не обрабатывается должным образом.

Для данного примера можно воспользоваться слегка модифицированной версией программы, показанной в упр. 4.24.

```
1 irmovl $5, %edx
2 irmovl $0x100, %esp
3 rrmovl %edx, 0 (%esp)
4 popl %esp
5 nor
6 nor
7 rrmovl %esp, %eax
```

Две команды `por` отправляют команду `rop1` на этап обратной записи, когда команда `irmovl` находится на этапе декодирования. Если двум источникам передвижения на этапе обратной записи неправильно установлен приоритет, тогда в регистр `%eax` будет установлен приращенный счетчик команд, а не значение, считанное из памяти.

РЕШЕНИЕ УПРАЖНЕНИЯ 4.26

Данная логика требует только проверки пяти источников передвижения:

```
int new_E_valB = {
d_srcB == E_dstE: e_valE; # Передвижение valE из этапа выполнения
d_srcB == M_dstM: m_valM; # Передвижение valM из этапа памяти
d_srcB == M_dstE: M_valE; # Передвижение valE из этапа памяти
d_srcB == W_dstM: W_valM; # Передвижение valM из этапа обратной записи
d_srcB == W_dstE: W_valE; # Передвижение valE из этапа обратной записи
1 : d_rvalB; # Использовать значение, считанное из регистрового файла
};
```

РЕШЕНИЕ УПРАЖНЕНИЯ 4.27

Следующая тестовая программа разработана для настройки комбинации управления (см. рис. 4.60) и проверки правильности всех операций:

```
1 # Код для генерирования комбинации невыбранной ветви и ret
2 irmovl Stack, %esp
3 irmovl rtmp, %eax
4 pushl %eax           # Установка обратного указателя
5 xorl %eax, %eax      # Установка кода условия Z
6 jne target           # Не выбрана (Первая часть комбинации)
7 irmovl $1, %eax       # Необходимо выполнить
8 halt
9 target:ret           # Вторая часть комбинации
10 irmovl $2, %ebx      # Не выполнять
11 halt
12 rtmpirmovl $3, %edx  # Не выполнять
13 halt
14 .pos 0x40
15 Stack
```

Данная программа спроектирована таким образом, что при возникновении каких-либо сбоев (например, при фактическом выполнении команды `ret`) она выполнит одну из дополнительных команд `irmovl` и остановится. Таким образом, ошибка в конвейере некорректно обновит тот или иной регистр. Данный код наглядно иллюстрирует осторожность, которую необходимо соблюдать при реализации тестовой программы. Он должен настроить условие потенциальной ошибки, после чего выяснить, будет она иметь место или нет.

РЕШЕНИЕ УПРАЖНЕНИЯ 4.28

Следующая тестовая программа спроектирована для настройки управляющей комбинации В (см. рис. 4.60). Имитатор выявит случай, когда управляющие сигналы кружка и останова для конвейерного регистра оба установлены в 0 так, что тестовой программе потребуется только настроить комбинацию для ее выявления. Наиболее сложно здесь заставить программу выполнять полезные действия при правильных манипуляциях.

```

1 # Команда тестирования, изменяющая %esp, за которым следует ret
2 irmovl mem, %ebx
3 irmovl 0(%ebx), %esp      # Настройка %esp для указания на точку возврата
4 ret                      # Возврат в точку возврата
5 halt                     #
6 rtnpt:irmovl $5, %esi    # Точка возврата
7 halt
8 .pos 0x40
9 mem:.long stack          # Удерживает нужный указатель стека
10 .pos 0x50
11 stack:.long rtnpt       # Верхний стек: удерживает точку возврата

```

Данная программа использует в памяти два инициализированных слова. Первое слово (mem) удерживает адрес второго (stack — нужный указатель стека). Второе слово удерживает адрес нужной точки возврата для команды ret. Программа загружает указатель стека в %esp и выполняет команду ret.

РЕШЕНИЕ УПРАЖНЕНИЯ 4.29

Из табл. 4.16 видно, что конвейерный регистр D необходимо приостановить для риска загрузки/использования:

```

bool D_stall =
# Условия для риска загрузки/использования
E_icode in { IMRMOVL, IPOPL } &&
E_dstM in { d_srcA, d_srcB };

```

РЕШЕНИЕ УПРАЖНЕНИЯ 4.30

Из табл. 4.17 видно, что конвейерный регистр E необходимо установить на кружок для риска загрузки/использования или для некорректно спрогнозированной ветви:

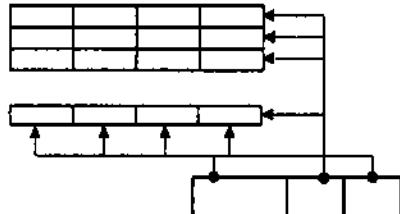
```

bool E_bubble =
# Некорректно спрогнозированная ветвь
(E_icode == IJXX && !e_Bch) ||
# Условия для риска загрузки/использования
E_icode in { IMRMOVL, IPOPL } &&
E_dstM in { d_srcA, d_srcB };

```

РЕШЕНИЕ УПРАЖНЕНИЯ 4.31

Получаем некорректно спрогнозированную частоту 0.35 с $mp = 0.20 \times 0.35 \times 2 = 0.14$ с общим числом циклов на команду (CPI), равным 1.25. Коэффициент усиления кажется довольно маржинальным, однако он того стоит, если стоимость реализации новой стратегии прогнозирования ветвей не слишком высока.



ГЛАВА 5

Оптимизация производительности программ

- Возможности и ограничения оптимизирующего компилятора.
 - Выражение производительности программы.
 - Пример программы.
 - Устранение недостаточности циклов.
 - Сокращение обращений к процедурам.
 - Устранение ненужных ссылок на ячейки памяти.
 - Общее описание современных процессоров.
 - Снижение непроизводительных издержек циклов.
 - Преобразование в код указателя.
 - Повышение параллелизма.
 - Окончательная сборка: сводка результатов оптимизации объединяющего кода.
 - Прогнозирование ветвей и штрафы за некорректное прогнозирование.
 - Понятие производительности памяти.
 - Жизнь в реальном мире: методы повышения производительности.
 - Выявление и устранение критических элементов производительности.
 - Резюме.
-

Процесс написания эффективной программы требует осуществления двух видов деятельности. Во-первых, необходимо подобрать набор алгоритмов и структур данных. Во-вторых, нужно написать исходный код, который бы компилятор эффективно оптимизировал в действенный исполняемый код. Для второй операции важно понимать возможности и ограничения оптимизирующих компиляторов. Внешне незначительные изменения в процессе написания программы могут радикально повлиять на успех ее оптимизации компилятором. Одни языки программирования оптимизируются проще, другие — сложнее. Некоторые особенности С, напри-

мер, возможность преобразования арифметики указателя и подбора типов, затрудняют его оптимизацию. Часто программисты могут создавать программы способами, облегчающими задачу компиляторов по генерированию эффективного кода.

В ходе разработки программы и ее оптимизации необходимо понимать то, как данный код будет использоваться, а также критические факторы, которые могут на него повлиять. В принципе, программистам всегда стоит искать компромиссы между тем, насколько просто реализовать и обслуживать программу, и тем, насколько быстро действующим будет ее выполнение. На уровне алгоритмов простую сортировку методом вставок можно написать за считанные минуты, тогда как реализация и оптимизация высокоеффективной процедуры сортировки может занять несколько дней. На уровне кодировки многие оптимизации низкого уровня в некотором роде снижают читаемость и изменяемость кодов, что делает программы более чувствительными к ошибкам и затрудняет возможность их модификации или расширения. Для программы, которая будет выполнена всего один раз для генерирования набора результатов обработки данных, более важным аспектом является реализация с минимальными затратами усилий со стороны программиста с одновременным сохранением ее полной корректности. Для кода, предназначенного для многократного выполнения в среде, где производительность является критичным фактором, например в сетевом маршрутизаторе, обычно более уместно применение расширенной оптимизации.

В данной главе описываются несколько методик повышения производительности кодов. Идеальным вариантом была бы ситуация, когда компилятор принимал бы любой написанный код и генерировал максимально производительную программу машинного уровня с заданным поведением. На практике же компиляторы могут выполнять только ограниченные преобразования программы; их функционированию могут мешать так называемые блокираторы оптимизации — аспекты поведения программы, сильно зависящие от среды выполнения. Программисты должны "помогать" компилятору написанием таких кодов, которые можно легко оптимизировать. В литературе, посвященной компиляторам, методики оптимизации классифицируются либо как "независимые от машины" — они должны применяться независимо от характеристик компьютера, на котором будет выполняться код, либо как "зависимые от машины", т. е. от множества особенностей низкого уровня машины. Представление материала организовано по сходным строкам, начиная с преобразований программы, что должно быть стандартной практикой при написании любой программы. Затем авторы переходят к преобразованиям, действенность и эффективность которых зависит от характеристик целевой машины и компилятора. Эти преобразования также снижают модульность и читабельность кода и, следовательно, должны применяться, когда во главу угла ставится максимальная производительность.

Для максимального повышения производительности программы ее создатель и компилятор должны иметь модель целевой машины с указанием способов обработки команд и временных характеристик различных операций. Например, компилятор должен обладать информацией синхронизации для того, чтобы определить, использовать команду умножения или некую комбинацию сдвигов и адресов. В современных компьютерах используются очень сложные методики обработки программ машинного уровня с выполнением многих команд параллельно и часто в другом порядке, нежели они появляются в программе. Для того чтобы иметь возможность

настройки программ на максимальную скорость выполнения, программисты должны понимать глубинные принципы работы процессоров. Здесь представлена высокоуровневая модель такой машины, на основе последних моделей процессоров Intel. Также авторами разработано графическое представление, которое можно использовать для визуализации выполнения процессором команд и для прогнозирования производительности программы.

Глава заканчивается обсуждением вопросов, имеющих отношение к оптимизации крупных программ. Описывается использование кодовых *профайлеров* — инструментов измерения производительности различных частей программы. Такой анализ может помочь обнаружить неэффективные элементы кода и идентифицировать части программы, на которые следует обращать особое внимание в процессе оптимизации. И в самом конце представлено важное наблюдение, известное как закон Эмдела, выражающий в количественном отношении общий эффект от оптимизации той или иной части системы программного обеспечения.

В представленном здесь материале кодовая оптимизация выглядит простым линейным процессом применения к коду серии преобразований в определенном порядке. На самом деле, задача не настолько прозрачна, и требуется большой объем экспериментов методом проб и ошибок. Это становится особенно видно при подходе к более поздним этапам оптимизации, когда внешние мелкие изменения могут вызвать переворот в показателях производительности, а какие-то многообещающие методы себя абсолютно не оправдывают. По приведенным в главе примерам станет понятно, что иногда довольно трудно вразумительно объяснить, почему та или иная кодовая последовательность имеет ту или иную продолжительность выполнения. Производительность может зависеть от множества мелких особенностей проектирования процессора, по которым недостаточно либо документации, либо досконального понимания. Это еще одна причина использования многих вариаций и комбинаций различных способов.

Изучение компонующего автокода — одно из самых эффективных средств понимания работы компилятора и выполнения сгенерированного кода. Хорошей стратегией для начала будет тщательное изучение кода на предмет наличия в нем внутренних циклов. Можно выявить атрибуты, снижающие производительность, например, чрезмерные ссылки на ячейки памяти и недостаточное использование регистров. Если начать с компонующего автокода, то можно даже спрогнозировать, какие операции будут выполняться параллельно и насколько полно они будут использовать ресурсы процессора.

5.1. Возможности и ограничения оптимизирующего компилятора

В современных компиляторах используются сложные алгоритмы для определения вычисляемых в программе значений и их использования. Затем применяются возможности упрощения выражений для осуществления единого вычисления в нескольких разных частях системы и для сокращения повторения того или иного вычисле-

ния. Способность компиляторов к оптимизации программ ограничена несколькими факторами, включая:

- требование к тому, чтобы корректное поведение программы никогда не менялось;
- ограниченное понимание поведения программы и среды, в которой она будет использоваться;
- необходимость оперативного проведения оптимизации.

Предполагается, что оптимизация компилятора невидима для пользователя. Когда программист компилирует код с активизированной оптимизацией (например, используя опцию `-O` командной строки), код должен повторять поведение, которое было демонстрировалось при компиляции другим способом, за исключением того, что его выполнение должно занимать меньше времени. Это требование ограничивает способность компилятора выполнять некоторые виды оптимизации.

Рассмотрим, к примеру, две следующие процедуры (листинг 5.1).

Листинг 5.1. Две процедуры

```

1 void twiddle1 (int *xp, int *yp)
2 {
3     *xp += *yp;
4     *xp += *yp;
5 }
6
7 void twiddle2 (int *xp, int *yp)
8 {
9     *xp += 2 * *yp;
10 }
```

На первый взгляд кажется, что обе процедуры одинаковы. Обе дважды добавляют значение, сохраненное в ячейке, обозначенной указателем `yp`, в ячейку, обозначенную указателем `xp`. С другой стороны, функция `twiddle2` более эффективна. Она требует только трех ссылок на ячейки памяти (`read *xp, read *yp, write *xp`), тогда как `twiddle1` требует шести ссылок (два считывания `*xp`, два считывания `*yp` и две записи `*xp`). Следовательно, если компилятор должен скомпилировать процедуру `twiddle1`, может сложиться впечатление о создании более производительного кода, на основании вычислений, выполненных функцией `twiddle2`.

Рассмотрим случай, когда `xp` и `yp` равны. Тогда функция `twiddle1` выполнит следующие вычисления:

```

3     *xp += *xp; /* Двойное значение по xp */
4     *xp += *xp; /* Двойное значение по xp */
```

В результате значение по `xp` будет увеличено на коэффициент 4. С другой стороны, функция `twiddle2` выполнит следующее вычисление:

```
9     *xp += 2 * *xp; /* Тройное значение при xp */
```

В результате значение по `xp` будет увеличено на коэффициент 3. Компилятор не имеет никакой информации о том, как будет вызываться `twiddle1`, поэтому он должен предположить, что аргументы `xp` и `yp` могут быть равными. Следовательно, компилятор не может сгенерировать код в стиле `twiddle2` в качестве оптимизированной версии `twiddle1`.

Данный феномен известен как использование псевдонимов памяти. Компилятор должен предположить, что разные указатели могут обозначать одну и ту же ячейку памяти. Это приводит к одному из основных блокираторов оптимизации — аспекту программ, который может значительно ограничить возможности компилятора по генерированию оптимизированного кода.

УПРАЖНЕНИЕ 5.1

Следующая проблема иллюстрирует то, как использование псевдонимов памяти может спровоцировать неожиданное поведение программы. Рассмотрим следующую процедуру обмена двух значений:

```
1 /* Обмен значения x при xp на значение у при yp */
2 void swap (int *xp, int *yp)
3 {
4     *xp = *xp + *yp; /* x+y */
5     *yp = *xp - *yp; /* x+y-y = x */
6     *xp = *xp - *yp; /* x+y-x = y */
7 }
```

Если эта процедура вызывается с `xp`, равным `yp`, каков будет эффект?

Второй блокиратор оптимизации происходит из-за обращения к функции. В качестве примера рассмотрим две следующие процедуры (листинг 5.2).

Листинг 5.2: Еще два процедуры

```
1 int f (int);
2
3 int func1(x)
4 {
5     return f (x) + f (x) + f (x);
6 }
7
8 int func2 (x)
9 {
10    return 4*f (x);
11 }
```

Сначала может показаться, что обе процедуры вычисляют один и тот же результат, только `func2` вызывает `f` один раз, а `func1` — четыре раза. Весьма заманчиво сгенерировать код в стиле `func2`, имея `func1` в качестве источника.

Впрочем, рассмотрим следующий код (листинг 5.3) для f.

Листинг 5.3. Пример функции

```
1 int counter = 0
2
3 int a (int x)
4 {
5     return counter++;
6 }
```

Эта функция имеет побочный эффект: она модифицирует некоторую часть глобального состояния программы. Изменение количества вызовов этой функции изменяет поведение программы. В частности, вызов `func1` возвратит $0 + 1 + 2 + 3 = 6$, тогда как вызов `func2` возвратит $4 \times 0 = 0$, если предположить, что обе функции запущены с глобальной переменной `counter`, установленной в 0.

Большинство компиляторов не пытается определить, свободна ли функция от побочных эффектов, и, следовательно, является кандидатом на оптимизацию, как в случае с `func2`. Вместо этого компилятор предполагает худший вариант и оставляет все обращения к функциям нетронутыми.

Из существующих компиляторов компилятор GCC GNU считается наиболее адекватным, но не единственным, в том, что касается возможностей оптимизации. Он выполняет базовые типы оптимизации, но не выполняет радикальных преобразований программ (эти задачи решаются более "агрессивными" компиляторами). Вследствие этого программисты, работающие с GCC, должны прикладывать больше усилий при написании программ методами, упрощающими для компилятора задачу генерирования эффективного кода.

5.2. Выражение производительности программы

Необходимо найти способ выражения производительности программы, который может оказать помощь в совершенствовании кода. Для многих программ полезной единицей измерения является CPE (cycles per element, циклов на элемент). Эта единица измерения помогает понять производительность цикла итеративной программы на детальном уровне. Она подходит для программ, выполняющих многократные вычисления, такие как обработка пикселов в графическом изображении или вычисление элементов в произведении матриц.

Планирование операций процессором управляет тиковым генератором, регулярно подающим сигналы на определенной частоте, измеряемой либо в мегагерцах (МГц) — миллионах циклов в секунду, либо в гигагерцах (ГГц) — в миллиардах циклов в секунду. Например, если в руководствах к системе указан процессор с частотой 1.4 ГГц, то это означает, что тиковский генератор процессора работает на частоте

1400 МГц. Время, необходимое для выполнения каждого цикла синхронизации, представлено как величина, обратная синхронизирующей частоте. Оно обычно выражается в наносекундах (миллиардная доля секунды). Тактовый генератор 2 ГГц имеет период в 0.5 наносекунд, а тактовый генератор 500 МГц — период в 2 наносекунды. С точки зрения программиста, в качестве единицы измерения более наглядно использовать циклы синхронизации, а не наносекунды, потому что при этом единица измерения менее зависит от конкретной модели оцениваемого процессора, и обеспечивается четкое понимание того, как программа выполняется машиной.

Многие процедуры содержат цикл, выполняющий итерацию над множеством элементов. Например, функции `vsum1` и `vsum2`, показанные в листинге 5.4, обе вычисляют сумму двух векторов с длиной n . Первая вычисляет один элемент вектора назначения за итерацию. Вторая применяет методику, известную как *развертка цикла*, для вычисления двух элементов за итерацию. Данная версия работает корректно только для четных значений n . Далее развертка цикла рассматривается более подробно, включая ее работу для произвольных значений n .

Необходимое для выполнения данной процедуры время можно охарактеризовать как константа плюс коэффициент, пропорциональный количеству обработанных элементов. Например, на рис. 5.1 показана диаграмма числа циклов синхронизации, необходимая двум данным функциям для диапазона значений n . Используя подбор кривых методом наименьших квадратов, обнаруживается, что значения времени выполнения двух функций (в циклах синхронизации) можно аппроксимировать кривыми с уравнениями $80 + 4.0n$ и $83.5 + 3.5n$ соответственно. Эти уравнения указали служебные сигналы или данные на инициацию процедуры от 80 до 84 циклов, задали цикл, завершили процедуру, а также указали линейный коэффициент в 3.5 или 4.0 циклов на элемент. Для больших значений n (например, более 50) над значениями времени выполнения будут преобладать линейные коэффициенты. В данном случае обращение к коэффициентам считается действующим количеством выполняемых циклов на элемент (CPE). Обратите внимание, что предпочтение отдается измерению количества циклов на элемент, а не на итерацию, потому что такие методики, как развертка цикла, позволяют использовать меньшее количество итераций для завершения вычислений, однако конечной целью является выяснение того, насколько быстро будет выполняться процедура для заданной длины вектора (листинг 5.4).

Листинг 5.4. Функции суммарных векторов

```

1 void vsum1 (int n)
2 {
3     int i;
4
5     for (i = 0; i < n; i++)
6         c [i] = a [i] + b [i];
7 }
8
9 /*Суммарный вектор n элементов (n должно быть четным) */
10 void vsum2 (int n)

```

```

11 {
12 int i;
13
14 for (i = 0; i < n; i +=2) {
15 /* Вычисление двух элементов на итерацию */
16 c [i]      = a [i]      + b [i];
17 c [i+1]    = a [i+1]    + b [i+1];
18}
19 }

```

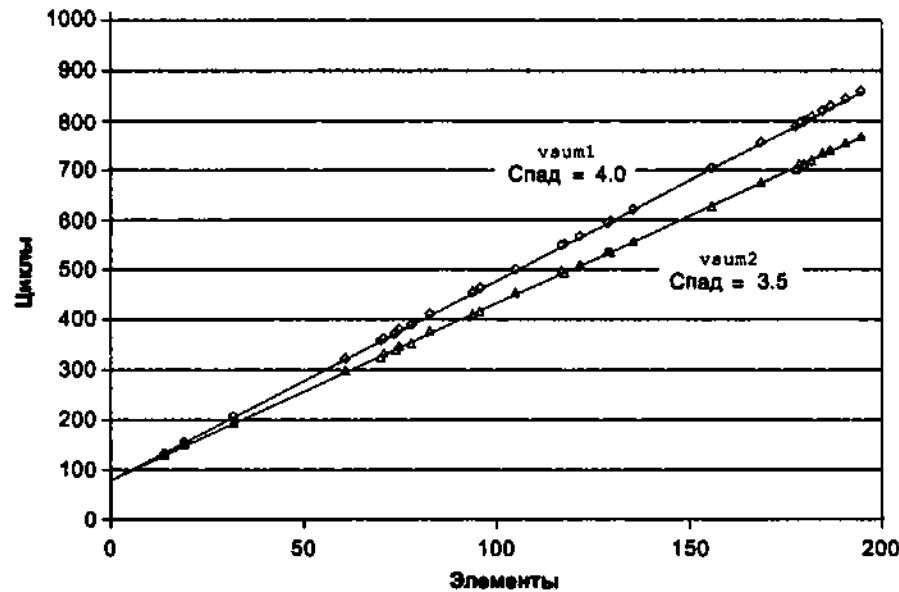


Рис. 5.1. Производительность функций суммарных векторов

Представленные в листинге 5.4 функции дают примеры выражения производительности программы.

Усилия сосредоточены на сведении СРЕ при вычислениях к минимуму. По данной единице измерения vsum2 с СРЕ 3.50 превышает vsum1 с СРЕ 4.0.

Что такое подбор кривых методом наименьших квадратов

Для множества результатов обработки $(x_1, y_1), \dots, (x_n, y_n)$ часто делаются попытки изображения кривой, наилучшим образом аппроксимирующей направленность $X-Y$, представленную этими данными. При подборе кривых методом наименьших квадратов делается попытка поиска кривой в форме

$$y = mx + b,$$

сводящей к минимуму следующее ошибочное измерение:

$$E(m, b) = \sum_{i=1, n} (mx_i + b - y_i)^2$$

Алгоритм вычисления m и b можно получить вычислением производных $E(m, b)$ с учетом m и b и их установки в 0.

УПРАЖНЕНИЕ 5.2

Далее в главе будет взята отдельная функция и сгенерировано много различных вариантов, сохраняющих поведение функции, но с разными характеристиками производительности. Для трех из этих вариантов было обнаружено, что показатели времени выполнения (в циклах синхронизации) можно аппроксимировать следующими функциями:

- $60 + 35n$
- $136 + 4n$
- $157 + 1.25n$

Для каких значений n каждая из версий будет выполняться быстрее всех? Следует помнить о том, что n всегда будет целым числом.

5.3. Пример программы

Для демонстрации того, как абстрактную программу можно систематически преобразовывать в более эффективный код, рассмотрим простую векторную структуру данных, показанную на рис. 5.2. Вектор представлен в виде двух блоков памяти. Заголовок является структурой, объявленной в листинге 5.5.

Листинг 5.5. Структура заголовка

```
1 /* Создать абстрактный тип данных для вектора */
2 typedef struct {
3     int len;
4     data_t *data;
5 } vec_rec, *vec_ptr
```

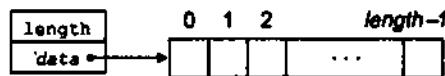


Рис. 5.2. Тип абстрактных векторных данных

Данное объявление использует тип данных `data_t` для обозначения типа данных базовых элементов. В нынешних расчетах измеряется производительность кода для

типов данных `int`, `float` и `double`. Это определяется компиляцией и выполнением программы отдельно для разных объявлений типов, как в следующем примере:

```
typedef int data_t;
```

В дополнение к заголовку распределяется массив объектов `len` типа `data_t` для хранения фактических векторных элементов.

В листинге 5.6 показаны некоторые базовые процедуры генерирования векторов с доступами к векторным элементам и определением длины вектора. Следует обратить внимание на важную особенность, что `get_vec_element` — процедура доступа к вектору — выполняет граничные проверки для каждой ссылки на вектор. Данный код схож с представлениями массива, используемыми во многих других языках, включая Java. Граничные проверки снижают шансы появления программной ошибки, но, как будет видно далее, также оказывает значительное влияние на производительность программы.

В качестве примера оптимизации рассмотрим листинг 5.7, объединяющий все элементы вектора в одно значение, в соответствии с определенной операцией. Используя разные определения выполняемых в процессе компиляции констант `IDENT` и `OPER`, код можно перекомпилировать для выполнения различных операций с данными. В частности, использованием объявлений

```
#define IDENT 0
#define OPER +
```

суммируются элементы вектора. Использованием объявлений

```
#define IDENT 1
#define OPER +
```

вычисляется произведение векторных элементов.

В качестве исходной точки в табл. 5.1 представлены измерения СРЕ для `combine1`, выполняющейся на Intel Pentium III, в которых испытываются все комбинации типов данных и операций. По данным измерениям обнаружено, что значения синхронизации обычно равны данным с плавающей точкой обычной и двойной точности. Следовательно, представлены только измерения обычной точности.

Таблица 5.1. Значения СРЕ для разных типов

Функция	Страница	Метод	Целое число		Число с плавающей точкой	
			+	*	+	*
combine1	387	Абстрактный, неоптимизированный	42.06	41.86	41.44	160.00
combine1	387	Абстрактный -O2	31.25	33.25	31.25	143.00

Листинг 5.6. Реализация типа абстрактных векторных данных

```
1 /* Создание вектора заданной длины */
2 vec_ptr new_vec (int len)
3 {
4 /* Распределение структуры заголовка */
5 vec_ptr result = (vec_ptr) malloc (sizeof (vec_rec));
6 if (!result)
7     return NULL; /* Невозможно распределить сохранение */
8 result ->len = len;
9 /* Распределение массива */
10 if (len > 0) {
11     data_t *data = (data_t *) calloc (len, sizeof (data_t));
12     if (!data) {
13         free ((void *) result);
14         return NULL; /* Невозможно распределить сохранение */
15     }
16     result ->data = data;
17 }
18 else
19 result ->data = NULL;
20 return result;
21 }
22
23 /*
24 * Извлечение элемента вектора и сохранение в месте назначения
25 * Возврат 0 (за пределами границ) или 1 (успешно)
26 */
27 int get_vec_element (vec_ptr v, int index, data_t *dest)
28 {
29 if (index < 0 || index >= v->len)
30     return 0;
31 *dest = v->data [index];
32 return 1;
33 }
34
35 /* Возврат длины вектора */
36 int vec_length (vec_ptr v)
37 {
38 return v->len;
39 }
```

В фактической программе листинга 5.6 тип данных `data_t` объявляется как `int`, `float` или `double`.

Листинг 5.7. Первоначальная реализация комбинирующей операции

```

1  /* Реализация с максимальным использованием абстракции данных */
2  void combinel (vec_ptr v, data_t *dest)
3  {
4      int i;
5
6      *dest = IDENT;
7      for (i = 0; i < vec_length (v); i++)  (
8          data_t val;
9          get_vec_element (v, i, &val);
10         *dest = *dest OPER val;
11     }
12 }
```

В листинге 5.7, используя различные объявления нейтрального элемента IDENT и комбинирующей операции OPER, можно измерить рутинную процедуру для различных операций.

По умолчанию компилятор генерирует код для пошагового выполнения символьного отладчика. Поскольку намерением является подведение объектного кода максимально близко к соответствию вычислениям, указанным в исходном коде, выполняется минимум оптимизации. Последние активизируются простой установкой переключателя командной строки на `-O2`. Можно видеть, что этим значительно повышается производительность программы. Вообще говоря, активизация этого уровня оптимизации может стать полезной привычкой, если программа компилируется без последующей цели ее отладки. Этот уровень компиляторной оптимизации активизируется для оставшейся части измерений.

Также следует обратить внимание на то, что показатели времени выполнения вполне сравнимы для разных типов данных и разных операций, за исключением умножения чисел с плавающей точкой. Эти одиночные импульсы счета циклов высокого уровня для умножения происходят из-за аномалий в исходных данных. Выявление подобных аномалий является важным компонентом анализа производительности и оптимизации. Авторы намерены вернуться к данному вопросу в разд. 5.11.1. Читателям станет понятно, что таким образом можно значительно повысить производительность программ.

5.4. Устранение недостаточности циклов

Обратите внимание, что процедура `combinel`, как показано в листинге 5.7, вызывает функцию `vec_length` как проверяемое условие цикла `for`. Вспомним из обсуждения циклов, что проверяемое условие должно оцениваться при каждой итерации цикла. С другой стороны, длина вектора не меняется при прохождении цикла. Таким образом, длину вектора можно вычислить только однажды и использовать это значение в проверяемом условии.

В листинге 5.8 показана модифицированная версия `combine2`, вызывающая `vec_length` в начале цикла и присваивающая результат локальной переменной `length`. Затем эта локальная переменная используется в проверяемом условии цикла `for`. Удивительно, что это небольшое изменение значительно влияет на производительность программы: Как показано в табл. 5.2, данное простое преобразование убирает приблизительно по 10 циклов для каждого векторного элемента.

Листинг 5.8. Повышение эффективности проверки конца цикла

```

1 /* Перемещение вызова в vec_length за пределы цикла */
2 void combine2 (vec_ptr v, data_t *dest)
3 {
4     int i;
5     int length = vec_length (v);
6
7     *dest = IDENT;
8     for (i = 0; i < length; i++) {
9         data_t val;
10        get_vec_element (v, i, &val);
11        *dest = *dest OPER val;
12    }
13 }
```

Таблица 5.2. Значения СРЕ для модифицированных типов

Функция	Страница	Метод	Целое число		Число с плавающей точкой	
			+	*	+	*
combine1	387	Абстрактный -O2	31.25	33.25	31.25	143.00
combine1	388	Перемещение vec_length	22.61	21.25	21.15	135.00

Данная оптимизация представляет собой экземпляр общего класса оптимизаций, называемый *вынесением части текста программы*. В их число входит выявление вычисления, выполняемого многократно (например, в рамках цикла), но такого, что результат этого вычисления не изменится. Таким образом, данное вычисление можно перенести в более ранний раздел кода, не оцениваемый столь часто. В этом случае вызов в `vec_length` перемещен из цикла в место непосредственно перед его началом.

Оптимизирующие компиляторы делают попытку выполнить вынесение части текста. Как уже упоминалось, к сожалению, они, как правило, ведут себя предельно осторожно при выполнении преобразований, изменяющие место обращения к процедуре или количество этих обращений. Компиляторы не могут с уверенностью определить, будет функция иметь побочные эффекты или нет, поэтому попросту предполагают их

возможное наличие. Например, если `vec_length` имеет побочный эффект, тогда `combine1` и `combine2` могут обладать различным поведением. В случаях, подобных данным, программист должен помочь компилятору явным вынесением части текста программы.

В качестве крайнего примера неэффективности цикла, видимого в `combine1`, рассмотрим процедуру `lower1`. Данная процедура стилизована по образцам, представленными несколькими студентами в качестве части проекта сетевого программирования. Целью ее является преобразование всех прописных букв в строчные. Процедура пошагово проходит строку, преобразовывая каждый прописной символ в строчный.

Библиотечная процедура `strlen` вызывается как часть проверки цикла `lower1`. Простая версия `strlen` также показана в листинге 5.9. Поскольку строки в С представляют собой последовательности, заканчивающиеся символом конца строки, `strlen` должна пошагово проходить строку до тех пор, пока не достигнет символа пробела. Для строки длиной n прохождение `strlen` занимает время, пропорциональное n . Поскольку `strlen` вызывается на каждую из n итераций `lower1`, общее время выполнения `lower1` квадратично длине строки.

Листинг 5.9. Рутинные процедуры преобразования в строчный формат

```

1  /* Преобразование строки в строчный формат: медленно */
2 void lower1 (char *S)
3 {
4     int i;
5     .
6     for (i = 0; i < strlen (s); i++)
7         if (s [i] >= 'A' && s [i] <= 'Z')
8             (s [i]) -= ('A' - 'a');
9 }
10
11 /* Преобразование строки в строчный формат: быстрее */
12 void lower2 (char *S)
13 {
14     int i;
15     int len = strlen (s);
16
17     for (i = 0; i < len; i++)
18         if (s [i] >= 'A' && s [i] <= 'Z')
19             (s [i]) -= ('A' - 'a');
20 }
21
22 /* Реализация библиотечной функции strlen */
23 /* Вычисление длины строки */
24 size_t strlen (const char *s)
25 {
26     int length = 0;

```

```

27 while (*s != '\0') {
28     s++;
29     length++;
30 }
31 return length;
32 }
32 return 1;

```

Две процедуры в листинге 5.9 имеют радикально различную производительность. Этот анализ подтверждается фактическими измерениями процедуры для строк разной длины, как показано на рис. 5.3. Граф времени выполнения для lower1 круто поднимается, по мере возрастания длины строки. В нижней части схемы показаны значения времени выполнения для восьми разных длин (не тех, что показаны на графике), каждая из которых является степенью двойки. Обратите внимание, что для lower1 каждое удвоение длины строки вызывает увеличение времени выполнения вчетверо. Это четкий указатель на квадратичную сложность. Для строки длиной 262 144 lower1 требует 3.1 минуты времени работы центрального процессора.

Функция lower2 в листинге 5.9 идентична функции lower1, за исключением того, что вызов вынесен в strlen за пределы цикла. Производительность повышается радикально (табл. 5.3). Для длины строки 262 144 выполнение этой функции требует всего 0.006 секунд, т. е. более чем в 30 тыс. раз меньше. Каждое удвоение длины строки вызывает удвоение значения времени выполнения, четкий указатель на линейную сложность. Для строк большей длины сокращение времени выполнения будет еще более заметным.

Первоначальный код lower1 имеет квадратичную асимптотическую сложность из-за неэффективной структуры цикла (рис. 5.3). Модифицированный код lower2 имеет линейную сложность.

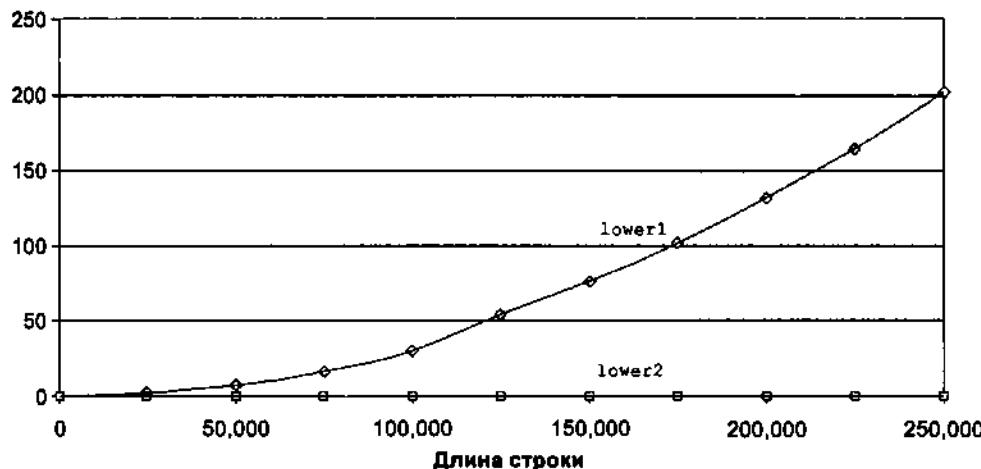


Рис. 5.3. Сравнительная производительность рутинных процедур преобразования в строчный формат

Таблица 5.3. Производительность процедур преобразования

Функция	Длина строки					
	8.192	16,384	32,768	65,536	131,072	262,144
lower1	0.15	0.62	3.19	12.75	51.01	186.71
lower2	0.0002	0.0004	0.0008	0.0016	0.0031	0.0060

В идеальном мире компилятор распознает, что каждый вызов `strlen` в проверке цикла возвратит один и тот же результат, и, следовательно, этот вызов будет перемещен из цикла. Все это потребует очень тщательного и сложного анализа, поскольку `strlen` проверяет элементы строки, а их значения изменяются с началом выполнения `lower1`. Компилятору потребуется выявить, несмотря на изменение символов в строке, ни один из них не меняет ненулевое значение на нулевое, и наоборот. Такого рода анализ выходит далеко за пределы возможностей самых агрессивных компиляторов, поэтому программисты должны выполнять такие преобразования самостоятельно.

Данный пример иллюстрирует общую проблему написания программ, когда, на первый взгляд, тривиальная часть кода имеет скрытую асимптотическую неэффективность. Сложно предположить, чтобы рутинная процедура преобразования в строчный формат могла бы стать ограничивающим фактором производительности программы. Обычно программы тестируются и анализируются по мелким наборам данных, для которых производительности `lower1` вполне хватает. Впрочем, когда программа полностью развернута, очень возможно, что данную процедуру (`lower1`) можно применить к строке длиной в миллион символов, на что потребуется порядка одного часа работы процессора. Так, ни с того ни с сего, этот "миленький" кусочек кода превратился в основной критический элемент производительности. В противовес ему, на выполнение `lower2` ушла бы одна секунда. Известно множество рассказов о крупных программных проектах, сталкивавшихся с проблемами подобного сорта, поэтому частью работы компетентного программиста является необходимость избегать появления асимптотической неэффективности.

УПРАЖНЕНИЕ 5.3

Рассмотрим следующие функции:

```
int min (int x, int y) { return x < y ? x : y; }
int max (int x, int y) { return x < y ? y : x; }
void incr (int *xp, int v) { *xp += v; }
int square (int x) { return x*x; }
```

Эти функции вызываются следующими тремя фрагментами кода:

- for (i = min (x, y); i < max (x, y); incr (&i, 1))
 t += square (i);
- for (i = max (x, y) - 1; i >= min (x, y); incr (&i, -1))
 t += square (i);

```
3. int low = min (x, y);
   int high = max (x, y);
   for (i = low; i < high; incr (&i, 1))
      t += square (i);
```

Предположим, что x равно 10, а у равно 100. Заполните следующую таблицу с указанием количества раз вызова этих четырех функций во фрагментах кода 1–3.

Код	min	max	incr	square
1				
2				
3				

5.5. Сокращение обращений к процедурам

Как уже отмечалось, обращения к процедурам вызывают значительное количество служебных сигналов и блокируют большую часть форм оптимизации программы. В коде для `combine2` (см. листинг 5.8) видно, что `gen_vec_element` вызывается при каждой итерации цикла для извлечения элемента следующего вектора. Эта процедура обходится особенно дорого, поскольку она выполняет граничную проверку. Последняя может быть полезной при необходимости доступа к произвольным массивам, однако простой анализ кода для `combine2` показывает, что все ссылки будут действительными.

Вместо этого предположим, что к типу абстрактных данных добавляется функция `get_vec_start`. Эта функция возвращает начальный адрес массива данных, как показано в листинге 5.10. Затем можно написать процедуру, обозначенную как `combine3` (листинг 5.11), не имеющую обращений к функции во внутреннем цикле. Вместо обращения к функции для получения каждого векторного элемента, она получает непосредственный доступ к массиву. Борцы за чистоту нравов могут заявить, что подобное преобразование серьезно вредит модульности программы. В принципе, пользователю типа абстрактных векторных данных даже не нужно знать, что содержимое вектора сохраняется в виде массива, а не в виде какой-то другой структуры данных, например связного списка. Более прагматичный программист оспорил бы преимущество данного преобразования на основании следующих экспериментальных результатов (табл. 5.4).

Повышение происходит на коэффициент до 3.5X. Для приложений, в которых производительность является критичной, часто ради скорости выполнения можно пожертвовать модульностью и абстракцией (листинги 5.10—5.11). Очень полезно включать документацию по применяемым преобразованиям, а также по приведшим к ним посылкам, на случай, если код будет модифицирован позже.

Таблица 5.4. Спорное преимущество

Функция	Страница	Метод	Целое число		Число с плавающей точкой	
			+	*	+	*
combine2	388	Перемещение vec_length	20.66	21.25	21.15	135.00
combine3	392	Прямой доступ к данным	6.00	9.00	8.00	117.00

Листинг 5.10. Удаление обращений к функции в рамках цикла

```

1 data_t *get_vec_start (vec_ptr v)
2 {
3 return V ->data;
4 }
```

Листинг 5.11. Результирующий код

```

1 /* Прямой доступ к векторным данным */
2 void combine3 (vec_ptr v, data_t *dest)
3 {
4 int i;
5 int length = vec_length (v);
6 data_t *data = get_vec_start (v);
7
8 *dest = IDENT;
9 for (i = 0; i < length; i++) {
10 *dest = *dest OPER data [i];
11 }
12 }
```

Выражение относительной производительности

Лучшим способом выражения повышения производительности является отношение формы T_{old}/T_{new} , где T_{old} — время, требуемое для выполнения первоначальной версии, а T_{new} — время, требуемое на выполнение модифицированной версии. Это число будет превышать 1.0, если будет иметь место реальное повышение. Сuffix X используется для обозначения такого отношения, где коэффициент 3.5X выражается вербально как "умноженное на 3.5".

Более традиционный способ выражения относительного изменения в процентном отношении целесообразен, если изменение незначительно, но его определение двусмысленно.

Это должно быть

$$100 \times (T_{old} - T_{new})/T_{new},$$

$$100 \times (T_{old} - T_{new})/T_{old}$$

или что-то еще? Кроме этого, определение менее информативно для радикальных изменений. Выражение "производительность повысилась на 250%" намного сложнее понять, нежели заявление, что производительность повысилась с коэффициентом 3.5.

5.6. Устранение ненужных ссылок на ячейки памяти

Код для `combine3` аккумулирует значение, вычисляемое комбинирующей операцией в ячейке, обозначенной указателем `dest`. Этот атрибут можно увидеть путем изучения компонующего автокода, сгенерированного для скомпилированного цикла, с целыми числами в качестве типа данных, и умножения в качестве комбинирующей операции (листинг 5.12). В данном коде регистр `%ecx` указывает на `data`, `%edx` содержит значение 1, а `%edi` указывает на `dest`.

Листинг 5.12. Ассемблерный код

```
combine3: type=INT, OPER = *
dest in %edi, data in %ecx, i in %edx, length in %esi

1 .L18                                loop:
2 movl (%edi), %eax                  Считывание *dest
3 imull(%ecx, %edx, 4), %eax        Умножение на data [i]
4 movl %eax, (%edi)                 Запись *dest
5 incl %edx                          i++
6 cmpl %esi, %edx                  Сравнение i:length
7 jl .L18                            If <, goto loop
```

Команда в строке 2 листинга 5.12 считывает значение, сохраненное в `dest`, а команда 4 выполняет обратную запись в эту ячейку. Это кажется неэкономным, поскольку значение, считываемое командой 2 при следующей итерации, как правило, будет только что записанным значением.

Все это приводит к оптимизации, обозначенной как `combine4` (листинг 5.13), где представлена временная переменная `x`, используемая в цикле для накопления вычисленного значения. Результат сохраняется в `*dest` только по завершении цикла. Как показано в следующем компонующем автокоде (листинг 5.14), теперь компилятор может использовать регистр `%eax` для удержания накопленного значения. По сравнению с циклом для `combine3` количество операций памяти на каждую итерацию было

сокращено с двух считываний и одной записи до одной операции считывания. Регистры `%ecx` и `%edx` используются, как и раньше, однако необходимость в ссылке на `*dest` отпадает.

```
combine4: type=INT, OPER = *
dest in %eax, x in %ecx, i in %edx, length in %esi

1 .L24                      loop:   .
2 imull(%eax, %edx, 4), %ecx    Умножение x на data [1]
3 incl %edx                  i++
4 cmpb %esi, %edx            Сравнение i:length
5 jl.L24                     If <, goto loop
```

/* Накопление результата в локальной переменной */

```
void combine4 (vec_ptr v, data_t *dest)
{
int i;
int length = vec_length (v);
data_t *data = get_vec_start (v);
data_t x = IDENT;

*dest = IDENT;
for (i = 0; i < length; i++) {
x = x OPER data [i];
}
*dest = x;
```

Тем самым устраняется необходимость считывания и записи промежуточных значений при каждой итерации цикла.

Значительное повышение производительности программы видно из табл. 5.5.

Наиболее впечатляющий спад по времени наблюдается для выполнения умножения чисел с плавающей точкой. Оно становится сопоставимым со значениями для других комбинаций типа данных и операции. В разд. 5.11.1 рассматриваются причины такого внезапного сокращения.

Опять же, можно подумать, что компилятор должен автоматически преобразовывать код `combine3`, показанный в листинге 5.11, для накопления значения в регистре, как это происходит с кодом для `combine4` (листинг 5.14).

Таблица 5.5. Повышение производительности

Функция	Страница	Метод	Целое число		Число с плавающей точкой	
			+	*	+	*
combine3	392	Прямой доступ к данным	6.00	9.00	8.00	117.00
combine4	394	Накопление во временной переменной	2.00	4.00	3.00	5.00

Однако, на самом деле, эти две функции могут демонстрировать различное поведение из-за использования псевдонимов памяти. Рассмотрим, к примеру, случай с целочисленными данными с умножением в качестве операции, и единицей в качестве нейтрального элемента. Пусть v — вектор, состоящий из трех элементов [2, 3, 5]; обратим внимание на следующие два обращения к функциям:

```
combine 3 (v, get_vec_start (v) + 2);
combine 4 (v, get_vec_start (v) + 2);
```

Таким образом, между последним элементом вектора и назначением для сохранения результата создается псевдосигнал. Тогда эти две функции будут выполняться следующим образом (табл. 5.6):

Таблица 5.6. Сравнительные характеристики циклов

Функция	Первонач.	До цикла	i = 0	i = 1	i = 2	Окончательн.
combine3	[2, 3, 5]	[2, 3, 1]	[2, 3, 2]	[2, 3, 6]	[2, 3, 36]	[2, 3, 36]
combine4	[2, 3, 5]	[2, 3, 5]	[2, 3, 5]	[2, 3, 5]	[2, 3, 5]	[2, 3, 30]

Как указывалось ранее, combine3 накапливает результат в точке назначения, которое в данном случае является окончательным векторным элементом. Таким образом, это значение сначала устанавливается в 1, затем на $2 \times 1 = 2$, а потом $3 \times 2 = 6$. При окончательной итерации это значение умножается на само себя для получения окончательного значения 36. Для случая с combine4 вектор остается неизменным до конца, когда окончательный элемент устанавливается на вычисленный результат

$$1 \times 2 \times 3 \times 5 = 30.$$

Разумеется, данный пример, показывающий разделение между combine3 и combine4, довольно путаный. Можно возразить, что поведение combine4 ближе соответствует цели описания функции. К сожалению, оптимизирующий компилятор не может анализировать ни условия, при которых могла бы использоваться функция, ни намерения программиста. Вместо этого, получив для компиляции combine3, он обязан в точ-

ности сохранить ее функциональность, даже если это будет означать генерирование неэффективного кода.

5.7. Общее описание современных процессоров

До сих пор применялись виды оптимизации, не полагающиеся на какие бы то ни было особенности целевой машины. Эти виды оптимизации просто сокращали количество обращений к процедурам и удаляли некоторые критичные "блокираторы оптимизации", создающие тупиковые ситуации и сложности для оптимизирующих компиляторов. Поскольку задачей является максимальное повышение производительности, необходимо начать рассмотрение оптимизаций, с большей степенью использующих средства, с помощью которых процессоры выполняют команды, а также возможности конкретных процессоров. Чтобы добиться максимально возможной производительности, необходимо провести подробнейший анализ программы, а также генерирование кода, настроенного на целевой процессор. Несмотря на это, можно применить некоторые базовые виды оптимизации, в результате чего можно добиться общего повышения производительности большого класса процессоров. Описываемые подробные результаты производительности могут не соответствовать каким-то другим машинам, однако общие принципы функционирования и оптимизации применимы к большому их диапазону.

Для понимания способов повышения производительности необходима простая операционная модель, демонстрирующая работу современных процессоров. Из-за большого числа транзисторов, которые можно интегрировать в одну микросхему, в современных микропроцессорах используются комплексные аппаратные средства, направленные на повышение производительности программы. Одним из результатов является то, что их фактическое функционирование радикально отличается от того, что пользователи видят при рассмотрении ассемблерных программ. На уровне компонующего кода кажется, будто команды выполняются по одной за раз, когда каждая из них выбирает значения из регистров или памяти, выполняет операцию и сохраняет результаты в ячейке памяти или регистра. На самом деле, в процессоре определенное количество команд выполняется одновременно. В некоторых проектах "в полете" могут находиться 80 и более команд. В обеспечении такого поведения параллельного выполнения, когда точно схватывается последовательная семантическая модель, необходимая для программы машинного уровня, задействованы сложные механизмы.

5.7.1. Общее функционирование

На рис. 5.4 показано очень упрощенное представление современного микропроцессора. В общем, описываемое в книге проектирование гипотетического процессора базируется на микроархитектуре Intel P6 [30] — основе для процессоров Intel Pentium Pro, Pentium II и Pentium III.

Блок формирования команд отвечает за считывание команд из памяти и генерирование последовательности примитивных операций. Затем блок выполнения выполняет эти операции и указывает на корректность прогнозирования ветвей.

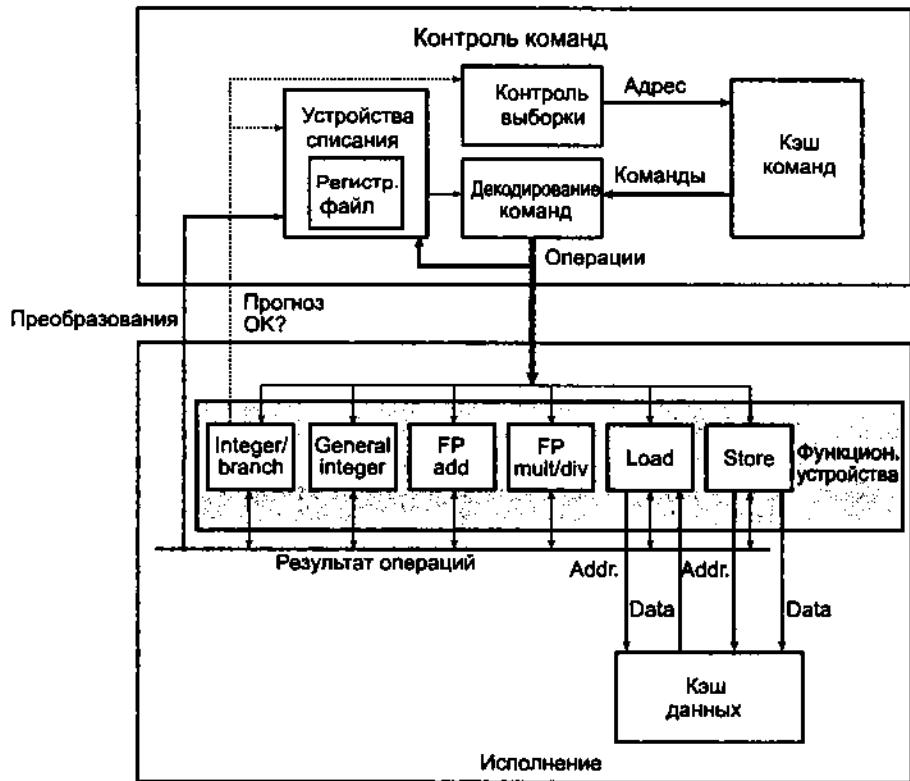


Рис. 5.4. Структурная схема современного процессора

Современный Pentium 4 имеет другую микроархитектуру, однако его общая структура сходна с представленной здесь. Микроархитектура Р6 является типичным примером высокотехнологичных процессоров, выпускаемых несколькими производителями с конца 90-х годов. В промышленной терминологии они носят название *суперскалярных*, что означает возможность выполнения множественных операций на каждом цикле синхронизации, и *нестандартных*, потому что в данном случае порядку выполнения команд не обязательно соответствовать их упорядочению в ассемблере. Общий проект состоит из двух частей: устройства формирования команд (ICU, Instruction control unit), отвечающего за считывание последовательности команд из памяти и генерирование из них набора примитивных операций для исполнения их с данными программ, и устройства исполнения (EU, Execution unit), выполняющего эти операции.

ICU считывает команды из кэша команд — особого быстродействующего блока памяти, содержащего последние команды, к которым осуществлялся доступ. Вообще говоря, ICU выполняет выборку команд гораздо раньше выполняемых в данный момент, поэтому оно имеет достаточно времени для их декодирования и отправления на EU. Впрочем, проблема заключается в том, что когда программа обращается

к ветви¹, то существует два направления, по которым она может начать двигаться. Ветвь может быть выбрана, когда управление передается мишени ветви. Либо ветвь не выбирается; в этом случае управление передается следующей команде в последовательности. В современных процессорах применяется методика, известная как *прогнозирование ветвей*, где делаются попытки "угадать", будет ветвь выбрана или нет, а также прогнозирования для ветви целевого адреса. С помощью методики, называемой *предположительным выполнением*, процессор начинает выборку и декодирование команд там, где в соответствии с прогнозом пройдет ветвь, и даже начинает выполнение этих операций до определения того, корректно спрогнозирована ветвь или нет. Если позже процессор обнаружит, что ветвь была спрогнозирована некорректно, он сбрасывает свое состояние до точки ветви и начинает выборку и выполнение команд в другом направлении. Еще более экзотической методикой явилось бы начало выборки и выполнения команд для обоих возможных направлений с последующим отбрасыванием результатов для неправильного направления. На сегодняшний день такой подход считается нерентабельным. Блок управления выборкой объединяет в себе прогнозирование ветвей для решения задачи определения того, какие команды подлежат выборке.

Логика декодирования команд использует фактические команды программы и преобразует их в набор примитивных операций. Каждая из этих операций выполняет некую простую вычислительную задачу, например сложение двух чисел, считывание данных из памяти или запись данных в память. Для машин с комплексными командами, например, для процессора IA32, команда может быть декодирована в переменное число операций. Разные процессоры различаются определенными тонкостями, однако здесь делается попытка описать типичную реализацию. В данной машине декодирование команды

```
addl %eax, %edx
```

дает в результате одну операцию сложения, тогда как декодирование команды

```
addl %eax, 4(%edx)
```

дает три операции: одну — для загрузки значения из памяти в процессор, вторую — для сложения загруженного значения со значением в регистре %eax, и третью — для сохранения результата в памяти. Такое декодирование делит команды, обеспечивая так называемое разделение труда между набором специализированных аппаратных устройств. Затем эти устройства могут параллельно выполнять различные части множественных команд. Для машин с простыми командами эти операции ближе соответствуют первоначальным командам.

EU получает операции из устройства выборки команд. Обычно оно может получить некоторое их количество за каждый цикл синхронизации. Эти операции отправляются на набор функциональных устройств, выполняющих фактические операции.

¹ Авторы используют термин "ветвь" специально для обращения к командам условного перехода. Другие команды, которые могут передавать управление на множественные пункты назначения, например возврат процедур и косвенные переходы, обеспечивают аналогичные сложности для работы процессора.

Функциональные устройства специализированы для обработки особых типов операций. На схеме показан типичный набор функциональных устройств. Он изображен по образцу и подобию последних процессоров от Intel. Устройства следующие:

- Целое число/Ветвь выполняет простые целочисленные операции (сложение, тестирование, сравнение, логические), также обрабатывает ветви.
- Общее целое число может обрабатывать все целочисленные операции, включая умножение и деление.
- Сложение чисел с плавающей точкой обрабатывает простые операции с плавающей точкой (сложение, преобразование формата).
- Умножение/деление чисел с плавающей точкой обрабатывает операции умножения и деления чисел с плавающей точкой. Более сложные команды с плавающей точкой, такие как трансцендентальные функции, преобразуются в последовательности операций.
- Загрузка обрабатывает операции считывания данных из памяти в процессор. Данное функциональное устройство имеет сумматор для выполнения адресных вычислений.
- Сохранение обрабатывает операции записи данных из процессора в память. Данное функциональное устройство имеет сумматор для выполнения адресных вычислений.

Как показано на рис. 5.4, устройства загрузки и сохранения получают доступ к памяти через *кэш данных* — быстродействующий блок памяти, содержащий самые последние значения данных, к которым осуществлялся доступ.

При выполнении операции оцениваются, однако окончательные результаты не сохраняются в регистрах программы или памяти для сохранения данных до тех пор, пока процессор не будет "уверен", что именно эти команды должны выполняться. Операции ветвления отправляются на EU не для определения того, куда должна быть направлена ветвь, а для определения корректности их прогнозирования. Если последнее не было корректным, EU отбрасывает результаты, вычисленные за пределами точки ветвления. Оно также сообщает устройству ветвления о том, что прогноз был некорректным и указывает корректное место назначения ветви. В этом случае устройство ветвления начинает выборку в новой ячейке. Такое некорректное прогнозирование очень отрицательно влияет на производительность. До выборки, декодирования и отправки новой команды на устройства выполнения проходит значительное количество времени. Более подробно этот вопрос рассматривается в разд. 5.12.

В рамках ICU устройство списания отслеживает текущую обработку и удостоверяется в том, что она удовлетворяет последовательной семантике программы машинного уровня. На рис. 5.4 показан *регистровый файл*, содержащий регистры целых чисел и чисел с плавающей точкой в качестве части устройства списания, потому что это устройство управляет обновлением этих регистров. Как только команда декодирована, информация о ней помещается в очередь с дисциплиной обработки FIFO ("первым пришел --- первым обслужен"). Эта информация остается в очереди до появления одного результата из двух. Во-первых, как только операция для команды завершается, и любые точки ветвления, приводящие к этой команде, подтверждаются как

спрогнозированные корректно, команда может списываться с любыми выполняемыми обновлениями регистров команд. С другой стороны, если какая-то из точек ветвления, приводящих к данной команде, спрогнозирована некорректно, тогда команда будет подавлена со сбросом любых вычисленных результатов. Из-за этого некорректные прогнозы не изменят состояния программы.

Как уже описывалось, любые обновления регистров команд имеют место только при списании команд, а это происходит только после того, как процессор может быть уверен в том, что любые ветви, приводящие к этой команде, были спрогнозированы корректно. Для ускорения передачи результатов от одной команды к другой между устройствами выполнения, обозначенными на схеме как результаты операции, происходит обмен большей частью этой информации. Стрелки на схеме показывают, что устройства выполнения могут отправлять результаты друг другу напрямую.

Наиболее широко используемый механизм управления передачей операндов между устройствами выполнения называется *переименованием регистров*. Когда команда, обновляющая регистр r , декодирована, генерируется метка t , обеспечивающая результату операции уникальный идентификатор. Запись (r, t) добавляется в таблицу, поддерживающую связь между каждым регистром программы и меткой для операции, обновляющей этот регистр. После того, как декодируется последующая команда, использующая в качестве операнда регистр r , операция, отправленная в устройство выполнения, будет содержать t в качестве источника для значения операнда. Когда некоторое устройство выполнения завершает первую операцию, оно генерирует результат (u, t) , указывая на то, что операция с меткой t выдала значение u . Затем любая операция, ожидающая t в качестве источника, будет использовать u в качестве исходного значения. С помощью такого механизма значения будут передаваться напрямую от одной операции к другой, а не будут записываться в регистровый файл и считываться из него. Таблица переименования содержит только записи для регистров, имеющих незавершенные операции записи. Когда декодированной команде требуется регистр r , а метки, связанной с этим регистром, нет, тогда operand извлекается непосредственно из регистрового файла. С помощью процесса переименования регистров вся последовательность операций может выполняться предположительно, несмотря на то, что регистры обновляются только после того, как процессор будет "уверен" в исходах ветвления.

Обработка с изменением последовательности

Впервые обработка с изменением последовательности была реализована в процессоре Control Data Corporation 6600 в 1964 г. Команды обрабатывались десятью разными функциональными устройствами, каждое из которых могло функционировать независимо. В то время эта машина с тактовой частотой в 10 МГц считалась самой лучшей для научных вычислений.

Компания IBM впервые реализовала обработку с изменением последовательности в процессоре IBM 360/91 в 1966 году, но только для выполнения команд с числами с плавающей точкой. На протяжении 25 лет обработка с изменением последовательности считалась экзотической технологией, реализованной только на машинах, от которых требовалась максимально возможная производительность, пока в 1990 г. IBM повторно представила на рынок линию рабочих станций RS/6000. Этот

проект стал фундаментом для линии IBM/Motorola PowerPC с моделью 601, представленной в 1993 г., которая стала первым микропроцессором, состоящим из одной микросхемы, в которой реализована обработка с изменением последовательности.

5.7.2. Производительность функционального устройства

В табл. 5.7 приведена производительность некоторых из основных операций процессора Intel Pentium III. Эти синхронизации также типичны и для других процессоров. Каждая операция характеризуется двумя счетчиками циклов: задержкой, указывающей на общее количество циклов, которое требуется функциональному устройству для завершения операции, и временем выдачи, указывающим количество циклов между последовательными независимыми операциями. Задержки входят в диапазон от одного цикла (для базовых целочисленных операций) до нескольких циклов (для загрузки, сохранения, умножения целых чисел и более общих операций с числами с плавающей точкой) и до множества циклов для деления и прочих комплексных операций.

Таблица 5.7. Производительность арифметических операций Pentium III

Операция	Задержка	Время выдачи
Сложение целых чисел	1	1
Умножение целых чисел	4	1*
Деление целых чисел	36	36
Сложение чисел с плавающей точкой	3	1
Умножение чисел с плавающей точкой	5	2
Деление чисел с плавающей точкой	38	38
Загрузка (результативное обращение в кэш)	3	1
Сохранение (результативное обращение в кэш)	3	1

Как показано в третьем столбце табл. 5.7, несколько функциональных устройств процессора объединены в *конвейер*; это означает, что очередная операция может начаться до полного завершения предыдущей. Время выдачи указывает количество циклов между последовательными операциями устройства. В конвейерном устройстве период выдачи короче, нежели задержка. Конвейерное функциональное устройство реализовано как серия этапов, каждый из которых выполняет часть операции. Например, типичный сумматор чисел с плавающей точкой содержит три этапа: один — для обработки значений экспонент, один — для сложения дробей, и один — для округления окончательного результата. Операции могут проходить этапы в плотной последовательности; очередная операция начинается, не дожидаясь завершения предыдущей.

Эту возможность можно использовать только при наличии последовательных и логически независимых операций, которые необходимо выполнить. Как указано, большинство устройств способно начать выполнение новой операции на каждом новом цикле синхронизации. Единственным исключением являются операции умножения чисел с плавающей точкой, требующие минимум двух циклов синхронизации между последовательными операциями и двух делителей, которые не являются частью конвейера.

Проектировщики циклов могут создавать функциональные устройства с целым диапазоном параметров производительности. Создание устройства с минимальным временем задержки или выдачи требует большего количества аппаратных средств, особенно для более сложных функций, таких как умножение и операции с числами с плавающей точкой. Поскольку предусмотренная на микросхеме площадь для этих устройств довольно ограничена, для достижения оптимальной общей производительности проектировщики CPU должны тщательно поддерживать равновесие между количеством функциональных устройств и их индивидуальной производительностью. Разработчики рассматривают множество эталонных тестовых программ и отводят большую часть ресурсов на выполнение наиболее критичных операций. В табл. 5.7 показано, что в проекте Pentium III умножение целых чисел и чисел с плавающей точкой, а также сложение последних рассматриваются как важные операции, несмотря на то, что для сокращения времени задержек и показанной высокой степени конвейеризации требуется довольно большое количество аппаратных средств. С другой стороны, операция деления используется не так часто, и ее сложно реализовать с малым временем задержки или выдачи, поэтому такие операции выполняются достаточно медленно.

5.7.3. Более пристальный взгляд на работу процессора

В качестве инструментального средства анализа производительности программы машинного уровня, выполняемой на современном процессоре, разработано более подробное текстовое представление для описания операций, генерированных декодером команд, а также графическое представление для отображения обработки операций функциональными устройствами. Ни одно из этих представлений не представляет в точности реализации специфичного процессора, который можно встретить в реальной жизни. Это просто методы, помогающие понять, как процессор использует преимущество параллелизма и прогнозирования ветвей при выполнении программы.

Перевод команд в операции

Разработанное представление иллюстрируется на примере `combine4` (листинг 5.15) — до сих пор наиболее быстродействующего кода. Упор делается только на вычисления, выполненные циклом, поскольку это — доминирующий фактор производительности больших векторов. Рассматриваются случаи целочисленных данных с умножением и сложением, используемых в качестве комбинирующих операций. Скомпилированный код для данного цикла с умножением состоит из четырех команд. В данном коде регистр `%eax` удерживает указатель `data`, `%edx` удерживает `i`, `%ecx` удерживает `x`, а `%esi` — `length`.

Листинг 5.16. Наиболее быстродействующий код

```

combine4: type=INT, OPER = *
dest in %eax, x in %ecx, i in %edx, length in %esi

1 .L24          loop:
2 imull(%eax, %edx, 4), %ecx Умножение x на data [i]
3 incl %edx        i++
4 cmpl %esi, %edx Сравнение i:length
5 jl.L24          If <, goto loop

```

Всякий раз при выполнении процессором цикла декодер команд преобразует четыре команды в последовательность операций для устройства выполнения. При первой итерации, когда $i = 0$, рассматриваемая гипотетическая машина выдает следующую последовательность операций (табл. 5.8).

Таблица 5.8. Устройство выполнения

Ассемблерные команды	Операции устройства выполнения
.L24: imull (%eax, %edx, 4), %ecx incl %edx cmpb %esi, %edx jl .L24	load (%eax, %edx.0, 4) t.1 imull t.1, %ecx.0 %ecx.1 incl %edx.0 %edx.1 cmpl %esi, %edx.1 cc.1 jl-taken cc.1

В данном преобразовании ссылка на ячейку памяти была преобразована из множественной команды в явную команду `load`, считывающую данные из памяти в процессор. Также значениям, изменяющимся с каждой итерацией, присваиваются ярлыки **операндов**. Эти ярлыки представляют собой стилизованную версию меток, генерированных переименованием регистров. Таким образом, значение в регистре `%ecx` идентифицируется ярлыком `%ecx.0` в начале цикла, и `%ecx.1` — после обновления. Значения регистров, не изменяющиеся от одной итерации к другой, принимаются непосредственно из регистрового файла во время декодирования. Здесь также представлен ярлык `t.1` для обозначения значения, считанного операцией `load` и переданного операции `imull`, и явно указывается место назначения операции. Таким образом, пара операций

```

load (%eax, %edx.0, 4) → t.1
imull t.1, %ecx.0           → %ecx.1

```

указывает на то, что сначала процессор выполняет операцию `load`, вычисляя адрес, используя значение `%eax` (не изменяющееся во время цикла), а также указывает значение, сохраненное в регистре `%edx`, в начале цикла. Это дает временное значение,

обозначаемое t.1. Затем операция умножения берет это значение и значение %edx в начале цикла и создает новое значение для регистра %edx. Как показывает данный пример, метки могут ассоциироваться с промежуточными значениями, которые никогда не записываются в регистровый файл. Операция

```
incl %edx.0 - %edx.1
```

указывает на то, что операция приращения добавляет 1 к значению %edx в начале цикла для создания нового значения для данного регистра. Операция

```
cmpl %esi, %edx.1 - cc.1
```

указывает на то, что операция сравнения (выполняемая любой целой единицей) сравнивает значение в %esi (которое не меняется в цикле) с вновь вычисленным значением для регистра %edx. Затем задаются коды условий, идентифицируемые с явно определенным ярлыком cc.1. Как показывает данный пример, для отслеживания изменений в регистрах кода условий процессор может использовать переименование. Наконец, команда перехода спрогнозирована как выбранная. Операция перехода

```
jl-taken cc.1
```

проверяет, указывают ли вновь вычисленные значения для кода условий (cc.1) на правильность выбора. Если нет, тогда операция перехода сигнализирует ICU о начале выборки команды с команды, следующей за jl. Для упрощения представления любая информация о возможных местах назначения переходов опускается. На практике процессор должен следить за местом назначения непрогнозируемого направления для того, чтобы начать с него выборку в случае некорректности прогноза.

Как показывает данный пример преобразования, рассматриваемые операции во многом воспроизводят структуру команд языка компоновщика, за исключением того, что они обращаются к своим исходным операциям и операциям назначения посредством ярлыков, обозначающих различные экземпляры регистров. В настоящем аппаратном обеспечении переименование регистров динамически присваивает метки для обозначения различных значений. Метки — это, скорее, комбинации битов, а не символические имена, такие как %edx.1, однако они служат той же цели.

Обработка операций устройством выполнения

На рис. 5.5 показаны операции в двух формах: сгенерированные декодером команд и представленные в виде *вычислительного графа*, где они обозначены рамками с закругленными углами; стрелки указывают обмен данными между операциями. Стрелки показаны только для operandов, изменяющихся от одной итерации к другой, поскольку только эти значения передаются напрямую между функциональными устройствами.

Высота каждой рамки оператора указывает количество циклов, необходимое для выполнения операции, т. е. время задержки выполнения той или иной функции. В этом случае функция умножения целого числа *imul* требует четырех циклов, загрузка — трех, а прочие операции — одного. При демонстрации синхронизации цикла рамки расположены вертикально для обозначения времени выполнения операций; значения времени при этом увеличиваются по нисходящей. На схеме видно, что пять операций

цикла образуют две параллельные цепи, указывающие две серии вычислений, которые должны быть выполнены последовательно. Цепь слева обрабатывает данные, считывая элемент массива из памяти и умножая его на накопленное произведение. Цепь справа обрабатывает указатель цикла *i* его приращением и последующим сравнением с *length*. Операция перехода проверяет результат этого сравнения для уверенности в том, что ветвь спрогнозирована корректно. Обратите внимание на то, что из рамки операции перехода нет исходящих стрелок. Если ветвь спрогнозирована корректно, то никакой другой обработки не требуется. Если ветвь спрогнозирована некорректно, функциональное устройство ветвления сообщит об этом в устройство управления выборкой команды, и последнее выполнит корректирующее действие. В любом случае, все прочие операции не зависят от результата операции перехода.

Операции выполнения	
load (%eax, %edx.0, 4)	t.1
imull t.1, %ecx.0	%ecx.1
incl %edx.0	%edx.1
cmpl %esi, %edx.1	cc.1
jl-taken cc.1	

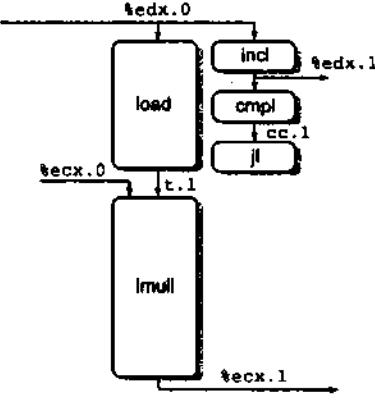


Рис. 5.5. Операции первой итерации умножения целых чисел

По сравнению с умножением единственным изменением является то, что операции сложения достаточно только одного цикла (рис. 5.6).

Операции выполнения	
load (%eax, %edx.0, 4)	t.1
addl t.1, %ecx.0	%ecx.1
incl %edx.0	%edx.1
cmpl %esi, %edx.1	cc.1
jl-taken cc.1	

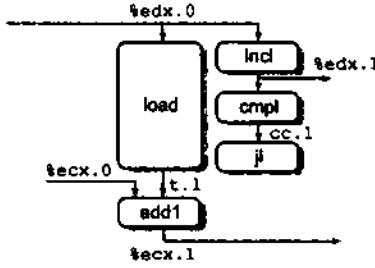


Рис. 5.6. Операции сложения целых чисел

Планирование операций с неограниченными ресурсами

Чтобы понять, как процессор выполнял бы серию итераций, представьте себе процессор с неограниченным количеством функциональных устройств и с безупречным прогнозированием ветвей. Тогда выполнение каждой операции могло бы начинаться

тогда, когда становятся доступными ее операнды данных. Производительность такого процессора была бы ограничена только задержками и пропускной способностью функциональных устройства и зависимости по данным в программе. На рис. 5.7 показан вычислительный граф для первых трех итераций цикла в `combine4` с целочисленным умножением на такой машине. Для каждой итерации существует набор из пяти операций с той же конфигурацией, что и показанные на рис. 5.5, с соответствующими изменениями в ярлыках operandов. Стрелки от операторов одной итерации к операторам другой показывают зависимости по данным между разными итерациями.

Каждый оператор размещен вертикально в самой высокой позиции из возможных, с тем ограничением, что ни одна из стрелок не может указывать вверх, поскольку это бы обозначало прохождение потока информации в обратном направлении во времени. Следовательно, операция `load` одной итерации могла бы начаться, как только операция `incl` предыдущей итерации сгенерирует обновленное значение указателя цикла.

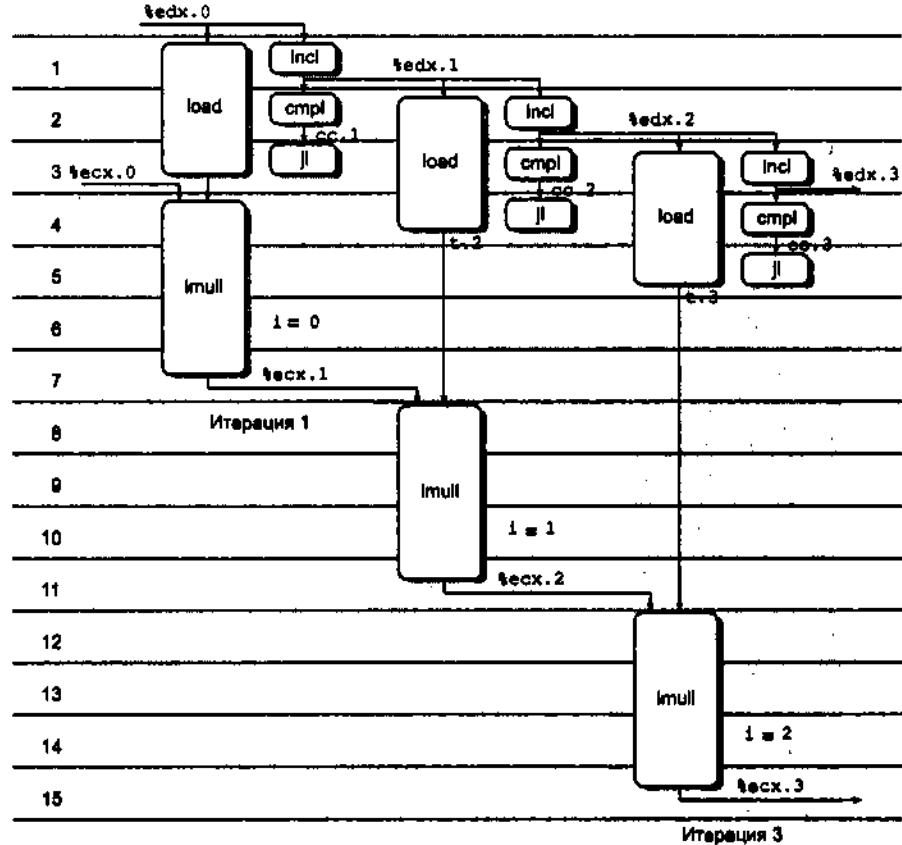


Рис. 5.7. Планирование операций для умножения целых чисел с неограниченным числом устройств выполнения

Вычислительный граф показывает параллельное выполнение операций устройством выполнения. На каждом цикле все операции на одной горизонтальной строке графа выполняются параллельно. Данный график также демонстрирует выполнение с изменением последовательности, предположительное. Например, операция `incl` в одной итерации выполняется до того, как начнется выполнение команды `j1` предыдущей итерации. Здесь также можно наблюдать эффект конвейера. Каждая итерация требует минимум семь циклов от начала и до конца, однако последовательные итерации завершаются каждые четыре цикла. Таким образом, эффективная скорость обработки составляет одну итерацию каждые четыре цикла, что дает СРЕ в 4.0.

Задержка на четыре цикла операции умножения целых чисел ограничивает производительность процессора для этой программы. Каждая операция `imull` должна дождаться завершения предыдущей операции, поскольку для начала `imull` ей требуется результат умножения. На показанной схеме операции умножения начинаются на циклах 4, 8 и 12. С каждой последующей итерацией новая операция умножения начинается на каждом четвертом цикле.

На рис. 5.8 показаны первые четыре итерации `combine4` для сложения целых чисел на машине с неограниченным числом функциональных устройств. С помощью одного цикла комбинирующей операции программа может достичь СРЕ в 1.0. Читатель обнаружит, что в ходе итераций устройство выполнения осуществляет части семи операций на каждом цикле синхронизации. Например, в цикле 4 видно, что машина выполняет `addl` для итерации 1; различные части операций `load` для итераций 2, 3 и 4; `j1` для итерации 2; `cmpl` для итерации 3 и `incl` для итерации 4.

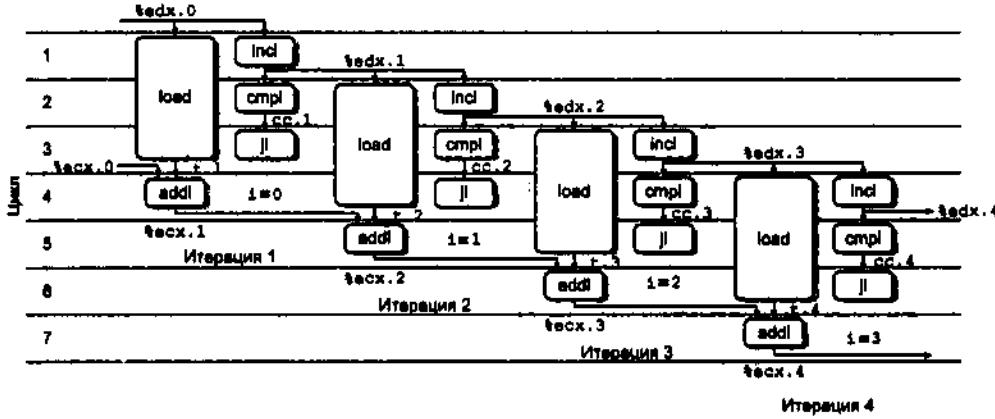


Рис. 5.8. Планирование операций для сложения целых чисел с неограниченным числом ограничений ресурсов

Планирование операций с ограничениями ресурсов

Разумеется, настоящий процессор имеет фиксированное число функциональных устройств. В отличие от приведенных выше примеров, где производительность была ограничена только зависимостями по данным и задержками функциональных уст-

ройств, здесь производительность ограничивается еще и ресурсами. В частности, рассматриваемый на рис. 5.8 процессор имеет только два устройства, способных выполнять целочисленные операции и операции ветвления. В противовес этому, граф на рис. 5.7 имеет три таких параллельных операций в цикле 3 и четыре параллельных операции в цикле 4.

На рис. 5.9 показано планирование операций для `combine4` с умножением целых чисел на процессоре с ограниченными ресурсами. Предполагается, что устройство общего целого числа и устройство ветви/целого числа могут начать новую операцию на каждом цикле синхронизации. Можно параллельно выполнять более двух целочисленных операций или операций ветвления, как показано в цикле 6, потому что операция `imull` находится к этой точке в третьем цикле.

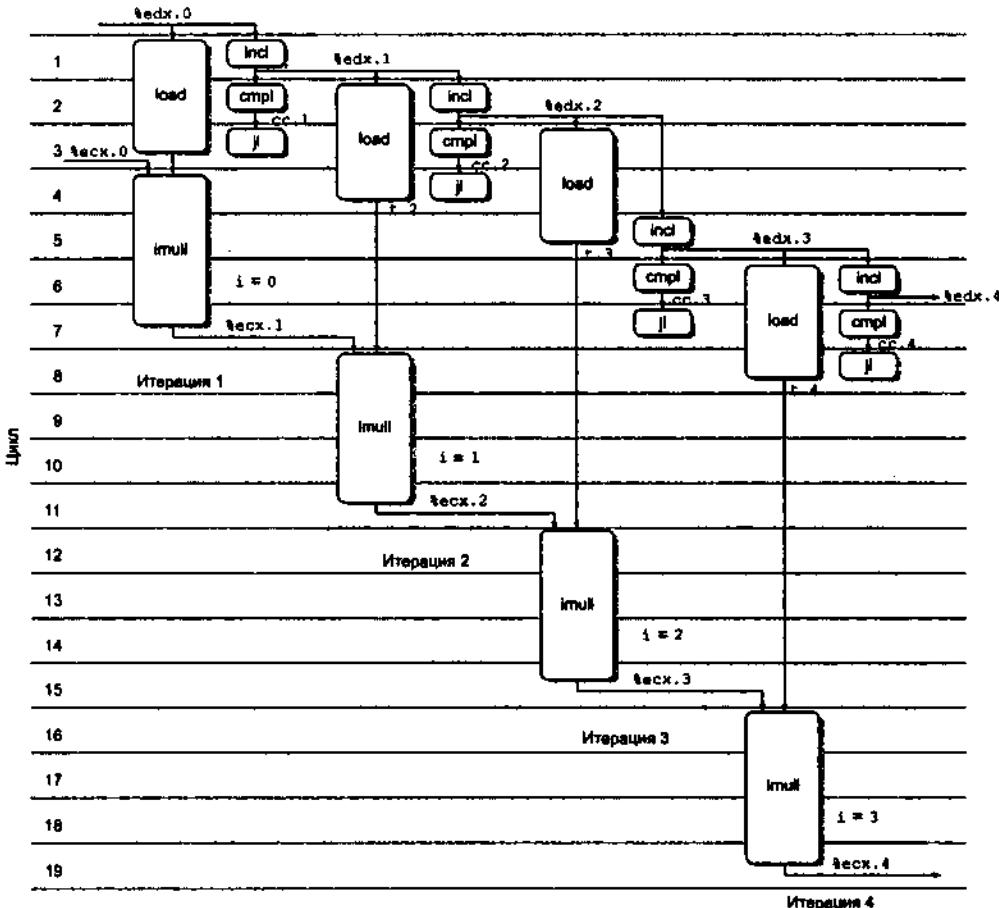


Рис. 5.9. Планирование операций для умножения целых чисел с фактическим числом ограничений ресурсов

С ограниченными ресурсами рассматриваемый процессор должен иметь на вооружении определенную *стратегию планирования*, определяющую операцию для выполнения при наличии выбора. Например, в цикле 3 графа на рис. 5.7 показаны три выполняемые целочисленные операции: *j1* итерации 1, *cmpl* итерации 2 и *incl* итерации 3. Для рис. 5.9 выполнение одной из этих операций необходимо задержать. Это осуществляется прослеживанием *программного порядка* операций, т. е. порядка, в котором будут выполняться операции при выполнении программы машинного уровня в строгой последовательности. Затем для операций устанавливается приоритет, в соответствии с их программным порядком. В данном примере задержана будет операция *incl*, поскольку любая операция итерации 3 появляется по программному порядку позже, нежели операции итераций 1 и 2. Подобным же образом в цикле 4 приоритет будет отдан операции *imull* итерации 1 и *j1* итерации 2, а не операции *incl* итерации 3.

Для данного примера ограниченное число функциональных устройств не замедляет выполнение программы. Производительность по-прежнему ограничивается задержкой на четыре цикла операции умножения целых чисел.

В случае со сложением целых чисел ограничения по ресурсам однозначно ограничивают производительность программы. Каждая итерация требует четыре целочисленные операции или операции ветвления, и для этих операций имеется всего два функциональных устройства. Следовательно, сложно надеяться на то, что скорость обработки можно повысить более чем на 2 цикла на итерацию. При создании графа

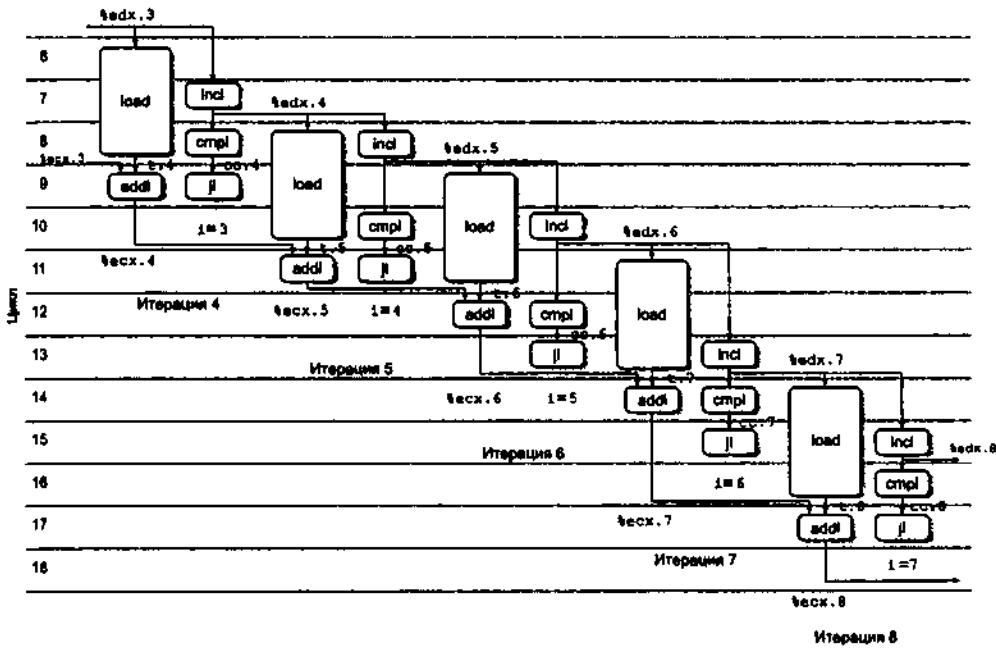


Рис. 5.10. Планирование операций для сложения целых чисел с фактическим числом ограничений ресурсов

множественных итераций `combine4` для сложения целых чисел возникает интересная структура. На рис. 5.10 показано планирование операций для итераций 4—8. Этот диапазон итераций выбран потому, что он демонстрирует регулярный образ синхронизации операций. Обратите внимание, что синхронизация всех операций в итерациях 4 и 8 идентична, за исключением того, что операции в итерации 8 выполняются на восемь циклов позже. С началом итераций структуры для итераций 4—7 будут повторяться. Таким образом, четыре итерации завершаются каждые восемь циклов, достигая оптимального СРЕ в 2.0.

Выводы о производительности `combine4`

Теперь можно рассмотреть измеренную производительность `combine4` для всех четырех комбинаций типов данных и комбинирующих операций (табл. 5.9).

Таблица 5.9. Типы данных

Функция	Метод	Целое число		Число с плавающей точкой	
		+	*	+	*
<code>combine4</code>	Накопление во временной переменной	2.00	4.00	3.00	5.00

За исключением сложения целых чисел эти значения времени цикла практически совпадают с задержкой комбинирующей операции, как показано в табл. 5.7. Преобразования в этой точке уменьшили значение СРЕ до точки, когда время на выполнение комбинирующей операции становится ограничивающим фактором.

В случае со сложением целых чисел становится понятно, что ограниченное количество функциональных устройств для целочисленных операций и операций ветвления ограничивает достижимую производительность. С четырьмя такими операциями на одну итерацию и всего лишь двумя функциональными устройствами сложно ожидать от программы более быстрого выполнения, нежели 2 цикла на одну итерацию.

Вообще говоря, производительность процессора ограничивается тремя типами факторов. Во-первых, зависимости по данным в программе вызывают задержку выполнения некоторых операций до тех пор, пока не будут вычислены их операнды. Поскольку функциональные устройства имеют задержку в один или более циклов, этим устанавливается нижняя граница на количество циклов, за которое может быть выполнена данная последовательность операций. Во-вторых, сдерживающие факторы ресурсов ограничивают число операций, которые могут быть выполнены за определенное установленное время. Как уже отмечалось, ограниченное число функциональных устройств является одним из сдерживающих факторов по ресурсам. В число других ограничений входят степень конвейеризации, выполняемая функциональными устройствами, а также ограничения других ресурсов в ICU и EU. Например, за каждый цикл синхронизации Intel Pentium III может декодировать три команды. Наконец, успех логики прогнозирования ветвей ограничивает степень, в которой про-

цессор может намного опережать поток команд для поддержания беспрерывной работы устройства выполнения. Результатом некорректного прогноза станет значительная задержка, вызывающая перезапуск процессора в корректном местоположении (ячейке).

5.8. Снижение непроизводительных издержек циклов

Производительность `combine4` для операции сложения целых чисел ограничена тем фактом, что в каждой итерации содержится четыре команды с двумя функциональными устройствами, которые могут их выполнить. Только одна из этих четырех команд работает с данными программы. Другие являются частью непроизводительных издержек цикла вычисления указателя цикла и тестирования состояния цикла (листинг 5.16).

Листинг 5.16. Развертка цикла на 3 элемента

```
1  /*          */
2 void combine5 (vec_ptr v, data_t *dest)
3 {
4     int length = vec_length (v);
5     int limit = length-2;
6     data_t *data = get_vec_start (v);
7     data_t x = IDENT;
8     int i;
9
10    /* Комбинирование 3 элементов за единицу времени */
11    for (i = 0; i < limit; i+=3)  {
12        x = x OPER data [i]  OPER data [i + 1]  OPER data [i + 2];
13    }
14
15    /* Окончание всех оставшихся элементов */
16    for (; i < length; i++)  {
17        x = x OPER data [i]
18    }
19    *dest = x;
20 }
```

Воздействие непроизводительных издержек можно снизить выполнением большего количества операций с данными в каждой итерации, используя методику под названием *развертка цикла*. Идея заключается в доступе и комбинировании элементов множественного массива в рамках одной итерации. Результирующая программа требует меньшего количества итераций, что приводит к снижению непроизводительных издержек цикла.

В листинге 5.16 показана версия комбинирующего кода с использованием трехсторонней развертки цикла. Первый цикл проходит за один раз массив из трех элементов. То есть, указатель цикла i увеличивается на 3 при каждой итерации, а комбинирующая операция применяется к массиву элементов $i, i + 1$ и $i + 2$ в одной итерации.

Вообще, длина вектора не будет кратна 3. Необходимо, чтобы код работал корректно с произвольными длинами вектора. Это требование можно объяснить двояко. Сначала нужно убедиться в том, что первый цикл не выходит за границы массива. Для вектора длиной n предел цикла устанавливается на $n - 2$. Затем нужно убедиться, что цикл будет выполняться, только если указатель цикла i удовлетворяет $i < n - 2$, откуда указатель максимального массива $i + 2$ будет удовлетворять уравнению $i + 2 < n - 2$, т. е. $i < (n - 2) + 2 = n$. Вообще говоря, если цикл развернут на k , верхний предел устанавливается на $n - k + 1$. Тогда указатель максимального цикла $i + k - 1$ будет меньше n . Кроме этого, для прохода нескольких конечных элементов вектора за единицу времени добавляется второй цикл. Тело этого цикла будет выполнено от 0 до 2 раз.

Для лучшего понимания производительности кода с разверткой цикла рассмотрим компонующий автокод для внутреннего цикла и его преобразование в операции (табл. 5.10).

Таблица 5.10. Внутренний цикл

Ассемблерные команды	Операции устройства выполнения
.L49:	
addl (%eax, %edx, 4), %ecx	load (%eax, %edx.0, 4) t.1a
addl 4 (%eax, %edx, 4), %ecx	addl t.1a, %ecx.0c %ecx.1a
addl 8 (%eax, %edx, 4), %ecx	load 4 (%eax, %edx.0, 4) t.1b
addl t.1b, %ecx.1a	addl t.1b, %ecx.1a %ecx.1b
addl 8 (%eax, %edx, 4)	load 8 (%eax, %edx.0, 4) t.1c
addl t.1c, %ecx.1b	addl t.1c, %ecx.1b %ecx.1c
addl %edx, 3	addl, %edx.0, 3 %edx.1
cmpl %esi, %edx	cmpl %esi, %edx.1 cc.1
jl .L49	jl-taken cc.1

Как упоминалось ранее, развертка циклов сама по себе повысит производительность кода только в случае с целочисленной суммой, потому что другие случаи ограничены задержками функциональных устройств. Для целочисленной суммы трехсторонняя развертка позволяет скомбинировать три элемента с шестью целочисленными операциями или операциями ветвления, как показано на рис. 5.11. Имея два функциональных устройства для этих операций, можно потенциально добиться СРЕ в 1.0. На рис. 5.12 показано, что по достижении итерации 3 ($i = 6$) операции будут придерживаться регулярной структуры. Операции итерации 4 ($i = 9$) имеют те же показатели синхронизации, но сдвинуты на три цикла. Это даст СРЕ 1.0.

Измерения, выполненные для данной функции, демонстрируют СРЕ 1.33, т. е. на каждую итерацию требуется четыре цикла. Очевидно, что определенный фактор

ограничения по ресурсам, не учтённый при анализе, задерживает вычисления на один дополнительный цикл на каждую итерацию. Несмотря на это, данная производительность демонстрирует преимущество над кодом, в котором не использовалась развертка циклов.

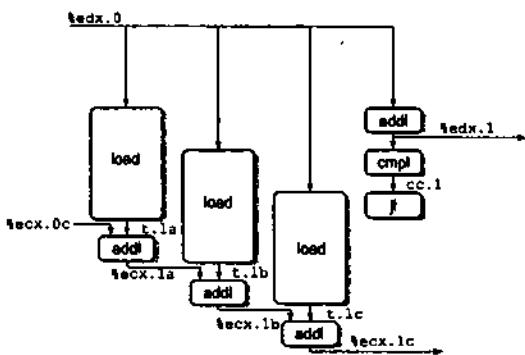
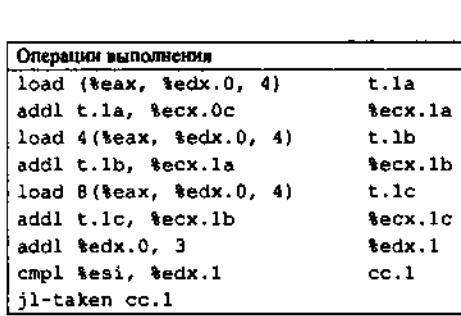


Рис. 5.11. Операции первой итерации внутреннего цикла трёхстороннего развернутого целочисленного сложения

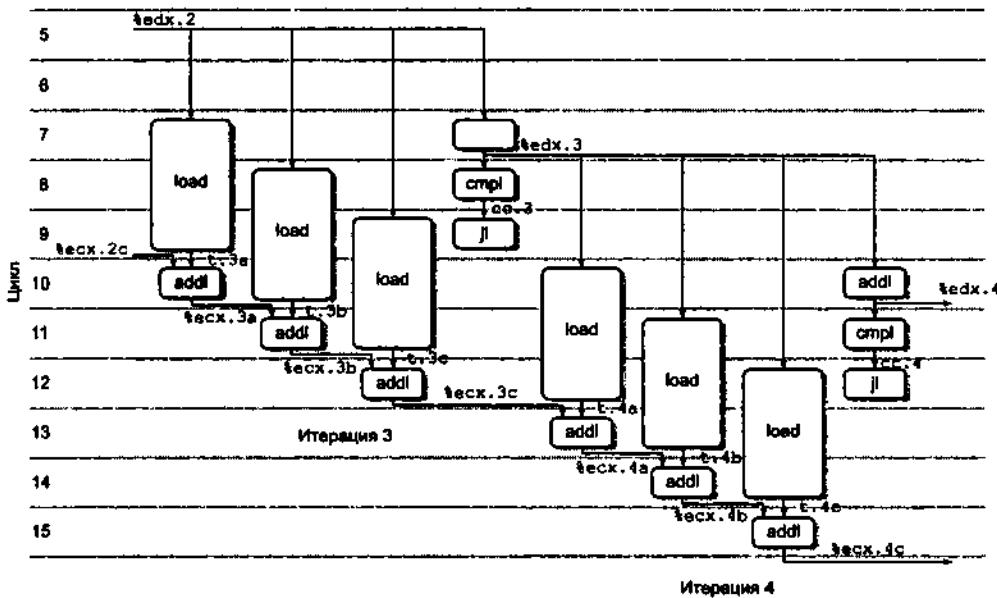


Рис. 5.12. Планирование операций для трехсторонней развернутой целочисленной суммы с ограниченными ресурсами

Измерение производительности для разных степеней развертки дает следующие значения СРЕ (табл. 5.11).

Таблица 5.11. Измерения производительности

Длина вектора	Степень развертки					
	1	2	3	4	8	16
CPE	2.00	1.50	1.33	1.50	1.25	1.06

Как показывают данные измерения, развертка цикла может понизить CPE. Если цикл развернут с коэффициентом 2, тогда каждая итерация основного цикла требует трех циклов синхронизации, что дает CPE в 1.5. По мере повышения степени развертки производительность, как правило, повышается, приближаясь к теоретическому пределу CPE в 1.0. Интересно будет отметить, что такое повышение не монотонно: развертка с коэффициентом 3 дает большую производительность, нежели развертка с 4. Очевидно, что планирование операций на устройствах выполнения менее эффективно для последнего случая.

Приведенные здесь расчеты CPE не учитывают такие непроизводительные факторы, как затраты на вызов процедур и на установку цикла. С разверткой цикла представляется новый источник непроизводительных издержек — необходимость завершения всех остающихся элементов, когда длину вектора нельзя разделить на степень развертки. Для исследования влияния непроизводительных издержек измеряется чистое значение CPE для различных длин векторов. Чистое CPE вычисляется как общее число циклов, необходимое для процедуры, деленное на число элементов. Для разных степеней развертки и для двух различных длин векторов получаем следующие данные (табл. 5.12):

Таблица 5.12. Измерения производительности

Длина вектора	Степень развертки					
	1	2	3	4	8	16
CPE	2.00	1.50	1.33	1.50	1.25	1.06
31 чист. CPE	4.02	3.57	3.39	3.84	3.91	3.66
1024 чист. CPE	2.06	1.56	1.40	1.56	1.31	1.12

Различие между CPE и чистым CPE для длинных векторов минимально, как видно в измерениях для длины 1024, однако его влияние значительно для коротких векторов, судя по измерениям, сделанным для длины 31. Измерения чистого CPE для вектора с длиной 31 демонстрируют один недостаток развертки цикла. Даже при отсутствии развертки значение чистого CPE (4.02) значительно выше, нежели 2.06 (измерения для длинных векторов). Непроизводительные издержки от запуска и завершения цикла намного возрастают, когда цикл выполняется меньшее количество раз. Кроме того, выгода от развертки цикла здесь менее значительна. Развернутый код должен запускать и останавливать два цикла, и он должен завершать конечные элементы по одному за каждый раз. Непроизводительные издержки уменьшаются с повышением

развертки циклов, тогда как количество операций, выполняемых в конечном цикле, увеличивается. Если длина вектора составляет 1024, производительность, как правило, повышается по мере повышения степени развертки. Если длина вектора — 31, то наилучшая производительность достигается разверткой цикла только на коэффициент 3.

Второй недостаток развертки циклов заключается в том, что она увеличивает объем генерируемого объектного кода. Объектный код для `combine4` требует 63 байта, а объектный код с циклом, развернутым на коэффициент 16, требует 142 байта. В этом случае создается впечатление, что это не такая уж и большая цена за код, выполняющийся почти в два раза быстрее. Впрочем, в других случаях оптимальная позиция в данном компромиссе времени и пространства не совсем ясна.

Выполнение развертки циклов компилятором

Развертку циклов можно легко выполнить с помощью компилятора. Многие из них выполняют ее в качестве рутинной процедуры всякий раз при установке достаточного высокого уровня оптимизации (например, с флагком оптимизации `-O2`). GCC выполняет развертку циклов при вызове с `-funroll-loops` в командной строке.

5.9. Преобразование в код указателя

Перед тем как продолжить описание, попробуем выполнить еще одно преобразование, которое иногда может повышать производительность программы (впрочем, за счет читаемости последней). Одной из уникальных особенностей С является способность к созданию указателей на произвольные объекты программы. Фактически, арифметический указатель обладает тесной связью со ссылками на массив. Комбинация арифметического указателя и ссылок, представленная выражением `* (a + i)`, в точности эквивалентна ссылке на массив `a[i]`. Иногда производительность программы можно повысить использованием указателей, а не массивов.

В листингах 5.17 и 5.18 показан пример преобразования процедур `combine4` и `combine5` в код указателя с созданием процедур `combine4p` и `combine5p` соответственно. Вместо того чтобы поддерживать указатель `data` фиксированным в начале вектора, он перемещается с каждой итерацией. Затем на элементы вектора делаются ссылки фиксированным смещением указателя `data` (между 0 и 2). Еще более важным является то, что переменную итерации `i` можно убрать из процедуры. Для определения того, когда цикл должен завершиться, указатель `dend` вычисляется как верхняя граница указателя `data`. Сравнение производительности этих процедур с их аналогами-массивами дает смешанные результаты.

Листинг 5.17. Преобразование процедуры `combine4`

```
1 /* Накопление в локальной переменной, версия указателя */
2 void combine4 (vec_ptr v, data_t *dest)
3 {
4     int length = vec_length (v);
```

```

5 data_t *data = get_vec_start (v);
6 data_t *dend = data+length;
7 data_t x = IDENT;
8
9 for (; data < dend; data++)
10 x = x OPER *data;
11 *dest = x;
12 }
```

Листинг 5.18. Преобразование процедуры combine5

```

1 /* Разворотка цикла на 3, версия указателя */
2 void combine5 (vec_ptr v, data_t *dest)
3 {
4 data_t *data = get_vec_start (v);
5 data_t *dend = data+vec_length (v);
6 data_t *dlimit = dend-2;
7 data_t x = IDENT;
8
9 /* Комбинирование 3 элемента за один раз */
10 for (; data < dlimit; data += 3)  {
11 x = x OPER data [0] OPER data [1] OPER data [2];
12 }
13
14 /* Завершение всех оставшихся элементов */
15 for (; data < dend; data++)  {
16 x = x OPER data [0]
17 }
18 *dest = x
19 }
```

В некоторых случаях это может привести к повышению производительности (табл. 5.13).

Для большинства случаев версии массива и указателя обладают абсолютно одинаковой производительностью. С указателем код CPE для целочисленной суммы без развертки фактически ухудшается на один цикл. Такой результат в некоторой степени удивительный, поскольку внутренние циклы для версий указателя и массива очень похожи (листинг 5.19). Трудно предположить, почему код указателя требует дополнительного цикла синхронизации на итерацию. Таким же таинственным способом версии процедур с четырехсторонней разверткой цикла дают улучшение кода указателя на один цикл по каждой итерации, что дает CPE в 1.25 (пять циклов на итерацию), а не 1.5 (шесть циклов на итерацию).

В рассматриваемом примере относительная производительность кода указателя против кода массива зависит от машины, компилятора и даже от конкретной процедуры.

Читатели уже встречались с компиляторами, применяющими к коду массива улучшенную оптимизацию, а к коду указателя — минимальную. Ради сохранения читаемости обычно предпочтителен код массива.

Таблица 5.13. Измерения производительности

Функция	Метод	Целое число		Число с плавающей точкой	
		+	*	+	*
combine4	Накопление во временной переменной	2.00	4.00	3.00	5.00
combine4p	Версия указателя	3.00	4.00	3.00	5.00
combine5	Развертка цикла ×3	1.33	4.00	3.00	5.00
combine5p	Версия указателя	1.33	4.00	3.00	5.00
combine5x4	Развертка цикла ×4	1.50	4.00	3.00	5.00
combine5px4	Версия указателя	1.25	4.00	3.00	5.00

Листинг 5.19. Аномалия производительности указателя кода

```

combine4: type = INT, OPER = '+'
    data in %eax, x in %ecx, i in %edx, length in %esi

1 .L24:loop:
2     addl(%eax, %edx, 4), %ecx Прибавление data [i] к x
3     incl %edx i++
4     cmpl %esi, %edx Сравнение i:length
5     jl.L24If <, goto loop

combine4p: type = INT, OPER = '+'
    data in %eax, x in %ecx, dend in %edx

1 .L30:loop:
2     addl(%eax), %ecx Прибавление data [0] к x
3     addl $4, %eaxdata++
4     cmpl %edx, %eax Сравнение data:dend
5     jb.L30If <, goto loop

```

Несмотря на то, что две программы очень похожи по своей структуре, код массива требует двух циклов синхронизации на итерацию, а код указателя — три.

УПРАЖНЕНИЕ 5.4

Временами GCC выполняет свою версию преобразования кода массива в код указателя. Например, с целочисленными данными и сложением в качестве комбинирующей операции GCC генерирует следующий код для внутреннего цикла варианта `combine5`, использующий восемистороннюю развертку цикла:

```

1 .L6:
2 addl (%eax), %edx
3 addl 4 (%eax), %edx
4 addl 8 (%eax), %edx
5 addl 12 (%eax), %edx
6 addl 16 (%eax), %edx
7 addl 20 (%eax), %edx
8 addl 24 (%eax), %edx
9 addl 28(%eax), %edx
10 addl $32, %eax
11 addl $8, %ecx
12 cmpl %esi, %ecx
13 j1 .L6

```

Обратите внимание на то, как регистр `%eax` увеличивается на 32 в каждой итерации.

Напишите код C для процедуры `combine5px8`, демонстрирующий вычисления указателей, переменных циклов и условий завершения. Покажите общую форму с произвольными данными и комбинирующей операции в стиле листинга 5.16. Опишите, чем он отличается от кода указателя, написанного вручную (листинги 5.17—5.18).

5.10. Повышение параллелизма

На данном этапе программы ограничены задержками функциональных устройств. Однако, как показано в третьем столбце табл. 5.7, несколько функциональных устройств процессора объединены в *конвейер*. Это означает, что они могут начать новую операцию до завершения предыдущей. Рассматриваемый код не может пользоваться такой возможностью даже с разверткой цикла, поскольку значение накапливается как единая переменная *x*. Нельзя вычислить новое значение *x* до завершения предыдущего вычисления. В результате этого процессор остановится в ожидании завершения текущей операции и начала новой. Это ограничение наглядно показано на рис. 5.7 и 5.9. Даже с неограниченными ресурсами процессора множитель может выдать новые результаты только каждые четыре цикла синхронизации. Подобные же ограничения имеют место со сложением чисел с плавающей точкой (три цикла) и умножением (пять циклов).

5.10.1. Разбиение циклов

Для ассоциативной и коммутативной комбинирующей операции, например, для сложения или умножения целых чисел, производительность можно повысить путем разбиения набора комбинирующих операций на две или более частей и конечного ком-

бинирования результатов. Например, пусть P_n обозначает произведение элементов a_0, a_1, \dots, a_{n-1} :

$$P_n = \prod_{i=0}^{n-1} a_i$$

Если предположить, что n — четное число, эту же формулу можно записать как

$$P_n = PE_n \times PO_n,$$

где PE_n — произведение элементов с четными индексами, а PO_n — произведение элементов с нечетными индексами:

$$PE_n = \prod_{i=0}^{n/2-2} a_{2i}$$

$$PO_n = \prod_{i=0}^{n/2-2} a_{2i+1}$$

Листинг 5.20 содержит код, использующий этот метод. Он применяет как двухстороннюю развертку цикла для комбинирования большего количества элементов на итерацию, так и двухсторонний параллелизм, накапливающий элементы с четным индексом в переменной $x0$ и элементы с нечетным индексом в переменной $x1$. Как и ранее, второй цикл включен для накапливания любых оставшихся элементов массива на случай, если длина вектора не является кратной двойке. Затем, для вычисления окончательного результата к $x0$ и $x1$ применяется комбинирующая операция.

Листинг 5.20. Развертка цикла на 2 и использование двухстороннего параллелизма

```

1 /* Развёртка цикла на 2, двухсторонний параллелизм */
2 void combine6 (vec_ptr v, data_t *dest)
3 {
4     int length = vec_length (v);
5     int limit = length-1;
6     data_t *data = get_vec_start (v);
7     data_t x0 = IDENT;
8     data_t x1 = IDENT;
9     int i;
10
11    /* Комбинирование 2 элементов за один раз */
12    for (i = 0; i < limit; i+=2)  {
13        x0 = x0 OPER data [i];
14        x1 = x1 OPER data [i+1];
15    }
16
17    /* Завершение всех оставшихся элементов */
18    for (; i < length; i++)  {

```

```

19     x0 = x0 OPER data [i];
20 }
21 *dest = x0 OPER x1;
22 }
```

Данный подход использует возможность конвейеризации функциональных устройств.

Для понимания того, как этот код обеспечивает повышенную производительность, рассмотрим преобразование цикла в операции для случая умножения целых чисел (табл. 5.14).

Таблица 5.14. Умножение целых чисел

Ассемблерные команды	Операции устройства выполнения	
.L151:		
imull (%eax, %edx, 4), %ecx	load (%eax, %edx.0, 4)	t.la
	imull t.la, %ecx.0	%ecx.1
imull 4(%eax, %edx, 4), %ebx	load 4 (%eax, %edx.0, 4)	t.lb
	imull t.lb, %ebx.0	%ebx.1
addl \$2, %edx	addl \$2, %edx.0	%edx.1
cmpl %esi, %edx	cmpl %esi, %edx.1	cc.1
jl .L151	jl-taken cc.1	

Операции выполнения	
load (%eax, %edx.0, 4)	t.la
imull t.la, %ecx.0	%ecx.1
load 4(%eax, %edx.0, 4)	t.lb
imull t.lb, %ebx.0	%ebx.1
addl \$2, %edx.0	%edx.1
cmpl %esi, %edx.1	cc.1
jl-taken cc.1	

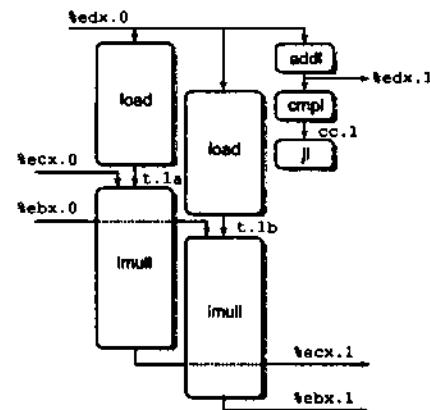


Рис. 5.13. Операции первой итерации внутреннего цикла двухстороннего развернутого двухстороннего параллельного целочисленного умножения

На рис. 5.13 показано графическое представление этих операций для первой итерации ($i = 0$). На диаграмме показано, что две операции умножения в цикле независимы друг от друга. Одна в качестве источника и места назначения имеет регистр $\%ecx$.

(соответствующий переменной x_0 программы), а другая — имеет регистр $\$ebx$ (соответствующий переменной x_1 программы). Вторая операция умножения может начаться через один цикл синхронизации после первой. Этим используются возможности конвейеризации устройства загрузки и целочисленного множителя.

На рис. 5.14 показано графическое представление первых трех итераций ($i = 0, 2$ и 4) для целочисленного умножения. Для каждой итерации две операции умножения должны подождать вычисления результатов предыдущей итерации. Машина по-прежнему может генерировать два результата каждые четыре цикла синхронизации, что теоретически дает СРЕ в 2.0. На данной схеме не учитывается ограниченный набор целочисленных функциональных устройств, но именно в этой процедуре это не рассматривается как ограничение.

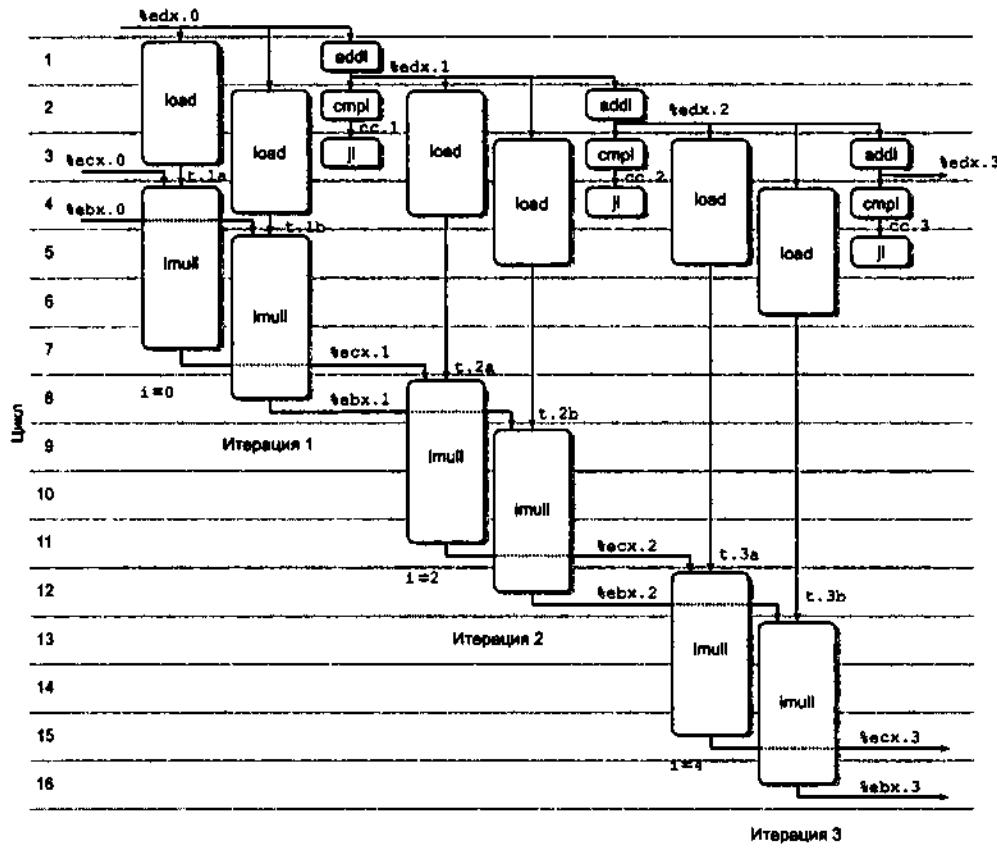


Рис. 5.14. Планирование операций для двухстороннего развернутого, двухстороннего параллельного целочисленного умножения с неограниченными ресурсами

Сравнение просто развертки цикла с разверткой цикла с двухсторонним параллелизмом дает следующие показатели производительности (табл. 5.15).

Таблица 5.15. Производительность функции

Функция	Метод	Целое число		Число с плавающей точкой	
		+	*	+	*
	Разворотка × 2	1.50	4.00	3.00	5.00
combine6	Разворотка × 2, параллелизм × 2	1.50	2.00	2.00	2.50

Целочисленной сумме параллелизм не помогает, потому что задержка целочисленного сложения составляет всего один цикл синхронизации. Однако для произведения целого числа и числа с плавающей точкой СРЕ уменьшается на коэффициент 2. По сути дела, этим удваивается использование функциональных устройств. Для суммы чисел с плавающей точкой другой сдерживающий фактор ресурсов ограничивает СРЕ до 2.0, а не до теоретического значения 1.5.

Ранее уже отмечалось, что арифметика дополнительных кодов является коммутативной и ассоциативной даже при возникновении переполнения. Отсюда, для типа целочисленных данных результат, вычисленный `combine6`, будет идентичен результату `combine5` при всех возможных условиях. Следовательно, оптимизирующий компилятор мог бы потенциально преобразовать код, показанный в `combine4`, в двухсторонний развернутый вариант `combine5`, путем развертки цикла, а затем — в `combine6` путем введения параллелизма. В литературе, посвященной оптимизирующими компиляторам, это называется *разбиением итерации*. Многие компиляторы выполняют развертку циклов автоматически, однако совсем немногие выполняют разбиение итераций.

С другой стороны, читатели уже имели возможность обратить внимание на то, что сложение и умножение чисел с плавающей точкой не ассоциативны. Таким образом, `combine5` и `combine6` могли выдавать разные результаты по причине округления или переполнения. Представим, к примеру, случай, в котором все элементы с четными индексами являются числами с очень большим абсолютным значением, тогда как элементы с нечетными индексами очень близки к 0.0. Тогда произведение PE_n может переполниться или PO_n — исчезнуть, даже если окончательное произведение P_n останется. Впрочем, в большинстве реальных программных приложений такие комбинации маловероятны. Поскольку большинство физических феноменов непрерывны, числовые данные стремятся к тому, чтобы быть обоснованно ровными и имеющими физический смысл. Даже при наличии разрывности они, как правило, не вызывают периодических комбинаций, приводящих к условию, подобному описанному. Вряд ли суммирование элементов в строгом порядке обеспечит фундаментально улучшенную точность, нежели независимое суммирование двух групп с последующим сложением полученных сумм. Для большинства программных приложений достижение прироста производительности на 2Х перевесит риск генерирования разных результатов для странных комбинаций данных. Несмотря на это, разработчик программы должен согласовывать с ее потенциальными пользователями те или иные условия, которые могут вызвать неприемлемость пересмотренного алгоритма.

Так же, как циклы можно разворачивать на произвольный коэффициент k , можно увеличивать параллелизм на любой коэффициент p , такой, что k будет кратно p . Здесь показаны некоторые результаты для различных степеней развертки и параллелизма (табл. 5.16).

Таблица 5.16. Различные степени развертки

Метод	Целое число		Число с плавающей точкой	
	+	*	+	*
Развертка × 2	1.50	4.00	3.00	5.00
Развертка × 2, параллелизм × 2	1.50	2.00	2.00	2.50
Развертка × 4	1.50	4.00	3.00	5.00
Развертка × 4, параллелизм × 2	1.50	2.00	1.50	2.50
Развертка × 8	1.25	4.00	3.00	5.00
Развертка × 8, параллелизм × 2	1.25	2.00	1.50	2.50
Развертка × 8, параллелизм × 4	1.25	1.25	1.61	2.00
Развертка × 8, параллелизм × 8	1.75	1.87	1.87	2.07
Развертка × 9, параллелизм × 3	1.22	1.33	1.66	2.00

Как показано в таблице, повышение степени развертки и степени параллелизма повышает производительность программы до определенной точки, однако оно снижает степень повышения или дает худшую производительность при экстремальных значениях. В следующем разделе описываются две причины данного феномена.

5.10.2. Вытеснение регистров

Преимущества параллелизма циклов ограничены способностью выражать вычисления в компонующем автокоде. В частности, система команд IA имеет небольшое количество регистров для сохранения накапливаемых значений. Если есть степень параллелизма p , превышающая количество имеющихся регистров, тогда компилятор обратится к процедуре *вытеснения*, сохраняя некоторые из временных значений на стеке. Когда это происходит, производительность сразу резко понижается. Для рассматриваемых эталонных тестовых программ это происходит при попытке получить значение $p = 8$. Выполненные измерения показывают, что производительность для данного случая хуже, нежели, когда p имеет значение 4.

Для случая с типом целочисленных данных имеется только восемь полных регистров целых чисел. Два из них (%ebp и %esp) указывают на области стека. С версией указателя кода один из оставшихся шести регистров удерживает указатель *data*, а один удерживает останавливающую позицию *dead*. Остаются четыре целочисленных реги-

стра для накопления значений. С версией указателя массива требуется три регистра для удержания индекса цикла *i*, останавливающего индекса *limit* и адреса массива *data*. Остаются три регистра для накопления значений. Для типа данных с плавающей точкой необходимы два из восьми регистров для удержания промежуточных значений; шесть регистров остаются для накопления значений. Таким образом, можно получить максимальный параллелизм из шести до того, как произойдет вытеснение регистров.

Такое ограничение восемью целочисленными регистрами и восемью — для чисел с плавающей точкой является неудачным артефактом системы команд IA32. Описанная ранее схема переименования устраниет прямое соответствие между названиями регистров и фактическим местоположением данных регистра. В современных процессорах названия регистров служат только для идентификации значений программы, передаваемых между функциональными устройствами. IA32 обеспечивает немного таких идентификаторов, ограничивая объем параллелизма, который может быть выражен в программе.

Возникновение вытеснения можно наблюдать, изучив компонующий автокод. Например, в рамках первого цикла для кода в восьмисторонним параллелизмом обнаруживается следующая последовательность команд (листинг 5.21):

Листинг 5.21. Первый цикл

```

type =INT, OPER = '*'
x6 in -12(%ebp), data+i in %eax
1    movl -12(%ebp), %edi  Получение x6 из стека
2    imull 24(%eax), %edi  Умножение на data [i+6]
3    movl %edi, -12(%ebp)  Возврат x6

```

В данном коде для удержания *x6* — одной из восьми локальных переменных, используемых для накопления сумм, — используется ячейка стека. Код загружает его в регистр, умножает на один из элементов данных и сохраняет в той же ячейке стека. В качестве общего правила, как только компилируемая программа демонстрирует вытеснение регистров в рамках какого-либо внутреннего цикла, который используется с большой нагрузкой, предпочтительно переписать код так, чтобы требовалось меньше временных значений. Это делается сокращением числа локальных переменных.

УПРАЖНЕНИЕ 5.5

Ниже показан код, сгенерированный из варианта *combineb*, в котором используется восьмисторонняя развертка цикла и четырехсторонний параллелизм.

```

1 .L1S2
2 addl (%eax), %ecx
3 addl 4(%eax), %esi
4 addl 8(%eax), %edi
5 addl 12(%eax), %ebx

```

```

6 addl 16 (%eax), %ecx
7 addl 20 (%eax), %esi
8 addl 24 (%eax), %edi
9 addl 28 (%eax), %ebx
10 addl $32 %eax
11 addl $8 %edx
12 cmpl -8(%ebp), %edx
13 j1 .L152

```

1. Какая программная переменная вытесняется в стек?
2. В какой ячейке стека?
3. Почему это хороший выбор значения для вытеснения?

С данными с плавающей точкой необходимо сохранять все локальные переменные в стеке регистра для данных с плавающей точкой. Также необходимо поддерживать верхнюю часть стека доступной для загрузки данных из памяти. Это ограничивает степень параллелизма до 7 и менее.

5.10.3. Ограничения параллелизма

Для рассматриваемых тестовых программ основные ограничения производительности связаны с возможностями функциональных устройств. Как показано в табл. 5.17, целочисленный множитель и сумматор чисел с плавающей точкой может только инициировать выполнение новой операции на каждом цикле синхронизации. Это обстоятельство, а также сходное ограничение на устройстве загрузки, ограничивают эти случаи до СРЕ 1.0. Множитель чисел с плавающей точкой может инициировать выполнение новой операции только каждые два цикла. Это ограничивает данный случай до СРЕ в 2.0. Целочисленная сумма ограничена СРЕ в 1.0 из-за ограничений устройства загрузки. Это приводит к следующему сравнению между достигнутой производительностью и теоретическими пределами:

Таблица 5.17. Показатели производительности

Метод	Целое число		Число с плавающей точкой	
	+	*	+	*
Достигнутая производительность	1.06	1.25	1.50	2.00
Теоретический предел	1.00	1.00	1.00	2.00

В этой таблице выбрана комбинация развертки и параллелизма, достигающая наилучшей производительности для каждого случая. Была возможность приблизиться к теоретическому пределу для целочисленной суммы и произведения, а также для произведения чисел с плавающей точкой. Определенный коэффициент (или коэффици-

енты), зависящий от используемой машины, ограничивает достигнутый СРЕ для умножения чисел с плавающей точкой значением 1.50, а не теоретическим пределом в 1.0.

УПРАЖНЕНИЕ 5.6

Рассмотрим следующую функцию вычисления произведения массива из целых чисел. Цикл развернут на коэффициент 3.

```
int aprod (int a [ ], int n)
{
int i, x, y, z;
int r = 1;
for (i = 0; i < n-2; i += 3) {
x = a [i]; y = a [i+1]; z = a [i+2];
r = r * x * y * z; /*Вычисление произведения */
}
for ( ; i < n; i++)
r *= a [i];
return r;
}
```

Для строки, обозначенной вычисления произведения, можно использовать круглые скобки для создания пяти различных ассоциаций вычисления следующим образом:

```
r = (r * x) * y * z; /*A1 */
r = (r * (x * y)) * z; /*A2 */
r = r * ((x * y) * z); /*A3 */
r = r * (x * (y * z)); /*A4 */
r = (r * x) * (y * z); /*A5 */
```

Измерены пять версий функции на Intel Pentium III. Вспомните по табл. 5.7, что операция целочисленного умножения на данной машине имеет задержку в 4 цикла, а время выдачи — 1 цикл.

В следующей таблице показаны некоторые значения СРЕ и другие недостающие значения. Измеренные значения СРЕ — те, что фактически имели место. "Теоретическое значение СРЕ" означает производительность, которая была бы достигнута, если бы единственным ограничивающим фактором была задержка и время выдачи целочисленного множителя.

Версия	Измеренное СРЕ	Теоретическое СРЕ
A1	4.00	
A2	2.67	
A3		
A4	1.67	
A5		

Заполните пустые графы таблицы. Для недостающих значений измеренного СРЕ можно использовать значения из других версий, имеющих то же самое вычислительное поведение. Для значений теоретического СРЕ можно определить число циклов, которые потребуются для итерации, рассматривающей только задержку и время выдачи множителя, после чего разделить ее на 3.

5.11. Сводка результатов оптимизации объединяющего кода

Авторами рассмотрены шесть версий объединяющего кода, некоторые из которых имели множественные варианты. На этом пока имеет смысл остановиться и обратить внимание на общий эффект от созданного кода и на то, как он будет выполняться на разных машинах. В табл. 5.18 показана измеренная производительность для всех рутинных процедур плюс несколько других вариантов. Как видно, максимальная производительность для целочисленной суммы достигается простой многократной разверткой цикла, тогда как максимальная производительность для других операций достигается представлением определенного параллелизма (но не в очень большом объеме). Общий прирост производительности в 27.6X и еще лучшая производительность первоначального кода впечатляют.

5.11.1. Аномалия производительности операций с числами с плавающей точкой

Одной из поразительных особенностей, показанных в табл. 5.18, является резкий перепад во времени цикла для операции умножения чисел с плавающей точкой при переходе от `combine3`, где произведение накапливается в памяти к `combine4`, где произведение накапливается в регистре с плавающей точкой. При этом небольшом изменении код внезапно начинает выполняться в 23.4 раза быстрее. При возникновении такого неожиданного результата важно предположить, что могло вызвать такое поведение, после чего разработать серию тестов для оценки гипотез.

Если внимательно изучить таблицу, выяснится, что для случая операций умножения чисел с плавающей точкой происходит что-то странное при накоплении результатов в памяти. Производительность здесь намного ниже, чем для операций сложения чисел с плавающей точкой или умножения целых чисел, несмотря на сравнимое количество циклов для соответствующих функциональных устройств. На процессоре IA32 все операции с плавающей точкой выполняются с повышенной точностью (80 бит), и регистры с плавающей точкой сохраняют значения в этом формате. Только когда значение в регистре записано в память, оно преобразуется в формат 32 бит (`float`) или 64 бит (`double`).

При изучении данных для измерений источник проблемы становится прозрачным. Измерения проводились на векторе длиной 1024, на котором каждый элемент i равен $i + 1$. Отсюда делается попытка вычислить $1024!$, что равно приблизительно 5.4×10^{2639} . Такое невообразимо огромное число можно представить в формате с плавающей точкой с повышенной точностью (в этом формате можно представлять

Таблица 5.18. Сравнительный результат на процессоре Pentium III

Функция	Метод	Целое число		Число с плавающей точкой	
		+	*	+	*
combine1	Абстрактный неоптимизированный	42.06	41.86	41.44	160.00
combine1	Абстрактный -O2	31.25	33.25	31.25	143.00
combine2	Перемещение vec_length	20.66	21.25	21.15	135.00
combine3	Прямой доступ к данным	6.00	9.00	8.00	117.00
combine4	Накопление во временной переменной	2.00	4.00	3.00	5.00
combine5	Развертка × 4	1.50	4.00	3.00	5.00
	Развертка × 16	1.06	4.00	3.00	5.00
	Развертка × 2, параллелизм × 2	1.50	2.00	2.00	2.50
	Развертка × 4, параллелизм × 2	1.50	2.00	1.50	2.50
	Развертка × 8, параллелизм × 4	1.25	1.25	1.50	2.00
Худшее, лучшее		39.7	33.5	2.00	80.0

числа до 10^{4932}), однако оно превышает то, что можно представить с одинарной точностью (до порядка 10^{38}) или с удвоенной точностью (до порядка 10^{308}). По достижении $i = 34$ случай с одинарной точностью переполняется; случай с двойной точностью переполняется при $i = 171$. По достижении данной точки каждое выполнение оператора

`*dest = *dest OPER val;`

во внутреннем цикле combine3 требует считывания значения $+\infty$ из dest, умножая его на val для получения $+\infty$, с последующим сохранением назад в dest. Очевидно, некоторая часть такого вычисления требует больше, чем обычные пять циклов синхронизации, необходимые для выполнения операции умножения чисел с плавающей точкой. Фактически, при запуске измерений на этой операции обнаруживается, что для умножения числа на бесконечность требуется от 110 до 120 циклов синхронизации. Вероятнее всего аппаратные средства определят это как особый случай и выдадут прерывание, заставляющее программу системы программного обеспечения выполнять фактическое вычисление. Проектировщики CPU предположили, что такие случаи будут довольно редкими, поэтому им не придется обрабатывать их в рамках проектирования аппаратных средств. Подобное поведение может иметь место с операцией исчезновения.

При запуске эталонной тестовой программы с данными, для которых каждый векторный элемент равен 1.0, `combine3` достигает СРЕ в 10 тыс. циклов, как для обычной, так и удвоенной точности. Это намного больше в строке со значениями времени, измеренными для других типов данных и операций, и сравнимо со значением времени для `combine4`.

Данный пример иллюстрирует одну из сложностей оценки производительности программы. На измерения могут оказывать сильное влияние характеристики данных и условия работы, поначалу кажущиеся незначительными.

5.11.2. Изменение платформ

Несмотря на то, что стратегии оптимизации представляются в контексте специальных машины и компилятора, общие принципы также применимы к другим комбинациям машин и компиляторов. Разумеется, оптимальная стратегия может очень сильно зависеть от машины. В качестве примера, в табл. 5.19 показаны результаты измеренной производительности процессора Compaq Alpha 211164 для условий, сравнимых для Pentium III (табл. 5.18). Эти измерения сделаны для кода, генерированного компилятором Compaq C, использующего более передовую оптимизацию,

Таблица 5.19. Сравнительный результат на процессоре Compaq

Функция	Метод	Целое число		Число с плавающей точкой	
		+	*	+	*
combine1	Абстрактный неоптимизированный	40.14	47.14	52.07	53.71
combine1	Абстрактный -O2	25.08	36.05	37.37	32.02
combine2	Перемещение <code>vec_length</code>	19.19	32.18	28.73	32.73
combine3	Прямой доступ к данным	6.26	12.52	13.26	13.01
combine4	Накопление во временной переменной	1.76	9.01	8.01	8.01
combine5	Развертка ×4	1.51	9.01	6.32	6.32
combine6	Развертка ×16	1.25	9.01	6.33	6.22
	Развертка ×2, параллелизм ×2	1.19	4.69	4.44	4.45
	Развертка ×8, параллелизм ×4	1.15	4.12	2.34	2.01
	Развертка ×8, параллелизм ×8	1.11	4.24	2.36	2.08
	Худшее, лучшее	36.2	11.4	22.3	26.7

нежели GCC. Обратите внимание, как значения времени циклов уменьшаются при перемещении в нижнюю часть таблицы — так же, как и для другой машины. Видно, что можно эффективно использовать более высокую (восьмистороннюю) степень параллелизма, потому что Alpha имеет 32 целочисленных регистра и 32 регистра для чисел с плавающей точкой. Как показывает данный пример, общие принципы оптимизации программы применяются к множеству различных машин, даже если конкретная комбинация свойств, приводящих к оптимальной производительности, зависит от моделей.

5.12. Прогнозирование ветвей и штрафы за некорректное прогнозирование

Как упоминалось ранее, современные процессоры прекрасно работают с опережением команд, выполняющихся в текущий момент времени; они считывают новые команды из памяти и декодируют их для определения предназначенных для выполнения операций и необходимых для этого операндов. Конвейерная обработка команд работает адекватно, пока команды придерживаются простой последовательности. Однако при столкновении с ветвью процессор должен "угадать" ее направление. Для случая с условным переходом это означает прогнозирование того, будет ветвь выбрана или нет. Для таких команд, как команда косвенного перехода (как было видно в коде перехода к адресу, обозначенному записью в таблице перехода) или команда возврата процедуры, это означает прогнозирование целевого адреса. В данном обсуждении упор сделан на условные ветви.

В процессоре, где применяется *предположительное выполнение*, команды начинают выполняться на спрогнозированной ветви. Это происходит так, что ни один фактический регистр или ячейка памяти не модифицируются до определения фактического выхода. Если прогноз корректен, процессор просто "передает" результаты предположительно выполненных команд путем сохранения их в регистрах или в памяти. Если прогноз некорректен, тогда процессор должен отбросить все предположительно выполненные результаты и перезапустить процесс выборки команд с корректного межстоположения. При этом налагается значительный штраф ветви, потому что конвейер команд должен быть повторно наполнен до того, как будут сгенерированы полезные результаты.

До недавнего времени технология, необходимая для поддержки предположительного выполнения, считалась весьма дорогостоящей и экзотической для любых машин, кроме промышленных суперкомпьютеров. Однако, где-то в 1998 г. технология интегральных схем позволила представить на одной микросхеме настолько большое количество различных цепей, что некоторые микросхемы стало возможно использовать для поддержки процессов прогнозирования ветвей и предположительного выполнения. С этого времени практически любой процессор настольной рабочей станции или сервера поддерживает предположительное выполнение.

При оптимизации комбинирующей процедуры ограничения производительности, вызванные циклической структурой, не рассматривались. То есть, оказалось, что

единственный фактор, сдерживающий производительность, связан с функциональными устройствами. Для этой процедуры процессор, как правило, мог спрогнозировать направление ветви в конце цикла. На самом деле, если сделан прогноз, что ветвь всегда будет выбрана, тогда процессор будет работать корректно всегда, кроме последней итерации.

Для прогнозирования ветвей разработано множество схем, и их производительность тщательно изучена. Общей эвристикой является прогноз, что любая ветвь на более младший адрес будет выбрана, а любая ветвь на более старший адрес выбрана не будет. Ветви на младшие адреса используются для закрытия циклов, а поскольку циклы, как правило, выполняются многократно, то хорошей идеей является прогнозирование этих ветвей как выбранных. С другой стороны, ветви с переходом вперед используются для условных вычислений. Экспериментально доказано, что эвристика BTFNT корректна приблизительно 65% времени. Однако прогнозирование всех ветвей как выбранных имеет долю успешных попыток в 60%. Были разработаны намного более сложные стратегии, требующие большего количества аппаратных средств. Например, в процессоре Pentium II и III используется стратегия прогнозирования ветвей с заявленной долей успешной попытки 90—95% времени.

Для тестирования возможности прогнозирования ветвей процессора и убытков от некорректного прогнозирования можно провести эксперименты. В качестве контрольного примера используется рутинная процедура абсолютного значения, показанная в листингах 5.22 и 5.23. Здесь также показана скомпилированная форма. Для неотрицательных аргументов ветвь будет выбрана, чтобы миновать команду отрицания. Эта функция синхронизируется вычислением абсолютного значения каждого элемента в массиве, состоящем из различных комбинаций +1 и -1. Для регулярных образов (например, все +1, все -1 или чередование +1 и -1) обнаруживается, что данная функция требует от 13.01 до 13.41 циклов. Этот диапазон используется для оценки производительности с идеальным условием ветвления. Если массив настроен на произвольные комбинации +1 и -1, тогда функция требует 20.32 циклов. Один из принципов вероятностного процесса: не важно, какая стратегия используется для понимания последовательности значений, но если основной процесс по-настоящему вероятностный (стохастический), тогда доля успешных попыток составит всего 50% времени. Например, не важно, какая применяется стратегия для угадывания стороны, на которую упадет брошенная монета: если она брошена "честно", тогда вероятность успешного угадывания составляет всего 0.5. Таким образом, становится понятно, что некорректно спрогнозированная для этого процессора ветвь "выливается" в штраф в размере 14 циклов синхронизации, поскольку при коэффициенте некорректного прогноза в 50% функция выполняется в среднем на 7 циклов медленнее. Это означает, что вызовы `absval` требуют от 13 до 27 циклов синхронизации, в зависимости от успешности предсказателя ветвей.

Листинг 5.22. Код абсолютного значения на C

```
1 int absval (int val)
2 {
3     return (val < 0) ? -val : val;
4 }
```

Листинг 5.23. Код абсолютного значения на ассемблере

```

1 absval :
2 pushl %ebp
3 movl %esp, %ebp
4 movl *(%ebp), %eax Получение val
5 testl %eax, %eax Тестирование val
6 jge .L3If >0, goto end
7 negl %eaxElse, negate it
8 .L3:end:
9 movl %ebp, %esp
10 popl %ebp
11 ret

```

Штраф в 14 циклов довольно крупный. Например, если бы точность предсказания составляла всего 65%, тогда бы процессор тратил впустую в среднем $14 \times 0.35 = 4.9$ циклов на каждую команду ветвления. Даже с точностью прогноза в 90—95%, заявленной для Pentium II и III, на каждую ветвь тратится порядка 1 цикла синхронизации, по причине некорректного прогнозирования. Изучение реальных программ показывает, что ветви составляют порядка 15% от всех выполняемых команд в типичных программах "целочисленных вычислений" (в которых не обрабатываются числовые данные), и от 3 до 12% всех выполняемых команд в типичных числовых программах [33]. Таким образом, время, потраченное впустую из-за неэффективной обработки ветвей, может значительно повлиять на производительность процессора.

Многие ветви с зависимостями по данным вообще невозможно спрогнозировать. Например, нет способа определения того, положительным или отрицательным будет аргумент к рутинной процедуре абсолютного значения. Для повышения производительности кода, включающего в себя условную оценку, проекты многих процессоров были расширены для включения команд *условного перемещения*. Эти команды позволяют реализовать некоторые формы условных операторов, без каких бы то ни было команд ветвления.

С системой команд IA32, начиная с процессора Pentium Pro, было добавлено несколько разных команд *смов*. Эти команды поддерживаются всеми последними процессорами Intel и совместимыми с Intel и выполняют операцию, сходную с кодом

```

if (COND)
    x = y;

```

где *у* — исходный operand, а *х* — operand назначения. Условие *COND*, определяющее наличие операции копирования, основано на некоторой комбинации значений условного кода, подобной командам тестирования и условного перехода. В качестве примера, команда *смов1* выполняет копирование, когда коды условия указывают на значение меньше нуля. Обратите внимание, что первый символ "!" этой команды обозначает "меньше", а второй — суффикс GAS для длинного слова.

Листинг 5.24 показывает реализацию абсолютного значения с условным перемещением.

Листинг 5.24: Код абсолютного значения с условным перемещением

1	movl 8(%ebp), %eax	Получение val в качестве результата
2	movl %eax, %edx	Копирование в %edx
3	negl %edx	Отрицание %edx
4	testl %eax, %eax	Тестирование val
5		Условное перемещение %edx в %eax
6	cmovl %edx, %eax	If < 0, копирование %edx в результат

Как показано в данном коде, стратегия следующая: val задается как возвращаемое значение, вычисляется -val и осуществляется его условное перемещение в регистр %eax для изменения возвращаемого значения, когда val — отрицательно. Проведенные измерения данного кода показывают, что он выполняется в течение 13.7 циклов, независимо от комбинаций данных. Это, очевидно, дает лучшую общую производительность, чем процедура, требующая от 13 до 27 циклов.

УПРАЖНЕНИЕ 5.7

Ваш друг написал оптимизирующий компилятор, использующий команды условного перемещения. Вы пытаетесь скомпилировать следующий код:

```
1 /* Разыменование указателя или возврат 0, если null */
2 int deref (int *)
3 {
4     return xp ? *xp : 0;
5 }
```

Компилятор генерирует следующий код для тела процедуры.

```
1 movl 8(%ebp), %edx      Получение xp
2 movl (%edx), %eax      Получение xp в качестве результата
3 testl %edx, %edx       Тестирование xp
4 cmovl %edx, %eax       If 0, копирование 0 в результат
```

Объясните, почему этот код не обеспечивает допустимой реализации deref?

Текущая версия GCC не генерирует никаких кодов, используя условные перемещения. Из-за желания сохранить совместимость с более ранними процессорами 486 и Pentium компилятор не пользуется этими новыми преимуществами. В экспериментах применялся написанный вручную компонующий автокод. Версия, использующая способность GCC встраивать компонующий автокод в программу C (см. разд. 3.15), требовала 17.1 циклов из-за не очень качественного генерирования кода.

К сожалению, для повышения производительности ветвей программы программист С мало что может сделать, кроме выяснения того, что ветви с зависимостями по данным заметно ее ухудшают. Помимо этого, программист не имеет возможности в полной мере контролировать подробную структуру ветвей, сгенерированную компилятором, а повысить прогнозируемость ветвей довольно сложно. В конечном счете,

необходимо полагаться на сочетание качественного генерирования кода компилятором для минимизации использования условных ветвей и эффективного прогнозирования ветвей процессором для сокращения числа случаев ошибок прогнозирования.

5.13. Понятие производительности памяти

Все написанные до сих пор коды и все проведенные процедуры тестирования требовали сравнительно небольшого объема памяти. Например, рутинные процедуры комбинирования измерялись по векторам длиной 1024, требующего не более 8,096 байтов данных. Все современные процессоры содержат один или более кэшей для обеспечения быстрого доступа к таким маленьким объемам памяти. Все значения синхронизации, показанные в табл. 5.7, предполагают, что считываемые или записываемые данные содержатся в кэше. В главе 6 будет подробно рассматриваться работа кэш-памяти, а также написание кода, максимально использующего кэш.

В настоящем разделе продолжается рассмотрение производительности операций загрузки и сохранения с предположением, что считываемые или записываемые данные хранятся в кэше. Как показано в табл. 5.7, оба эти устройства имеют время задержки 3, а время выдачи — 1. Во всех программах, рассматриваемых до сих пор, использовались только операции загрузки, а они имели такое свойство, что адрес одной загрузки зависел от приращения определенного регистра, а не от результата другой операции загрузки. Таким образом, как показано на рис. 5.7—5.10, 5.12 и 5.14, операции загрузки могут воспользоваться преимуществом конвейеризации для инициализации новых операций загрузки на каждом цикле. Сравнительно длительная задержка операции загрузки не имела обратного воздействия на производительность программы.

5.13.1. Задержка операций загрузки

В качестве примера кода, производительность которого ограничена задержкой операции загрузки, рассмотрим функцию `list_len`, показанную в листинге 5.25. Данная функция вычисляет длину связного списка. В цикле этой функции каждое последующее значение переменной `ls` зависит от значения, считанного ссылкой на указатель `ls->next`. Измерения показывают, что функция `list_len` имеет СРЕ 3.00, что объясняется прямым отображением задержки операции загрузки. Для понимания этого рассмотрим компонующий автокод для цикла и преобразование его первой итерации в операции (табл. 5.20).

Листинг 5.25. Функции связного списка

```

1  typedef struct ELE {
2      struct ELE *next;
3      int data;
4  } list_ele, *list_ptr;
5

```

```

6 int list_len (list_ptr ls)
7 {
8     int len = 0;
9
10    for (; ls; ls = ls->next)
11        len++;
12    return len;
13 }

```

Таблица 5.20. Первая итерация

Ассемблерные команды	Операции устройства выполнения
.L27: incl %eax movl (%edx), %edx testl %edx, %edx j1 .L24	incl %eax.0 %eax.1 load(%edx.0) %edx.1 testl %edx.1, %edx.1 cc.1 j1-taken cc.1

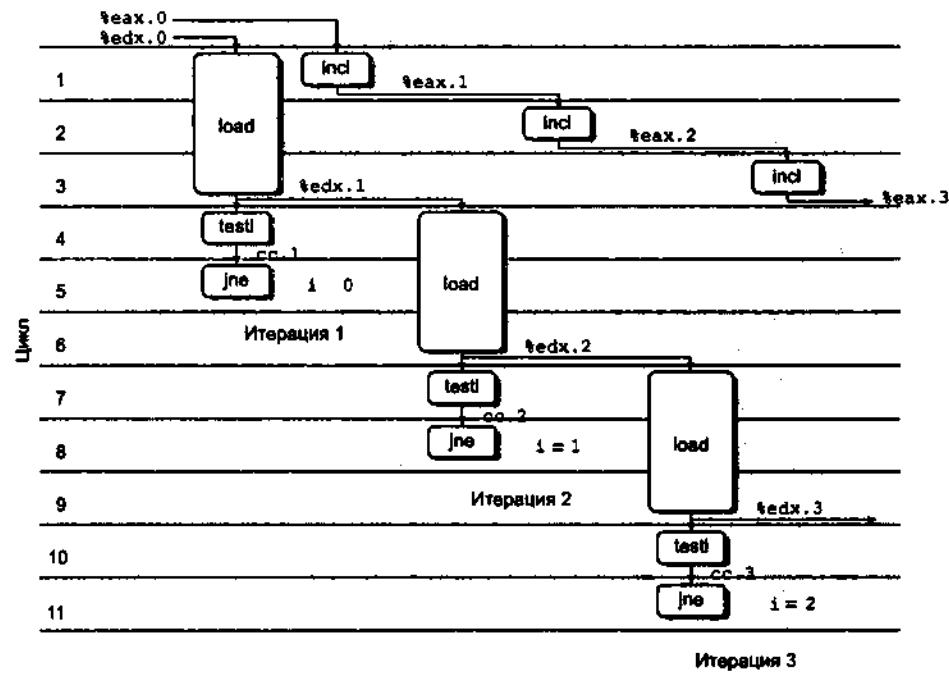


Рис. 5.15. Планирование операций для функции длины списка

Каждое последующее значение регистра %edx зависит от результата операции загрузки, имеющей %edx в качестве операнда. На рис. 5.15 показано планирование операций

для первых трех итераций данной функции. Можно заметить, что задержка операции загрузки ограничивает СРЕ до 3.0.

5.13.2. Задержка операций сохранения

До сих пор во всех примерах взаимодействие с памятью осуществлялось только использованием операции загрузки для считывания из ячейки памяти в регистр. Ее эквивалент, операция сохранения записывает в память значение регистра. Как показано в табл. 5.7, эта операция также обладает номинальной задержкой в три цикла, а время выдачи — 1 цикл. Однако ее поведение и взаимодействие с операцией загрузки включает в себя несколько "щекотливых" моментов.

Как и с операцией загрузки, в большинстве случаев операция сохранения может работать полностью в конвейерном режиме, начинаясь с нового сохранения на каждом цикле. Например, рассмотрим функции, показанные в листинге 5.26, устанавливающие элементы массива dest с длиной n на 0. Измерения для первой версии демонстрируют СРЕ в 2.00. Поскольку каждая итерация требует операции сохранения, понятно, что процессор может начать новую операцию сохранения минимум через каждые 2 цикла. Для дальнейших проб попытаемся восемь раз развернуть цикл, как показано в коде для array_clear_8. Для последнего СРЕ измеряется в 1.25. То есть, каждая итерация требует порядка 10 циклов и выдает восемь операций сохранения. Таким образом, практически достигнут оптимальный предел в одну новую операцию сохранения на каждый цикл синхронизации.

Листинг 5.26. Функции сброса массива

```
1 /* Установка элемента массива на 0 */
2 void array_clear (int *src, int *dest, int n)
3 {
4     int i;
5
6     for (i = 0; i < n; i++)
7         dest [i] = 0;
8 }
9
10 /* Установка элемента массива равным 0, развертка на 8 */
11 void array_clear_8 (int *src, int *dest, int n)
12 {
13     int i;
14     int len = n - 7;
15
16     for (i = 0; i < len; i +=8)  {
17         dest [i] = 0;
18         dest [i+1] = 0;
19         dest [i+2] = 0;
20         dest [i+3] = 0;
```

```

21 dest [i+4] = 0;
22 dest [i+5] = 0;
23 dest [i+6] = 0;
24 dest [i+7] = 0;
25 }
26 for (; i < n; i++)
27 dest [i] = 0;
28 }
```

В отличие от других рассмотренных операций, операция сохранения не влияет на значения регистров. Таким образом, по своей сути серии операций сохранения должны быть независимыми друг от друга. На самом деле на операцию загрузки влияет результат операции сохранения, поскольку только загрузка может считывать данные из ячейки памяти, записанной операцией сохранения. Функция `write_read`, показанная в листинге 5.27, иллюстрирует потенциальные взаимодействия между данными загрузки и сохранения. На схеме также показаны два примерных выполнения этой функции, когда она вызывается для двухэлементного массива `a` с начальным содержанием `-10` и `17` и с аргументом `cnt`, равным `3`. Эти выполнения иллюстрируют некоторые тонкости операций загрузки и сохранения.

Листинг 5.27. Функцииброса массива

```

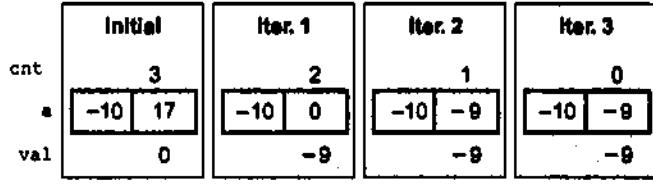
1 /* Запись в dest, считывание из src */
2 void write_read (int *src, int *dest, int n)
3 {
4     int cnt = n;
5     int val = 0;
6
7     while (cnt--) {
8         *dest = val;
9         val = (*src)+1;
10    }
11 }
```

На рис. 5.16 в первом примере аргумент `src` является указателем на элемент массива `a [0]`, тогда как `dest` — указатель на элемент массива `a [1]`. В этом случае каждая загрузка, выполняемая указателем ссылки `*src`, выдаст значение `-10`. Следовательно, после двух итераций массив элементов остается зафиксированным на `-10` и `-9` соответственно. Запись в `dest` не влияет на результат считывания из `src`. Измерение этого примера на примере большого количества итераций дает СРЕ в 2.00.

Во втором примере оба аргумента, `src` и `dest`, являются указателями на элемент `a [0]`. В этом случае каждая загрузка, выполняемая указателем ссылки `*src`, выдаст значение, сохраненное предыдущим выполнением указателем ссылки `*dest`. Вследствие этого серия восходящих значений будет сохраняться в этой ячейке памяти. Вообще говоря, если функция `write_read` вызывается с аргументами `src` и `dest`, указы-

вающими на ту же ячейку памяти, и с аргументом `cnt`, имеющим некоторое значение $n > 0$, тогда чистым эффектом будет установка данной ячейки в $n - 1$. Данный пример иллюстрирует феномен, который в книге называется *зависимостью от записи и считывания*: выход считывания из памяти зависит от самой последней записи в память. Проведенные измерения производительности показали, что второй пример рис. 5.16 имеет СРЕ 6.00. Зависимость от записи и считывания вызывает замедление обработки.

Пример 1: `write_read(sa[0], sa[1], 3)`



Пример 2: `write_read(sa[0], sa[0], 3)`

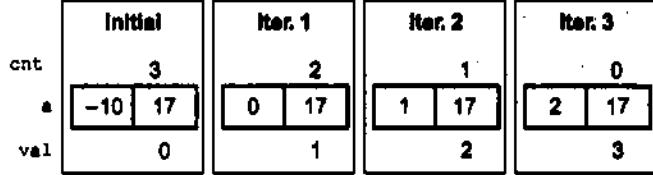


Рис. 5.16. Код к ячейкам памяти записи и считывания вместе с наглядными выполнениями

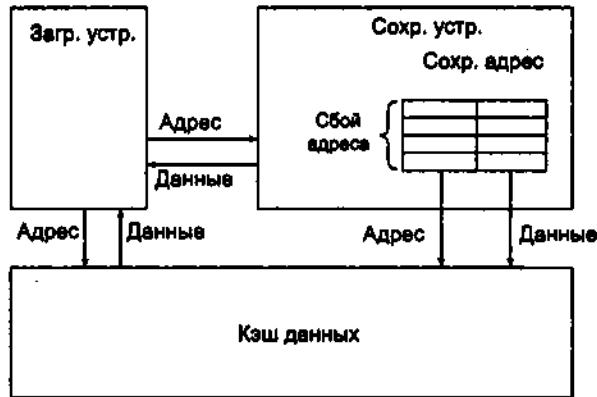


Рис. 5.17. Подробности функционирования устройств загрузки и сохранения

Для понимания того, как процессор различает эти два случая и почему один работает медленнее другого, необходимо более детально рассмотреть устройства выполнения загрузки и сохранения, как показано на рис. 5.17. Устройство сохранения содержит буфер *сохранения* с адресами и данными операций сохранения, выданными на устройство сохранения, но еще не завершенными, т. к. завершение требует обновления

кэша данных. Этот буфер устроен так, что серия операций сохранения может быть выполнена без необходимости ожидания, пока каждая операция не обновит кэш. При возникновении операции загрузки он должен проверять записи в буфере сохранения на предмет совпадения адресов. При обнаружении совпадения кэш извлекает соответствующую запись данных в результате операции загрузки.

Устройство сохранения поддерживает буфер незавершенных данных для записи. Устройство загрузки должно проверить его адрес, а также адреса в устройстве сохранения для обнаружения зависимости от записи и считывания.

Компонующий автокод для внутреннего цикла и его преобразование в операцию во время итерации представлен в табл. 5.21.

Таблица 5.21. Внутренний цикл

Ассемблерные команды	Операции устройства выполнения
.L32:	
movl %edx, (%ecx)	storeaddr (%ecx)
	storedata %edx.0
movl (%ebx), %edx	load (%ebx) %edx.1a
incl %edx	incl %edx.1a %edx.1b
decl \$eax	decl \$eax.0 %eax.1
jnc .L32	jnc-taken cc.1

Обратите внимание, что команда `movl %edx` преобразуется в две операции: команда `storeaddr` вычисляет адрес для операции сохранения, создает запись в буфере сохранения и устанавливает для этой записи поле адреса. Команда `storedata` устанавливает поле данных для этой записи. Поскольку устройство сохранения только одно, и операции сохранения обрабатываются в программном порядке, то не возникает двусмысленности в отношении того, как сочетаются две операции. Далее будет видно, что тот факт, что два вычисления выполняются независимо, является важным для производительности программы.

На рис. 5.18 показана синхронизация операций первых двух итераций `write_read` для первого примера рис. 5.16. Пунктирная линия между операциями `storeaddr` и `load` указывает на то, что операция `storeaddr` создает в буфере сохранения запись, которая впоследствии проверяется командой `load`. Поскольку эти команды не равны, `load` начинает считывать данные из кэша. Даже если операция сохранения не завершена, процессор может определить, что она повлияет на другую ячейку памяти, нежели на ту, что команда `load` пытается считать. Этот процесс повторяется также на второй итерации. Здесь видно, что операция `storedata` должна дождаться загрузки и приращения результата предыдущей итерации. Задолго до этого операция `storeaddr` и операции загрузки могут сравнить свои адреса, выяснить, что они разные, и обеспечить начало загрузки. На вычислительном графе показана загрузка для второй итерации, начинаящейся всего лишь через 1 цикл после загрузки первой. Если продолжить график для большего числа итераций, то он будет указывать СРЕ в 1.0. Очевидно, что

некоторый сдерживающий фактор ограничивает фактическую производительность до CPE 2.0.

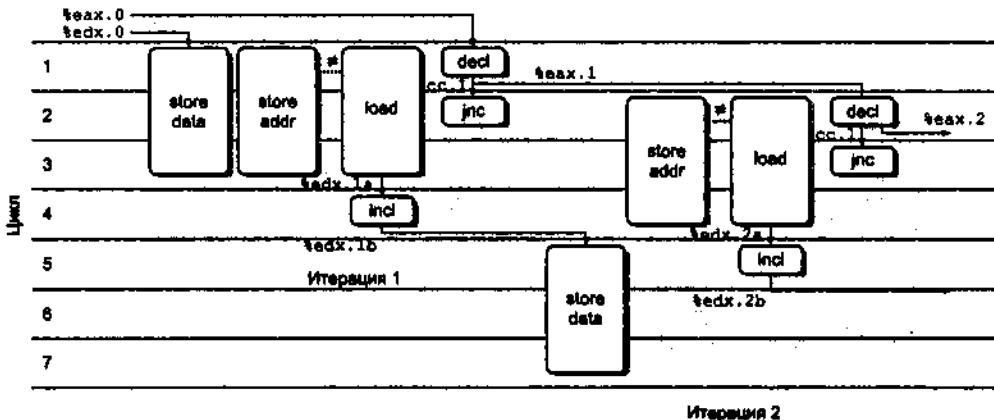


Рис. 5.18. Синхронизация для примера 1

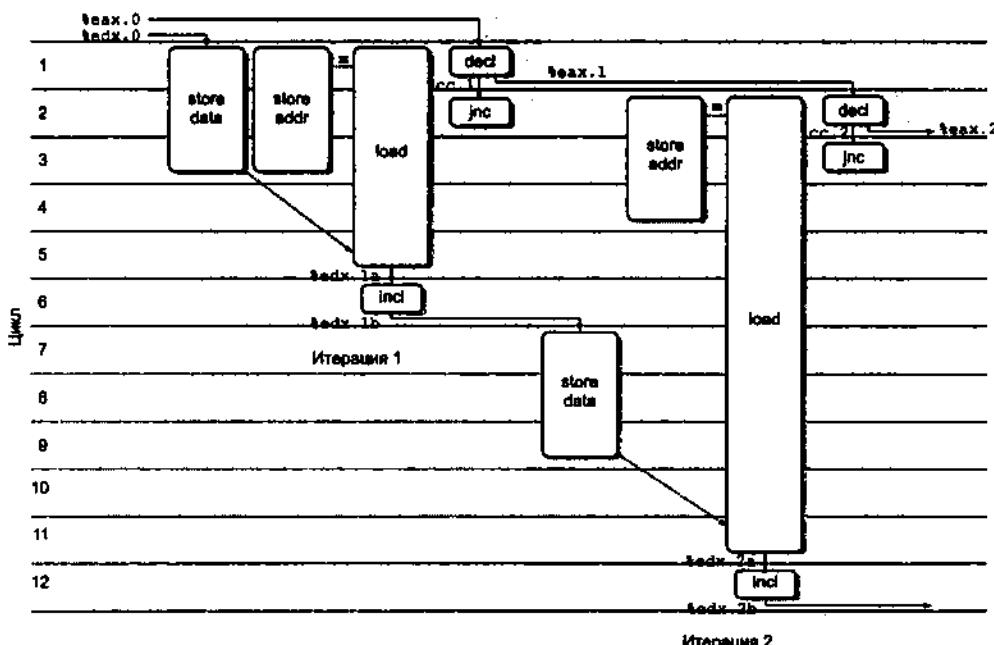


Рис. 5.19. Синхронизация для примера 2

На рис. 5.19 показана синхронизация операций для первых двух итераций write_read для случая второго примера на рис. 5.16. Опять же, пунктирная линия между операциями storeaddr и load указывает, что операция storeaddr создает запись в буфере

сохранения, которая потом проверяется операцией загрузки. Поскольку эти операции равны, load должна дождаться завершения операции storeaddr, после чего получить данные из буфера сохранения. Это ожидание обозначено на графике растянутой рамкой для операции загрузки.

Кроме этого, от операции storedata к load проведена пунктирная линия, указывающая на то, что результат storedata передается к load в качестве результата последней. Синхронизация этих операций изображена для отражения измеренного CPE в 6.00. Впрочем, не совсем ясно, как могло получиться такое значение CPE, поэтому предполагается, что эти цифры должны быть более иллюстративными, нежели фактическими. Вообще говоря, интерфейс "процессор-память" — одна из наиболее сложных областей проектирования процессора. Без доступа к подробной документации и инструментам машинного анализа описание фактического поведения можно составить только гипотетически.

Как показано в этих двух примерах, реализация операций памяти таит в себе множество тонкостей. С операциями в регистрах процессор может определить команды, которые влияют на другие, по мере их декодирования в операции. С другой стороны, с операциями в памяти процессор не может спрогнозировать команды, выполнение которых влияет на поведение других команд до вычисления адресов загрузки и сохранения. Поскольку операции памяти образуют значительную долю программы, подсистема памяти оптимизирована на собственное выполнение с большим параллелизмом для каждой отдельной операции памяти.

УПРАЖНЕНИЕ 5.8

В качестве очередного примера кода с потенциальным взаимодействием операций загрузки и сохранения рассмотрим следующую функцию копирования одного массива в другой:

```

1     void copy_array (int *src, int *dest, int n)
2     {
3         int, i;
4
5         for (i = 0; i < n; i++)
6             dest [i] = src [i];
7     }

```

Предположим, что a — массив длиной 1000, инициализированный так, что каждый элемент a [i] равен i.

1. Каков будет эффект от вызова copy_array (a+1, a, 999)?
2. Каков будет эффект от вызова copy_array (a, a+1, 999)?
3. Измерения производительности указывают на то, что вызов примера 1 имеет CPE в 3.00, тогда как вызов примера 2 имеет CPE в 5.00. За счет каких факторов возникает такое различие в производительности?
4. Какой производительности пользователь ожидает от вызова copy_array (a, a, 999)?

5.14. Жизнь в реальном мире: методы повышения производительности

Несмотря на то, что рассматривался достаточно ограниченный набор программных приложений, можно сделать важные выводы о том, как писать эффективные коды. Описано несколько базовых стратегий оптимизации производительности программ:

1. Проектирование на высоком уровне. Выберите подходящие алгоритмы и структуры данных для задачи, решаемой в данный момент. Особо следите, чтобы в число выбранных алгоритмов не попали алгоритмы или методы кодировки, дающие на выходе асимптотично низкую производительность.
2. Основные принципы кодировки. Избегайте блокираторов оптимизации для того, чтобы компилятор имел возможность генерирования эффективного кода.
 - Удаляйте чрезмерные обращения к функциям. По возможности, выводите вычисления за пределы циклов. Для повышения эффективности выборочно рассматривайте различные компромиссы модульности программы.
 - Удаляйте необязательные ссылки на ячейки памяти. Введите временные переменные для хранения промежуточных результатов. Сохраняйте результат в массиве или глобальной переменной.
3. Оптимизация низкого уровня:
 - Попробуйте использовать различные формы указателя относительно кода массива.
 - Удалите непроизводительные издержки циклов путем их развертки.
 - Найдите способы использования конвейерных функциональных устройств, используя такие методики, как разбиение итераций.

И последним советом читателю будет следующий: по возможности, старайтесь не тратить время и деньги на достижение заведомо некорректных результатов. Одним из полезных методов в этом направлении будет проверка кода с тестированием каждой его версии так, будто она оптимизирована, для того чтобы исправить во время этого процесса все ошибки. При проверке кода программа подвергается серии процедур тестирования, когда все полученные результаты проверяются на предмет их адекватности заявленным перед началом разработки. При вводе новых переменных, изменении границ циклов и общем усложнении кода очень легко допустить ошибки. Кроме того, очень важно отмечать необычные или неожиданные изменения производительности. Как было показано, выбор исходных данных может привести к очень большим расхождениям при сравнении производительности из-за аномалий последней, а также из-за выполнения очень коротких последовательностей команд.

5.15. Выявление и устранение критических элементов производительности

До настоящего момента рассматривалась оптимизация небольших программ, когда в них так или иначе находились места, явно требующие оптимизации. При работе с

крупными программами затруднение может вызвать даже место программы, в котором следует сосредоточить оптимизацию. В данном разделе описывается использование *кодовых профайлеров* — инструментов анализа, собирающих данные о производительности программы по мере выполнения. Здесь также описан общий принцип системной оптимизации, известный как закон Эмдала.

5.15.1. Профилирование программ

В рамках профилирования программ входит запуск версии программы, в которую встроен регистрационный код для определения того, какое количество времени на срабатывание требуют разные части программы. Этот метод может быть очень полезным при выявлении определенных частей программы, на которые следует обратить внимание при попытках оптимизации. Одной из сильных сторон профилирования является то, что его можно выполнять при работающей "настоящей" программе на вполне реальных исходных данных.

В системах Unix представлена программа профилирования GPROF. Она генерирует две формы информации. Во-первых, этой программой определяется количество времени, потраченное CPU на выполнение каждой функции. Во-вторых, она вычисляет количество вызовов каждой функции по категориям, откуда осуществляется этот вызов. Обе эти формы информации могут быть весьма полезными. Значения времени определяют относительную важность различных функций при расчете общего времени выполнения. Информация о вызовах позволяет понять динамическое поведение программы.

Профилирование с GPROF требует трех шагов, как показано для программы *prog.c*, запускаемой с помощью аргумента командной строки *file.txt*.

1. Программа должна быть скомпилированной и связанной для профилирования. С GCC (и другими компиляторами C) эта задача заключается в простом включении флагка выполнения *-pg* в командной строке:

```
unix> gcc -O2 -pg prog.c -o prog
```

2. После этого программа выполняется в обычном порядке:

```
unix> .prog file.txt
```

Программа выполняется немного (с коэффициентом 2) медленнее, чем обычно, в остальном единственное различие заключается в том, что генерируется файл *gmon.out*.

3. Для анализа данных в *gmon.out* генерируется GPROF.

```
unix> gprof prog
```

В первой части отчета профиля приведены значения времени, потраченного на выполнение различных функций, сохраненных в нисходящем порядке. Для примера в листинге 5.28 представлена эта часть отчета для первых трех функций программы.

В каждой строке представлено время, затраченное на вызов определенной функции. В первом столбце указан процент всего времени, затраченного на функцию. Во втором

ром показано накопленное время, затраченное функциями, до включения функции в эту строку. В третьем столбце показано время, затраченное на конкретную функцию, а в четвертом — количество раз обращения к ней (не считая рекурсивных вызовов).

В нашем примере функция `sort_words` вызывалась только один раз, но этот вызов потребовал 7.80 секунд, тогда как функция `lower1` вызывалась 946 596 раз, на которые потребовалось всего 0.41 секунд.

Листинг 5.28. Затраты времени

время	сек	сек	вызывы	мс/вызов	мс/вызов	имя
85.62	7.80	7.80	1	7800.00	7800.00	<code>rt_words</code>
6.59	8.40	0.60	946596	0.00	0.00	<code>file_ele_rec</code>
4.50	8.81	0.41	946596	0.00	0.00	<code>lower1</code>

Во второй части отчета профиля показана история вызовов функции. В листинге 5.29 приведена история рекурсивной функции `find_ele_rec`.

Листинг 5.29. Рекурсивная функция

		4872758	<code>find_ele_rec [5]</code>
[5] 6.7	0.60	0.01	<code>946596/946596 insert_string [4]</code>
	0.60	0.01	<code>946596 + 4872758 find_ele_rec [5]</code>
	0.00	0.01	<code>26946/26946 save_string [9]</code>
	0.00	0.00	<code>26946/26946 new_ele [11]</code>
		4872758	<code>find_ele_rec [5]</code>

Данная история показывает функции, вызывавшие `find_ele_rec`, а также функции, которые вызывала последняя. В верхней части показано, что функция фактически вызывалась 5 819 354 раз (показано как 946596 + 4872758): 4 872 758 раз сама по себе, а 946 596 раз функцией `insert_string` (которая сама вызывалась 946 596 раз). Функция `find_ele_rec`, в свою очередь, вызвала две другие функции: `save_string` и `new_ele`, каждую по 26 946 раз.

Из этой информации о вызовах часто можно извлечь полезную информацию о поведении программы. Например, функция `find_ele_rec` является рекурсивной процедурой, просматривающей связный список в поисках конкретной строки. При условии, что коэффициент от рекурсивных вызовов до вызовов верхнего уровня составлял 5.15, можно сделать вывод, что каждый раз приходилось просматривать (сканировать) порядка шести элементов.

Следует отметить некоторые свойства GPROF:

- Синхронизация не очень точна. Она основана на простой схеме счета интервалов (*описывается в главе 9*). Коротко говоря, скомпилированная программа поддерживает счетчик для каждой функции, записывая время, затраченное на выполнение этой функции. Операционная система приостанавливает выполнение программы на временном интервале δ . Типичные значения δ варьируются в диапазоне

не между 1.0 и 10.0 мс. Затем определяется функция, которую выполняла программа на момент прерывания, и счетчик для этой функции увеличивается на 8. Конечно, может случиться так, что эта функция только начала выполняться и очень скоро будет завершена, однако ей присвоен статус выполнения с последнего прерывания. Какая-либо другая функция может выполняться между двумя прерываниями, следовательно, она не требует временных затрат.

- На длинном интервале эта схема работает сравнительно хорошо. Статистически каждая функция должна рассматриваться в соответствии с относительным временем, необходимым на выполнение. Впрочем, для программ, выполняемых менее чем за одну секунду, показатели должны рассматриваться как весьма приблизительные.
- Информация о вызовах вполне надежна. Скомпилированная программа поддерживает счетчик для каждой комбинации вызывающей и вызываемой. Соответствующий счетчик увеличивается каждый раз при вызове процедуры.
- По умолчанию значения синхронизации для библиотечных функций не показаны. Вместо этого значения прибавляются к значениям времени, необходимым для вызова функций.

5.15.2. Использование профайлера при управлении оптимизацией

С целью использования профайлера для управления оптимизацией создано программное приложение, включающее в себя несколько задач и структур данных. Это приложение считывает текстовый файл, создает таблицу уникальных слов и количества раз, которое встречается каждое слово, после чего слова сортируются в нисходящем порядке, по мере их появления. В качестве точки отсчета данное приложение было запущено с файлом, содержащим полное собрание сочинений Шекспира. Было вычислено, что всего Шекспир написал 946 596 слов, 26 946 из которых — уникальны. Самым распространенным словом стал английский artikel "the", встречающийся 29 801 раз. Слово "любовь" встречается 2249 раз, а "смерть" — 933 раза.

Программа состоит из следующих частей. Создана серия версий, начинающаяся с простых алгоритмов для разных частей с последующей заменой их более сложными:

1. Каждое слово считывается из файла и преобразуется в строчные буквы. Исходная версия использовала функцию `lower1` (см. рис. 5.7), которая, как известно, имеет квадратичную сложность.
2. Для создания числа в диапазоне между 0 и $s-1$ хэш-таблицы с s областями памяти к строке применяется хэш-функция. Исходная функция просто суммировала коды ASCII для символов по модулю s .
3. Каждая хэш-область памяти организована в виде связного списка. Программа просматривает данный список в поисках совпадающей записи. При ее обнаружении частота для данного слова приращивается. В противном случае создается новый элемент списка. Первоначальная версия выполняла эту операцию рекурсивно, вставляя новые элементы в конец списка.

4. После того как таблица сгенерирована, все элементы сортируются по частотам. Первоначальная версия использовала сортировку методом вставок.

На рис. 5.20 показаны результаты профилирования для разных версий программы анализа частоты слов. Для каждой версии время разделено на пять категорий:

- Сортировка слов по частотам.
- Просмотр связного списка на предмет поиска совпадающих слов; при необходимости вставляется новый элемент.
- Преобразование строки в формат строчных символов.
- Вычисление хэш-функции.
- Сумма всех оставшихся функций.

Как показано на рис. 5.20, первоначальная версия требует более 9 секунд, большая часть из которых затрачивается на сортировку. Это не удивительно, поскольку сортировка методом вставок имеет квадратичную сложность, а программа сортирует порядка 27 тыс. значений.

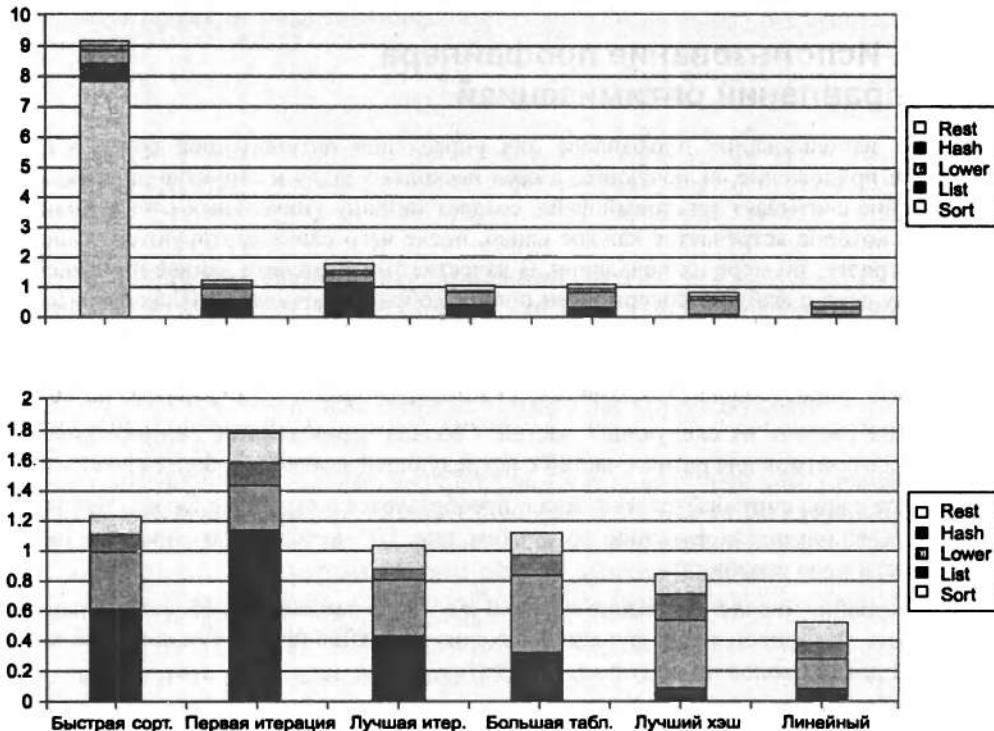


Рис. 5.20. Результаты профилирования для различных версий программы подсчета частоты слов

В следующей версии сортировка осуществляется с использованием библиотечной функции `qsort`, базирующейся на алгоритме быстрой сортировки. На схеме эта версия обозначена как "быстрая сортировка". Более эффективный алгоритм сортировки уменьшает время, затрачиваемое на сортировку, практически делая его ничтожным, а общее время — до 1.2 секунды. В второй части схемы показаны значения времени для оставшейся версии на более подробной шкале.

С усовершенствованной сортировкой обнаруживается, что сканирование списка становится критичной ситуацией. С предположением, что неэффективность имеет место из-за рекурсивной структуры функции, она меняется на итерационную функцию. Удивительно, но время выполнения увеличивается до 1.8 секунд. При более скрупулезном рассмотрении обнаруживается небольшое расхождение между двумя функциями списка. Рекурсивная версия вставила новые элементы в конец списка, тогда как итерационная — в начало. Для максимизации производительности необходимо, чтобы наиболее часто встречающиеся слова появлялись ближе к началам списков. Таким образом, функция быстро выявит общие случаи. Если предположить, что слова в документе распределены равномерно, то ожидать появления чаще встречающегося слова можно раньше, нежели слова, встречающегося реже. Вставкой новых слов в конец списка первая функция стремилась располагать слова в нисходящем порядке, тогда как вторая функция — наоборот. Таким образом, создана третья функция, просматривающая список и использующая итерацию, но со вставкой новых элементов в конец этого списка. С использованием этой версии время сократилось до 1.0 секунды, что немного лучше по сравнению с рекурсивной версией.

Далее будет рассмотрена структура хэш-таблицы. Первоначальная версия имела только 1021 сегмент (обычно количество сегментов выбирается как простое число, для повышения способности хэш-функции к равномерному распределению ключей по сегментам). Для таблицы с 26 946 записями это подразумевает среднюю загрузку в $26946/1007 = 26.4$. Это объясняет, почему столько много времени тратится на выполнение операций списка: в процедуру поиска входит тестирование значительного количества слов-вариантов. Это также объясняет повышенную чувствительность выполнения к упорядочиванию списка. Затем количество сегментов увеличивается до 10 007, что снижает среднюю загрузку до 2.70. Это покажется странным, но общее время выполнения увеличилось до 1.11 секунд. Результаты профилирования указывают на то, что это дополнительное время в основном уходило на рутинную процедуру преобразования в строчный формат, хотя это маловероятно. Значения времени выполнения довольно коротки, поэтому с этими значениями синхронизации трудно ожидать особой точности.

Была построена гипотеза о том, что низкая производительность в случае с более крупной таблицей имела место из-за неправильного выбора хэш-функции. Простое суммирование кодов символов дает не очень большой диапазон значений и не дифференцирует в соответствии с упорядочиванием символов. Например, слова "god" (бог) и "dog" (собака) будут хэшированы в ячейку $147 + 157 + 144 = 448$, поскольку они содержат одинаковые символы. Слово "foe" (недруг) также будет хэшировано в эту ячейку, поскольку $146 + 157 + 145 = 448$. Тут сделан переход к хэш-функции, использующей операцию EXCLUSIVE-OR. С этой версией, обозначенной как улучшенный хэш, время сокращается до 0.84. Более систематическим подходом будет

более тщательное изучение распределения ключей по сегментам, чтобы оно приближалось к тому, чего можно было бы ожидать, если бы хэш-функция обладала равномерным распределением выходных данных.

Наконец, время выполнения сокращено до точки, когда половина времени тратится на выполнение преобразования в строчный формат. Читатели уже могли заметить, что функция `lower1` имеет очень низкую производительность, особенно для длинных строк. Слова в данном документе достаточно короткие, что позволяет избежать катастрофических последствий квадратичного выполнения; самое длинное слово "honorificabilitudinitatibus" состоит из 27 символов. Все-таки переход к `lower2`, обозначенной как линейная `lower`, дает значительную производительность с общим временем выполнения, сокращающимся до 0.52 с.

В этом упражнении показано, что профилирование кодов может помочь сократить время, необходимое на выполнение простого программного приложения, от 9.11 до 0.52, коэффициент повышения — 17.5. Профайлер помогает сосредоточить внимание на наиболее затратных по времени частях программы, а также предоставляет полезную информацию о структуре обращения к процедуре.

Понятно, что функцию профилирования полезно иметь на панели инструментов, однако ей также должны сопутствовать дополнительные функции. Измерения значений времени синхронизации несовершенны, особенно для коротких значений времени выполнения (менее одной секунды). Результаты применяются только к конкретным протестированным данным. Например, если первоначальная функция запускается с данными, состоящими из меньшего количества длинных строк, то обнаружилось бы, что рутинная процедура преобразования строк в строчный формат стала бы основным критическим параметром производительности. Более того, если бы профилировались только документы с короткими словами, то выявить скрытые факторы, снижающие производительность (например, квадратичную производительность `lower1`), было бы весьма затруднительно. Вообще говоря, профилирование может помочь в оптимизации типичных случаев, предполагая, что программа запускается на представительных данных, однако также необходимо убедиться в том, что программа будет демонстрировать достойную производительность при всех возможных случаях. Сюда в основном входит избежание использования алгоритмов (например, сортировки методом вставок) и неэффективных стратегий программирования (например, `lower1`), дающих низкую асимптотическую производительность.

5.15.3. Закон Эмдала

Джин Эмдал (Gene Amdahl), один из пионеров вычислительной техники, сделал простое, но проницательное наблюдение, связанное с эффективностью повышения производительности одной части системы. Это наблюдение получило название закона Эмдала. Основной его идеей является то, что при разгоне одной части системы влияние общей системной производительности зависит как от значимости этой части системы, так и от степени ее ускорения. Рассмотрим систему, в которой выполнение одного из приложений требует времени T_{old} . Предположим, что некая часть системы требует доли α этого времени, и что производительность повышается на фактор k . То

есть, компонент первоначально требовал времени αT_{old} , а теперь требует времени $(\alpha T_{old})/k$. Таким образом, общее время выполнения составит

$$T_{new} = (1 - \alpha)T_{old} + (\alpha T_{old})/k = T_{old} [(1 - \alpha) + \alpha/k]$$

По этому выражению можно вычислить увеличение быстродействия $S = T_{old}/T_{new}$ как

$$S = \frac{1}{(1 - \alpha) + \alpha/k} \quad (5.1)$$

В качестве примера рассмотрим случай, когда часть системы, изначально требующая 60% времени ($\alpha = 0.6$), ускоряется на коэффициент 3 ($k = 3$). Затем получаем ускорение в $1/[0.4 + 0.6/3] = 1.67$. Таким образом, несмотря на то, что для основной части системы осуществлено значительное повышение производительности, чистое повышение быстродействия оказалось значительно ниже. Это основная идея закона Эмдала: для значительного повышения быстродействия всей системы необходимо повышать быстродействие очень большой доли всей системы.

УПРАЖНЕНИЕ 5.9

Предположим, что человек работает водителем грузовика, и ему поступил заказ на перевозку картофеля из Буазе (штат Айдахо) в Миннеаполис (штат Миннесота). Общее расстояние составляет 2500 километров. Водитель рассчитал, что весь рейс займет 25 часов, если двигаться со скоростью 100 км/ч (не превышая скоростного режима).

1. В выпуске новостей сообщили, что в штате Монтана отменили предельно допустимую скорость (доля всего расстояния — 1500 км). Грузовик может ехать со скоростью 150 км/ч. Каково будет ускорение на протяжении всего рейса?
2. Для грузовика в Internet (www.fasttrucks.com) можно заказать новый турбокомпрессор. Моделей много, но наибольшую скорость обеспечит только самая дорогостоящая. С какой скоростью следует ехать из Монтаны, чтобы добиться общего ускорения в 5/3?

УПРАЖНЕНИЕ 5.10

Отдел продаж компании пообещал клиентам, что следующая версия программного обеспечения продемонстрирует двукратное повышение производительности. Перед сотрудником поставлена задача выполнения этого обещания. Последний делает вывод о том, что усовершенствовать можно только 80% системы. Сколько потребуется средств (т. е. каково должно быть значение k) для усовершенствования производительности этой части системы для того, чтобы достичь общей цели производительности?

Одним из интересных особых случаев закона Эмдала является рассмотрение эффекта установки k на ∞ . То есть можно взять некоторую часть системы и разогнать ее до точки, в которой ее выполнение занимает ничтожное количество времени.

Получаем

$$S_{\infty} = \frac{1}{(1-\alpha)} \quad (5.2)$$

Таким образом, например, если порядка 60% системы можно разогнать до точки, когда ее выполнение практически не требует времени, ускорение все равно составит только $1/0.4 = 2.5$. Такую производительность продемонстрировал электронный словарь, когда сортировка методом вставки была заменена на быструю сортировку. Первоначальная версия затрачивала 7.8 из 9.1 секунды на выполнение вставки методом сортировки (α составляет .86). При быстрой сортировке затрачиваемое на нее время — ничтожно, и спрогнозированное ускорение равно 7.1. Фактически, реальное ускорение было выше: $9.11/1.22 = 7.5$, из-за неточности измерений профилирования первоначальной версии. Большее ускорение стало возможным получить из-за того, что сортировка составляла очень большую часть общего времени выполнения.

Закон Эмдала описывает общий принцип совершенствования любого процесса. Кроме ускорения работы компьютерных систем, он может дать некую рекомендацию компании-производителю бритвенных лезвий по повышению их остроты, или студенту, пытающемуся повысить свой средний балл. Возможно, он более применим и существенен в мире компьютеров, где производительность повышается рутинно на коэффициент 2 или более. Таких высоких коэффициентов можно добиться оптимизацией большей части системы.

5.16. Резюме

Несмотря на то, что во многих презентациях оптимизации кодов описывается способность компиляторов генерировать эффективные коды, очень многое зависит от программиста, которому надлежит оказывать компилятору активную "помощь". Никакой компилятор не заменит неэффективный алгоритм или структуру данных на более качественные, поэтому эти аспекты проекта программы должны всегда находиться в поле пристального внимания разработчиков. Также отмечалось, что блокаторы оптимизации, такие как использование псевдонимов памяти и обращения к процедурам, серьезно ограничивают способность компиляторов к выполнению расширенных видов оптимизации. Опять же, программист должен нести первоочередную ответственность за их устранение.

Помимо всего сказанного, в главе была рассмотрена серия методик, в которую входят развертка циклов, разбиение итераций и арифметика указателей. По мере углубления в изучение оптимизации, особую важность приобретают рассмотрение генерированного компонующего автокода и попытка понимания того, как машина осуществляет вычисления. Для выполнения кода на современном нестандартном процессоре большой объем полезной информации можно получить в результате анализа того, как программа будет выполняться на машине с неограниченными ресурсами обработки, на которой значения времени задержки и выдачи функциональных устройств соответствуют тем же значениям целевого процессора. Для совершенствования указанного анализа также необходимо принимать во внимание такие сдерживающие факторы, как количество и типы функциональных устройств.

Программы, включающие в себя условное ветвление или комплексные итерации с системой памяти, проанализировать и оптимизировать намного труднее, нежели простые рассмотренные петлевые программы. Основной стратегией является попытка сделать петли (циклы) более прогнозируемыми, а также сократить итерации между операциями сохранения и загрузки.

При работе с крупными программами важность приобретает сосредоточение попыток оптимизации на частях системы, выполнение которых занимает большую часть времени. Профайлеры кодов и аналогичные инструментальные средства могут оказать помощь в систематической оценке и улучшении производительности программы. В главе описан GPROF — стандартный инструмент профилирования от Unix. Также доступны и более усовершенствованные профайлеры, такие как система разработки программ VTUNE от Intel. Для измерения производительности каждого стандартного блока эти инструментальные средства могут разбить время выполнения на уровень ниже программного. Стандартный блок — это последовательность команд без условных операций.

Закон Эмдала обеспечивает простое, но действенное понимание прироста производительности, получаемое усовершенствованием всего лишь одной части системы. Этот прирост зависит как от степени усовершенствования, так и от того, насколько большая доля времени от общего требуется на выполнение этой части системы.

Библиографические примечания

На тему способов оптимизации с помощью компиляторов написано много книг. В книге Мачника (Muchnick) они описаны наиболее полно [55]. В книге Уодлея (Wadleigh) и Кроуфорда (Crawford) по оптимизации программного обеспечения [85] описывается уже представленный материал, а также процесс достижения высокой производительности на параллельных машинах.

Функционирование нестандартного процессора довольно короткое и абстрактное. Более полное раскрытие общих принципов представлено в учебниках по усовершенствованной компьютерной архитектуре, например, в книге Хеннеси и Паттерсона [33, глава 3]. Шрайвер (Shriver) и Смит (Smith) в подробностях описывают процессор AMD [69].

Закон Эмдала представлен во многих материалах по компьютерной архитектуре. Делая основной упор на количественную оценку системы, Хеннеси и Паттерсон в своей книге [33] рассматривают данный предмет вполне исчерпывающе.

Задачи для домашнего решения

УПРАЖНЕНИЕ 5.11 ◆◆

Предположим, что необходимо написать процедуру, вычисляющую скалярное произведение двух векторов. Абстрактная версия этой функции имеет СРЕ 54 как для целочисленных данных, так и для данных с плавающей точкой. Выполнением того же типа преобразований, что и для преобразования абстрактной программы `combine1` в более эффективную программу `combine4`, получаем следующий код:

```

1  /* Накопление во временной переменной */
2  void inner4 (vec_ptr u, vec_ptr v, data_t *dest)
3  {
4      int i;
5      int length = vec_length (u);
6      data_t *udata = get_vec_start (u);
7      data_t *vdata = get_vec_start (v);
8      data_t sum = (data_t) 0;
9
10     for (i = 0; i < length; i++) {
11         sum = sum + udata[i] * vdata[i];
12     }
13     *dest = sum;
14 }

```

Измерения показывают, что функция требует CPI 3.11 для целочисленных данных. Компонующий автокод для внутреннего цикла — следующий:

udata in %esi, vdata in %ebx, i in %edx, sum in %ecx, length in %edi

```

1 .L24:                                loop:
2         movl (%esi,%edx,4),%eax          Получение udata [i]
3         imull (%ebx,%edx,4),%eax        Умножение на vdata [i]
4         addl %eax,%ecx                Добавление к сумме
5         incl %edx                   i++
6         cmpl %edi,%edx              Сравнение i:length
7         jle .L24                    If <, goto loop

```

Предположим, что целочисленное умножение выполняется общим целочисленным функциональным устройством, и что это устройство конвейеризовано. Это означает, что через один цикл синхронизации после начала умножения может начаться новая операция (умножения или другая). Предположим также, что функциональное устройство Integer/Branch может выполнять простые целочисленные операции.

1. Покажите преобразование этих строк компонующего автокода в последовательность операций. Команда `movl` преобразуется в единую операцию `load`. Регистр `%eax` обновляется в этом цикле дважды. Отметьте ярлыками разные версии `%eax.1a` и `%eax.1b`.
2. Объясните, как данная функция может выполнятся быстрее, чем количество циклов, необходимое для целочисленного умножения.
3. Объясните, какой фактор ограничивает производительность данного кода до лучшего CPI в 2.5.
4. Для данных с плавающей точкой получаем значение CPI, равное 3.5. Без необходимости изучения компонующего автокода опишите фактор, ограничивающий производительность, в лучшем случае, до 3 циклов на итерацию.

УПРАЖНЕНИЕ 5.12 ♦

Напишите версию процедуры скалярного произведения, описанной в упр. 5.11, в котором применяется четырехсторонняя развертка цикла.

Измерения для этой процедуры дают СРЕ в 2.20 для целочисленных данных и 3.50 для данных с плавающей точкой.

1. Объясните, почему любая версия любой процедуры скалярного произведения не может достичь СРЕ выше 2?
2. Объясните, почему производительность для данных с плавающей точкой не повышается с разверткой цикла?

УПРАЖНЕНИЕ 5.13 ♦

Напишите версию процедуры скалярного произведения, описанной в упр. 5.11, в которой применялись бы четырехсторонняя развертка цикла и двухсторонний параллелизм.

Измерения для этой процедуры дают СРЕ в 2.25 для данных с плавающей точкой. Опишите два фактора, ограничивающих СРЕ, в лучшем случае, до 2.0.

УПРАЖНЕНИЕ 5.14 ♦♦

Программист присоединился к рабочей группе, занимающейся созданием самой быстродействующей в мире факториальной процедуры. Начав с рекурсивного факто-риала, группа преобразовала код для использования итерации:

```

1     int fact (int n)
2     {
3         int i;
4         int result = 1;
5
6         for (i = n; i > 0; i--)
7             result = result * i;
8         return result;
9     }

```

При этом число СРЕ для данной функции было сокращено с 63 до 4 (измерения проводились на Intel Pentium III (в самом деле!). Однако рабочая группа хочет добиться еще более впечатляющих результатов.

Один из программистов слышал о развертке циклов и сгенерировал следующий код:

```

1     int fact_u2 (int n)
2     {
3         int i;
4         int result = 1;
5         for (i = n; i > 0; i-=2)  (
6             result = (result * i) * (i-1);
7         }
8         return result;
9     }

```

К сожалению, рабочая группа обнаружила, что для некоторых значений аргумента *n* данный код возвращает 0.

- Для каких значений *n* *fact_u2* и *fact* возвращают различные значения?
- Покажите, как можно зафиксировать *fact_u2*. Обратите внимание, что для этой процедуры существует особый прием, заключающийся в изменении границы цикла.
- Разметка *fact_u2* не дает повышения производительности. Как это объяснить?
- Строка в рамках цикла модифицируется. К общему изумлению, измеренная производительность теперь имела показатель СРЕ в 2.5. Как объяснить такое повышение производительности?

```
result = result * (i * (i - 1));
```

УПРАЖНЕНИЕ 5.15 +

Используя команду условного перехода, напишите компонующий автокод для тела следующей команды:

```
1      /* Возврат максимального x и y */
2      int max(int x, int y)
3      {
4          return (x < y) ? y : x
5      }
```

УПРАЖНЕНИЕ 5.16 +

С использованием условных переходов общая техника преобразования оператора формы

```
val = cond-expr ? then-expr : else-expr;
```

должна генерировать код формы

```
val = then-expr;
temp = else-expr;
test = cond-expr;
if (test) val = temp;
```

Последняя строка реализуется командой условного перехода. Используя пример из упр. 5.7 в качестве руководства, изложите общие требования для того, чтобы данное преобразование было действительно.

УПРАЖНЕНИЕ 5.17 ++

Следующая функция вычисляет сумму элементов в связном списке:

```
1      int list_sum (list_ptr ls)
2      {
3          int sum = 0;
4      }
```

```

5         for ( ; ls; ls = ls->next)
6             sum += ls ->data;
7
8     }

```

Компонующий автокод для цикла и преобразования первой итерации в операции дает следующее:

Ассемблерные команды	Операции устройства выполнения	
.L43:		
addl 4(%edx), %eax	movl 4(%edx.0)	t.1
movl (%edx), %edx	addl t.1, %eax.0	%eax.1
testl %edx, %edx	load (%edx.0)	%edx.1
jne .L43	testl %edx.1, %edx.1	cc.1
	jne-taken	cc.1

- Изобразите граф, демонстрирующий планирование операций для первых трех итераций цикла, по стилю рис. 5.15. Помните о том, что устройство загрузки только одно.
- Измерения для данной функции дают CPE 4.00. Не противоречит ли это графу, изображеному в пункте 1?

УПРАЖНЕНИЕ 5.18 ◆◆◆

Следующая функция является вариантом функции суммы списка, приведенной в упр. 5.17:

```

1     int list_sum2 (list_ptr ls)
2     {
3         int sum = 0;
4         list_ptr old;
5
6         while (ls) {
7             old = ls;
8             ls = ls->next;
9             sum += old ->data;
10        }
11        return sum;
12    }

```

Этот код написан так, что доступ к памяти для выборки следующего элемента списка возникает до доступа к памяти для получения поля данных из текущего элемента.

Компонующий автокод для цикла и преобразования первой итерации в операции дает следующее:

Ассемблерные команды	Операции устройства выполнения
.L48: movl %edx, %ecx movl (%edx), %edx addl 4(%ecx), %eax testl %edx, %edx jne .L43	load (%edx.0) %edx.1 movl 4(%edx.0) t.1 addl t.1, %eax.0 %eax.1 testl %edx.1, %edx.1 cc.1 jne-taken cc.1

Обратите внимание на то, что операция перемещения регистра

movl %edx, %ecx

не требует никаких операций для реализации. Она управляется простым связыванием метки edx.0 с регистром %ecx, поэтому последующая команда

addl 4 (%ecx), %eax

преобразуется для использования edx.0 в качестве исходного операнда.

1. Изобразите граф, демонстрирующий планирование операций для первых трех итераций цикла, по стилю рис. 5.15. Помните о том, что устройство загрузки только одно.
2. Измерения для данной функции дают СРЕ 3.00. Не противоречит ли это графу, изображеному в пункте 1?
3. Насколько лучше эта функция использует устройство загрузки, чем функция, описанная в упр. 5.17?

УПРАЖНЕНИЕ 5.19 ◆

Предположим, что получено задание на повышение производительности программы, состоящей из трех частей:

- часть А требует 20% общего времени выполнения;
- часть В требует 20% общего времени выполнения;
- часть С требует 20% общего времени выполнения.

Определено, что за 1000 долларов можно повысить быстродействие части В с коэффициентом 3.0 или части С с коэффициентом 1.5. Какой из этих двух вариантов максимально повысит производительность?

Решение упражнений

РЕШЕНИЕ УПРАЖНЕНИЯ 5.1

Данная задача иллюстрирует некоторые тонкости использования псевдонимов памяти. Как показано в приведенном прокомментированном коде, эффектом будет установка значения при хр нулевым:

```

1      *xp = *xp + *xp; /* 2x */
2      *xp = *xp - *xp; /* 2x-2x = 0 */
3      *xp = *xp - *xp; /* 0 - 0 = 0 */

```

Данный пример иллюстрирует интуитивное предположение о том, что поведение программы часто бывает некорректным. Естественной кажется мысль о случае, когда xp и ur явно выражены, но при этом не придается значения вероятности того, что они могут быть равными. Ошибки и сбои часто возникают из-за условий, о которых программист даже не догадывается.

РЕШЕНИЕ УПРАЖНЕНИЯ 5.2

Данный пример иллюстрирует отношение между СРЕ и временем работы до первого отказа. Он решается с помощью элементарной алгебры. Вычисляется, что для $n \leq 2$ версия 1 является самой быстродействующей. Версия 2 является самой быстродействующей для $3 \leq n \leq 7$, а версия 3 — для $n \geq 8$.

РЕШЕНИЕ УПРАЖНЕНИЯ 5.3

Упражнение очень простое, но является важным для признания того, что четыре оператора цикла `for` — первоначальный, тестирование, обновление и тело — выполняются разное количество раз.

Код	min	max	incr	square
1	1	91	90	90
2	91	1	90	90
3	1	1	90	90

РЕШЕНИЕ УПРАЖНЕНИЯ 5.4

Как описано в главе 3, восстановление структурной схемы и алгоритма работы по исходным текстам (конструирование) из компонующего автокода в код С дает полезную информацию о процессе компиляции. В приведенном коде дается форма общих данных и общей операции компоновки:

```

1      void combine5px8 (vec_ptr v, data_t *dest)
2      {
3          int length = vec_length (v);
4          int limit = length - 3;
5          *data_t *data = get_vec_start (v);
6          data_t x = IDENT;
7          int i;
8
9          /* Компоновка 8 элементов за один раз */
10         for (i = 0; i < limit; i+=8) {

```

```

11          x = x OPER data [0]
12                  OPER data [1]
13                  OPER data [2]
14                  OPER data [3]
15                  OPER data [4]
16                  OPER data [5]
17                  OPER data [6]
18                  OPER data [7]
19          data += 8;
20      }
21
22     /* Завершение всех оставшихся элементов */
23     for (; i < length; i++) {
24         x = x OPER data [0];
25         data++
26     }
27     * dest = x;
28 }

```

Написанный вручную код способен удалить переменную цикла *i* путем вычисления окончного значения указателя. Это другой пример того, как хорошо подготовленный программист может выявлять преобразования, не принимаемые во внимание компилятором.

РЕШЕНИЕ УПРАЖНЕНИЯ 5.6

Сброшенные значения обычно сохраняются во фрейме локального стека. Следовательно, они имеют отрицательный оффсет по отношению к `%ebp`. Такую ссылку можно увидеть в строке 12 компонующего автокода.

1. Переменная `limit` была сброшена в стек.
2. Относительно `%ebp` она смешена на `-8`.
3. Это значение необходимо только для определения, должна ли выбираться команда `j1`, закрывающая цикл. Если логика прогнозирует ветвь как выбранную, тогда следующая итерация может начаться до завершения тестирования цикла. Следовательно, команда сравнения не является частью критического пути, определяющего производительность цикла. Более того, поскольку эта переменная не изменяется по ходу цикла, ее наличие в стеке не требует никаких дополнительных операций сохранения.

РЕШЕНИЕ УПРАЖНЕНИЯ 5.6

Данный пример демонстрирует то, как изменения в программе могут вызывать драматические перепады производительности, особенно на машине с нестандартным выполнением. На рис. 5.21 показано планирование операций умножения для одной итерации функции каждой из ассоциаций. В каждую из итераций входит три операции умножения, каждая из которых берет старое значение `r` (показанное как `r.0`) и

вычисляет новое значение, обозначенное как $r.1$. Однако, как показывают сплошные линии, критический путь, т. е. минимальное время между последовательными обновлениями и r , может быть 12 (A1), 8 (A2 и A5) или 4 (A3 и A4). Предполагая, что процессор достигает максимального параллелизма, данный критический путь представляет единственное ограничение теоретического СРЕ.

Следовательно, в таблицу заносятся следующие данные:

Версия	Измеренное СРЕ	Теоретическое СРЕ
A1	4.00	$12/3 = 4.00$
A2	2.67	$8/3 = 2.67$
A3	1.67	$4/3 = 1.33$
A4	1.67	$4/3 = 1.33$
A5	2.67	$8/3 = 2.67$

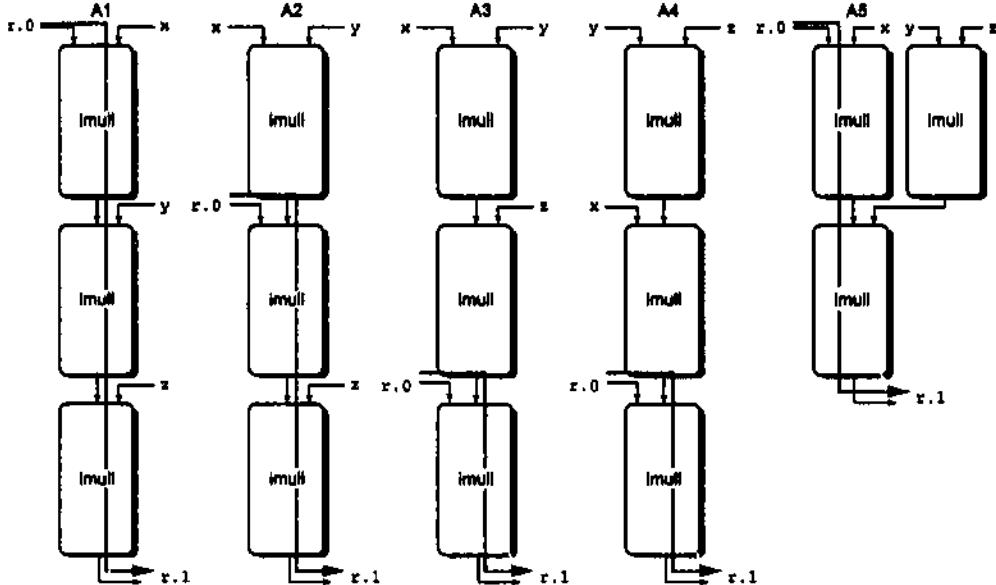


Рис. 5.21. Планирование операций умножения

Из данной таблицы и рис. 5.21 видно, что ассоциации A1, A2 и A5 достигают своего теоретического оптимума, тогда как A3 и A4 требуют 5 циклов на итерацию, а не 4.

РЕШЕНИЕ УПРАЖНЕНИЯ 5.7

Данный пример демонстрирует необходимость соблюдать осторожность при использовании условных переходов. Они требуют оценки значения исходного операнда, даже когда это значение не используется.

Данный код всегда принимает во внимание xp (команда B2). Это вызовет нулевую ссылку на указатель в случае, когда xp равно 0.

РЕШЕНИЕ УПРАЖНЕНИЯ 5.8

Данный пример требует анализа потенциальных взаимодействий "загрузка-сохранение" в программе.

- Каждый элемент $a[i]$ будет установлен как $i + 1$ для $0 \leq i \leq 998$.
- Каждый элемент $a[i]$ будет установлен нулевым для $1 \leq i \leq 999$.
- Во втором случае загрузка одной итерации зависит от результата сохранения из предыдущей итерации. Таким образом, между последовательными итерациями существует зависимость запись/считывание.
- Значение СРЕ составит 5.00, поскольку между сохранениями и последующими загрузками не существует зависимостей.

РЕШЕНИЕ УПРАЖНЕНИЯ 5.9

Упражнение иллюстрирует тот факт, что закон Эмдала применим не только в области разработки компьютерных систем.

- При рассмотрении уравнения (5.1) видно, что $\alpha = 0.6$, а $k = 1.5$. Если выражаться более понятно, то 1500 километров по Монтане займет 10 часов; остаток пути также займет 10 часов. Это дает ускорение в $25/(10 + 10) = 1.25$.
- При рассмотрении уравнения (5.1) видно, что $\alpha = 0.6$, а требуется $S = 5/3$, из чего можно вывести решение для k . Кроме того, для ускорения движения с коэффициентом $5/3$ общее время поездки необходимо снизить до 15 часов. На проезд областей за пределами штата Монтана также потребуется 10 часов, поэтому через Монтану нужно проехать за 5 часов. Значит, необходимо ехать со скоростью 300 км/ч, что для тяжелого грузовика практически нереально.

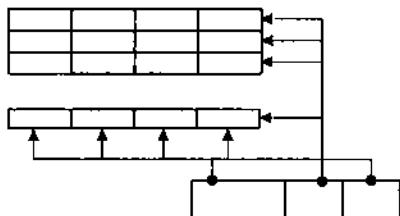
РЕШЕНИЕ УПРАЖНЕНИЯ 5.10

Легче всего закон Эмдала понять на конкретных примерах. Приведенный пример требует рассмотрения уравнения (5.1) с необычной точки зрения.

Дано $S = 2$ и $\alpha = 0.8$, после чего выполняется расчет для k :

$$\begin{aligned} 2 &= \frac{1}{(1 - 0.8) + 0.8/k} \\ 0.4 + 1.6/k &= 1.0 \\ k &= 2.67 \end{aligned}$$

ГЛАВА 6



Иерархия памяти

- Технологии сохранения информации.
- Локальность.
- Иерархия памяти.
- Кэш-память.
- Написание кодов, дружественных кэш-памяти.
- Влияние кэша на производительность программ.
- Использование локальности в программах.
- Резюме.

В предыдущих главах при изучении компьютерных систем читатели рассматривали простую модель центрального процессора, выполняющего команды и обладающего системой памяти, сохраняющей для процессора эти команды и данные. В этой простой модели система памяти представляет собой линейный массив данных, и процессор может осуществлять доступ к каждой ячейке памяти за постоянный отрезок времени. Несмотря на то, что в своем роде эта модель достаточно эффективна, она все-таки не отражает реальных принципов работы современных компьютерных систем.

На практике система памяти представляет собой иерархию запоминающих устройств различных объемов, стоимости и времени доступа. В регистрах CPU содержатся наиболее часто используемые данные. Небольшие быстродействующие устройства кэш-памяти, расположенные рядом с CPU, выполняют функции этапных областей для подмножеств данных и команд, сохраняемых в сравнительно медленной основной памяти. Последняя организует данные, хранящиеся на больших, медленнодействующих дисках, которые, в свою очередь, служат этапными областями для данных, сохраняемых на дисках или лентах других машин, объединенных в сети.

Общим эффектом является достаточно обширная буферная память, стоящая как дешевое запоминающее устройство в нижней части иерархии, но передающая данные в программы со скоростью быстродействующего запоминающего устройства, стоящего в иерархии на более высоком уровне.

Любому программисту необходимо досконально разбираться в принципах работы иерархии памяти, потому что она серьезно влияет на производительность создаваемых приложений. Если необходимые для программы данные хранятся в регистре центрального процессора, тогда доступ к ним может осуществляться во время выполнения команды. Если данные хранятся в кэш-памяти, тогда на доступ к ним требуется 1—10 циклов. Соответственно, на доступ к данным, хранящимся в основной памяти, требуется 50—100 циклов синхронизации, а на диске — порядка 20 млн циклов!

Здесь программисты опять имеют дело с фундаментальной и непреходящей идеей любой компьютерной системы: программы можно писать так, что элементы их данных будут сохраняться на более высоких уровнях иерархии, где CPU будет иметь возможность оперативного доступа к ним.

Данная мысль тесно связана с фундаментальным принципом создания компьютерных программ, называемым *локальностью*. Программы с хорошей локальностью стремятся к многократному доступу к одному и тому же множеству элементов данных, либо к множествам элементов данных, расположенных рядом. Программы с хорошей локальностью стремятся к доступу к большему количеству элементов данных из верхних уровней иерархии памяти, нежели программы с пониженной локальностью, и поэтому работают быстрее. Например, время выполнения перемножения матриц с разными степенями локальности может варьироваться в шесть раз!

В настоящей главе авторами рассматриваются основные технологии сохранения информации — память SRAM, память DRAM, память ROM и диски, а также описывается их организация в иерархии. В частности, делается упор на устройства кэш-памяти в роли этапных областей между CPU и основной памятью, потому что они оказывают наибольшее влияние на производительность программного приложения. Будет показано, как анализировать написанные программы на предмет локальности, а также представлены методики ее повышения. Читатели научатся характеризовать иерархию памяти конкретной машины "горой памяти", отображающей время доступа как функцию локальности.

6.1. Технологии сохранения информации

Большая часть успешности компьютерной технологии исходит от невероятного прогресса технологий сохранения информации. Первые компьютеры имели всего несколько килобайт ОЗУ. Самые первые IBM не имели даже жесткого диска. Все изменилось с выпуском на рынок в 1982 г. модели IBM PC-XT с диском в 10 Мбайт. К 2000 г. самая обычная машина уже имела в 1000 раз больше дисковой памяти, и каждые два или три года этот коэффициент возрастает в 10 раз.

6.1.1. Память с произвольной выборкой

Память с произвольной выборкой (RAM) выпускается в двух вариантах: статическом и динамическом. Статическая RAM (SRAM) — намного более быстродействующая и дорогостоящая, нежели динамическая RAM (DRAM). SRAM используется для кэш-памяти как на чипе процессора, так и вне его. DRAM используется для основной па-

мяти и для буфера графических систем. Как правило, обычные настольные системы имеют не более нескольких мегабайт SRAM, но сотни или тысячи мегабайт DRAM.

Статическая RAM

SRAM сохраняет каждый бит в бистабильной (с двумя устойчивыми состояниями) ячейке памяти. Каждая ячейка реализуется с помощью цепи из шести транзисторов. Эта цепь обладает свойством бесконечного нахождения в любой из двух разных конфигураций напряжений, называемых состояниями. Любое другое состояние будет неустойчивым: из него цепь быстро переместится в одно из устойчивых состояний. Такая ячейка памяти аналогична обратному маятнику, показанному на рис. 6.1.



Рис. 6.1. Обратный маятник

Динамическая RAM

DRAM сохраняет каждый бит как заряд конденсатора. Последний — очень маленький, как правило, порядка 30 фемтофарад (30×10^{-15} фарад). Однако следует помнить, что фарад — это очень большая единица измерения. Память DRAM можно значительно уплотнить: каждая ячейка состоит из конденсатора и одного транзистора доступа. В отличие от SRAM, ячейка памяти DRAM очень чувствительна к любым возбуждениям. При изменении напряжения конденсатора оно никогда не восстанавливается. Значения напряжений конденсатора может измениться даже под воздействием солнечных лучей. На самом деле, датчики в цифровых фотоаппаратах и видеокамерах являются массивами ячеек DRAM.

Различные источники тока утечки вызывают разрядку ячейки DRAM в течение порядка 10—100 мс. К счастью, для компьютеров, работающих по циклам синхронизации, измеряемых в нс, такое время задержки довольно значительно. Система памяти должна периодически обновлять каждый бит памяти путем его считывания и перезаписи. В некоторых системах также применяются коды с исправлениями ошибок, когда в компьютерные слова добавляется несколько дополнительных битов (например, 32-битовое слово можно закодировать с помощью 38 битов) так, что схема может выявить и исправить любой одиночный ошибочный бит в рамках слова.

В табл. 6.1 в общем виде представлены характеристики памяти SRAM и DRAM. SRAM устойчива до тех пор, пока ячейки находятся под напряжением. В отличие от DRAM, обновление здесь не обязательно. Доступ к SRAM осуществляется быстрее, нежели к DRAM. Память SRAM не восприимчива к таким возбудителям, как свет и электрические помехи. Компромисс заключается в том, что ячейки SRAM использу-

ют больше транзисторов, чем ячейки DRAM, и, следовательно, имеют меньшую плотность, более дорогие и потребляют больше электроэнергии.

Таблица 6.1. Характеристики памяти SRAM и DRAM

	Транзисторов на бит	Относительное время доступа	Устойчивая	Воспринимчивая	Относительные затраты	Приложения
SRAM	6	1X	Да	Нет	100X	Кэш-память
DRAM	1	10X	Нет	Да	1X	Основная память, кадровые буферы

Стандартные DRAM

Ячейки (биты) в чипе DRAM разделены на d суперячеек, каждая из которых состоит из w ячеек DRAM. DRAM сохраняет всего dw битов информации. Суперячейки расположены в виде прямоугольного массива с r строк и c столбцов, где $rc = d$. Каждая суперячейка имеет адрес в форме (i, j) , где i обозначает строку, а j — столбец.

Например, на рис. 6.2 показана организация чипа DRAM 16×8 с $d = 16$ суперячейками, $w = 8$ битов на суперячейку, $r = 4$ строки и $c = 4$ столбца. Заштрихованная клетка обозначает суперячейку в адресе $(2,1)$. Информация поступает на чип и выходит из него через внешние соединители, называемые контактами. Каждый контакт передает однобитовый сигнал. На рис. 6.2 показаны два из этих наборов контактов: 8 контактов данных для переноса одного байта данных на чип или с него, и 2 контакта адреса, передающих двухбитовую строку и столбец адресов суперячеек. Другие контакты, передающие управляющую информацию, не показаны.

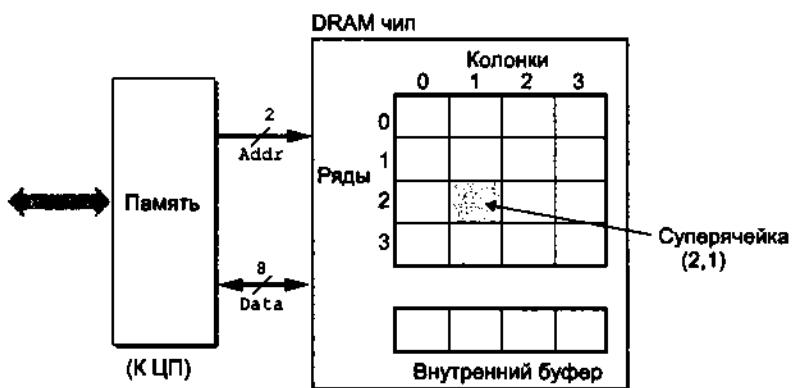


Рис. 6.2. Представление высокого уровня 128-битового чипа DRAM 16×8

Примечание по терминологии

Представители сообщества разработчиков устройств памяти не смогли договориться о едином названии элемента массива DRAM. Компьютерные архитекторы называют его "ячейкой", перегружая термин названием "ячейкой памяти DRAM". Проектировщики цепей называют его "словом", перегружая термин названием "слово основной памяти". Во избежание путаницы, авторы используют недвусмыслиенный термин "суперячейка".

Каждый чип DRAM подключен к определенной цепи, называемой *контроллером памяти*, по которой за одну единицу времени может передаваться w бит на чип DRAM и с него. Для считывания содержимого суперячейки (i, j) контроллер памяти отправляет адрес строки i на DRAM, за которым следует адрес столбца j . DRAM отвечает отправкой содержимого суперячейки (i, j) назад на контроллер. Адрес строки i называется *запросом RAS* (Row Access Strobe, строб адреса строки). Адрес столбца j называется *запросом CAS* (Column Access Strobe, строб адреса столбца). Следует обратить внимание, что запросы RAS и CAS совместно используют одни и те же контакты адреса DRAM.

Например, для считывания суперячейки $(2,1)$ с DRAM 16×8 на рис. 6.2 контроллер памяти отправляет адрес строки 2 (рис. 6.3, а). DRAM отвечает копированием всего содержимого строки 2 в буфер внутренней строки. Далее контроллер памяти отправляет адрес столбца 1 (рис. 6.3, б). DRAM отвечает копированием 8 битов в суперячейку $(2,1)$ из буфера внутренней строки и отправкой их на контроллер памяти.

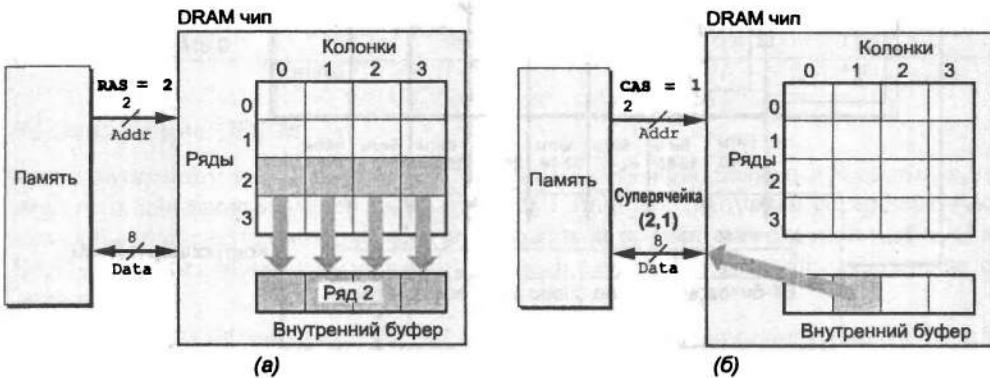


Рис. 6.3. Считывание содержимого суперячейки DRAM: а — запрос RAS; б — запрос CAS

Одной из причин организации проектировщиками цепей устройств памяти DRAM в виде двумерных, а не линейных массивов является сокращение числа адресных контактов на чипе (микросхеме). Например, если бы уже упоминавшаяся 128-битовая DRAM была организована в виде линейного массива, состоящего из 16 суперячейек с адресами от 0 до 15, тогда микросхеме потребовалось бы четыре адресных контакта, а не два. Недостаток двумерного массива заключается в том, что адреса должны отправляться двумя раздельными этапами, что увеличивает время доступа.

Модули памяти

Кристаллы DRAM объединены в пакеты модулей памяти, подключаемые к слотам расширения на материнской плате. В обычные пакеты входит 168-контактный Dual Inline Memory Module (DIMM, модуль памяти с двухрядным расположением выводов), передающий данные на контроллер памяти и из него в виде 64-битовых порций, а также 72-битовый Single Inline Memory Module (SIMM, модуль памяти с однорядным расположением выводов), передающий данные 32-битовыми порциями.

На рис. 6.4 представлена базовая идея модуля памяти. Модуль сохраняет всего 64 Мбайт, используя восемь 64-Мбайтовых $8\text{M} \times 8$ чипов DRAM, пронумерованных от 0 до 7. Каждая суперячейка сохраняет один байт основной памяти, а каждое 64-битовое двойное слово¹ в байтовом адресе A в основной памяти представлено восемью суперячейками, с соответствующим адресом (i, j) . В примере, показанном на рис. 6.4, DRAM 0 сохраняет первый байт (младшего разряда), DRAM 1 сохраняет следующий байт и т. д.

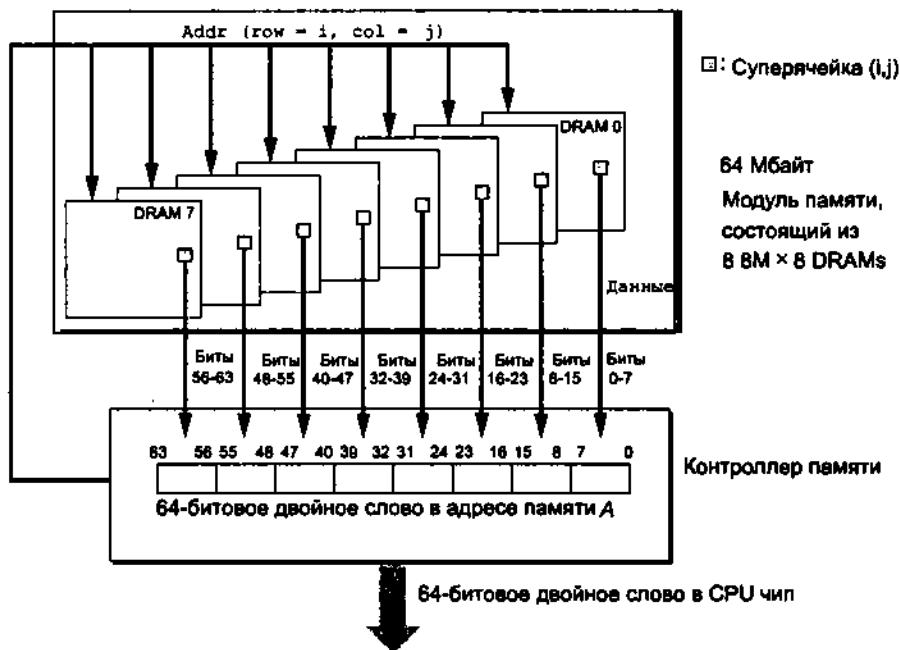


Рис. 6.4. Считывание содержимого модуля памяти

Для извлечения 64-битового двойного слова адреса памяти A контроллер памяти преобразует A в адрес суперячейки (i, j) и отправляет его на модуль памяти, который затем передает i и j на каждую DRAM. В ответ на это DRAM выдает 8-битовое содержимое своей суперячейки (i, j) . Цепь модуля собирает эти выходные данные и

¹ В IA32 это 64-битовое количество называлось бы "quadword" (квадратичное слово).

формирует их в 64-битовое двойное слово, после чего возвращает его в контроллер памяти.

Основную память можно агрегировать путем подключения множественных модулей памяти к контроллеру памяти. В этом случае, когда контроллер получает адрес A , тогда он выбирает модуль k , содержащий A , преобразует A в его форму (i, j) и отправляет (i, j) в модуль k .

УПРАЖНЕНИЕ 6.1

Пусть r — количество строк в массиве DRAM, c — число столбцов, b_r — число бит, необходимое для адресации строк, а b_c — число бит, необходимое для адресации столбцов. Для каждой из следующих DRAM определите размеры массива степени двойки, минимизирующего $\max(b_c, b_r)$ — максимальное число битов, необходимое для адресации строк и столбцов массива.

Организация	r	c	b_r	b_c	$\max(b_c, b_r)$
16×1					
16×4					
128×8					
512×4					
1024×4					

Расширенные DRAM

Существует много типов памяти DRAM, и новые регулярно появляются на рынке, по мере того, как производители пытаются "идти в ногу" с постоянно совершенствующимся быстродействием процессоров. Каждый тип основан на традиционной ячейке DRAM с оптимизациями, совершенствующими скорость, с которой осуществляется доступ.

□ Режим быстрой страницы DRAM (FPM DRAM). Традиционная память DRAM копирует всю строку суперячеек во внутренний буфер строки, использует одну суперячейку, а остальные отбрасывает. При этом FPM DRAM совершенствуется путем обеспечения последовательных доступов к той же строке для обслуживания напрямую из буфера строки. Например, для считывания четырех суперячеек из строки i традиционной DRAM контроллер памяти должен отправить четыре запроса RAS/CAS, несмотря на то, что адрес строки i в каждом случае одинаков. Для считывания суперячеек из той же строки FPM DRAM контроллер памяти направляет первоначальный запрос RAS/CAS, за которым следуют три запроса CAS. Первый запрос RAS/CAS копирует строку i в буфер строки и запрашивает первую суперячейку. Следующие три суперячейки обрабатываются непосредственно из буфера строки, и поэтому быстрее, чем первоначальная суперячейка.

- Выход расширенных данных DRAM (EDO DRAM). Расширенная форма FPM DRAM, позволяющая сближать отдельные сигналы CAS во времени.
- Синхронная DRAM (SDRAM). Традиционная, FPM и EDO DRAM — асинхронны в том смысле, что они взаимодействуют с контроллером памяти через набор явных управляющих сигналов. SDRAM замещает многие из этих управляющих сигналов нарастающими фронтами того же внешнего синхронизирующего сигнала, что приводит в действие контроллер памяти. Не вдаваясь в детали, можно сказать, что чистым эффектом станет выдача SDRAM содержимого суперчек с большей скоростью, чем ее асинхронные эквиваленты.
- Синхронная DRAM с удвоенной скоростью обработки данных (DDR DRAM). BBK DRAM — это усовершенствованная SDRAM, удваивающая скорость DRAM использованием в качестве управляющих сигналов обоих фронтов синхроимпульса.
- Rambus DRAM (RDRAM). Это альтернативная патентованная технология с более высокой максимальной полосой пропускания, чем у DDR SDRAM.
- Video RAM (VRAM). Применяется в кадровых буферах графических систем. По духу VRAM близка к FPM DRAM. Два основных различия: выходные данные VRAM создаются сдвигом всего содержимого внутреннего буфера в последовательность, и VRAM обеспечивает параллельные считывание и запись в память. Таким образом, система может раскрашивать экран пикселями в кадровом буфере (считывание) с параллельной записью новых значений для следующего обновления (запись).

Историческая популярность технологий DRAM

До 1995 г. большинство персональных компьютеров были оснащены FPM DRAM. С 1996 до 1999 г. на рынке доминировали EDO DRAM, а FPM DRAM практически сошли со сцены. SDRAM впервые появились в 1995 г. в высокотехнологичных системах, а к 2002 г. большинство компьютеров уже оснащено SDRAM и DDR SDRAM.

Энергонезависимая память

Типы памяти DRAM и SDRAM являются энергозависимыми в том смысле, что при выключении электропитания информация разрушается. Энергонезависимая же память сохраняет информацию даже при отключении питания. Существует множество типов энергонезависимой памяти. По историческим причинам они носят общее название *постоянной памяти* (Read-Only Memory, ROM), хотя в некоторых типах ROM информацию можно как считывать, так и записывать. Типы памяти ROM различаются по количеству раз, которое их можно перепрограммировать (записывать), а также механизмами такого перепрограммирования.

Программируемую ROM (PROM) можно запрограммировать только один раз. В PROM входит нечто, похожее на плавкий предохранитель для каждой ячейки памяти, который может "перегореть" один раз при скачке тока.

Стираемая программируемая ROM (EPROM) имеет прозрачное кварцевое окошко, обеспечивающее доступ света к ячейкам хранения информации. Информация в ячей-

как EEPROM стирается до нуля при попадании на них через окошко ультрафиолетовых лучей. Программирование EEPROM осуществляется с помощью специального устройства для записи ячеек в EEPROM. Последнюю можно очистить и перепрограммировать порядка 1000 раз. Электрически стираемая ROM (EEPROM) сходна с EPROM, но не требует физически независимого программируемого устройства и, следовательно, может перепрограммироваться на месте на печатных платах. EEPROM можно перепрограммировать порядка 10^5 раз. Флэш-память — это семейство небольших энергонезависимых плат памяти на базе EEPROM, которые можно как подключить, так и снять с настольных, карманных компьютеров или игровой приставки.

Программы, сохраняемые в устройствах ROM, часто называются "зашитыми". При подаче питания на компьютерную систему, она запускает зашитые программы, сохраненные в ROM. В некоторых системах в зашитых программах предусмотрены небольшие наборы примитивных функций ввода и вывода, например, рутинные процедуры BIOS персонального компьютера (базовая система ввода-вывода). Сложные устройства, такие как видеокарты и дисковые накопители, также полагаются на зашитые приложения для преобразования запросов ввода-вывода из центрального процессора.

Доступ к основной памяти

Данные перемещаются между процессором и основной памятью DRAM через совместно используемые электрические цепи, называемые *шинами*. Каждый перенос данных между CPU и памятью завершается серией шагов, называемых передачей по шине. Транзакция считывания передает данные из основной памяти в CPU. Транзакция записи передает данные из CPU в основную память.

Шина — это набор параллельных проводов, передающих адрес, данные и управляющие сигналы. В зависимости от конструкции той или иной шины, данные и адресные сигналы могут совместно использовать либо один и тот же набор проводов, либо разные. Также одна шина может использоваться совместно более чем двумя устройствами. Управляющие провода передают сигналы, синхронизирующие транзакцию и идентифицирующие тип выполняемой в данный момент транзакции. Например, представляет эта транзакция интерес для основной памяти или для какого-либо другого устройства ввода-вывода (для контроллера диска)? Это транзакция считывания или записи? Является ли информация на шине адресом или данными?

На рис. 6.5 показана конфигурация обычной настольной рабочей станции. Основными компонентами являются чип CPU, микропроцессорный набор, который называется *мостом ввода-вывода* (куда входит контроллер памяти), и модули памяти DRAM, составляющие основную память. Эти компоненты соединены парой шин: *системной шиной*, подключающей CPU к мосту ввода-вывода, и *шиной памяти*, подключающей мост ввода-вывода к основной памяти.

Мост ввода-вывода преобразует электрические сигналы в электрические сигналы шины памяти. Как будет видно, мост ввода-вывода также подключает системную шину и шину памяти к мосту ввода-вывода, совместно используемому устройствами ввода-вывода, такими как диски и графические карты. Впрочем, на данном этапе со средоточим внимание на шине памяти.

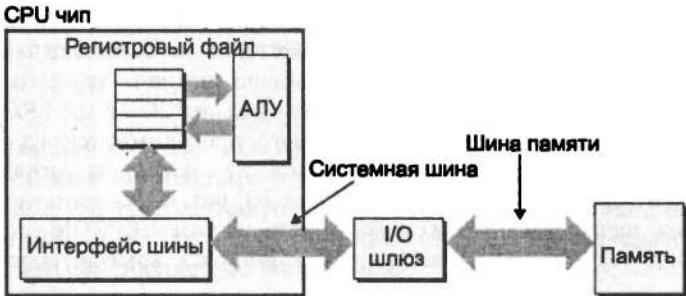
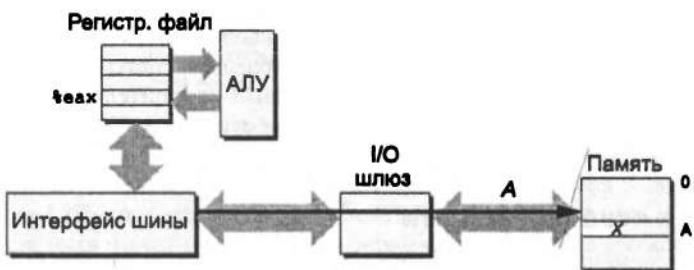
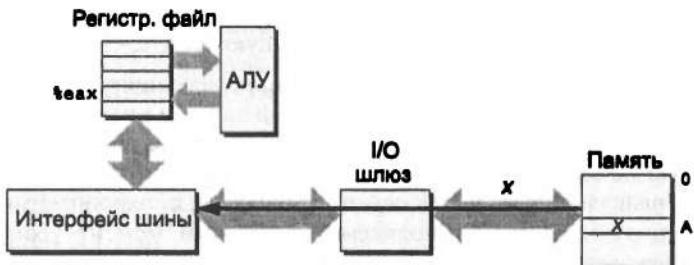


Рис. 6.5. Типичная структура шин, соединяющая CPU с основной памятью

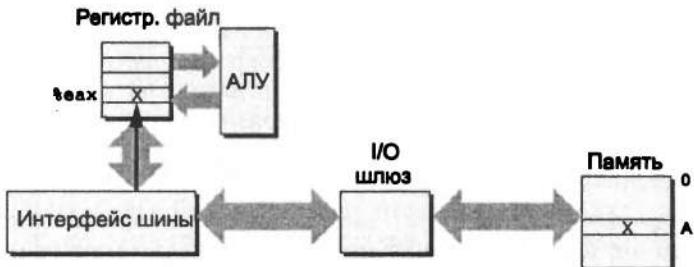


CPU помещает адрес A в шину



Память считывает адрес A

Получает слово X и помещает в шину



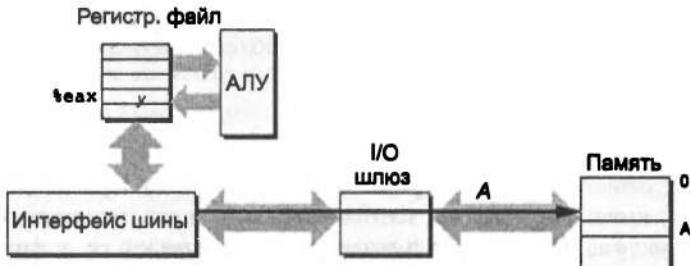
CPU считывает слово X с шины, копирует в регистр

Рис. 6.6. Транзакция считывания из памяти для операции загрузки

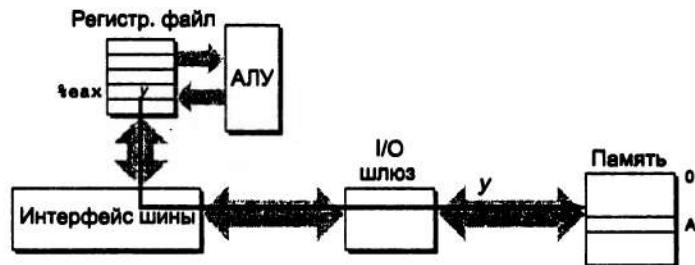
Рассмотрим, что произойдет, когда CPU будет выполнять такую операцию загрузки, как

```
movl A, %eax
```

где содержимое адреса A загружается в регистр %eax. Электрическая цепь на чипе CPU, называемая *интерфейсом шины*, инициирует на шине транзакцию считывания. Последняя состоит из трех шагов. Во-первых, CPU помещает адрес A на системную шину. Мост ввода-вывода передает сигнал на шину памяти (см. рис. 6.6). Далее, основная память распознает адресный сигнал на шине памяти, считывает его, выбирает слово данных из DRAM и записывает данные в шину памяти. Мост ввода-вывода преобразует сигнал шины памяти в сигнал системной шины и передает его на системную шину. Наконец, CPU распознает данные на системной шине, считывает их и копирует в регистр %eax.



CPU помещает адрес A в шину памяти Память читает и ждет слова данных



CPU помещает слово У в шину

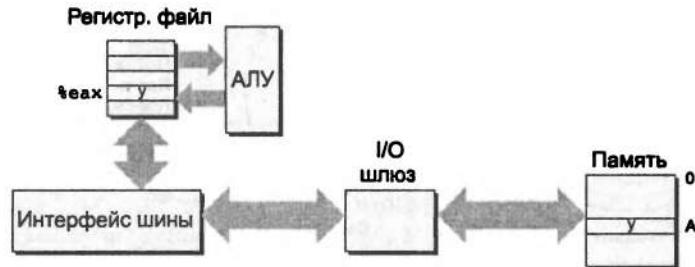


Рис. 6.7. Транзакция записи в память для операции сохранения

Наоборот, когда CPU, например, выполняет команду сохранения

```
movl %eax, A
```

где содержимое регистра `%eax` записывается в адрес `A`, CPU инициирует транзакцию записи. Здесь тоже присутствуют три шага. Сначала CPU размещает адрес на системной шине. Память считывает адрес с шины памяти и ожидает поступления данных (рис. 6.7). Далее CPU копирует слово данных в `%eax` в системную шину. Наконец, основная память считывает слово данных из шины памяти и сохраняет биты в DRAM.

6.1.2. Дисковый накопитель

Диски являются основными устройствами сохранения информации; на них содержатся огромные объемы данных — от 10 до сотен гигабайт (RAM-память содержит сотни или тысячи мегабайт). Однако время считывания информации с диска измеряется в миллисекундах — в сотни тысяч раз больше, нежели из DRAM, и в миллион раз больше, нежели из SDRAM.

Геометрия диска

Диски построены из так называемых "тарелок". Каждая тарелка (или компонент общего диска) имеет две стороны, или поверхности, покрытые магнитным записывающим материалом. Шпиндель в центре тарелки вращает ее с фиксированной скоростью, обычно составляющей 5400—15000 оборотов в минуту (об/мин). Диск обычно состоит из одной или нескольких таких тарелок, запечатанных в герметичный контейнер.

На рис. 6.8 показана геометрия типичной поверхности диска. Каждая поверхность состоит из множества концентрических колец, называемых дорожками. Каждая дорожка разделена на множество секторов. Каждый сектор содержит равное количество информационных разрядов (обычно 512 байтов), записанных на магнитном материале в секторе. Сектора разделены пропусками, в которых информационные биты не сохраняются. В пропусках сохраняются биты форматирования, идентифицирующие сектора.

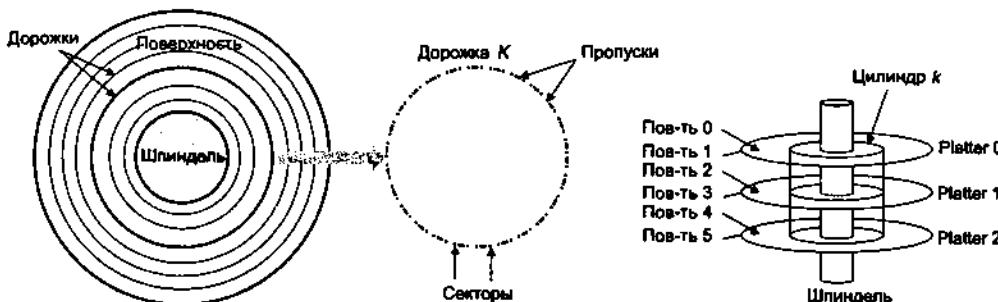


Рис. 6.8. Геометрия диска

Диск состоит из одной или нескольких тарелок (или пластин), расположенных одна над другой, запечатанных в герметичный контейнер. Производители дисков часто описывают геометрию многопластинчатых дисководов, используя термин "цилиндры", где цилиндром называется набор треков на всех поверхностях, равноудаленных от центра шпинделя. Например, если дисковод имеет три тарелки и шесть поверхностей, а дорожки на каждой поверхности пронумерованы последовательно, тогда цилиндр k представляет собой набор из шести экземпляров дорожки k .

Емкость диска

Максимальное число битов, которое можно записать на диск, называется его максимальной емкостью (или просто емкостью). Емкость диска определяется следующими технологическими факторами:

- плотность записи (бит/дюйм) — количество битов, которое можно уместить на однодюймовом сегменте дорожки;
- плотность дорожки (дорожка/дюйм) — количество дорожек, которые можно уместить на однодюймовом сегменте радиуса от центра тарелки;
- плотность размещения (бит/дюйм²) — произведение плотности записи и плотности дорожки.

Производители дисков неустанно работают над повышением плотности размещения (и следовательно, емкости), которые удваиваются каждые несколько лет. В первых дисках, разработанных в эру низкой плотности размещения, каждая дорожка была разделена на одинаковое количество секторов, определяемое количеством секторов, которое могло бы быть записано на самую внутреннюю дорожку. Для поддержания фиксированного количества секторов на каждую дорожку сектора отстояли дальше друг от друга на внешних дорожках. Это считалось разумным подходом, когда плотность размещения была сравнительно низкой. Однако с ее возрастанием пропуски между секторами (в которых не сохранялись биты данных) стали неприемлемо большими. Таким образом, в современных дисках стала применяться технология, известная как многозонная запись, при которой набор дорожек разбивался в несвязанные подмножества, называемые зонами записи. Каждая зона содержит смежный набор дорожек. Каждая дорожка в зоне имеет одинаковое количество секторов, которое можно уместить на самой внутренней дорожке зоны. Следует обратить внимание на то, что в дискетах до сих пор используется устаревшая технология с постоянным количеством секторов на каждую дорожку.

Обратите внимание, что производители выражают емкость диска в гигабайтах (Гбайт), где 1 Гбайт = 10^9 байтов.

1 гигабайт – это сколько?

К сожалению, значение таких префиксов, как "кило" (K), " mega" (M) и "гига" (G), зависит от контекста. Для измерений, относящихся к емкости DRAM и SRAM, обычно K = 2^{10} , M = 2^{20} , а G = 2^{30} . Для измерений, относящихся к емкости устройств ввода-вывода (диски и сети), K = 10^3 , M = 10^6 , а G = 10^9 . Эти же значения префиксов применяются для оценки быстродействия и пропускной способности.

К счастью, для "огибающих" оценок, на которые обычно приходится полагаться, на практике любое предположение "работает" прекрасно. Например, относительная разница между $2^{20} = 1\ 048\ 576$ и 10^6 — мала: $(2^{20} - 10^6)/10^6 \approx 5\%$. Подобным же образом, для $2^{30} = 1\ 073\ 741\ 824$ и 10^9 : $(2^{30} - 10^9)/10^9 \approx 7\%$.

УПРАЖНЕНИЕ 6.2

Какова емкость диска с 2 тарелками, 10 000 цилиндрами, в среднем 400 секторов на дорожку и 512 байтов на сектор?

Функционирование диска

Диски считывают и записывают биты, сохраненные на магнитной поверхности, с помощью *головки считывания/записи* (универсальной головки), подключенной к концу консоли *исполнительного механизма*, как показано на рис. 6.9. Перемещая консоль вперед и назад по ее радиальной оси, дисковод может менять положение головки относительно любой дорожки на поверхности. Это механическое движение называется *поиском*. Как только головка доходит до нужной дорожки, тогда, по мере прохождения под ней каждого бита, головка либо распознает значение бита (читывает), либо меняет его значение (записывает). Диски с несколькими тарелками имеют отдельные головки для каждой поверхности. Головки выровнены вертикально и двигаются в унисон. В любой точке времени все головки расположены в одном и том же цилиндре.

В конце консоли головка считывания/записи перелетает (в буквальном смысле) на тончайшей воздушной подушке над поверхностью диска на высоте порядка 0.1 микрон со скоростью примерно 80 км/ч. Это сродни тому, если бы Ширз Тауэр (Sears Tower — небоскреб в Чикаго) можно было положить на бок и запустить вокруг Земли на высоте 2.5 см (1 дюйм) над поверхностью с прохождением каждой орбиты за 8 секунд! При таких допусках малейшая пылинка на поверхности покажется огромным булыжником. Если головка сталкивается с одним из таких "булыжников", то ее полет прекращается, и она разбивается о поверхность (происходит так называемая авария головки). По этой причине диски всегда герметично упаковываются.

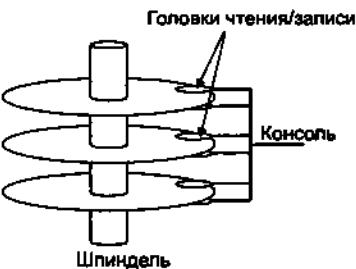


Рис. 6.9. Динамика диска

Диски считывают и записывают данные в блоки размером в сектор. Время доступа к сектору имеет три основных компонента: время поиска, задержка из-за вращения диска и время передачи.

- Время поиска. Для считывания содержимого какого-либо сектора консоль сначала располагает головку над дорожкой, на которой находится этот сектор. Время, необходимое для перемещения консоли, называется *временем поиска*. Время поиска T_{seek} зависит от предыдущего положения головки и от скорости, с которой консоль движется по поверхности. В современных дисководах среднее время поиска $T_{avg\ seek}$, измеренное, если брать среднее от нескольких тысяч поисков произвольных секторов, обычно составляет порядка 6—9 мс. Максимальное время одного поиска $T_{max\ seek}$ может достигать 20 мс.
- Задержка из-за вращения диска. Как только головка находится над дорожкой, дисковод ожидает прохождения под ней первого бита целевого сектора. Производительность на данном этапе зависит как от положения поверхности, когда головка достигает целевого сектора, так и от скорости вращения диска. В худшем случае головка пропускает целевой сектор и дожидается, пока диск сделает полный оборот. Таким образом, максимальная задержка из-за вращения диска в секундах вычисляется по следующей формуле:

$$T_{max\ rotation} = \frac{1}{RPM} \times \frac{60 \text{ сек.}}{1 \text{ мин.}}$$

Средняя задержка из-за вращения диска $T_{avg\ rotation}$ составляет половину от $T_{max\ rotation}$.

- Время передачи. Когда первый бит целевого сектора оказывается под головкой, дисковод начинает считывать или записывать содержимое сектора. Время передачи для одного сектора зависит от скорости вращения и от количества секторов на одной дорожке. Таким образом, можно приблизительно вычислить среднее время передачи для одного сектора в секундах по следующей формуле:

$$T_{avg\ transfer} = \frac{1}{RPM} \times \frac{1}{\text{среднее_кол-во_секторов}} \times \frac{60 \text{ сек.}}{1 \text{ мин.}}$$

Среднее время доступа к содержимому сектора диска можно вычислить как сумму среднего времени поиска, средней задержки из-за вращения диска и среднего времени передачи. Например, рассмотрим диск со следующими параметрами (табл. 6.2):

Таблица 6.2. Параметры диска

Параметр	Значение
Скорость вращения	7200 об/мин
$T_{avg\ seek}$	9 мс
Среднее кол-во секторов/дорожка	400

Для данного диска средняя задержка из-за вращения диска (в мс) составит:

$$\begin{aligned}T_{avg\ rotation} &= 1/2 \times T_{max\ rotation} = \\&= 1/2 \times (60c / 7.200 RPM) \times 1000 \text{ мс/с}\end{aligned}$$

Среднее время передачи составляет

$$\begin{aligned}T_{avg\ transfer} &= 60 / 7.200 RPM \times 1/400 \text{ секторов/дорожку} \times 1000 \text{ мс/с} = \\&= 0.02 \text{ мс}\end{aligned}$$

Если сложить все вместе, то общее вычисленное время доступа составляет

$$\begin{aligned}T_{access} &= T_{avg\ seek} + T_{avg\ rotation} + T_{avg\ transfer} \\&= 9 + 4 + 0.02 \\&= 13.02 \text{ мс}\end{aligned}$$

Данный пример иллюстрирует некоторые важные наблюдения:

- На время для доступа к 512 байтам дискового сектора влияют время поиска и задержка из-за вращения диска. Доступ к первому байту сектора занимает сравнительно много времени, однако доступ к остальным байтам времени практически не занимает.
- Поскольку время поиска и задержка из-за вращения диска — это приблизительно одно и то же, то удвоение времени поиска может рассматриваться как простое правило вычисления времени доступа к диску.
- Время доступа к двойному слову, сохраненному в SRAM, составляет приблизительно 4 нс, а для DRAM — 60 нс. Таким образом, время для считывания из памяти 512-байтового блока размером с сектор составляет приблизительно 256 нс для SRAM и 4000 нс для DRAM. Время доступа к диску (приблизительно 10 мс) — в 40 000 раз больше, чем SRAM, и примерно в 2500 раз больше, чем DRAM. Разница будет еще более разительной, если сравнить время доступа к одному слову.

УПРАЖНЕНИЕ 6.3

Вычислите среднее время доступа (в мс) к сектору на следующем диске:

Параметр	Значение
Скорость вращения	15 000 об/мин
$T_{avg\ seek}$	8 мс
Среднее кол-во секторов/дорожка	500

Логические блоки диска

Как уже отмечалось, геометрия современных дисков весьма сложна; они имеют много поверхностей с различными зонами записи. Для того чтобы скрыть эту сложность от операционной системы, геометрия современных дисков представлена проще,

в виде последовательности b логических блоков размером с сектор, пронумерованных как 0, 1, ..., $b - 1$. Маленькое программно-аппаратное устройство в диске, называемое контроллером диска, поддерживает преобразование данных между числами логических блоков и фактическими (физическими) секторами диска.

Когда операционной системе необходимо выполнить операцию ввода-вывода, например, считывание сектора диска в основную память, она отправляет команду на контроллер диска с запросом считать то или иное число логического блока. "Защищая" в контроллер программы выполняет быстрый поиск команды в таблице, преобразующей число логического блока в "троицу" (поверхность, дорожка, сектор), которая уникально идентифицирует соответствующий физический сектор. Аппаратное средство на контроллере заставляет указанную троицу переместить головки в соответствующий цилиндр, дожидается прохождения сектора под головкой, помещает распознанные головкой биты в небольшой буфер на контроллере и копирует их в основную память.

Емкость отформированного диска

Перед тем как диск можно будет использовать для хранения данных, его необходимо отформатировать с помощью контроллера диска. Эта процедура заключается в заполнении пропусков между секторами информацией, идентифицирующей секторы, идентификации любых цилиндров с дефектами поверхности и выведении их за пределы операций, "откладыванием" определенного количества цилиндров в каждой зоне "про запас", чтобы их можно было использовать, если один или более цилиндров в зоне выйдет из строя в течение срока эксплуатации диска. Форматная емкость, на которую ссылаются производители дисков, меньше максимальной емкости из-за наличия указанных запасных цилиндров.

Оценка дисков

Графические карты, мониторы, мыши, клавиатуры и диски подключены к центральному процессору и основной памяти посредством шины ввода-вывода, такой как, например, шина PCI (Peripheral Component Interconnect) от Intel. В отличие от системной шины и шины памяти, зависящих от CPU, такие шины ввода-вывода, как PCI, спроектированы так, что взаимодействие через них осуществляется без участия CPU. Например, компьютеры марок PC и Macintosh имеют встроенные шины PCI. На рис. 6.10 показана структура типичной шины ввода-вывода (смоделированная на PCI), соединяющая CPU, основную память и устройства ввода-вывода.

Шина ввода-вывода работает медленнее, нежели системная шина и шина памяти, но она может объединить много сторонних устройств ввода-вывода. Например, шина, схематически показанная на рис. 6.10, имеет три различных типа подключенных к ней устройств:

- контроллер универсальной последовательной шины (USB, Universal Serial Bus) — кабель для устройств, подключаемых к USB. USB имеет пропускную способность в 12 Мбит/с и предназначен для последовательных устройств низкого и среднего быстродействия (клавиатуры, мыши, модемы, цифровые камеры, джойстики, дисководы CD-ROM и принтеры);

- графическая карта (или адаптер) содержит аппаратную и программную логику, отвечающую за раскрашивание пикселов на экране монитора "от лица" CPU;
- контроллер диска содержит аппаратную и программную логику для считывания данных с диска и их записи "от лица" CPU.

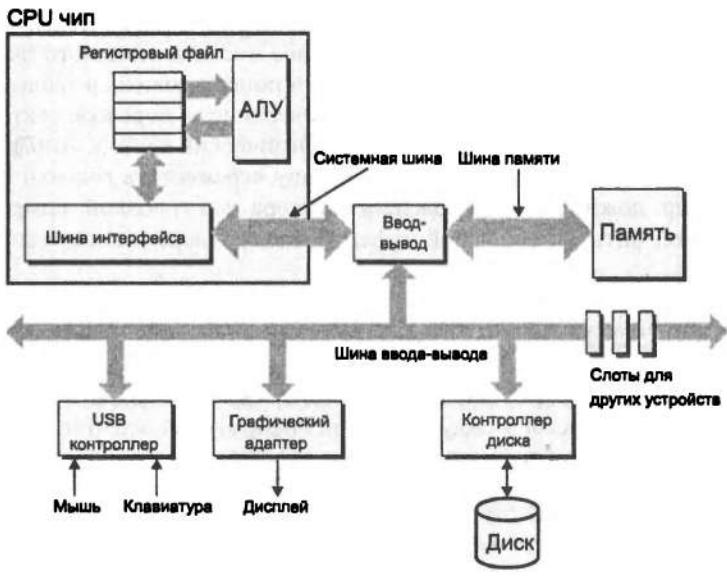


Рис. 6.10. Структура типичной шины, соединяющей CPU, основную память и устройства ввода-вывода

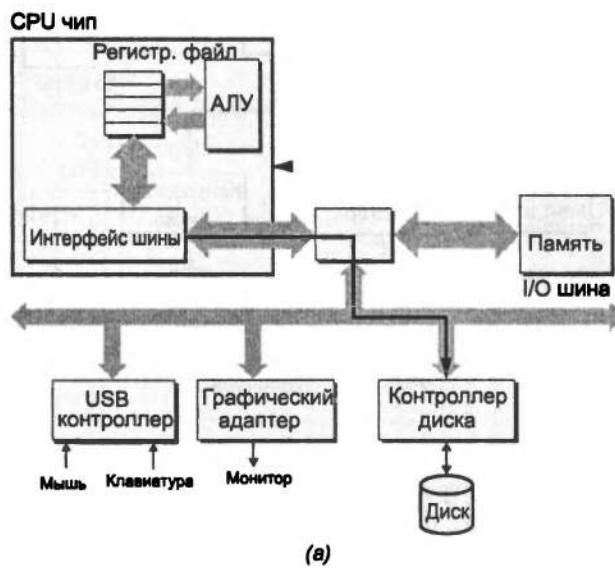
Дополнительные устройства, такие как *сетевые адAPTERы*, можно подсоединить кшине ввода-вывода путем простого подключения адаптера к свободным *расширяющим слотам* на материнской плате, обеспечивающим прямое электрическое подключение кшине.

Подробное описание принципов работы и программирования устройств ввода-вывода не входит в задачи настоящей книги, однако авторы намерены изложить лишь общую идею. Например, на рис. 6.10 показаны шаги, которые выполняет CPU при считывании данных с диска.

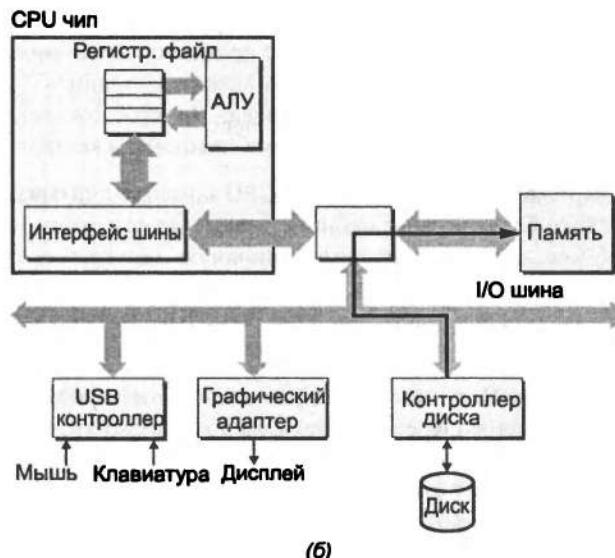
CPU направляет команды на устройства ввода-вывода, используя методику, называемую размещенным в памяти вводом-выводом (рис. 6.11, а). В системе с таким вводом-выводом блок адресов в адресном пространстве зарезервирован для взаимодействия с устройствами ввода-вывода. Каждый из этих адресов называется *портом ввода-вывода*. Каждое устройство связывается (или отображается) на одном или нескольких портах, когда подключается кшине.

В качестве примера предположим, что контроллер диска отображается на порт 0xa0. Затем CPU может инициировать считывания диска через выполнение трех команд сохранения на адрес 0xa: первая из этих команд отправляет командное слово, информирующее диск об инициировании считывания, вместе с другими параметрами, на-

пример, информацией о том, приостанавливать ли работу CPU по окончании считывания. (Прерывания рассматриваются в разд. 8.1.) Вторая команда указывает номер логического блока, который должен быть считан. Третья команда указывает адрес основной памяти, в котором должно сохраняться содержимое сектора диска. На рис. 6.11, а CPU инициирует процедуру считывания диска путем записи команды,

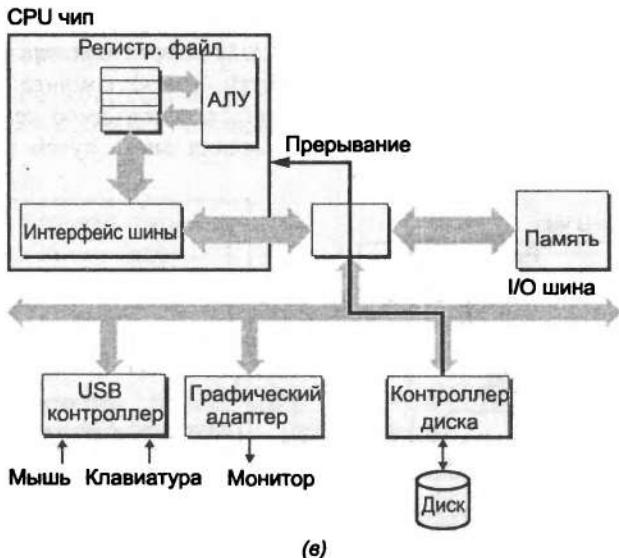


(a)



(б)

Рис. 6.11. Считывание сектора диска (см. продолжение)



(e)

Рис. 6.11. Окончание

номера логического блока и адреса назначения памяти в отображенный в памяти адрес, связанный с диском. Контроллер диска считывает сектор и выполняет перенос DMA в основную память (рис. 6.11, б). По завершении переноса DMA контроллер диска уведомляет об этом CPU через прерывание (рис. 6.11, в).

После выполнения запроса обычно CPU выполняет другие задачи, в то время как диск осуществляет считывание. Вспомните, что процессор с тактовой частотой в 1 Гц с циклом синхронизации в 1 нс потенциально может выполнить 16 млн команд за 16 мс, которые требуются для считывания диска. Простое ожидание переноса без выполнения каких-либо задач было бы слишком большой роскошью.

После того как контроллер диска получает от CPU команду считывания, он преобразует число логического блока в адрес сектора, считывает содержимое этого сектора и переносит содержимое непосредственно в основную память без "вмешательства" центрального процессора (рис. 6.11, б). Этот процесс, когда устройство самостоятельно, без участия CPU, выполняет транзакцию считывания или записи, называется **прямым доступом к памяти (DMA)**. Перенос данных называется **переносом DMA**.

По завершении переноса DMA, когда содержимое сектора диска благополучно сохранено в основной памяти, контроллер диска уведомляет об этом CPU отправкой на него сигнала прерывания (рис. 6.11, в). Основной идеей здесь является то, что прерывание сигнализирует внешнему контакту чипа CPU. При этом CPU прекращает обработку текущей операции и переходит к выполнению рутинных процедур операционной системы. Эта процедура записывает тот факт, что ввод-вывод завершен, и возвращает управление в точку, на которой работа CPU была прервана.

Анатомия коммерческого диска

Производители дисков публикуют на своих сайтах огромные объемы технической информации. Например, если зайти на сайт за информацией о диске IBM Ultrastar 36LZX, то можно получить сведения о геометрии и производительности, показанные в табл. 6.3—6.4.

Таблица 6.3. Сведения о геометрии

Атрибут геометрии	Значение
Тарелки	6
Поверхности (головки)	12
Размер сектора	512 байтов
Зоны	11
Цилиндры	15 110
Плотность записи (максимальная)	352 000 бит/дюйм
Плотность дорожки	20 000 дорожек/дюйм
Плотность размещения (максимальная)	7040 Мбит/кв. дюйм
Форматная емкость	36 Гбайт

Таблица 6.4. Сведения о производительности

Атрибут производительности	Значение
Скорость вращения	10 000 об/мин
Средняя задержка из-за вращения диска	2.99 мс
Среднее время поиска	4.9 мс
Незатухающая скорость передачи	21—36 Мбайт/с

Производители дисков часто пренебрегают публикацией подробной технической информации о геометрии отдельных зон записи. Однако исследователи записывающих устройств разработали полезный инструмент под названием DIXtrac, автоматически выявляющий ценность информации нижнего уровня о геометрии и производительности дисков SCSI [68]. Например, инструмент DIXtrac способен выявить подробную геометрию зоны диска-образца IBM, приведенную в табл. 6.5. Каждая строка в таблице характеризует одну из 11 зон на поверхности диска в том, что касается количества секторов в зоне, диапазона логических блоков, отображенного на секторах в зоне, а также диапазона и числа цилиндров в зоне.

Отображение по зонам подтверждает некоторые интересные факты о диске IBM. Во-первых, на внешних зонах можно разместить большее количество дорожек (имеющих большую длину окружности), чем на внутренних зонах. Во-вторых,

каждая зона имеет больше секторов, чем логических блоков (читатели могут в этом убедиться). Неиспользованные сектора формируют накопитель "запасных" цилиндров. Если записываемый в секторе материал оказывается запорченным, контроллер диска автоматически перераспределяет логические блоки на этом цилиндре на доступный запасной цилиндр. Таким образом, становится ясно, что понятие логического блока не только обеспечивает операционной системе упрощенный интерфейс, но и уровень выполнения нескольких операций физического обращения к памяти для реализации одного логического обращения (*indirection*), что делает диск более устойчивым. При изучении виртуальной памяти в главе 10 эта генеральная идея окажется очень важной.

Таблица 6.5. Характеристики зон

Номер зоны	Секторов на одной дорожке	Начальный логический блок	Конечный логический блок	Начальный цилиндр	Конечный цилиндр	Цилиндров на зону
(внешний) 0	504	0	2 292 096	1	380	380
1	476	2 292 097	11 949 751	381	2078	1698
2	462	11 949 752	19 416 566	2079	3340	1352
3	420	19 416 567	36 409 689	3431	6815	3385
4	406	36 409 690	39 844 151	6816	7523	708
5	392	39 844 152	46 287 903	7524	8898	1375
6	378	46 287 904	52 201 829	8899	10 207	1309
7	364	52 201 830	56 691 915	10 208	11 239	1032
8	352	56 691 916	60 087 818	11 240	12 046	807
9	336	60 087 819	67 001 919	12 047	13 768	1722
(внутр.) 10	308	67 001 920	71 687 339	13 769	15 042	1274

6.1.3. Направления развития технологий записывающих устройств

Существует несколько важных положений для того, чтобы отойти от приведенных здесь концепций рассмотрения технологий записывающих устройств.

- Различные технологии сохранения имеют разную цену и разные альтернативы производительности. SRAM работает немного быстрее, нежели DRAM, а DRAM — более быстродействующая, нежели диск. С другой стороны, любое быстродействующее запоминающее устройство всегда дороже работающего медленнее. Затраты SRAM на байт — выше, чем DRAM, а DRAM — намного дороже диска.

- Цена и свойства производительности разных технологий сохранения информации меняются с разными скоростями. В табл. 6.6—6.9 в общем виде показаны цена и свойства производительности технологий сохранения информации, начиная с 1980 г., когда появились первые персональные компьютеры. Показатели выбраны по торговым каталогам. Несмотря на то, что собирались они в процессе неофициального исследования, они демонстрируют определенные интересные тенденции.

Таблица 6.6. Тенденции совершенствования SRAM

Метрическая	1980	1985	1990	1995	2000	2000:1980
\$/Мбайт	19 200	2900	320	256	100	190
Доступ (нс)	300	150	35	15	3	100

Таблица 6.7. Тенденции совершенствования DRAM

Метрическая	1980	1985	1990	1995	2000	2000:1980
\$/Мбайт	8000	880	100	30	1	8000
Доступ (нс)	375	200	100	70	60	6
Обычный размер (Мбайт)	0.064	0.256	4	16	64	1000

Таблица 6.8. Тенденции совершенствования диска

Метрическая	1980	1985	1990	1995	2000	2000:1980
\$/Мбайт	500	100	8	0.30	0.01	50 000
Время поиска (мс)	87	75	28	10	8	11
Обычный размер (Мбайт)	1	10	160	1000	20 000	20 000

Таблица 6.9. Тенденции совершенствования CPU

Метрическая	1980	1985	1990	1995	2000	2000:1980
Intel CPU	8080	80 286	80 386	Pentium	P-III	—
Тактовая частота CPU (МГц)	1	6	20	150	600	600
Время цикла CPU (нс)	1000	166	50	6	1/6	600

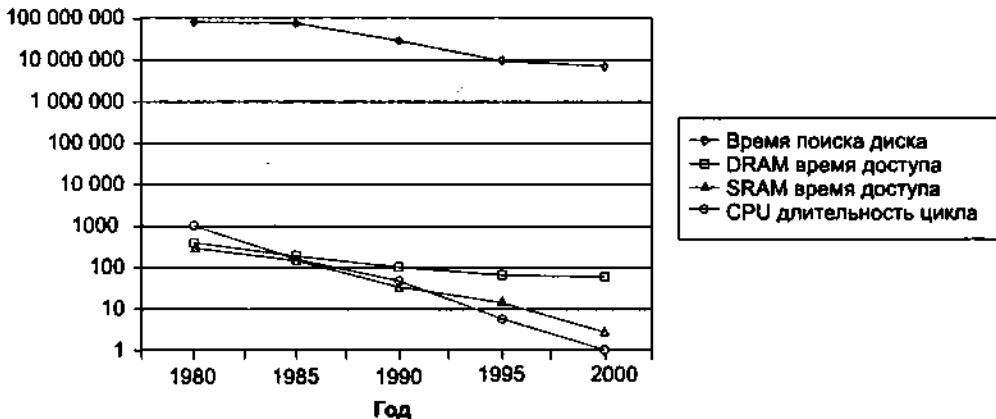


Рис. 6.12. Увеличивающийся промежуток между скоростями действия DRAM, диска и ЦП

- Начиная с 1980 г., стоимость и производительность технологии SRAM совершенствовались с равномерной скоростью. Время доступа сократилось в 100 раз, а затраты на мегабайты — в 200 (табл. 6.6). Однако тенденции для DRAM и дисков разительны. В то время, как затраты на мегабайты для DRAM сократились в 8000 раз (почти на четыре порядка), время доступа для DRAM сократилось только в 6 раз (табл. 6.7). Технология разработки дисков развивалась еще более драматично, чем DRAM. В то время, как затраты на мегабайты дисковой памяти с 1980 г. упали на коэффициент 50 000, время доступа сокращалось гораздо медленнее (коэффициент порядка 10) (табл. 6.8). Такие удивительные долгосрочные тенденции выявляют базовую истину технологий памяти и диска: проще повысить плотность (и следовательно, сократить затраты), нежели сократить время доступа.
- Время доступа к DRAM и диску отстает от времени цикла CPU. Как показано в табл. 6.9, время цикла CPU за период с 1980 до 2000 г. повысилось в 600 раз. Производительность SRAM отстает, но по-прежнему высока, хотя расхождение между DRAM, производительностью диска и производительностью CPU фактически повышается. На рис. 6.12 реальные тенденции показаны достаточно четко с изображением времени цикла и доступа на полулогарифмической шкале.

В разд. 6.4 показано, что современные компьютеры в полной мере используют кэш-память на базе SRAM для образования моста между процессором и памятью. Данный подход срабатывает из-за фундаментального свойства программных приложений, называемого локальностью, которое рассматривается далее.

6.2. Локальность

Хорошо написанные компьютерные программы, как правило, демонстрируют высокую локальность. То есть, в них делаются обращения к элементам данных, расположенных рядом с другими элементами данных, к которым недавно делались обращения, или эти данные обращались к другим данным. Данная тенденция, называемая

принципом локальности, является устойчивой концепцией, оказывающей значительное воздействие на проектирование и производительность систем аппаратного и программного обеспечения.

Локальность обычно описывается как принимающая две отчетливые формы: временную локальность и так называемую пространственную локальность. В программе с хорошей временной локальностью на ячейку памяти, на которую один раз делается ссылка, наверняка в ближайшем будущем будут другие ссылки. В программе с хорошей пространственной локальностью, если на ячейку памяти один раз делается ссылка, тогда программа, скорее всего, в будущем сделает ссылку на расположенную рядом ячейку памяти.

Программисты должны хорошо понимать принцип локальности, потому что, как правило, программы с высокой степенью локальностью выполняются быстрее, чем программы с низкой степенью локальности. Все уровни современных компьютерных систем, от аппаратных средств до операционных систем и отдельных приложений, спроектированы с применением локальности. На уровне аппаратных средств принцип локальности позволяет проектировщикам ускорять доступ к основной памяти через ввод небольших быстродействующих устройств памяти, называемых кэшами, содержащих последние по времени обращения к блокам команд и элементам данных. На уровне операционных систем принцип локальности позволяет системе использовать основную память в качестве кэша самых последних по времени обращения блоков виртуального адресного пространства. Аналогичным же образом операционная система использует основную память для кэширования последних блоков диска системы дискового файла, к которым осуществлялось обращение. Принцип локальности также играет ключевую роль при проектировании программных приложений. Например, Web-браузеры используют временную локальность кэшированием на локальный диск последних документов, к которым осуществлялось обращение. В высокобъемных Web-серверах последние запрошенные документы сохраняются на предварительных дисковых кэшах, удовлетворяющих запросам этих документов без вмешательства сервера.

6.2.1. Локальность обращений к данным программы

Рассмотрим простую функцию, показанную в листинге 6.1, суммирующую элементы вектора. Обладает ли эта функция высокой степенью локальности? Для ответа на этот вопрос рассмотрим комбинацию обращения к каждой переменной. В данном примере к переменной `sum` осуществляется однократное обращение в каждом цикле итерации, и поэтому в том, что касается функции `sum`, налицо хорошая временная локальность. С другой стороны, поскольку `sum` — функция скалярная, в отношении ее не имеет места пространственная локальность.

Как видно по табл. 6.10, элементы вектора `v` считаются последовательно, один за другим, в том порядке, в котором они сохранены в памяти (для удобства предполагается, что массив начинается в адресе 0). Таким образом, с учетом переменной `v` функция имеет хорошую пространственную локальность, но низкую временную локальность, поскольку к каждому элементу вектора доступ осуществляется только один

раз. Из-за того, что данная функция обладает либо хорошей временной, либо пространственной локальностью в том, что касается каждой переменной в теле цикла, можно сделать вывод о том, что функция sumvec обладает хорошей локальностью.

Листинг 6.1. Функция с высокой локальностью

```

1     int sumvec (int v[N])
2     {
3         int i, sum = 0;
4
5         for (i = 0; i < N; i++)
6             sum += v[i];
7         return sum;
8     }

```

Таблица 6.10. Комбинация обращения для вектора

Адрес	0	4	8	12	16	20	24	28
Содержимое	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7
Порядок доступа	1	2	3	4	5	6	7	8

Говорят, что такая функция, как sumvec, последовательно вызывающая каждый элемент вектора, имеет комбинацию обращений с шагом по индексу 1 (с учетом размера элемента). Вызов каждого элемента k непрерывного вектора называется комбинацией обращений шага по индексу k . Комбинации обращений шага по индексу 1 — широко распространенный и важный источник пространственной локальности в программах. Вообще говоря, с увеличением шага по индексу пространственная локальность уменьшается.

Шаг по индексу также является важным моментом для программ, обращающихся к многомерным таблицам. Рассмотрим функцию sumarrayrows, показанную в листинге 6.2, суммирующую элементы двумерного массива. Двойной вложенный цикл считывает элементы массива развертыванием по строкам. То есть, внутренний цикл считывает элементы массива развертыванием по строкам. То есть, внутренний цикл считывает элементы первой строки, затем — второй и т. д. Функция sumarrayrows обладает хорошей пространственной локальностью, потому что она осуществляет обращение к массиву в том же развертывании по строкам, в каком массив сохранен (табл. 6.11). Результатом является комбинация обращений с шагом по индексу 1 с великолепной пространственной локальностью.

Внешне тривиальные изменения в программе могут оказывать серьезное влияние на ее локальность. Например, функция sumarraycols (листинг 6.3) вычисляет тот же результат, что и функция sumarrayrows, показанная в листинге 6.1. Единственным различием является то, что циклы i и j оказались взаимно замещенными. Какое воздействие окажет взаимное замещение циклов на локальность программы?

Функция `sumarraycols` страдает от низкой пространственной локальности из-за того, что она просматривает массив по столбцам, а не по строкам. Поскольку массивы С располагаются в памяти по строкам, результатом является комбинация обращений с шагом по индексу, как показано в табл. 6.12.

Листинг 6.2. Функция с высокой пространственной локальностью

```

1     int sumarrayrows (int a[M] [N])
2     {
3         int i, j, sum = 0;
4
5         for (i = 0; i < M; i++)
6             for (j = 0; j < N; j++)
7                 sum += a[i] [j];
8         return sum;
9     }

```

Таблица 6.11. Комбинация обращения для массива

Адрес	0	4	8	12	16	20
Содержимое	a_{00}	a_{01}	a_{02}	a_{10}	a_{11}	a_{12}
Порядок доступа	1	2	3	4	5	6

Листинг 6.3. Функция с низкой пространственной локальностью

```

1     int sumarraycols (int a[M] [N])
2     {
3         int i, j, sum = 0;
4
5         for (j = 0; i < N; j++)
6             for (i = 0; i < M; i++)
7                 sum += a[i] [j];
8         return sum;
9     }

```

Таблица 6.12. Пример низкой локальности

Адрес	0	4	8	12	16	20
Содержимое	a_{00}	a_{01}	a_{02}	a_{10}	a_{11}	a_{12}
Порядок доступа	1	3	5	2	4	6

6.2.2. Локальность выборки команд

Поскольку команды программы сохранены в памяти и должны выбираться (считываться) центральным процессором, локальность программы можно оценить с учетом выборки команд. Например, в листинге 6.1 команды в теле цикла `for` выполняются в последовательном порядке памяти, и следовательно, данный цикл обладает хорошей пространственной локальностью. По причине того, что тело цикла выполняется многократно, оно также обладает хорошей временной локальностью.

Важным свойством кода, отличающим его от данных программы, является то, что его нельзя модифицировать во время выполнения. Во время выполнения программы CPU только считывает команды из памяти. CPU никогда не перезаписывает и не модифицирует эти команды.

6.2.3. Резюме локальности

В этом разделе представлено фундаментальное понятие локальности и обозначены некоторые простые правила качественной оценки локальности в программах:

- Программы, которые многократно обращаются к одной и той же переменной, обладают высокой степенью локальности.
- Для программ с комбинациями обращений с шагом по индексу k , чем меньше шаг, тем лучше пространственная локальность. Программы с комбинациями обращений с шагом 1 обладают хорошей пространственной локальностью. Программы, "прыгающие" по памяти большими шагами, обладают более низкой пространственной локальностью.
- Циклы имеют хорошую временную и пространственную локальность с учетом выборки команд. Чем меньше тело цикла и больше число итераций цикла, тем лучше локальность.

Далее в главе, после того как будут подробно рассмотрены устройства кэш-памяти и принципы их работы, авторы расскажут об определении идеи локальности с точки зрения результативных обращений в кэш и промахов. Читателям также станет понятно, почему программы с высокой локальностью, как правило, выполняются быстрее, нежели программы с низкой локальностью. Несмотря на это, для программиста очень важно овладеть полезным и важным навыком рассмотрения исходного кода и высокуюровневого понимания принципа локальности.

УПРАЖНЕНИЕ 6.4

Перемените местами циклы в следующей функции так, чтобы она просматривала трехмерный массив a с комбинацией обращений с шагом по индексу 1:

```

1     int sumarray3d (int a [N] [N] [N])
2     {
3         int i, j, k, sum = 0;
4
5         for (i = 0; i < N; i++)    {
6             for (j = 0; j < N; j++)  {
7                 for (k = 0; k < N; k++) {
8                     sum += a[i][j][k];
9                 }
10            }
11        }
12        return sum;
13    }
```

```

7                     for (k = 0; k < N; k++) {
8                         sum += a [k] [i] [j];
9                     }
10                }
11            }
12        return sum;
13    }

```

УПРАЖНЕНИЕ 6.5

Три функции, показанные в листингах 6.5—6.7, выполняют ту же операцию, что и структура, описанная в листинге 6.4, с различными степенями пространственной локальности. Упорядочите эти функции с учетом пространственной локальности каждой. Обоснуйте выбранный принцип упорядочивания.

Листинг 6.4. Массив структур

```

1 #define N 1000
2
3 typedef struct {
4     int vel [3];
5     int acc [3];
6 } point;
7
8 point p [N]

```

Листинг 6.5. Функция clear1

```

1 void clear1 (point *p, int n)
2 {
3     int i, j;
4
5     for (i = 0; i < n; i++) {
6         for (j = 0; j < 3; j++)
7             p[i].vel[j] = 0;
8         for (j = 0; j < 3; j++)
9             p[i].acc[j] = 0;
10    }
11 }

```

Листинг 6.6. Функция clear2

```

1 void clear2 (point *p, int n)
2 {
3     int i, j;
4

```

```

5         for (i = 0; i < n; i++)    (
6             for (j = 0; j < 3; j++)
7                 p[i].vel[j] = 0;
8                 p[i].acc[j] = 0;
9             )
10        )
11    )

```

Листинг 6.7. Функция clear3

```

1 void clear3 (point *p, int n)
2 {
3     int i, j;
4
5     for (j = 0; j < n; j++)    {
6         for (i = 0; i < 3; i++)
7             p[i].vel[j] = 0;
8         for (i = 0; i < n; i++)
9             p[i].acc[j] = 0;
10    }
11 }

```

6.3. Иерархия памяти

В разд. 6.1 и 6.2 описываются некоторые фундаментальные и устойчивые свойства технологии сохранения информации и программного обеспечения:

□ Различные технологии сохранения информации обладают широким диапазоном времени доступа. Более быстродействующие (на байт) технологии стоят дороже, чем менее быстродействующие, а также обладают меньшей емкостью. Расхождение между CPU и основной памятью расширяется.

□ Грамотно написанные программы демонстрируют большую степень локальности.

При одном из удачных совпадений компьютерных вычислений эти фундаментальные свойства аппаратных и программных средств прекрасно дополняют друг друга. Их комплементарная природа предполагает подход к организации систем памяти, называемый *иерархией памяти*, используемый во всех современных компьютерных системах. На рис. 6.13 показана типичная иерархия памяти.

Вообще говоря, записывающие устройства работают более медленно, они дешевле и больше по размерам при переходе от более высокого уровня к более низкому. На самом высоком уровне (L0) находится небольшое число быстродействующих регистров CPU, доступ к которому CPU может получить за один цикл синхронизации. Далее следуют небольшие и средних размеров устройства кэш-памяти, основанные на SRAM, доступ к которым осуществляется за несколько циклов синхронизации CPU. За ними следует основная память большого размера, основанная на DRAM, доступ к

которой осуществляется за десятки или сотни циклов синхронизации. Затем следуют медленнодействующие локальные диски больших размеров. Наконец, некоторые системы включают в себя даже дополнительный уровень дисков на удаленных серверах, доступ к которым может осуществляться по сети. Например, системы распределенных файлов AFS (Andrew File System) или NFS (Network File System) позволяют программе осуществлять доступ к файлам, хранящимся на удаленных сетевых серверах. Аналогичным же образом глобальная сеть позволяет программам осуществлять доступ к удаленным файлам, сохраненным на Web-серверах в любой точке мира.

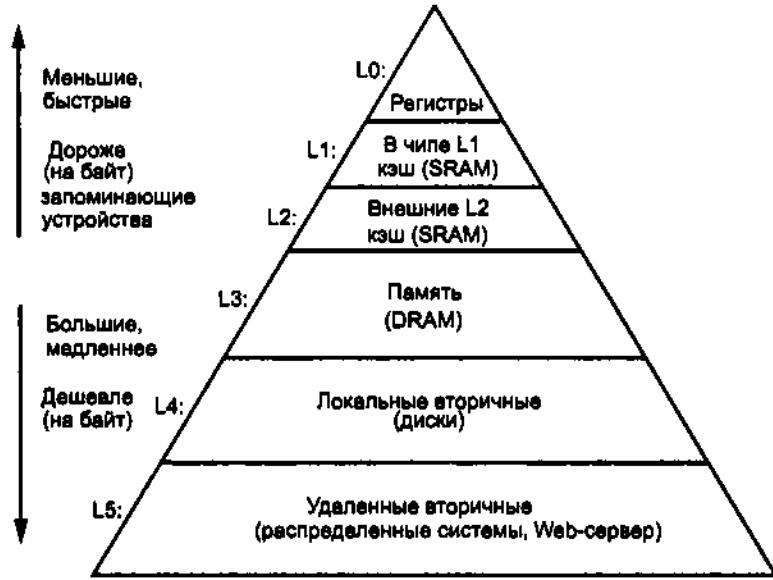


Рис. 6.13. Иерархия памяти

Другие иерархии памяти

Здесь приведен всего лишь один пример иерархии памяти, однако возможны и другие комбинации, которые распространены довольно широко. Например, многие вычислительные системы создают резервные копии локальных дисков на магнитной ленте, предназначенней для архивирования. На других системах специальные роботы выполняют эту задачу автоматически. В любом случае коллекция лент представляет собой уровень в иерархии памяти, расположенный ниже уровня диска, к которому применяются те же общие принципы. Использование лент дешевле дисков (на байт), и это позволяет системам архивировать множественные моментальные копии на локальных дисках. Уступкой (или компромиссом) в данном случае является то, что доступ к лентам занимает больше времени, нежели к дискам.

6.3.1. Кэширование в иерархии памяти

Кэш представляет собой небольшое по размеру быстродействующее записывающее устройство в роли этапной области для объектов данных, сохраняемых на более

крупных и менее быстродействующих записывающих устройствах. Процесс использования кэш-памяти называется **кэшированием**.

Центральной концепцией иерархии памяти является то, что для каждого k более быстродействующее и меньшее по размеру записывающее устройство на уровне k служит кэшем для менее быстродействующего и более габаритного записывающего устройства на уровне $k+1$. Другими словами, каждый уровень в иерархии кэширует объекты данных из следующего, более низкого уровня. Например, локальный диск служит кэшем для файлов (например, для Web-страниц), полученных с удаленных дисков по сети; основная память служит кэшем для данных на локальных дисках и т. д. до наименьшего кэша из всех существующих — набора регистров центрального процессора.

На рис. 6.14 схематически показана общая идея кэширования в иерархии памяти. Записывающее устройство на уровне $k+1$ разделено на смежные куски объектов данных, называемые блоками. Каждый блок обладает уникальным адресом или именем, отличающим его от других блоков. Блоки могут быть либо фиксированного размера (обычный случай), либо изменяемого (например, HTML-файлы, сохраняемые на Web-серверах). Например, показанное на рис. 6.14 записывающее устройство уровня $k+1$ разделено на 16 блоков фиксированного размера с номерами от 0 до 15.



Рис. 6.14. Базовый принцип кэширования в иерархии памяти

Подобным же образом записывающее устройство на уровне k разделено на меньшее количество блоков, имеющих тот же размер, что и блоки на уровне $k+1$. В любой момент времени кэш-память на уровне k содержит копии подмножества блоков из уровня $k+1$. Например, на рис. 6.14 кэш на уровне k обладает пространством для четырех блоков и в показанный момент содержит копии блоков 4, 9, 14 и 3.

Данные всегда копируются попарно между уровнями k и $k+1$ в устройствах переноса, имеющих размеры блока. Важно понять, что в то время, как размер блока фиксирован между той или иной парой смежных уровней иерархии, другие пары уровней могут иметь другие размеры блоков. Например, на рис. 6.13 переносы между

L1 и L0 обычно используют блоки размером в одно слово. А переносы между L4 и L3 используют блоки с сотнями или тысячами байтов. Вообще говоря, устройства более низких уровней в иерархии (отстоящие дальше от CPU) имеют большие значения времени доступа, следовательно, стремятся использовать большие размеры блоков для так называемой "амортизации" затрат на доступ.

Результативные обращения

Когда программе необходим доступ к конкретному объекту данных d из уровня $k+1$, она сначала ищет d в одном из блоков, сохраненных в настоящий момент на уровне k . Если действительно оказывается так, что d кэшировано на уровне k , это называется результативным обращением в кэш. Программа считывает d непосредственно с уровня k , который — по самой природе иерархии памяти — срабатывает быстрее, нежели при считывании d с уровня $k+1$. Например, программа с хорошей временной локальностью может считывать объект данных из блока 14 (рис. 6.14), результатом чего становится результативное обращение в кэш из уровня k .

Промахи кэша

С другой стороны, если объект данных d не кэширован на уровне k , тогда пользователь сталкивается с так называемым *промахом кэша*. При возникновении промаха кэша на уровне k выбирается блок, содержащий d , из кэша на уровне $k+1$, возможно, с перезаписью существующего блока, если кэш уровня k полон.

Процесс перезаписи существующего блока называется заменой, или вытеснением блока. Вытесненный блок иногда называется блоком-жертвой. Решение о том, какой блок должен быть вытеснен, регулируется стратегией замены кэша. Например, кэш с произвольной стратегией замены выберет произвольный блок-жертву. Кэш, в котором применяется стратегия наиболее давнего использования (LRU), вытеснит блок, последнее обращение к которому было осуществлено на самом дальнем этапе в прошлом.

После того как кэш на уровне k выбрал блок с уровня $k+1$, программа может считать d с уровня k , как и прежде. Например, на рис. 6.14 показано, что результатом считывания объекта данных из блока 12 кэша на уровне k станет промах кэша, потому что блок 12 в данный момент времени не сохранен в кэше уровня k . Как только он скопируется из уровня $k+1$ на уровень k , блок 12 останется там, ожидая последующих доступов.

Виды промахов кэша

Иногда имеет смысл разделять промахи кэша по видам. Если кэш на уровне k пуст, тогда никакого доступа ни к каким объектам данных не состоится. Пустой кэш иногда называется "холодным", и промахи такого рода называются принудительными, или также "холодными". Холодные промахи важны, потому что они часто являются переходными событиями, которые могут не иметь места в устойчивом состоянии после того, как кэш "подогревается" многократными обращениями к памяти.

Всякий раз при возникновении промаха кэш на уровне k должен реализовать некоторую стратегию замены, определяющую место для расположения блока, полученного

с уровня $k+1$. Наиболее гибкой стратегией замены является допущение сохранения любого блока с уровня $k+1$ на уровне k . Для более высоких уровней иерархии (ближе к CPU), реализованных в аппаратных средствах, где быстродействие является основным приоритетом, эта стратегия, как правило, требует больших затрат на реализацию, потому что произвольно расположенные блоки требуют больших затрат.

Таким образом, аппаратные кэши обычно реализуют более ограниченную стратегию замены, ограничивающую тот или иной блок на уровне $k+1$ небольшим подмножеством (иногда одним элементом) блоков на уровне k . Например, по рис. 6.14 можно принять решение о том, что блок i на уровне $k+1$ должен быть помещен в блок i по модулю 4 на уровне k . Например, блоки 0, 4, 8 и 12 на уровне $k+1$ отобразятся в блок 0 на уровне k , блоки 1, 5, 9 и 13 — в блок 1 и т. д. Обратите внимание на то, что кэш-пример, показанный на рис. 6.14, использует эту стратегию.

Ограничительные стратегии замены такого типа приводят к типу промаха, называемого *конфликтным промахом*, когда кэш обладает довольно большим размером для удержания запрашиваемых объектов данных, но из-за отображения в тот же блок кэша все равно имеет место промах. Например, как показано на рис. 6.14, если программа запрашивает блок 0, затем блок 8, затем блок 0, затем опять блок 8, то каждое из обращений к этим двум блокам не попадет в кэш на уровне k , даже если этот кэш может содержать 4 блока.

Программы часто выполняются как последовательность фаз (циклов), когда каждая фаза получает доступ к более-менее постоянному множеству блоков кэша. Например, вложенный цикл может получать многократный доступ к элементам одного и того же массива. Это множество блоков называется *рабочим множеством* фазы. Когда размер рабочего множества превышает размер кэша, тогда последний испытывает то, что называется *промахами емкости*. Другими словами, кэш слишком мал для управления данным конкретным рабочим множеством.

Управление кэшем

Как уже отмечалось, квинтэссенцией иерархии памяти является то, что записывающее устройство на каждом уровне — кэш для следующего более низкого по отношению к нему уровня. На каждом уровне определенная форма логики должна управлять кэшем. Этим подразумевается, что нечто должно делить записывающее устройство кэша на блоки, переносить блоки между разными уровнями, определять, где результативные обращения в кэш, а где — промахи, после чего обрабатывать всю эту информацию. Логикой, управляющей кэшем, могут быть аппаратные, программные средства или их комбинация.

Например, компилятор управляет регистровым файлом — самым высоким уровнем иерархии кэш-памяти. Он определяет, когда подавать команду загрузки, наличие промахов, а также определяет регистр, в котором должны сохраняться данные. Кэши на уровнях L1 и L2 целиком управляются логикой встроенных в них аппаратных средств. В системе с виртуальной памятью основная память DRAM служит кэш-памятью для блоков данных, сохраненных на диске, и управляется комбинацией программных средств операционной системы и аппаратными средствами преобразования адресов на CPU. Для машины с системой распределенных файлов (например,

AFS) локальный диск служит кэш-памятью, управляемой процессом AFS-клиента, запущенным на локальной машине. В большинстве случаев кэш работает автоматически и не требует никаких особых или явных действий со стороны программы.

6.3.2. Резюме концепции иерархии памяти

В качестве резюме можно еще раз отметить, что иерархии памяти базируются на работе кэша, потому что менее быстродействующие записывающие устройства дешевле, чем более быстродействующие, а также потому, что программы стремятся демонстрировать локальность.

- Использование временной локальности. Из-за временной локальности одни и те же объекты данных могут использоваться многократно. Как только объект данных скопирован в кэш с первым промахом, тогда можно ожидать некоторого количества последовательных результативных обращений к этому объекту. Поскольку кэш-память работает быстрее, нежели записывающее устройство на следующем более низком уровне, эти последовательные результативные обращения могут обслуживаться намного быстрее, нежели первоначальный промах.
- Использование пространственной локальности. В блоках, как правило, содержатся множественные объекты данных. Из-за пространственной локальности можно ожидать, что затраты на копирование блока после промаха будут амортизированы последующими обращениями к другим объектам в рамках данного блока.

В современных системах кэш используется повсеместно: в чипах центральных процессоров, операционных системах, системах распределенных файлов, а также в глобальной сети. Они создаются различными комбинациями аппаратных и программных средств, и ими же управляются. Обратите внимание на то, что в табл. 6.13 приведено определенное количество еще не описанных терминов и акронимов. Они включены здесь для демонстрации типичных устройств кэш-памяти.

Таблица 6.13. Абсолютное кэширование в современных компьютерных системах

Вид	Предмет кэширования	Где осуществляется кэширование	Задержка (циклов)	Управляется
Регистры CPU	4-байтовое слово	Внутренние регистры CPU	0	Компилятор
TLB	Преобразования адресов	Внутренний TLB	0	Аппаратный MMU
Кэш уровня L1	32-байтовый блок	Внутренний кэш уровня L1	1	Аппаратные средства
Кэш уровня L2	32-байтовый блок	Внешний кэш уровня L2	10	Аппаратные средства
Виртуальная память	Страница 4-Кбайт	Основная память	100	Аппаратные средства + операционная система

Таблица 6.13 (окончание)

Вид	Предмет кэширования	Где осуществляется кэширование	Задержка (циклов)	Управляется
Буферная кэш-память	Части файлов	Основная память	100	Операционная система
Сетевая буферная кэш-память	Части файлов	Локальный диск	10 000 000	AFS/NFS-клиент
Кэш-память браузера	Web-страницы	Локальный диск	10 000 000	Web-браузер
Web-кэш	Web-страницы	Диски удаленного сервера	1 000 000 000	Web-сервер прокси

6.4. Виды кэш-памяти

Иерархии памяти первых компьютерных систем состояли только из трех уровней: регистров CPU, основной памяти DRAM и записывающих устройств на дисках. Однако из-за растущего расхождения между CPU и основной памятью проектировщики системы были вынуждены подключить небольшое записывающее устройство SRAM — кэш L1 (кэш первого уровня) между регистровым файлом CPU и основной памятью. В современных системах кэш L1 расположен на чипе центрального процессора (он называется внутренним кэшем), как показано на рис. 6.15. Доступ к кэшу L1 — почти такой же оперативный, как и доступ к регистрам 1 или 2 цикла синхронизации.



Рис. 6.15. Типичная структура шины кэш L1 и L2

На увеличение расхождения между CPU и основной памятью проектировщики отреагировали представлением дополнительной кэш-памяти — кэш L2 (кэш второго уровня), между кэш L1 и основной памятью; доступ к нему осуществляется за несколько циклов синхронизации. Кэш L2 может быть подключен к шине памяти либо к собственной *шине кэша* (рис. 6.15). В некоторых высокопроизводительных системах (на-

пример, Alpha 21164) на шине памяти установлен дополнительный уровень кэш-памяти, называемый кэш L3, располагаемый в иерархии между кэш L2 и основной памятью. Несмотря на большое разнообразие проектов, основные принципы остаются одинаковыми.

6.4.1. Родовая организация кэш-памяти

Рассмотрим компьютерную систему, в которой каждый адрес памяти имеет m битов, формирующих $M = 2^m$ уникальных адресов. На рис. 6.16, а кэш-память для такой машины организована как массив из $S = 2^s$ множеств кэша. Каждое состоит из E строк кэша. Каждая строка состоит из блока данных из $B = 2^b$ байтов, бита достоверности, указывающего на то, содержит строка значащую информацию или нет, и $t = m - (b + s)$ теговых разрядов (подмножества битов из адреса памяти текущего блока), которые уникально идентифицируют блок, сохраненный в строке кэша.

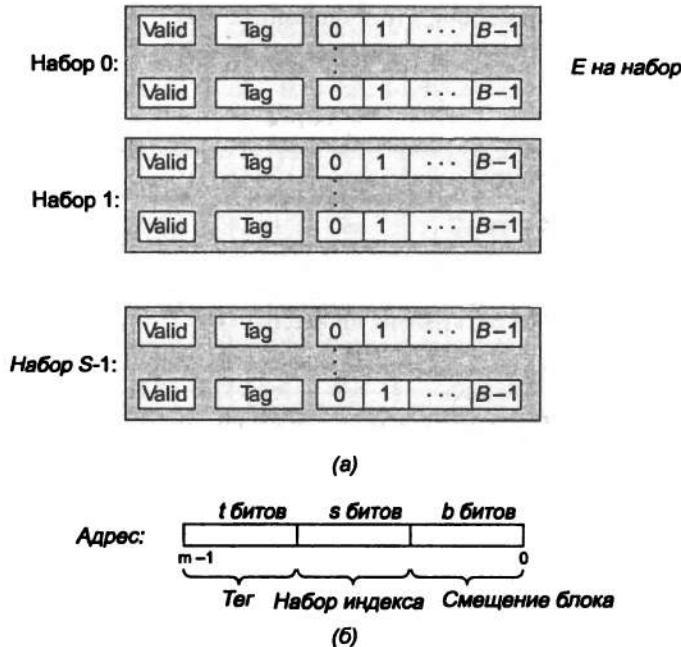


Рис. 6.16. Общая организация кэш-памяти: а — массив наборов; б — структура

Организацию кэш-памяти можно охарактеризовать кортежем (S, E, B, m) . Размер (или емкость) кэша C описывается в том, что касается совокупного размера всех блоков. Теговые биты и бит достоверности не включены. Таким образом, $C = B \times E \times S$.

Когда команда загрузки инструктирует CPU на считывание слова из адреса A основной памяти, она отправляет адрес A в кэш. Если кэш содержит копию слова в адресе A, тогда он незамедлительно отправляет слово назад в CPU. Как кэш распознает, содержит он копию этого слова в адресе A или нет? Кэш-память устроена так, что она

может найти запрошенное слово путем простого просмотра битов адреса, как это происходит в хэш-таблице с предельно простой хэш-функцией. Далее описан принцип работы.

Параметры S и B инициируют разбиение m адресных битов на три поля, показанные на рис. 6.16, б. S битов указателя множества в А формируют из указателя массив, состоящий из S множеств. Первое множество — 0, второе — 1 и т. д. При интерпретации как целого числа без знака биты указателя множества определяют, в каком множестве должно быть сохранено слово. Как только это становится известно, t теговых битов в А сообщают строку (если она существует) множества, в которой содержится слово. Стока множества содержит слово только в том случае, если задан бит достоверности, и теговые биты в строке совпадают с теговыми битами в адресе А. Как только местоположение строки определяется тегом множества, идентифицированным, в свою очередь, указателем множества, тогда биты сдвига блоков b сообщают сдвиг слова в блоке данных, состоящем из B байтов.

Читатели наверняка уже отметили, что в описаниях кэш присутствует много символов. В табл. 6.14—6.15 дается расшифровка этих символов.

Таблица 6.14. Резюме параметров множества

Фундаментальные параметры	
Параметр	Описание
$S = 2^t$	Количество множеств
E	Количество строк на множество
$B = 2^b$	Размер блока (в байтах)
$m = \log_2(M)$	Количество физических (основной памяти) адресных битов

Таблица 6.15. Резюме параметров кэша

Выведенные количества	
Параметр	Описание
$M = 2^m$	Максимальное количество уникальных адресов памяти
$s = \log_2(S)$	Количество битов указателя множества
$b = \log_2(B)$	Количество битов сдвига блоков
$t = m - (s + b)$	Количество теговых битов
$C = B \times E \times S$	Размер кэша (в байтах), исключая служебные (бит достоверности и теговые)

УПРАЖНЕНИЕ 6.6

В таблице представлены параметры для некоторого количества различных кэшей. Определите для каждого количество множеств кэша (S), теговых битов (t), битов указателя множества (s) и битов сдвига блоков.

Кэш	m	C	B	E	S	t	s	b
1	32	1024	4	1				
2	32	1024	8	4				
3	32	1024	32	32				

6.4.2. Кэш прямого отображения

Кэши сгруппированы в разные классы, исходя из E — количества строк кэша на каждое множество. Кэш с одной строкой на множество ($E = 1$) называется *кэшем прямого отображения* (рис. 6.17). Кэши прямого отображения — самые простые как для реализации, так и понимания принципов их работы, поэтому они будут использоваться для иллюстрирования некоторых общих концепций.

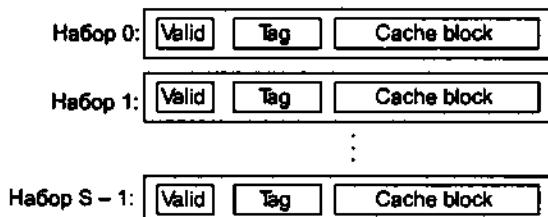


Рис. 6.17. Кэш прямого отображения

Предположим, что имеется система с CPU, регистровым файлом, кэшем L1 и основной памятью. Когда CPU выполняет команду,читывающую слово памяти w , он запрашивает слово из кэша L1. Если последний является кэшированной копией w , тогда налицо результативное отображение в кэш L1, кэш быстро извлекает w и возвращает его в CPU. В противном случае, пользователь сталкивается с промахом кэша, и CPU должен дождаться, пока кэш L1 запросит из основной памяти копию блока, содержащего w . Когда нужный блок поступает из памяти, кэш L1 сохраняет этот блок в одной из его строк, извлекает слово w из сохраненного блока и возвращает его в CPU. Процесс, при котором кэш проходит через определение того, является запрос результативным или промахом, после чего происходит извлечение запрошенного слова, состоит из трех шагов:

1. Выбора множества.
2. Сопоставления строк.
3. Извлечения слова.

Выбор множества в кэше прямого отображения

В этом шаге кэш извлекает s битов указателя множества из центра адреса для w . Эти биты интерпретируются как целое число без знака, соответствующее номеру множества. Другими словами, если кэш рассматривать как одномерный массив множеств, тогда биты указателя множества формируют указатель в этот массив. На рис. 6.18 показано функционирование выбора множества для кэша прямого отображения. В данном примере биты указателя множества 00001_2 рассматриваются как целочисленный указатель, выбирающий множество 1.

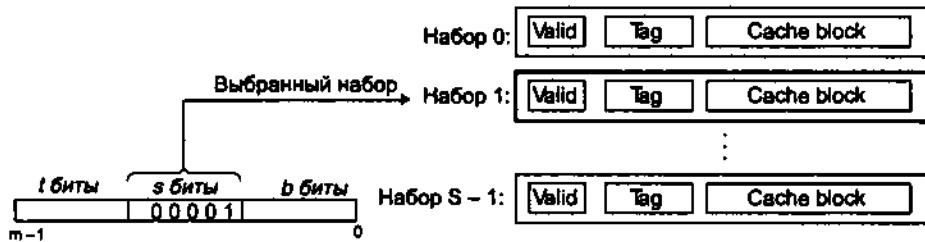


Рис. 6.18. Выбор множества в кэше прямого отображения

Сопоставление строк в кэше прямого отображения

Теперь, когда выбрано некоторое множество i , следующим шагом является определение того, сохранена ли копия слова w в одной из строк кэша, содержащихся в множестве i . В случае с кэшем прямого отображения это просто и быстро, потому что для множества имеется только одна строка. Копия w содержится в этой строке, только если задан бит достоверности, и тег в строке кэша совпадает с тегом в адресе w .

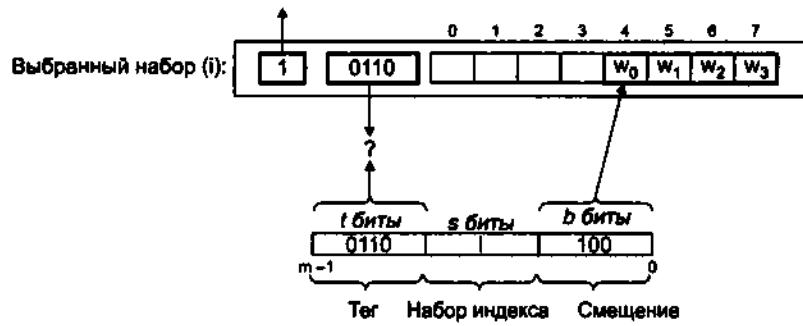


Рис. 6.19. Сопоставление строк и извлечение слова в кэше прямого отображения

На рис. 6.19 показано сопоставление строк в кэше прямого отображения. В данном примере в выбранном множестве существует только одна строка. Бит достоверности для этой строки задан, поэтому известно, что биты в теге и блоке являются значимыми. Поскольку теговые биты в строке кэша совпадают с теговыми битами в адресе, известно, что копия нужного слова действительно сохранена в этой строке. Другими

словами, налицо результативное обращение в кэш. С другой стороны, если бы не был задан бит достоверности, либо не совпадали бы теговые биты, тогда бы место имел промах кэша.

Извлечение слова в кэшах прямого отображения

При наличии результативного обращения в кэш известно, что w находится где-то в блоке. Этот последний шаг определяет то, где нужное слово начинается в блоке. Как показано на рис. 6.19, биты сдвига блока обеспечивают сдвиг первого байта нужного слова. Подобно представлению кэша как массива строк, блок можно рассматривать как массив байтов, а сдвиг байтов — как указатель на этот массив. В примере биты сдвига блока 100₂ указывают на то, что копия w начинается в блоке с байта 4. (Предполагается, что длина слов — 4 байта.)

Вытеснение строк при промахах обращения в кэш прямого отображения

При возникновении промаха кэша запрошенный блок необходимо извлечь из следующего уровня иерархии памяти и сохранить новый блок в одной из строк кэша множества, указанного битами указателя множества. Вообще говоря, если кэш заполнен достоверными строками кэша, тогда одну из существующих строк необходимо вытеснить. Для кэша прямого отображения, где каждое множество состоит только из одной строки, стратегия замены тривиальна: текущая строка вытесняется вновь выбранной строкой.

Окончательная сборка: кэш прямого обращения в действии

Механизмы, которые кэш использует для выбора множеств и идентификации строк, на редкость просты. Они и должны такими быть, потому что аппаратные средства должны выполнять их всего за несколько наносекунд. Впрочем, такого рода манипулирование с битами может показаться простым машине, но не человеку. Прояснить процесс поможет конкретный пример. Предположим, что имеется кэш прямого обращения, описываемый как

$$(S, E, B, m) = (4, 1, 2, 4)$$

Другими словами, данный кэш имеет четыре множества с одной строкой на каждое, 2 байта на блок и 4-битовые адреса. Также предположим, что каждое слово представляет собой один байт. Такие предположения, разумеется, нереальны, однако они помогут предельно упростить пример.

При изучении кэша весьма информативным способом может стать нумерация всего адресного пространства и разбиение битов, как было сделано в табл. 6.16 для 4-битового примера. В том, что касается пронумерованного пространства, следует отметить несколько интересных моментов:

- Сцепление теговых битов и битов указателей уникально идентифицирует каждый блок памяти. Например, блок 0 состоит из адресов 0 и 1, блок 1 — из адресов 2 и 3, блок 2 — из адресов 4 и 5 и т. д.

- Поскольку имеется восемь блоков памяти, но только четыре множества кэша, множественные блоки отображаются на то же множество кэша (т. е. они имеют тот же указатель множества). Например, блоки 0 и 4 отображаются на множество 0, блоки 1 и 5 оба отображаются на множество 1 и т. д.
- Блоки, отображающиеся на одно и то же множество кэша, уникально идентифицируются тегом. Например, блок 0 имеет теговый бит 0, а блок 4 — теговый бит 1, блок 1 имеет теговый бит 0, тогда как блок 5 — теговый бит 1.

Таблица 6.16. 4-битовое адресное пространство кэша прямого отображения

Адресные биты				
Адрес (десятичный)	Теговые биты ($t = 1$)	Биты указателя ($s = 2$)	Биты сдвига ($b = 1$)	Число блока (десятичное)
0	0	00	0	0
1	0	00	1	0
2	0	01	0	1
3	0	01	1	1
4	0	10	0	2
5	0	10	1	2
6	0	11	0	3
7	0	11	1	3
8	1	00	0	4
9	1	00	1	4
10	1	01	0	5
11	1	01	1	5
12	1	10	0	6
13	1	10	1	6
14	1	11	0	7
15	1	11	1	7

Смоделируем кэш в действии в процессе выполнения CPU последовательности операций считывания. Помните, что для данного примера предполагается, что CPU считывает однобайтовые слова. Несмотря на то, что такого рода моделирование вручную — дело довольно утомительное, и читателю захочется его пропустить, по мнению авторов, понять принцип работы кэш-памяти будет сложно, если не выполнить хотя бы несколько таких примеров.

Первоначально кэш — пустой, каждый бит достоверности — 0 (табл. 6.17).

Таблица 6.17. Первоначальное пространство кэша прямого отображения

Множество	Достоверность	Тег	блок [0]	блок [1]
0	0			
1	0			
2	0			
3	0			

Каждая строка в таблице представляет строку кэша. Первый столбец обозначает множество, которому принадлежит строка, однако следует помнить о том, что данный столбец приведен только для удобства и не является фактической частью кэша. Оставшиеся четыре столбца представляют фактические биты в каждой строке кэша. Посмотрим, что происходит, когда CPU выполняет последовательность операций считывания:

- Считывание слова в адресе 0 (табл. 6.18). Поскольку бит достоверности для множества 0 представляет 0, имеет место промах кэша. Кэш выбирает из памяти блок 0 (или кэш нижнего уровня) и сохраняет его в множестве 0. Затем кэш возвращает $m[0]$ (содержимое ячейки памяти 0) из блока 0 вновь выбранной строки кэша.

Таблица 6.18. Считывание слова в адресе 0

Множество	Достоверность	Тег	блок [0]	блок [1]
0	1	0	$m[0]$	$m[1]$
1	0			
2	0			
3	0			

- Считывание слова в адресе 1. Это — результативное обращение в кэш. Последний немедленно возвращает $m[1]$ из блока 1 строки кэша. Состояние кэша не меняется.
- Считывание слова в адресе 13 (табл. 6.19). Поскольку строка кэша во множестве 2 — достоверна, налицо — промах кэша. Кэш загружает блок 6 во множество 2 и возвращает $m[13]$ из блока 1 новой строки кэша.
- Считывание слова в адресе 8 (табл. 6.20). Промах кэша. Страна кэша во множестве 0 — достоверна, однако теги не соответствуют друг другу. Кэш загружает блок 4 во множество 0 (замещая строку, оставшуюся там после считывания адреса 0) и возвращает $m[8]$ из блока 0 новой строки кэша.

Таблица 6.19. Считывание слова в адресе 13

Множество	Достоверность	Тег	блок [0]	блок [1]
0	1	0	m[0]	m[1]
1	0			
2	1	1	m[12]	m[13]
3	0			

Таблица 6.20. Считывание слова в адресе 8

Множество	Достоверность	Тег	блок [0]	блок [1]
0	1	1	m[8]	m[9]
1	0			
2	1	1	m[12]	m[13]
3	0			

5. Считывание слова в адресе 0 (табл. 6.21). Это — еще один промах кэша, вследствие неудачного факта того, что блок 0 был замещен во время предыдущего обращения к адресу 8. Такой тип промаха, когда в кэше очень много свободного места, но приходится чередовать обращения к блокам, отображенными на одно и то же множество, является примером конфликтного промаха.

Таблица 6.21. Промах кэша

Множество	Достоверность	Тег	блок [0]	блок [1]
0	1	0	m[0]	m[1]
1	0			
2	1	1	m[12]	m[13]
3	0			

Конфликтные промахи в кэше прямого отображения

В реальных программах конфликтные промахи — довольно обычны и могут вызывать серьезные проблемы производительности. Конфликтные промахи в кэшах прямого отображения обычно имеют место, когда программы получают доступ к массивам, размеры которых составляют степень двойки. Например, рассмотрим функцию, вычисляющую внутреннее произведение двух векторов (листинг 6.8).

Листинг 6.8. Внутреннее произведение векторов

```

1     float dotprod (float x [8], float y [8])
2     {
3         float sum = 0.0
4         int i;
5
6         for (i = 0; i < 8; i++)
7             sum += x[i] * y[i];
8         return sum;
9     }

```

Данная функция обладает хорошей пространственной локальностью с учетом x и y , поэтому можно ожидать, что она продемонстрирует хорошее число результативных обращений в кэш. К сожалению, это не всегда верно.

Предположим, что запасы времени составляют 4 байта, что x загружен в 32 байта смежной памяти, начинающихся в адресе 0, и что y запускается сразу после x в адресе 32. Для простоты предположим, что блок состоит из 16 байтов (достаточно большой для сохранения четырех запасов времени), а кэш — из двух множеств, для общего размера кэша в 32 байта. Делается допущение, что переменная sum фактически сохранена в регистре CPU и, следовательно, не требует обращения к памяти. При всех указанных допущениях каждый $x[i]$ и $y[i]$ будет отображен в идентичное множество кэша (табл. 6.22).

Таблица 6.22. Конфликтные промахи

Элемент	Адрес	Указатель множества	Элемент	Адрес	Указатель множества
$x[0]$	0	0	$y[0]$	32	0
$x[1]$	4	0	$y[1]$	36	0
$x[2]$	8	0	$y[2]$	40	0
$x[3]$	12	0	$y[3]$	44	0
$x[4]$	16	1	$y[4]$	48	1
$x[5]$	20	1	$y[5]$	52	1
$x[6]$	24	1	$y[6]$	56	1
$x[7]$	28	1	$y[7]$	60	1

Во время выполнения первая итерация цикла обращается к $x[0]$ — промаху, вызывающему загрузку в множество 0 блока, содержащего $x[0]$ – $x[3]$. Следующее обращение осуществляется к $y[0]$ — другой промах, вызывающий копирование блока, содержащего $y[0]$ – $y[3]$, в множество 0 с наложением записи на значения x , которые

были скопированы при предыдущем обращении. Во время следующей итерации обращение к $x[1]$ вызывает промах, вызывающий загрузку блока $x[0] - x[3]$ назад во множество 0, с наложением записи на блок $y[0] - y[3]$. Теперь налицо конфликтный промах, и фактически, результат каждого последующего обращения к x и y вызовет конфликтный промах, по мере пробуксовки между блоками x и y . Термин "пробуксовка" описывает любую ситуацию, когда кэш многократно загружает и вытесняет одни и те же множества блоков кэша.

В нижней строке, несмотря на то, что программа обладает хорошей пространственной локальностью, и в кэш имеется место для удержания блоков как для $x[i]$, так и для $y[i]$, результатом каждого обращения будет конфликтный промах, потому что эти блоки отображаются в одно и то же множество кэша. Для такого рода пробуксовки обычным результатом будет замедление быстродействия на коэффициент 2 или 3. Также следует учитывать, что, несмотря на простоту примера, для более крупных и более реалистичных кэш прямого обращения решение данной проблемы представляют определенную сложность.

К счастью, программисты могут легко избавиться от пробуксовки, поняв, что происходит. Одним из простейших решений будет размещение B байтов заполнения (незначащей информации) в конце каждого массива. Например, вместо определения того, что x — это float $x[8]$, определяем его как float $x[12]$. Допуская, что y запускается в памяти непосредственно после x , имеем следующее отображение элементов массива во множествах (табл. 6.23).

Таблица 6.23. Отображение элементов

Элемент	Адрес	Указатель множества	Элемент	Адрес	Указатель множества
$x[0]$	0	0	$y[0]$	48	1
$x[1]$	4	0	$y[1]$	52	1
$x[2]$	8	0	$y[2]$	56	1
$x[3]$	12	0	$y[3]$	60	1
$x[4]$	16	1	$y[4]$	64	0
$x[5]$	20	1	$y[5]$	68	0
$x[6]$	24	1	$y[6]$	72	0
$x[7]$	28	1	$y[7]$	76	0

Имея заполнение в конце x , $x[i]$ и $y[i]$, выполняем отображение в разные множества, что устраняет "буксующие" конфликтные промахи.

УПРАЖНЕНИЕ 6.7

Какая доля от общего количества обращений к x и y в предыдущем примере dotprod станет результативным обращением в кэш после ввода заполнения в массив x ?

Чему служит указатель со срединными битами

У читателя может возникнуть вопрос о том, зачем в кэш нужны срединные биты вместо старших битов для задания указателя множества. Существует обоснованное объяснение этого факта. Это показано на рис. 6.20. Если в качестве указателя использовать старшие биты, тогда некоторые блоки смежной памяти отобразятся в то же множество кэша. Например, на схеме показано, что первые четыре блока отображаются в первое множество кэша, вторые четыре блока отображаются во второе множество и т. д. Если программа обладает хорошей пространственной локальностью и просматривает элементы массива последовательно, тогда в любой момент времени кэш может удерживать только кусок массива размером с блок. Такое использование кэша неэффективно. Читателю следует сравнить такой случай с индексированием срединными битами, когда смежные блоки всегда отображаются на разные строки кэша. В этом случае кэш может удерживать весь кусок массива размера C , где C — размер кэша.

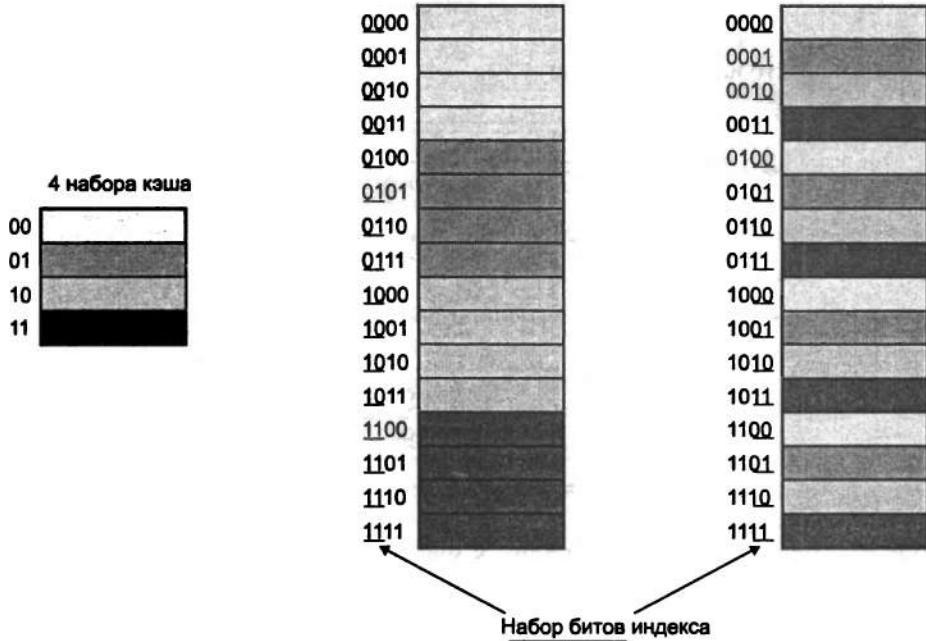


Рис. 6.20. Зачем индексировать кэш срединными битами

УПРАЖНЕНИЕ 6.8

Вообще, если старшие биты s любого адреса используются в качестве указателя массива, тогда смежные куски блоков памяти будут отображаться в то же множество кэша.

1. Сколько блоков входит в каждый из этих смежных кусков массива?
2. Рассмотрим следующий код, выполняющийся на системе с кэш-формы $(S, E, B, m) = (512, 1, 32, 32)$:

```

int array [4096];

for (i = 0; i < 4096; i++)
    sum += array [i];

```

Каково максимальное число блоков массива, сохраненных в кэше в любой момент времени?

6.4.3. Ассоциативные множествам кэши

Проблема с конфликтными промахами в кэшах прямого отображения происходит из ограничения, что каждое множество имеет только одну строку (в терминологии книги — $E = 1$). Ассоциативные множествам кэши снижают это ограничение так, что в каждом множестве содержится более одной строки. Кэш с $1 < E < C/B$ часто называют кэшем, ассоциативным E -стороннему множеству. В следующем разделе рассматривается особый случай, где $E = C/B$. На рис. 6.21 показана организация кэша, ассоциативного двустороннему множеству.

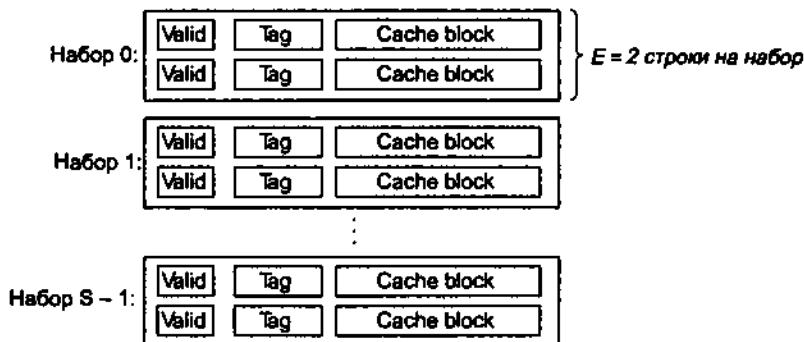


Рис. 6.21. Ассоциативный множествам кэш ($1 < E < C/B$)

В ассоциативном множеству кэше каждое множество содержит более одной строки. Данный конкретный пример показывает двухсторонний ассоциативный множеству кэш.

Выбор множества в ассоциативных множеству кэшах

Выбор множества идентичен кэшу с прямым обращением с битами указателя множества, идентифицирующими данное множество. На рис. 6.22 показан этот принцип.

Сопоставление строк и извлечение слова в ассоциативных множеству кэшах

Сопоставление строк в большей степени используется в ассоциативных множеству кэшах, нежели в кэшах с прямым обращением, потому что при этом необходима проверка тегов и битов достоверности множественных строк для определения того, при-

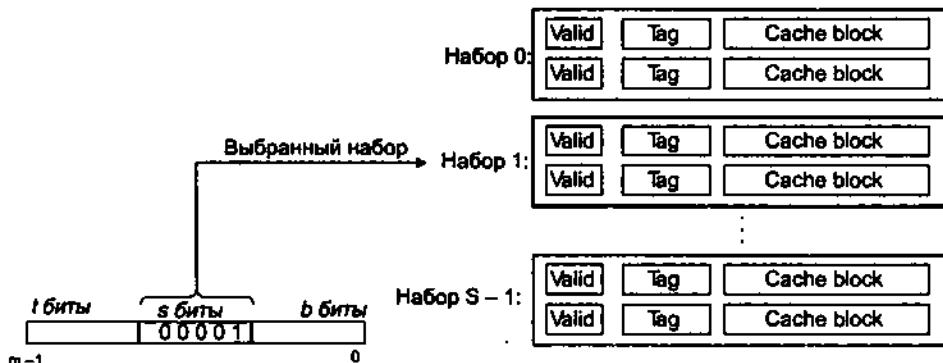


Рис. 6.22. Выбор множества в ассоциативном множеству кэша

существует ли запрошенное слово в множестве. Традиционная память представляет собой массив значений, принимающих адрес в качестве входного значения и возвращающий значение, сохраненное в этом адресе. С другой стороны, *ассоциативная память* — это массив пар (ключ, значение), принимающий в качестве входного значения ключ и возвращающий значение из одной пары (ключ, значение), совпадающее с ключом ввода. Таким образом, каждое множество в ассоциативном множеству кэша можно рассматривать как небольшую ассоциативную память, где ключи являются конкатенацией (цеплением) теговых битов и битов достоверности, а значениями является содержимое блока.

На рис. 6.23 отображен основной принцип сопоставления строк в ассоциативном множеству кэша. Важной идеей здесь является то, что любая строка множества может содержать любой из блоков памяти, отображающихся в это множество. Поэтому кэш должен осуществлять поиск каждой строки во множестве с целью нахождения достоверной строки, тег которой совпадает с тегом в адресе. Если кэш находит такую строку, тогда налицо результативное обращение в кэш, и сдвиг блока, как и раньше, выбирает из блока слово.

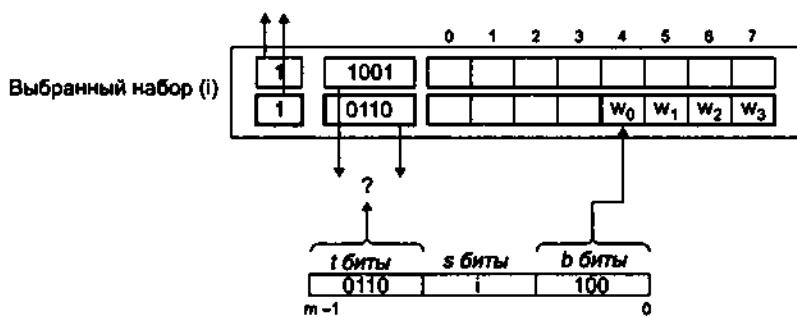


Рис. 6.23. Сопоставление строк и извлечение слова в ассоциативном множеству кэша

Вытеснение строк при промахах в ассоциативных множеству кэшах

Если запрошенное центральным процессором слово не сохранено ни в одной строке множества, тогда пользователь сталкивается с промахом кэша, и последний должен выбрать из памяти блок, содержащий данное слово. Однако, как только кэш извлекает блок, какую строку необходимо вытеснить? Разумеется, если существует пустая строка, тогда она-то и будет подходящим кандидатом на вытеснение. Но если во множестве пустых строк нет, тогда необходимо выбрать одну из них и надеяться на то, что CPU больше никогда не запросит вытесненную строку.

При написании кодов программистам очень сложно использовать знание стратегии замены кэша, поэтому авторы не будут вдаваться в подробности этой темы. Самой простой стратегией замены является произвольный выбор строки на вытеснение. В других, более сложных стратегиях используется принцип локальности с тем, чтобы по возможности свести к минимуму вероятность очередного обращения процессора к вытесненной строке. Например, стратегия наименьшей частоты использования (LFU) вытеснит строку, запрос которой осуществлялся наименьшее количество раз в течение определенного временного окна в прошлом. Стратегия наиболее давнего использования (LRU) вытеснит строку, последнее обращение к которой было осуществлено на самом "дальнем" этапе в прошлом. Все эти стратегии требуют дополнительного времени и аппаратных средств. Однако, двигаясь в направлении от CPU вниз по иерархии памяти, затраты на промах становятся более значительными, и минимизация промахов с помощью действенных стратегий замен имеет определяющее значение.

6.4.4. Полностью ассоциативные кэши

Полностью ассоциативный кэш состоит из одного множества ($E = C/B$), содержащего все строки кэша. На рис. 6.24 показана его базовая структура.

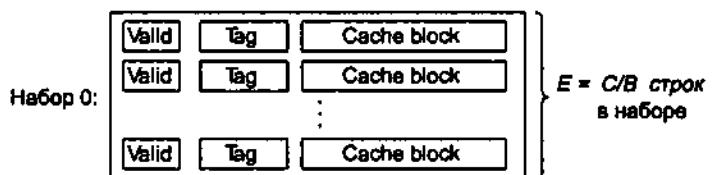


Рис. 6.24. Полностью ассоциативный множеству кэш

Выбор множества в полностью ассоциативных множеству кэшах

Выбор множества в полностью ассоциативном множеству кэше — процесс тривиальный, потому что имеется только одно множество, структура которого показана на рис. 6.25. Обратите внимание на то, что в адресе нет битов указателя множества; адрес разбит только на тег и сдвиг блока.

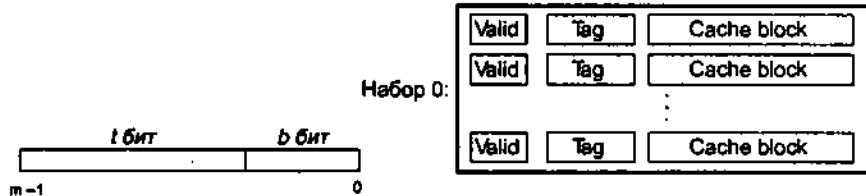
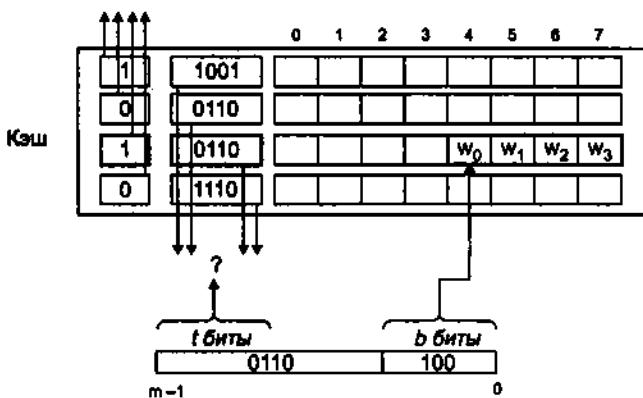


Рис. 6.25. Выбор множества в полностью ассоциативном множеству кэше

Сопоставление строк и извлечение слова в полностью ассоциативных множествах кэшах

Сопоставление строк и извлечение слова в полностью ассоциативном множеству кэше работает так же, как и с ассоциативным кэше (рис. 6.26). Различие заключается в масштабе.

Весь кэш состоит из одного множества, поэтому по умолчанию всегда выбирается множество 0.

Рис. 6.26. Сопоставление строк и извлечение слова
в полностью ассоциативном множеству кэше

По причине того, что цепи кэш должны осуществлять параллельный поиск большого количества совпадающих тегов, построить крупный и быстродействующий кэш весьма трудно и дорого. В результате этого полностью ассоциативные кэши подходят только для небольших устройств кэш-памяти, таких как буферы быстрого преобразования адресов (TLB) в системах виртуальной памяти, кэширующих записи в таблицах страниц (см. разд. 10.6.2).

УПРАЖНЕНИЕ 6.9

Описанные проблемы помогают укрепить понимание принципов работы устройств кэш-памяти.

Допустим следующее:

- память адресуема по байтам;
- доступ к памяти осуществляется как к однобайтовым словам (а не 4-байтовым);
- адреса имеют ширину в 13 битов;
- кэш — двусторонний ассоциативный множеству ($E = 2$) с размером блока 4 байта ($B = 4$) и 8 множествами ($S = 8$).

Содержимое кэша показано следующим образом (все числа приведены в шестнадцатеричной записи):

Двусторонний ассоциативный множеству кэш													
			Строка 0				Строка 1						
Указатель множества	Ter	Дост.	Байт 0	Байт 1	Байт 2	Байт 3	Ter	Дост.	Байт 0	Байт 1	Байт 2	Байт 3	
0	09	1	86	30	3F	10	00	0	-	-	-	-	-
1	45	1	60	4F	E0	23	38	1	00	BC	0B	37	
2	EB	0	-	-	-	-	0B	0	-	-	-	-	
3	06	0	-	-	-	-	32	1	12	08	7B	AD	
4	C7	1	06	78	07	C5	05	1	40	67	C2	3B	
5	71	1	0B	DE	18	4B	6E	0	-	-	-	-	
6	91	1	A0	B7	26	2D	F0	0	-	-	-	-	
7	46	0	-	-	-	-	DE	1	12	C0	88	37	

Ниже показан формат адреса (один бит на рамку)

12 11 10 9 8 7 6 5 4 3 2 1 0

Укажите (путем разметки диаграммы) поля, которые будут использоваться для определения следующего:

1. CO — сдвиг блока кэша;
2. CI — указатель множества кэша;
3. CT — тег кэша.

УПРАЖНЕНИЕ 6.10

Представьте, что программа, выполняющаяся на машине упр. 6.9, обращается к однобайтовому слову в адресе 0х0E34. Укажите запись кэша, к которой осуществлен доступ и возвращенное значение байта в шестнадцатеричном представлении. Укажите,

возникают ли промахи кэша. При их наличии введите "—" в графу "Возвращенный байт кэша".

1. Формат адреса:

12 11 10 9 8 7 6 5 4 3 2 1 0

2. Обращение к ячейке памяти:

Параметр	Значение
Сдвиг блока кэша (CO)	0x
Указатель множества кэша (CI)	0x
Тег кэша (CT)	0x
Результативное обращение в кэш (Да/Нет)	
Возвращенный байт кэша	0x

УПРАЖНЕНИЕ 6.11

Повторите упр. 6.10 для адреса ячейки памяти 0x0DD5.

1. Формат адреса:

12 11 10 9 8 7 6 5 4 3 2 1 0

2. Обращение к ячейке памяти:

Параметр	Значение
Сдвиг блока кэша (CO)	0x
Указатель множества кэша (CI)	0x
Тег кэша (CT)	0x
Результативное обращение в кэш (Да/Нет)	
Возвращенный байт кэша	0x

УПРАЖНЕНИЕ 6.12

Повторите упр. 6.10 для адреса ячейки памяти 0x1FE4.

1. Формат адреса:

12 11 10 9 8 7 6 5 4 3 2 1 0

2. Обращение к ячейке памяти:

Параметр	Значение
Сдвиг блока кэша (CO)	0x
Указатель множества кэша (CI)	0x
Тег кэша (CT)	0x
Результативное обращение в кэш (Да/Нет)	
Возвращенный байт кэша	0x

УПРАЖНЕНИЕ 6.13

Для кэша, описанного в упр. 6.9, перечислите все шестнадцатеричные адреса памяти, которые будут иметь результативное обращение в кэш во множестве 3.

6.4.5. Работа с операциями записи

Как уже отмечалось, функционирование кэша в том, что касается операций считывания — прямолинейно. Прежде всего, в кэш необходимо искать копию нужного слова *w*. При результативном обращении в кэш слово *w* должно сразу возвращаться в CPU. При промахе кэша из памяти выбирается блок, содержащий слово *w*, после чего данный блок сохраняется в определенной строке кэша (с возможным вытеснением действующей строки), и слово *w* возвращается в CPU.

Ситуация с операциями записи — немного более сложная. Предположим, что CPU записывает уже кэшированное слово *w* (обращение к записи). После того, как кэш обновляет копию *w*, что происходит с обновлением копии *w* в памяти? Простейшим подходом, известным как "сквозной", является мгновенная запись блока кэша *w* в память. Будучи простым методом, сквозная запись обладает недостатком вызова транзакции записи на шину с выполнением каждой команды сохранения. Другой подход, называемый *обратной записью*, откладывает обновление памяти на максимально длительное время путем записи обновленного блока в память только после его вытеснения из кэша алгоритмом замещения. Из-за локальности обратная запись может значительно сократить количество операций сшиной, однако она обладает недостатком дополнительной сложности. Кэш должен поддерживать дополнительный "запорченный" бит для каждой строки кэша, указывающий на то, имела ли место модификация блока кэша.

Другой вопрос заключается в том, как обрабатывать промахи записи. Один способ, называемый *записью с выделением*, загружает нужный блок памяти в кэш, после чего обновляет блок кэша. При записи с выделением (строк в кэш-памяти) делается попытка использования пространственной локальности операций записи, однако недостаток заключается в том, что результатом каждого промаха становится перенос блока из памяти в кэш. При применении альтернативного способа — *записи без выделения строк* — кэш обходится, и слово записывается напрямую в память. Кэши со сквозной записью обычно осуществляют запись без выделения строк. Кэши с обратной записью, как правило, осуществляют запись с выделением строк.

Оптимизация кэша для операций записи — дело весьма деликатное и сложное, поэтому авторы рассматривают его здесь только "по касательной". Многочисленные подробности варьируются, в зависимости от особенностей каждой конкретной системы; все они, как правило, запатентованы и мало описаны. Программисту, пытающемуся писать "дружественные" кэшам — в разумном смысле — программы, можно предложить воспользоваться умозрительной моделью: кэш с обратной записью и кэш с выделением строк. Для подобного предположения существует несколько причин.

Как правило, кэши на нижних уровнях иерархии памяти, вероятнее всего, будут использовать стратегию обратной записи, нежели сквозной, из-за более длительных значений времени переноса. Например, в системах виртуальной памяти (использующих основную память в качестве кэша для блоков, сохраненных на диске) применяется исключительно стратегия обратной записи. Однако, по мере повышения логической плотности, повышающаяся сложность обратной записи перестает быть серьезной помехой, и на всех уровнях современных компьютерных систем пользователи наблюдают кэш с обратной записью. Поэтому данное допущение соответствует современным тенденциям. Другой причиной использования подхода обратной записи и записи с выделением строк является его симметричность тому, как обрабатываются операции считывания, в том, что обратная запись и запись с выделением строк пытаются использовать локальность. Следовательно, для демонстрации высокой временной и пространственной локальности можно создавать программы высокого уровня, вместо того, чтобы пытаться осуществлять оптимизацию под конкретную систему организации памяти.

6.4.6. Кэши команд и унифицированные кэши

До сих пор предполагалось, что в кэш содержатся только данные программы. Однако в действительности, помимо данных, в кэше также могут содержаться команды. Кэш, содержащий только команды, называется i-кэш (instruction-cache). Кэш, содержащий только данные программы, называется d-кэш (data-cache). Кэш, содержащий команды и данные программы, называется унифицированным. В обычную настольную рабочую станцию входят i-кэш L1 и d-кэш L2 на чипе CPU, а также внешний унифицированный кэш L2. На рис. 6.27 показана базовая структура.

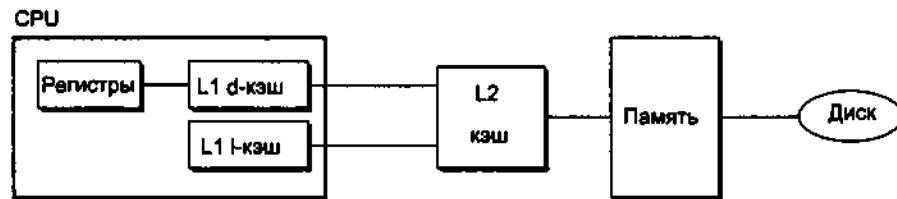


Рис. 6.27. Типичная многоуровневая организация кэша

В некоторых высокотехнологичных системах, например, на базе Alpha 21164, кэш L1 и L2 размещены на чипе CPU и имеют дополнительный внешний кэш L3. В конструкцию современных процессоров включены отдельные внутрикристальные i-кэш и

д-кэш, с целью повышения производительности. С двумя отдельными кэшами процессор может считывать слово команды и слово данных в течение одного цикла синхронизации. На данном этапе неизвестно, существует ли система, включающая кэш L4, хотя, по мере расхождения скоростей процессора и памяти, вполне вероятно, что это может иметь место.

Какая структура кэша входит в реальную систему

В системах Intel Pentium включена структура кэша, показанная на рис. 6.27 с внутристальным i-кэш L1, внутристальным d-кэш L1 и внешним унифицированным кэш L2. В табл. 6.24 показаны базовые параметры этих кэш.

Таблица 6.24. Базовые параметры кэша

Тип кэша	Ассоциативность (E)	Размер блока (B)	Множества (S)	Размер кэша (C)
Внутренний i-кэш L1	4	32 B	128	16 Кбайт
Внутренний d-кэш L1	4	32 B	128	16 Кбайт
Внешний унифицированный кэш L2	4	32 B	1024—16384	128 Кбайт—2 Мбайт

6.4.7. Влияние параметров кэша на производительность

Производительность кэша оценивается по нескольким показателям:

- Коэффициент неудач — доля обращений к памяти во время выполнения программы или части программы, демонстрирующая промахи. Коэффициент неудач рассчитывается как количество промахов/количество обращений.
- Коэффициент успеха — доля успешных обращений к памяти. Рассчитывается как коэффициент неудач.
- Время результативного обращения — время, необходимое для передачи слова из кэша в CPU, включая время на выбор множества, идентификацию строк и выборку слова. Время результативного обращения для кэш L1 обычно составляет 1—2 цикла синхронизации.
- Накладные расходы при отсутствии нужных данных. Любое дополнительное необходимо время из-за промаха. Накладные расходы для промахов L1, использованных из L2, обычно составляют 5—10 циклов синхронизации. Накладные расходы для промахов L1, использованных из основной памяти, обычно составляют 25—100 циклов синхронизации.

Оптимизация компромиссов между затратами и производительностью кэш-памяти — весьма деликатная операция, требующая расширенного моделирования кодов контрольных задач; ее рассмотрение не входит в рамки данной книги. Однако здесь можно идентифицировать некоторые качественные альтернативы.

Влияние размера кэша

С одной стороны, более крупные кэши имеют тенденцию к увеличению коэффициента успеха. С другой стороны, повышать быстродействие крупных устройств памяти — всегда сложно. В результате этого, более крупные кэши имеют тенденцию к увеличению времени результативного обращения. Это особенно важно для внутренних кэш L1, которые должны иметь время результативного обращения размером в один цикл синхронизации.

Влияние размера блока

Крупные блоки представляют собой смесь успехов и неудач. С одной стороны, крупные блоки могут помочь увеличить коэффициент успеха путем использования пространственной локальности, которая может присутствовать в программе. Однако для данного размера кэша крупные блоки подразумевают меньшее количество строк кэша, которые могут отрицательно повлиять на коэффициент удач в программах с большей степенью временной, но не пространственной локальностью. Также крупные блоки отрицательно влияют на накладные расходы при отсутствии нужных данных, поскольку крупные блоки обуславливают более длительные значения времени переноса. В современных системах, как правило, используются кэш-блоки, содержащие от 4 до 8 слов.

Влияние ассоциативности

Вопрос здесь заключается во влиянии выбора параметра E — количества строк кэш на каждое множество. Преимущество высокой ассоциативности (т. е. более высоких значений E) заключается в повышении чувствительности кэша к пробуксовке из-за конфликтных промахов. Однако высокая ассоциативность достигается с большими затратами. Высокая ассоциативность требует слишком больших затрат на реализацию и добиться ее быстродействия достаточно трудно. Она требует большего количества теговых битов на строку, дополнительных битов состояния LRU на строку, а также дополнительной управляющей логики. Высокая ассоциативность может увеличить время результативного обращения из-за повышенной сложности, а также увеличить накладные расходы из-за повышенной сложности выбора строки-«жертвы».

Выбор ассоциативности сводится к компромиссу между временем результативного обращения и накладными расходами. По традиции, высокопроизводительные системы, стимулирующие тактовую частоту, предпочитают кэш L1 (где накладные расходы составляют всего несколько циклов) и невысокую степень ассоциативности (2—4) для более низких уровней. Однако строгих и быстро срабатывающих правил не существует. В системах Intel Pentium все кэши L1 и L2 — четырехсторонне ассоциативны множествам. В системах Alpha21164 кэш команд и данных — прямого отображения, кэш L2 — трехсторонне ассоциативный множествам, а кэш L3 — прямого отображения.

Влияние стратегии записи

Кэши сквозной записи просты для реализации, и в них может использоваться **буфер записи**, работающий на обновление памяти независимо от кэша. Более того, затраты на промахи считывания снижаются, потому что они не допускают запись в память. С другой стороны, результатом наличия кэша обратной записи является меньшее количество переносов, обеспечивающее повышенную пропускную способность памяти для устройств ввода-вывода, выполняющих DMA (прямой доступ к памяти). Более того, сокращение количества переносов приобретает особую важность, по мере понижения по иерархии и увеличения времени переноса. Вообще кэши, расположенные ниже по иерархии, вероятнее будут использовать обратную запись, нежели сквозную.

Строки кэш, множества и блоки: Чем они различаются

Строки кэша, множества и блоки очень просто перепутать. Далее предлагается подробное обсуждение этих понятий.

- **Блок** — это пакет информации фиксированного размера, перемещающийся между кэшем и основной памятью (или кэшем нижнего уровня).
- **Строка** — это контейнер в кэше, сохраняющий блок, а также другую информацию: бит достоверности и теговые биты.
- **Множество** — это набор одной или более строк. Множества в кэшах прямого отображения состоят из одной строки. Множества в ассоциативных и полностью ассоциативных множествах кэшах состоят из множественных строк.

В кэшах прямого отображения множества и строки — эквивалентны. Однако в ассоциативных кэшах множества и строки очень различаются, и эти термины нельзя использовать взаимозаменяя.

Поскольку в строке всегда сохраняется один блок, термины "строка" и "блок" часто взаимозаменямы. Например, системные профессионалы обычно используют термин "размер строки" кэша, когда на самом деле подразумевается размер блока. Такое использование — вполнеично, и не должно вызывать путаницы, пока существует понимание того, что такое блок, а что — строка.

6.5. Написание кодов, дружественных кэш-памяти

В разд. 6.2 представлено понятие локальности и в общих чертах обсуждалось то, что составляет структуру высокой степени локальности. Теперь, понимая принципы работы кэш-памяти, имеет смысл определить некоторые подробности. Программы с улучшенной локальностью будут стремиться к пониженному коэффициенту неудач, а программы с пониженными коэффициентами неудач будут стремиться к более быстрому выполнению, нежели программы с повышенными коэффициентами неудач. Таким образом, хорошие программисты должны всегда делать попытки к написанию программ, благополучно взаимодействующих с кэш-памятью, в том смысле, что они будут иметь хорошую локальность.

Базовый подход, подтверждающий то, что написанный код благополучно взаимодействует с кэш-памятью, предполагает:

1. Обеспечение быстродействия общего случая. Часто при выполнении программ большая часть времени затрачивается на несколько основных функций. При выполнении последних большая часть времени затрачивается на несколько циклов. Поэтому следует сосредоточиться на внутренних циклах основных функций, не принимая во внимание остальные.
2. Сведение к минимуму количество промахов кэша в каждом внутреннем цикле. При всех равных условиях, например общем числе операций загрузки и сохранения, циклы с пониженными значениями коэффициента неудач будут срабатывать быстрее.

Для понимания того, как это происходит на практике, рассмотрим функцию `sumvec` (листинг 6.9).

Листинг 6.9. Функция суммирования вектора

```

1     int sumvec (int v[N])
2     {
3         int i, sum = 0;
4
5         for (i = 0; i < N; i++)
6             sum += v[i];
7         return sum;
8     }

```

Благоприятна ли эта функция использованию кэша? Во-первых, обратите внимание на то, что в теле цикла в отношении переменных `i` и `sum` наблюдается хорошая временная локальность. Фактически, по причине наличия локальных переменных, любой приемлемый оптимизирующий компилятор будет кэшировать их в регистровый файл — на самый высокий уровень иерархии памяти. Теперь рассмотрим обращения шага 1 к вектору `v`. Вообще, если кэш имеет размер блока в B байт, тогда результатом комбинации обращений шага k (где k выражено словами) в среднем становится $\min(1, (\text{длина слова} \times k)/B)$ промахов на итерацию цикла. Например, предположим, что `v` выровнен по блоку, слова состоят из 4 байтов, блоки кэша — из 4 слов, а кэш изначально пуст (холодный кэш). Тогда, независимо от структуры кэша, результатом обращений к `v` станет следующая комбинация промахов и результативных обращений (табл. 6.25).

В данном примере обращение к `v[0]` является промахом, и соответствующий блок, содержащий `v[0]–v[3]`, загружается в кэш из памяти. Таким образом, следующие три обращения будут результативными. Обращение к `v[4]` вызывает очередной промах при загрузке в кэш следующего блока, а три последующих обращения оказываются результативными и т. д. Три из четырех обращений будут результативными, и это — лучшее, чего можно добиться в случае с холодным кэшем.

Таблица 6.25. Комбинация промахов

<i>v[i]</i>	<i>i = 0</i>	<i>i = 1</i>	<i>i = 2</i>	<i>i = 3</i>	<i>i = 4</i>	<i>i = 5</i>	<i>i = 6</i>	<i>i = 7</i>
Порядок доступа, [h] результативное обращение или [m] — промах	1[m]	2[h]	3[h]	4[h]	5[m]	6[h]	7[h]	8[h]

В качестве резюме, простой приведенный пример sumvec иллюстрирует два важных положения о написании кодов, дружественных кэш-памяти:

1. Многократные обращения к локальным переменным полезны, потому что компилятор может кэшировать их в регистровый файл (временная локальность).
2. Комбинации обращений по шагу 1 полезны, потому что кэши на всех уровнях иерархии памяти сохраняют данные в виде смежных блоков (пространственная локальность).

Пространственная локальность особенно важна в программах, работающих на многомерных массивах. Например, рассмотрим функцию sumarrayrows, описанную в разд. 6.2, которая суммирует элементы двумерного массива в развертывании по строкам (листинг 6.10).

Листинг 6.10. Суммирование в двумерном массиве

```

1     int sumarrayrows (int a [M] [N])
2     {
3         int i, j, sum = 0;
4
5         for (i = 0; i < M; i++)
6             for (j = 0; j < N; j++)
7                 sum += a[i] [j];
8
9     }

```

Поскольку язык C сохраняет массивы в развертывании по строкам, внутренний цикл данной функции имеет ту же желаемую комбинацию доступа по шагу 1, что и sumvec. Например, предположим, что те же допущения, что и для sumvec, сделаны для кэша. Тогда результатом обращения к массиву *a* станет следующая комбинация промахов и результативных обращений (табл. 6.26).

Обратите внимание, что происходит, если внести, на первый взгляд, безобидное изменение порядка циклов (листинг 6.11).

Таблица 6.26. Комбинация промахов

$a[i][j]$	$j = 0$	$j = 1$	$j = 2$	$j = 3$	$j = 4$	$j = 5$	$j = 6$	$j = 7$
$i = 0$	1[m]	2[h]	3[h]	4[h]	5[m]	6[h]	7[h]	8[h]
$i = 1$	9[m]	10[h]	11[h]	12[h]	13[m]	14[h]	15[h]	16[h]
$i = 2$	17[m]	18[h]	19[h]	20[h]	21[m]	22[h]	23[h]	24[h]
$i = 3$	25[m]	26[h]	27[h]	28[h]	29[m]	30[h]	31[h]	32[h]

Листинг 6.11

```

1     int sumarraycols (int a [M] [N])
2     {
3         int i, j, sum = 0;
4
5         for (j = 0; j < N; j++)
6             for (i = 0; i < M; i++)
7                 sum += a[i][j];
8
9     }

```

В данном случае массив просматривается по столбцам, а не по строкам. При удачном стечении обстоятельств, когда весь массив входит в кэш, тогда коэффициент неудач не превысит 0.25. Однако если размер массива превышает размер кэша (что наиболее вероятно), тогда каждый доступ к $a[i][j]$ будет промахом (табл. 6.27).

Таблица 6.27. Комбинация промахов в зависимости от размера массива

$a[i][j]$	$j = 0$	$j = 1$	$j = 2$	$j = 3$	$j = 4$	$j = 5$	$j = 6$	$j = 7$
$i = 0$	1[m]	5[m]	9[m]	13[m]	17[m]	21[m]	25[m]	29[m]
$i = 1$	2[m]	6[m]	10[m]	14[m]	18[m]	22[m]	26[m]	30[m]
$i = 2$	3[m]	7[m]	11[m]	15[m]	19[m]	23[m]	27[m]	31[m]
$i = 3$	4[m]	8[m]	12[m]	16[m]	20[m]	24[m]	28[m]	32[m]

Более высокие коэффициенты неудач могут оказать значительное влияние на время выполнения. Например, на рассматриваемой гипотетической настольной рабочей станции `sumarraycols` выполняется за 20 циклов синхронизации на одну итерацию, тогда как `sumarrayrows` выполняется за 10 циклов на итерацию. Подводя итоги, можно сделать вывод, что программисты должны осознавать локальность программ и в полной мере использовать ее в процессе написания.

УПРАЖНЕНИЕ 6.14

Транспонирование строк и столбцов матрицы представляет собой серьезную проблему в программных приложениях обработки сигналов и научных вычислений. Оно также интересно с точки зрения локальности, потому что комбинация обращений структурирована как по рядам, так и по столбцам. Рассмотрим, к примеру, следующую процедуру транспонирования:

```

1     typedef int array [2] [2];
2
3     void transpose1 (array dst, array src)
4     {
5         int i, j;
6
7         for (i = 0; i < 2; i++)  {
8             for (j = 0; j < 2; j++)  {
9                 dst [j] [i] = src [i] [j];
10            }
11        }
12    }

```

Предположим, что данный код выполняется на машине со следующими свойствами:

- `sizeof (int) == 4;`
- массив `src` начинается в адресе 0, а массив `dst` — в адресе 16 (десятичном);
- существует один кэш данных L1 с прямым обращением, сквозной и с распределением строк, с размером блока в 8 байтов;
- кэш имеет общий размер в 16 байтов данных и изначально пуст;
- доступы к массивам `src` и `dst` являются единственными источниками промахов считывания и записи.

Укажите для каждого `row` и `col`, является доступ к `src [row] [col]` и `dst [row] [col]` результативным обращением (`h`) или промахом (`m`). Например, считывание `src [0] [0]` — промах, и запись `dst [0] [0]` — также промах.

Массив dst		
	столбец 0	столбец 1
Строка 0	m	
Строка 1		

Массив src		
	столбец 0	столбец 1
Строка 0	m	
Строка 1		

Повторите решение данного упражнения для кэша с 32 байтами данных.

УПРАЖНЕНИЕ 6.15

Ядром игры SimAquarium является плотный цикл, высчитывающий среднее положение 256 водорослей. Оценка производительности кэша оценивается на машине с 1024-байтовым кэшем данных прямого отображения с 16-байтовыми блоками ($B = 16$). Даны следующие определения:

```

1     struct algae_position {
2         int x;
3         int y;
4     };
5
6     struct algae_position grid [16] [16];
7     int total_x = 0, total_y = 0
8     int i, j;
```

Необходимо сделать следующие допущения:

- `sizeof (int) == 4`;
- `grid` (сетка) начинается в адресе ячейки памяти 0;
- изначально кэш пуст;
- единственный доступ к памяти осуществляется к записям массива `grid`. Переменные `i, j, total_x` и `total_y` сохраняются в регистрах.

Определите производительность кэша для следующего кода:

```

1     for (i = 0; i < 16; i++) {
2         for (j = 0; j < 16; j++) {
3             total_x += grid [i] [j] .x;
4         }
5     }
6
7     for (i = 0; i < 16; i++) {
8         for (j = 0; j < 16; j++) {
9             total_y += grid [i] [j] .y;
10        }
11    }
```

1. Каково общее количество считываний?
2. Каково общее количество промахов считываний в кэше?
3. Каков коэффициент неудач?

УПРАЖНЕНИЕ 6.16

С допущениями упр. 6.15 определите производительность кэша следующего кода:

```

1     for (i = 0; i < 16; i++) {
2         for (j = 0; j < 16; j++) {
```

```

3             total_x += grid [j] [i] .x;
4             total_y += grid [j] [i] .y;
5         }
6     }

```

1. Каково общее количество считываний?
2. Каково общее количество промахов считываний в кэше?
3. Каков коэффициент неудач?
4. Каков был бы коэффициент неудач, если бы размер кэша был в два раза больше?

УПРАЖНЕНИЕ 6.17

С допущениями упр. 6.15 определите производительность кэша следующего кода:

```

1     for (i = 0; i < 16; i++) {
2         for (j = 0; j < 16; j++) {
3             total_x += grid [i] [j] .x;
4             total_y += grid [i] [j] .y;
5         }
6     }

```

1. Каково общее количество считываний?
2. Каково общее количество промахов считываний в кэше?
3. Каков коэффициент неудач?
4. Каков был бы коэффициент неудач, если бы размер кэша был в два раза больше?

6.6. Влияние кэша на производительность программ

В данном разделе подводится итог обсуждения иерархии памяти через изучение влияния, оказываемое кэшем на производительность программ, выполняемых на реальных машинах.

6.6.1. Гора памяти

Скорость, с которой программа считывает данные из системы памяти, называется *пропускной способностью считывания* (иногда *полосой считывания*). Если программа считывает n байтов за период времени в s секунд, тогда пропускная способность считывания за этот период составляет n/s , выражаемая обычно в мегабайтах в секунду (Мбайт/с).

Если стоит задача написания программы, которая бы выдавала последовательность запросов считывания из плотного цикла программы, тогда измерение пропускной способности считывания в определенной степени дает понимание производительности системы памяти для конкретной последовательности считывания. В листин-

где 6.12 показана пара функций, измеряющих пропускную способность считывания для определенной последовательности операций считывания.

Листинг 6.12. Функции, измеряющие пропускную способность считывания

```

1 void test (int elems, int stride) /* Функция критерия*/
2 {
3     int i, result = 0;
4     volatile int sink;
5
6     for (i = 0; i < elems; i += stride)
7         result += data [i];
8     sink = result; /* Компилятор не оптимизирует цикл */
9 }
10
11 /* Запуск теста (elems, stride) и возврат пропускной способности
   считывания (Мбайт/с) */
12 double run (int size, int stride, double Mhz)
13 {
14     double cycles
15     int elems = size / sizeof (int);
16
17     test (elems, stride); /* разогрев кэша */
18     cycles = fcyc2 (test, elems, stride, 0); /* вызов теста
   (elems, stride) */
19     return (size / stride) / (cycles / Mhz); /* преобразование циклов
   в Мбайт/с */
20 }
```

Функция `test` генерирует последовательность считывания путем просмотра первых элементов `elems` целочисленного массива с шагом `stride`. Функция `run` — это упаковщик, который вызывает функцию `test` и возвращает измеренную пропускную способность считывания. Функция `fcyc2` в строке 18 (не показана) оценивает время выполнения функции `test` в циклах CPU, использованием схемы измерений по наилучшему коэффициенту K , описанной в главе 9. Обратите внимание, что аргумент `size` функции `run` измеряется в байтах, соответствующий аргумент `elems` функции `test` — в словах. Также в строке 19 10^6 Мбайт/с высчитывается как 10^6 байтов/с, а не 2^{20} байтов/с.

Аргументы `size` и `stride` функции `run` позволяют контролировать степень локальности в результирующей последовательности считывания. Результатом меньших значений `size` является меньший размер рабочего множества и, следовательно, большая временная локальность. Результатом меньших значений `stride` является большая пространственная локальность. Если многократно вызывать функцию `run` с разными значениями `size` и `stride`, тогда можно восстановить двумерную функцию полосы считывания относительно временной и пространственной локальности, называемую *горой памяти*. В листинге 6.13 приведена программа, называемая *горой* (*mountain*), генерирующая гору памяти.

Листинг 6.13. Программа, генерирующая гору памяти

```

1  #include <stdio.h>
2  #include "fcyc2.h" /* Рутинные процедуры синхронизации схемы измерения
по коэф. K*/
3  #include "clock.h" /* Рутинные процедуры доступа к счетчику циклов */
4
5  #define MINBYTES (1 << 10) /* Размер рабочего множества варьируется от 1
Кбайт */
6  #define MAXBYTES (1 << 23) /* Размер рабочего множества варьируется от 8
Мбайт */
7  #define MAXSTRIDE 16 /* Шаги варьируются от 1 до 16 */
8  #define MAXELEMS MAXBYTES /sizeof (int)
9
10 int data [MAXELEMS]; /* Массив для прослеживания */
11
12 int main ()
13 {
14     int size;           /* Размер рабочего множества (в байтах) */
15     int stride;         /* Шаг (в элементах массива) */
16     double Mhz;        /* Тактовая частота */
17
18     init_data (data, MAXELEMS); /* Инициализация каждого элемента
в данных до 1 */
19     Mhz = mhz (0); /* Определение тактовой частоты */
20     for (size = MAXBYTES; size >= MINBYTES; size >>= 1) {
21         for (stride = 1; stride <= MAXSTRIDE; stride++) {
22             printf ("%lf\t", run (size, stride, Mhz));
23         }
24         printf ("\n");
25     }
26     exit (0);
27 }
```

Программа mountain вызывает функцию run с разными размерами и шагами рабочего множества. Размеры рабочего множества начинаются с 1 Кбайт, увеличиваются на коэффициент 2 и достигают максимального размера в 8 Мбайт. Шаги варьируются от 1 до 16. Для каждой комбинации размера и шагов рабочего множества mountain распечатывает пропускную способность считывания в Мбайт/с. Функция mhz в строке 19 (не показана) представляет собой рутинную процедуру, зависимую от системы, вычисляющую тактовую частоту CPU, используя методики, описанные в главе 9.

Каждый компьютер имеет свою собственную уникальную гору памяти, характеризующую возможности системы памяти. Например, на рис. 6.28 показана гора памяти для системы Intel Pentium III Xeon.

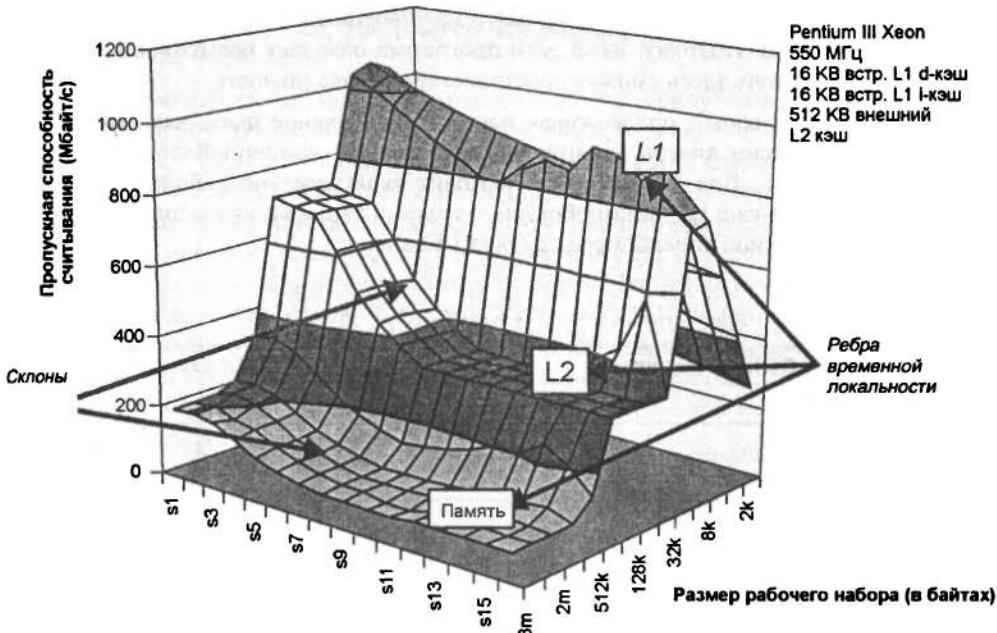


Рис. 6.28. Гора памяти

География горы системы памяти Хеон обнаруживает богатую структуру. Перпендикулярно оси размера расположены три ребра, соответствующие областям временной локальности, где рабочее множество полностью входит в кэш L1, кэш L2 и основную память соответственно. Следует обратить внимание на то, что между самой высокой точкой ребра L1, в которой CPU осуществляет считывание со скоростью 1 Гбайт/с, и самой нижней точкой ребра основной памяти, в которой CPU осуществляет считывание со скоростью 80 Мбайт/с, существует порядок величины.

Две особенности ребра L1 следует отметить особо. Во-первых, для постоянного шага обратите внимание на резкое падение пропускной способности считывания по мере уменьшения размера рабочего множества с 16 до 1 Кбайт (с задней стороны ребра). Во-вторых, для рабочего множества размера 16 Кбайт пик ребра L1 уменьшается с увеличением шага. Поскольку в кэше L1 содержится все рабочее множество, эти особенности не отражают истинной производительности кэша L1. Они являются артефактами служебных непроизводительных сигналов (накладных расходов), возникающих при вызове функции `test` и подготовки к выполнению цикла. Для небольших размеров рабочего множества вдоль ребра L1 эти накладные расходы не амортизируются, поскольку они связаны с более крупными размерами рабочих множеств.

На ребрах L2 и основной памяти имеет скос пространственной локальности, понижающийся с увеличением шага. Этот скос имеет самую крутую точку на ребре L2, из-за крупных абсолютных накладных расходов, которые испытывает кэш L2 при необходимости переноса блоков из основной памяти. Обратите внимание на то, что даже если рабочее множество слишком велико для вмещения в кэши L1 или L2, са-

мая высокая точка на ребре основной памяти на удвоенный показатель выше, нежели самая нижняя точка. Поэтому, даже если программа обладает невысокой временной локальностью, помочь здесь сможет пространственная локальность.

Если сделать "срез горы", поддерживая постоянное значение шага, как показано на рис. 6.29, можно ясно видеть влияние размера кэша и временной локальности на производительность. Для размеров до 16 Кбайт включительно рабочее множество целиком входит в d-кэш L2; таким образом, операции считывания выполняются от L1 на пиковой пропускной способности порядка 1 Гбайт.

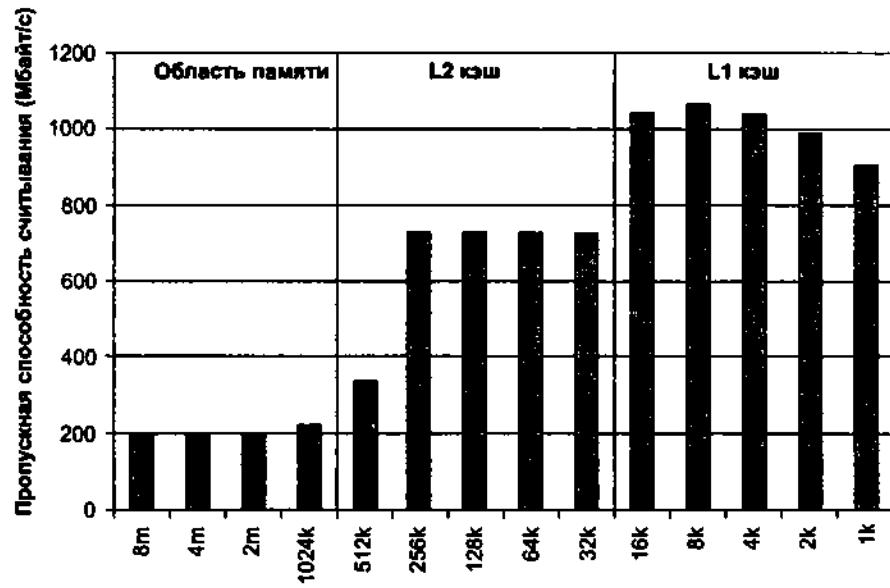


Рис. 6.29. Ребра временной локальности в горе памяти

Для размеров до 256 Кбайт включительно рабочее множество целиком входит в унифицированный кэш L2. Рабочие множества более крупных размеров обслуживаются, главным образом, из основной памяти. С скачок в пропускной способности считывания между 256 и 512 Кбайт довольно интересен. Поскольку размер кэша L2 составляет 512 Кбайт, можно ожидать, что скачок будет иметь место при 512 Кбайт, вместо 256 Кбайт. Единственным способом убедиться в этом будет детальное моделирование кэша, однако авторы подозревают, что истина заключается в том факте, что кэш L2 Pentium III — унифицированный кэш, в котором сохраняются как команды, так и данные. В результате можно наблюдать конфликтные промахи между командами и данными в L2, делающие невозможным размещение целого массива в кэше L2.

Срез горы в обратном направлении с поддержанием постоянного размера рабочего множества обеспечивает определенное понимание влияния пространственной локальности на пропускную способность считывания. Например, на рис. 6.30 показан срез рабочего множества фиксированного размера в 256 Кбайт. Этот срез проходит

вдоль ребра L2 на рис. 6.28, где рабочее множество целиком входит в кэш L2, но слишком велико для кэша L1.

Обратите внимание на то, как неуклонно понижается пропускная способность считывания, по мере повышения шага от 1 до 8 слов. В данной области горы промахи считывания в L1 вызывает перенос блока из L2 в L1. За ним следует некоторое количество результативных обращений на блоке в L1, в зависимости от шага. По мере увеличения шага коэффициент промахов L1 по сравнению с результативными обращениями в L1 увеличивается. Поскольку промахи обслуживаются медленнее результативных обращений, пропускная способность считывания понижается. Как только размер слова достигает 8 слов, что в этой системе равно размеру блока, каждый запрос на считывание в L1 является промахом и должен обслуживаться из кэша L2. Таким образом, пропускная способность считывания для шагов с минимальным размером в 8 слов является постоянной скоростью, определяемой скоростью, с которой блоки кэша могут переноситься из L2 в L1.

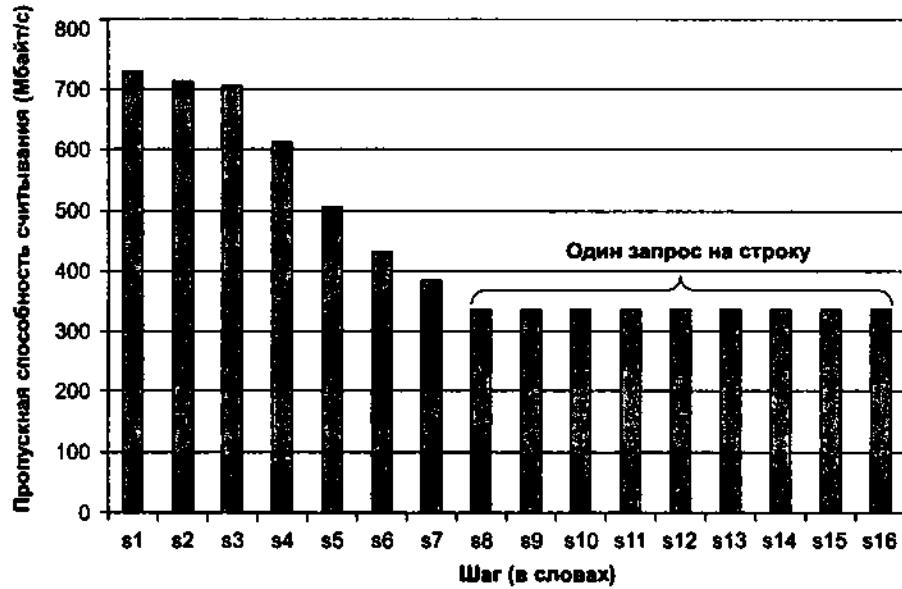


Рис. 6.30. Скос пространственной локальности

Для подведения итогов обсуждения понятия горы памяти можно сделать вывод, что производительность системы памяти не характеризуется единственным числом. Она, скорее, представляет собой "гору" временной и пространственной локальности, повышение которых может варьироваться в соответствии с порядком величины. Высокопрофессиональные программисты стараются структурировать свои программы так, чтобы они выполнялись на пиках горы, а не на ее склонах. Целью является использование временной локальности так, чтобы наиболее часто используемые слова выбирались из кэша L1 пространственной локальности так, чтобы доступ к как можно большему количеству слов осуществлялся из одной строки L1.

УПРАЖНЕНИЕ 6.18

Гора памяти, показанная на рис. 6.28, имеет две оси: шаг и размер рабочего множества. Какая ось соответствует пространственной локальности, а какая — временной?

УПРАЖНЕНИЕ 6.19

Для читателей, считающих себя программистами, придающими особое значение производительности создаваемых программ, важно знать приблизительные значения времени доступа к разным частям иерархии памяти. Используя гору памяти, показанную на рис. 6.28, вычислите время, выраженное в циклах синхронизации CPU, для считывания 4-байтового слова:

- из внутреннего d-кэша L1;
- из внешнего кэша L2;
- из основной памяти.

Сделайте допущение, что пропускная способность считывания при размере 16 Мбайт и шаге 16 составляет 80 Мбайт/с.

6.6.2. Реконфигурация циклов для повышения пространственной локальности

Рассмотрим проблему умножения пары матриц $C = AB$ размера $n \times n$ при $n=2$.

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

где:

$$c_{11} = a_{11} b_{11} + a_{12} b_{21}$$

$$c_{12} = a_{11} b_{12} + a_{12} b_{22}$$

$$c_{21} = a_{21} b_{11} + a_{22} b_{21}$$

$$c_{22} = a_{21} b_{12} + a_{22} b_{22}$$

Функция мультипликации (умножения) матрицы обычно реализуется с использованием трех вложенных циклов, обозначаемых указателями i , j и k . Если поменять циклы местами и внести в код некоторые другие незначительные изменения, тогда можно создать шесть функционально эквивалентных версий умножения матриц (листинги 6.14—6.19). Каждая версия уникально идентифицирована упорядочиванием циклов.

На высоком уровне эти шесть версий очень сходны. Если сложение ассоциативно, тогда каждая версия вычисляет идентичный результат¹. Каждая версия выполняет

¹ В главе 2 обсуждался вопрос о том, что сложение чисел с плавающей точкой — коммутативно, но, по большому счету, не ассоциативно. На практике, если матрицы не смешивают очень большие значения с очень маленькими — как часто случается, когда в матрицах сохраняются физические свойства, — тогда допущение ассоциативности вполне обосновано.

общее количество $O(n^3)$ операций, а также идентичное количество операций сложения и умножения. Каждый из n^2 элементов A и B считывается n раз. Каждый из n^2 элементов C вычисляется суммированием значений n . Однако, если проанализировать поведение итераций самого дальнего цикла, тогда обнаружится, что имеют место различия в количестве доступов и локальности. С целью данного анализа сделаем следующие допущения:

- каждый массив — это массив $n \times n$ чисел двойной точности при `sizeof (double) == 8`;
- существует единичный кэш с размером блока 32 байта ($B = 32$);
- размер массива n настолько большой, что одна строка матрицы не умещается в кэше L1;
- компилятор сохраняет локальные переменные в регистрах, и таким образом обращения к локальным переменным в рамках циклов не требуют ни команд загрузки, ни сохранения.

Листинг 6.15. Умножение матриц

```

1   for (i = 0; i < n; i++)
2       for (j = 0; j < n; j++) {
3           sum = 0.0;
4           for (k = 0; k < n; k++)
5               sum += A[i] [k] * B [k] [j];
6           C [i] [j] += sum;
7       }

```

Листинг 6.16. Умножение матриц

```

1   for (j = 0; j < n; j++)
2       for (i = 0; i < n; i++) {
3           sum = 0.0;
4           for (k = 0; k < n; k++)
5               sum += A[i] [k] * B [k] [j];
6           C [i] [j] += sum;
7       }

```

Листинг 6.17. Умножение матриц

```

1   for (j = 0; j < n; j++)
2       for (k = 0; k < n; k++) {
3           r = B [k] [j];
4           for (i = 0; i < n; i++)
5               C [i] [j] += A[i] [k] * r;
6       }

```

Листинг 6.17. Умножение матриц $A \cdot B$

```

1   for (k = 0; k < n; k++)
2       for (j = 0; j < n; j++) {
3           r = B [k] [j];
4           for (i = 0; i < n; i++)
5               C [i] [j] += A[i] [k] * r;
6       }

```

Листинг 6.18. Умножение матриц $A \cdot B$

```

1   for (k = 0; k < n; k++)
2       for (i = 0; i < n; i++) {
3           r = A [i] [k];
4           for (j = 0; j < n; j++)
5               C [i] [j] += r*B[k] [j];
6       }

```

Листинг 6.19. Умножение матриц $A \cdot B$

```

1   for (i = 0; i < n; i++)
2       for (k = 0; k < n; k++) {
3           r = A [i] [k];
4           for (j = 0; j < n; j++)
5               C [i] [j] += r*B[k] [j];
6       }

```

В табл. 6.28 показаны общие результаты анализа внутренних циклов. Обратите внимание на то, что шесть версий сводятся в три эквивалентных класса, обозначаемых парой матриц, к которым во внутреннем цикле осуществляется доступ. Например, версии ijk и jik являются членами класса AB , потому что они обращаются к массивам A и B (но не к C) в самом дальнем цикле. Для каждого класса подсчитано количество операций загрузки (считывания) и сохранения (записи) в каждой итерации внутреннего цикла, количество обращений к A , B и C , которые будут промахами кэша в каждой итерации цикла, а также общее число промахов кэша на каждую итерацию.

Таблица 6.28. Анализ мультигликации матрицы внутренних циклов

Мультигликация матрицы (класс)	Загр. на итер.	Сохран. на итер.	Промахов А на итер.	Промахов В на итер.	Промахов С на итер.	Всего промахов на итер.
ijk & jik (AB)	2	0	0.25	1.00	0.00	1.25
jki & kji (AC)	2	1	1.00	0.00	1.00	2.00
kij & ikj (BC)	2	1	0.00	0.25	0.25	0.50

Внутренние циклы рутинных процедур класса *AB* (листинги 6.14 и 6.15) просматривают строку массива *A* с шагом 1. Поскольку в каждом блоке кэша содержится четыре двойных слова, тогда коэффициент неудач для *A* составляет 0.25 промаха на итерацию. С другой стороны, внутренний цикл просматривает столбец *B* с шагом *n*. Поскольку *n* — большое число, результатом каждого доступа к массиву *B* становится промах (всего 1.25 промахов на итерацию).

Внутренние циклы в рутинных процедурах класса *AC* (листинги 6.16 и 6.17) демонстрируют определенные проблемы. Каждая итерация выполняет две загрузки и сохранение (в отличие от рутинных процедур класса *AB*, выполняющих 2 операции загрузки без сохранений). Внутренний цикл просматривает столбцы *A* и *C* с шагом *n*. В результате получается промах при каждой операции загрузки, что в итоге дает 2 промаха на каждую итерацию. Обратите внимание на то, что перестановка циклов сократила объем пространственной локальности (по сравнению с рутинными операциями класса *AB*).

Рутинные процедуры *BC* (листинги 6.18 и 6.19) представляют интересный компромисс: с двумя операциями загрузки и сохранением они, в отличие от рутинных процедур *AB*, требуют еще одну операцию с памятью. С другой стороны, поскольку внутренний цикл просматривает по строкам *B* и *C* с комбинацией доступа в 1 шаг, то коэффициент неудач на каждый массив составляет всего 0.25 промахов на итерацию, что в целом составляет 0.50 промахов на итерацию.

На рис. 6.31 представлена общая производительность различных версий мультилипликации матриц на системе Pentium III Xeon. На графике схематически показано измеренное количество циклов синхронизации CPU на каждую итерацию внутреннего цикла в виде функции размера массива (*n*).

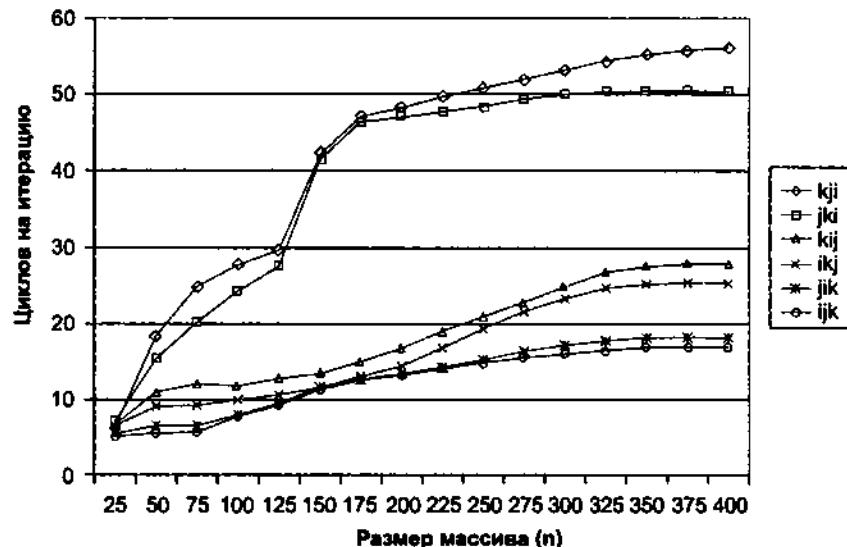


Рис. 6.31. Производительность мультилипликации матрицы Pentium III Xeon

В данном графе имеется несколько интересных моментов, заслуживающих внимания.

- Для больших значений n самая быстродействующая версия работает в три раза быстрее самой медленнодействующей версии, несмотря на то, что каждая выполняет одно и то же количество арифметических операций с числами с плавающей точкой.
- Версии с одинаковым количеством и локальностью доступов к памяти демонстрируют приблизительно одинаковую производительность.
- Две версии с наихудшим поведением памяти в том, что касается количества доступов и промахов на каждую итерацию, выполняются значительно медленнее, нежели остальные четыре версии, имеющие меньше промахов, меньше доступов, либо того и другого.
- Рутинные процедуры класса AB — 2 доступа к памяти и 1.25 промахов на итерацию демонстрируют несколько более высокую производительность на данной машине, чем рутинные процедуры класса BC — 3 доступа к памяти и 0.5 промахов на итерацию, в результате чего образуется дополнительное обращение к памяти для более низкого коэффициента неудач. Идея заключается в том, что в отношении общей производительности промахи кэша — это не единственная проблема. Здесь также значение имеет количество доступов к памяти, и во многих случаях получение наилучшей производительности сводится к поиску и нахождению компромиссного решения между двумя аспектами (промахи и результативные обращения). Более детально данный вопрос рассматривается в упр. 6.32 и 6.33.

6.6.3. Использование блокирования для повышения временной локальности

В последнем разделе читатели видели, как простая перестановка циклов смогла повысить пространственную локальность. Однако стоит отметить, что даже с хорошим расположением циклов время на выполнение итерации цикла увеличивается с увеличением размера массива. При увеличении размера массива происходит уменьшение временной локальности, и в кэше имеет место увеличение количества промахов емкости. Для устранения этой проблемы можно воспользоваться известной методикой блокирования. Однако стоит отметить, что, в отличие от простых преобразований циклов для повышения пространственной локальности, блокирование затрудняет читаемость и прозрачность кода. По этой причине оно лучше всего подходит для оптимизирующих компиляторов или часто выполняемых библиотечных процедур. Несмотря на это, данная методика представляет интерес для изучения и понимания, поскольку является общей концепцией, которая может обеспечить значительный выигрыш производительности.

Общей идеей блокирования является организация структур данных в программе в большие куски, называемые блоками. (В данном контексте "блок" относится к куску данных уровня приложения, а не к блоку кэш-памяти.) Программа структурирована так, что она загружает кусок в кэш L1, выполняет все необходимые для его обработ-

ки операции считывания и записи, после чего просто отбрасывает этот кусок, загружает следующий и т. д.

Блокирование процедуры мультиплексирования матрицы осуществляется разбиением матриц на матрицы меньшего ранга, после чего используется математический факт: с этими подматрицами можно обращаться, как со скалярами. Например, если $n = 8$, тогда каждую матрицу можно разделить на 4×4 матриц, где

$$C_{11} = A_{11} B_{11} + A_{12} B_{21}$$

$$C_{12} = A_{11} B_{12} + A_{12} B_{22}$$

$$C_{21} = A_{21} B_{11} + A_{22} B_{21}$$

$$C_{22} = A_{21} B_{12} + A_{22} B_{22}$$

Листинг 6.20. Блокированное мультипликация матриц

```

1 void bijk(array A, array B, array C, int n, int bsize)
2 {
3     int i, j, k, kk, jj;
4     double sum;
5     int en = bsize * (n/bsize); /* Объем, точно входящий в блоки */
6
7     for (i = 0; i < n; i++)
8         for (j = 0; j < n; j++)
9             C[i][j] = 0.0;
10
11    for (kk = 0; kk < en; kk += bsize) {
12        for (jj = 0; jj < en; jj += bsize) {
13            for (i = 0; i < n; i++)
14                for (j = jj; j < jj + bsize; j++) {
15                    sum = C[i][j];
16                    for (k = kk; k < kk + bsize; k++) {
17                        sum += A[i][k] * B[k][j];
18                    }
19                    C[i][j] = sum;
20                }
21            }
22        }
23    }
24 }
```

В листинге 6.20 показана одна версия мультипликации блочной матрицы, называемая авторами версией *bijk*. Основной идеей данного кода является разбиение A и C на пряди строки $1 \times bsize$ и разбиение B на блоки $bsize \times bsize$. Самая дальняя (j, k) пара цикла умножает прядь A на блок B и аккумулирует результат в пряди C . Цикл i итерирует через n прядей строки A и C , используя тот же блок в B .

На рис. 6.32 представлена графическая интерпретация кода листинга 6.20. Ключевой идеей является то, что программа загружает блок B в кэш, использует его, после чего отбрасывает. Обращения к A демонстрируют хорошую пространственную локальность из-за того, что доступ к каждой пряди осуществляется по шагу 1. Временная локальность здесь также хорошая потому, что обращение ко всей пряди осуществляется последовательно $bsize$ раз. Обращения к B также демонстрируют хорошую временную локальность, потому что доступ ко всему блоку $bsize \times bsize$ осуществляется последовательно n раз. И наконец, обращения к C обладают хорошей пространственной локальностью, потому что каждый элемент пряди записан последовательно один за другим. Обратите внимание на то, что обращения к C не обладают хорошей временной локальностью, потому что к каждой пряди доступ осуществляется только один раз.

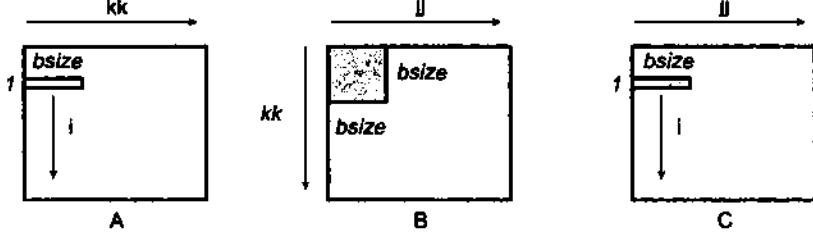


Рис. 6.32. Графическое представление мультиплексирования блочного матрицы

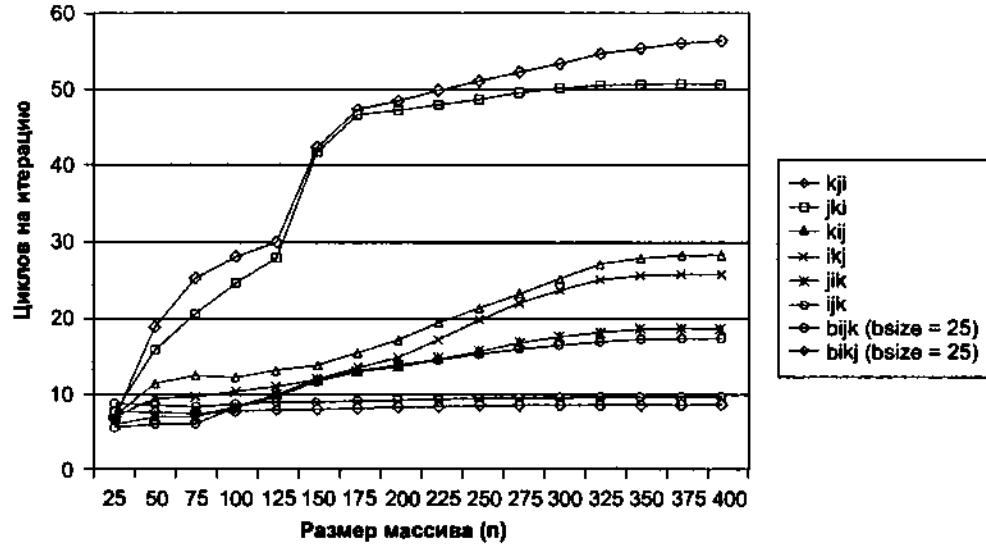


Рис. 6.33. Производительность мультипликации матрицы Pentium III Xeon

Блокирование может усложнить читабельность кода, однако оно вносит значительный вклад в повышение производительности. На рис. 6.33 показана производитель-

ность двух версий мультиплексирования блочной матрицы в системе Pentium III Хеоп ($bsize = 25$). Обратите внимание, что блокирование сокращает время выполнения вдвое (с 20 циклов синхронизации на итерацию до 10 циклов на итерацию). Другим интересным моментом блокирования является то, что время, затрачиваемое на одну итерацию, с увеличением размера массива остается практически неизменным. Для небольших размеров массива дополнительные накладные расходы (непроизводительные сигналы) в блочной версии замедляют ее выполнение. В области $n = 100$ существует точка пересечения, после которой блочная версия выполняется быстрее.

Устройства кэш-памяти и рабочие нагрузки потоковых сред

В настоящее время особую важность приобретают программные приложения, обрабатывающие мультимедийные данные в реальном режиме времени. В эти программы данные поступают непрерывным потоком с определенного устройства ввода: микрофона, фотоаппарата или сети (см. главу 12). Данные поступают, обрабатываются, отправляются на устройство вывода и вскоре сбрасываются, освобождая место для вновь поступающих данных. Насколько подготовлена иерархия памяти для подобных нагрузок потоковыми средами? Поскольку вся информация обрабатывается последовательно, по мере ее поступления, из пространственной локальности можно извлечь определенную выгоду, как получилось в примере разд. 6.6 с мультиплексированием матрицы. Однако, поскольку данные обрабатываются только один раз, после чего сбрасываются, объем временной локальности ограничен.

Для решения этой проблемы системные разработчики и создатели компиляторов придерживаются стратегии, называемой предварительной выборкой. Ее смысл — скрыть время ожидания промахов кэша путем предположения того, доступ к каким блокам будет осуществлен в ближайшем будущем, с последующей выборкой этих блоков заранее в кэш с помощью специальных машинных команд. Если предварительная выборка выполнена идеально, тогда каждый блок копируется в кэш непосредственно перед обращением к нему программы, и таким образом результатом каждой команды загрузки будет результативное обращение в кэш. Впрочем, предварительная выборка связана с рисками. Поскольку трафик предварительной выборки использует шину совместно с трафиком DMA, проходящим потоком от устройства ввода-вывода в основную память, то слишком большая предварительная выборка мешает трафику DMA и замедляет общее быстродействие системы. Другой потенциальной проблемой является то, что каждый предварительно выбранный блок кэша должен вытеснять существующий блок. При слишком большой предварительной выборке пользователь рискует "загрязнить" кэш вытеснением ранее выбранного блока, к которому программе еще предстоит обратиться.

6.7. Использование локальности в программах

Как уже отмечалось, система памяти структурирована в виде иерархии записывающих устройств с быстродействующими устройствами меньшего размера в верхней части и медленнодействующими устройствами — в нижней. Из-за такой иерархии

эффективная скорость, с которой программа может осуществлять доступ к ячейкам памяти, не характеризуется одним числом. Это, скорее, чрезвычайно переменчивая функция локальности программы (названная горой памяти), которая может изменяться по порядкам величины. Программы с хорошей локальностью осуществляют доступ к большей части данных из кэш-памяти L1 и L2. Программы с плохой локальностью осуществляют доступ к большей части данных из сравнительно медленнодействующей основной памяти DRAM.

- Сосредоточьтесь на внутренних циклах, где осуществляется большая часть вычислений и доступов к памяти.
- Попытайтесь добиться максимальной пространственной локальности программ последовательным считыванием объекта данных, в том порядке, в котором они сохранены в памяти.
- Попытайтесь добиться максимальной временной локальности программ максимально частым использованием объекта данных после того, как он считан из памяти.
- Помните, что коэффициенты неудач — единственный фактор (но немаловажный!), определяющий производительность кода. Количество доступов к памяти также играет важную роль, и иногда необходимо делать между ними выбор.

6.8. Резюме

Основные технологии сохранения — оперативные запоминающие устройства (RAM), энергонезависимые записывающие устройства (ROM) и диски. RAM поставляется в двух формах. Статическая память RAM (SRAM) — более быстродействующая и дорогая, используется для кэш-памяти на чипе центрального процессора и вне его. Динамическая память RAM (DRAM) — медленнодействующее и более дешевое устройство, используемое для основной памяти и графических кадровых буферов. Энергонезависимые записывающие устройства, также называемые постоянными (ROM), сохраняют информацию даже при отключении питания, и они используются для хранения "зашитых" программ. Диски являются энергонезависимыми записывающими устройствами, сохраняющими огромные объемы данных с меньшими затратами на бит. Компромиссом является увеличение времени доступа.

Быстродействующие технологии сохранения требуют больше затрат на бит и имеют меньшую емкость. Свойства "цена-качество" этих технологий изменяются с очень разными скоростями. В частности, значения времени доступа к DRAM и диску отличаются от времени выполнения циклов CPU. Системы сокращают эти промежутки путем структурирования памяти в иерархию записывающих устройств, где меньшие по размеру и быстродействующие устройства располагаются в верхней части, а медленнодействующие — внизу. По причине того, что грамотно написанные программы обладают хорошей локальностью, большая часть данных обслуживается с верхних уровней, результатом чего является система памяти, работающая со скоростью выполнения верхних уровней, но с затратами и емкостью более низких уровней.

Программисты могут значительно увеличить время выполнения программ написанием их с хорошей пространственной и временной локальностью. Здесь крайне важное

значение имеет использование кэш-памяти на основе SRAM. Программы, выбирающие данные в основном из кэш L1, могут выполняться на порядок быстрее, чем программы, выбирающие данные из памяти.

Библиографические примечания

Технологии памяти и дисков очень быстро совершенствуются. По опыту известно, что лучшими источниками технической информации являются Web-сайты, обслуживаемые производителями. Такие компании, как Micron, Toshiba, Hyundai, Samsung, Hitachi и Kingston Technology, предоставляют очень большие объемы технической информации о записывающих устройствах. Сайты IBM, Maxtor и Seagate предоставляют также полезную информацию о дисках.

В учебниках по логическому проектированию и проектированию цепей представлена подробная информация о технологии организации системы памяти [39, 62]. IEEE Spectrum опубликовал серию исследовательских статей по DRAM [36]. Международный симпозиум по компьютерной архитектуре (ISCA) — самый известный форум, на котором изучаются характеристики производительности записывающих устройств DRAM [22, 23].

Уилкис (Wilkes) написал первый документ, посвященный устройствам кэш-памяти [87]. Пржибильский (Przybylski) написал авторитетную книгу по проектированию кэш [59]. В работах Хеннеси (Hennesy) и Паттерсона (Patterson) также представлены подробные материалы по вопросам проектирования кэшей.

Стрикер (Stricker) ввел идею горы памяти в качестве исчерпывающей характеристики системы памяти в [82], а также предложил использовать собственно термин "гора памяти" в последующих выпусках работы. Разработчики компиляторов работают над повышением локальности путем автоматического выполнения типов ручного преобразования кодов, рассмотренных в разд. 6.6 [14, 25, 45, 48, 54, 60, 89]. Картер (Carter) с коллегами предложил контроллер памяти с распознаванием кэша [11]. Сьюард (Seward) разработал профайлер кэша открытого источника, называемый cacheprof, который характеризует поведение промахов программ на языке C в произвольном смоделированном кэше (адрес www.cacheprof.org).

Существует очень большое количество литературы по построению и использованию дисковых накопителей. Многие исследователи ОЗУ ищут способы объединения отдельных дисков в более крупные, мощные и безопасные пулы памяти [12, 28, 29, 57, 90]. Другие ищут способы использования кэш и локальности для повышения эффективности доступа к дискам [6, 13]. Такие системы, как Exokernel, обеспечивают повышенное управление диском и ресурсами памяти на пользовательском уровне [38]. Такие системы, как Andrew File System [53] и Coda [67], распространяют иерархию памяти по компьютерным сетям и переносным компьютерам (ноутбукам). Шиндлер (Schindler) и Гэнгер (Ganger) разработали интересное инструментальное средство, которое автоматически характеризует геометрию и производительность дисковых накопителей SCSI [68].

Задачи для домашнего решения

УПРАЖНЕНИЕ 6.20 ◆◆

Предположим, что получено задание на проектирование дискеты, где количество битов на дорожку постоянно. Разработчик знает, что количество битов, приходящееся на каждую дорожку, определяется окружностью самой внутренней дорожки, которую можно также принять за окружность "дырки". Таким образом, если дырку в центре дискеты сделать больше, тогда количество битов, приходящееся на каждую дорожку, увеличивается, однако общее количество дорожек уменьшится. Если принять, что r — это радиус тарелки, а $x \times r$ — радиус дырки, то какое значение x сделает емкость дискеты максимальной?

УПРАЖНЕНИЕ 6.21 ◆

В приведенной таблице представлены параметры нескольких разных кэш. Задайте для каждого кэша число множеств кэша S , теговых битов t , битов указателя множества s и битов сдвига блока b .

Кэш	m	C	B	E	S	t	s	b
1	32	1024	4	4				
2	32	1024	4	256				
3	32	1024	8	1				
4	32	1024	8	128				
5	32	1024	32	1				
6	32	1024	32	4				

УПРАЖНЕНИЕ 6.22 ◆

Данная задача относится к кэшу, описанному в упр. 6.9.

- Перечислите все шестнадцатеричные адреса ячеек памяти, которые во множестве 1 будут иметь результативное обращение в кэш.
- Перечислите все шестнадцатеричные адреса ячеек памяти, которые во множестве 6 будут иметь результативное обращение в кэш.

УПРАЖНЕНИЕ 6.23 ◆◆

Рассмотрите следующую процедуру транспозиции матрицы:

```

1     typedef int array [4] [4];
2
3     void transpose2 (array dst, array src)
4     {
5         int i, j;
6
7         for (i = 0; i < 4; i++)
8             for (j = 0; j < 4; j++)
9                 dst[i][j] = src[j][i];
10    }
11
12    void main()
13    {
14        array a[4][4], b[4][4];
15
16        for (int i = 0; i < 4; i++)
17            for (int j = 0; j < 4; j++)
18                a[i][j] = i * 4 + j;
19
20        for (int i = 0; i < 4; i++)
21            for (int j = 0; j < 4; j++)
22                b[i][j] = 0;
23
24        transpose2 (b, a);
25
26        for (int i = 0; i < 4; i++)
27            for (int j = 0; j < 4; j++)
28                cout << b[i][j] << " ";
29    }

```

```

7         for (i = 0; i < 4; i++) {
8             for (j = 0; j < 4; j++) {
9                 dst [j] [i] = src [i] [j];
10            }
11        }
12    )

```

Предположим, что этот код выполняется на машине со следующими свойствами:

- `sizeof (int) == 4;`
- массив `src` начинается в адресе 0, а массив `dst` — в адресе 64 (десятичном);
- существует один кэш данных `L1` — с прямым отображением, сквозной, с выделением строк, с размером блока 16 байтов;
- общий размер кэша — 32 байта данных; изначально данный кэш пуст;
- массивы `src` и `dst` — единственные источники считывания и записи промахов соответственно.

Укажите, будет доступ к `src [row] [col]` и `dst [row] [col]` для каждого `row` и `col` результативным обращением (`h`) или промахом (`m`). Например, считывание `src [0] [0]` — промах, и запись `dst [0] [0]` — также промах кэша.

Массив `dst`

	столбец 0	столбец 1	столбец 2	столбец 3
строка 0	m			
строка 1				
строка 2				
строка 3				

Массив `src`

	столбец 0	столбец 1	столбец 2	столбец 3
строка 0	m			
строка 1				
строка 2				
строка 3				

УПРАЖНЕНИЕ 6.24 ++

Повторите упр. 6.23 для кэша с общим размером в 128 бит данных.

Массив dst				
	столбец 0	столбец 1	столбец 2	столбец 3
строка 0	m			
строка 1				
строка 2				
строка 3				

Массив src				
	столбец 0	столбец 1	столбец 2	столбец 3
строка 0	m			
строка 1				
строка 2				
строка 3				

УПРАЖНЕНИЕ 6.25 +

Компания 3М принимает решение сделать пояснительные заметки путем печати желтых квадратов на белых листах бумаги. В процессе печати необходимо задать значения CMYK (голубой (cyan), пурпурный (magenta), желтый (yellow), черный (black)) для каждой точки квадрата. Нанят сотрудник для определения эффективности следующих алгоритмов на машине с 2048-байтовом кэшем данных прямого отображения с 32-байтовыми блоками. Даны следующие определения:

```

1   struct point_color {
2       int c;
3       int m;
4       int y;
5       int k;
6   };
7
8   struct point_color square [16] {16};
9   int i, j;
```

Допустим следующее:

- sizeof (int) == 4;
- square начинается в адресе памяти 0;

- изначально кэш пуст;
- единственные доступы к памяти — к записям массива square. Переменные i и j сохраняются в регистрах.

Определите производительность кэша следующего кода:

```

1     for (i = 0; i < 16; i++) {
2         for (j = 0; j < 16; j++) {
3             square [i] [j].c = 0;
4             square [i] [j].m = 0;
5             square [i] [j].y = 1;
6             square [i] [j].k = 0;
7         }
8     }

```

1. Каково общее количество записей?
2. Каково общее количество записей-промахов в кэше?
3. Каков коэффициент неудач?

УПРАЖНЕНИЕ 6.26 ♦

При допущениях упр. 6.25 определите производительность кэша следующего кода:

```

1     for (i = 0; i < 16; i++) {
2         for (j = 0; j < 16; j++) {
3             square [i] [j].c = 0;
4             square [i] [j].m = 0;
5             square [i] [j].y = 1;
6             square [i] [j].k = 0;
7         }
8     }

```

1. Каково общее количество записей?
2. Каково общее количество записей-промахов в кэше?
3. Каков коэффициент неудач?

УПРАЖНЕНИЕ 6.27 ♦

При допущениях упр. 6.25 определите производительность кэша следующего кода:

```

1     for (i = 0; i < 16; i++) {
2         for (j = 0; j < 16; j++) {
3             square [i] [j].y = 1;
4         }
5     }
6     for (i = 0; i < 16; i++) {
7         for (j = 0; j < 16; j++) {

```

```

8             square [i] [j].c = 0;
9             square [i] [j].m = 0;
10            square [i] [j].k = 0;
11         }
12     }

```

1. Каково общее количество записей?
2. Каково общее количество записей-промахов в кэше?
3. Каков коэффициент неудач?

УПРАЖНЕНИЕ 6.28 *

Некий программист пишет новую трехмерную игру в надежде прославиться и неплохо заработать. В данный момент он работает над функцией скрытия буфера дисплея перед тем, как нарисовать следующий кадр. Экран, над которым происходит работа, представляет собой массив 640×480 пикселов. Машина, на которой выполняется работа, имеет 64 Кбайт кэш прямого отображения с 4-байтовыми строками. Используемые структуры C:

```

1   struct pixel {
2       char r;
3       char g;
4       char b;
5       char a;
6   };
7
8   struct pixel buffer [480] [640];
9   int i, j;
10  char *cptr;
11  int *iptr;

```

Допустим следующее:

- `sizeof (int) == 4;`
- `buffer` начинается в адресе памяти 0;
- изначально кэш пуст;
- единственные доступы к памяти — к записям массива `buffer`. Переменные `i`, `j`, `cptr` и `iptr` сохраняются в регистрах.

Какой процент операций записи в следующем коде составят промахи кэша?

```

1   for (j = 0; j < 640; j++) {
2       for (i = 0; i < 480; i++) {
3           buffer [i] [j].r = 0;
4           buffer [i] [j].g = 0;
5           buffer [i] [j].b = 0;
6           buffer [i] [j].a = 0;
7       }
8   }

```

УПРАЖНЕНИЕ 6.29 ◆◆

При допущениях упр. 6.28, какой процент операций записи в следующем коде составят промахи кэша?

```
1     char *cptr = (char *) buffer;
2     for (; cptr < ((char *) buffer) + 640 * 480 * 4; cptr++)
3         *cptr = 0;
```

УПРАЖНЕНИЕ 6.30 ◆◆

При допущениях упр. 6.28, какой процент операций записи в следующем коде составят промахи кэша?

```
1     int *iptr = (int *) buffer;
2     for (; iptr < ((int *) buffer) + 640 * 480; iptr++)
3         *iptr = 0;
```

УПРАЖНЕНИЕ 6.31 ◆◆◆

Загрузите программу `mountain` с Web-сайта CS:APP и запустите ее в любимой системе PC/Linux. Воспользуйтесь результатами для оценки размера кэшей L1 и L2 в системе.

УПРАЖНЕНИЕ 6.32 ◆◆◆◆

В данном задании будут применяться концепции, изученные в главах 5 и 6, для решения проблемы оптимизации кода для программного приложения с интенсивной системой памяти. Рассмотрим процедуру для копирования и транспозиции элементов матрицы $N \times N$ типа `int`. То есть, для исходной матрицы S и матрицы назначения D необходимо скопировать каждый элемент s_{ij} в d_{ji} . Этот код можно записать простым циклом:

```
1     void transpose (int *dst, int *src, int dim)
2     {
3         int i, j;
4
5         for (i = 0; i < dim; i++)
6             for (j = 0; j < dim; j++)
7                 dst [j*dim + i] = src [i*dim + j];
8     }
```

где аргументы процедуры являются указателями места назначения `dst` и исходной матрицы `src`, а также матрицы размером N (`dim`). Чтобы ускорить выполнение данного кода, необходимы два типа оптимизации. Во-первых, несмотря на то, что данная процедура прекрасно использует пространственную локальность исходной матрицы, она не подходит для больших значений N , относящихся к матрице назначения. Во-вторых, код, генерированный GCC, не очень эффективен. При рассмотрении компонующего автокода видно, что внутренний цикл требует 10 команд, 5 из которых об-

рашаются к памяти: одна для источника, другая — для назначения, а три — для считывания локальных переменных из стека. Работа заключается в рассмотрении данных проблем и разработке процедуры транспозиции, выполняемой с максимально возможной скоростью.

УПРАЖНЕНИЕ 6.33 ***

Данная задача представляет собой интригующую вариацию упр. 6.32. Рассмотрим проблему преобразования ориентированного графа g в его неориентированный аналог g' . Граф g' имеет ребро от вершины i к вершине v , если существует ребро от i к v или от v к i в графе-оригинале g . Граф g представлен матрицей G следующим образом. Если N — число вершин в g , тогда G — матрица $N \times N$, и ее записи все равны либо 0, либо 1. Предположим, что вершины g поименованы как $v_0, v_1, v_2, \dots, v_{N-1}$. Тогда $G(i, j) = 1$, если существует ребро от v_i к v_j , и 0 — в противном случае. Обратите внимание, что элементы на диагонали матрицы G всегда равны 1, а матрицы неориентированного графа симметричны. Данный код можно записать в виде простого цикла:

```

1     void col_convert (int    *G, int dim)  {
2             int i, j
3
4             for (i = 0; i < dim; i++)
5                 for (j = 0; j < dim; j++)
6                     G[j*dim + i] = G[j*dim + i] + G[i *dim + j];
7     }

```

Задача заключается в разработке процедуры преобразования, которая выполнялась бы с максимальной скоростью. Как и ранее, для нахождения адекватного решения потребуется применение концепций, изученных в главах 5 и 6.

Решение упражнений

РЕШЕНИЕ УПРАЖНЕНИЯ 6.1

Идея заключается в минимизации числа адресных битов путем минимизации характеристического отношения $\max(r, c)/\min(r, c)$. Другими словами, чем квадратнее массив, тем меньше адресных битов.

Организация	r	c	b_r	b_c	$\max(b_r, b_c)$
16×1	4	4	2	2	2
16×4	4	4	2	2	2
128×8	16	8	4	3	4
512×4	32	16	5	4	5
1024×4	32	32	5	5	5

РЕШЕНИЕ УПРАЖНЕНИЯ 6.2

Целью данного упражнения является закрепление понимания взаимосвязи между цилиндрами и дорожками. Когда понимание будет доскональным, подключайтесь и — вперед:

$$\text{емкость} = 512 \text{ байтов} \times 400 \text{ секторов} \times 10000 \text{ дорожек} \times 2 \text{ поверхности} \times 2 \text{ тарелки} \\ = 8192 \, 000 \, 000 \text{ байтов} = 8192 \text{ Гбайт.}$$

РЕШЕНИЕ УПРАЖНЕНИЯ 6.3

Решение этой упражнения является непосредственным приложением формулы для времени доступа к диску. Средняя задержка из-за вращения диска (в мс) составляет:

$$T_{\text{avg rotation}} = 0.5 \times T_{\text{max rotation}} \\ = 0.5 \times (60 \text{ с}/15 \, 000 \text{ об}/\text{мин}) \times 1000 \text{ мс}/\text{с} \\ \approx 2 \text{ мс}$$

Среднее время передачи составляет:

$$T_{\text{avg transfer}} = (60 \text{ с}/15 \, 000 \text{ об}/\text{мин}) \times 1/500 \text{ секторов}/\text{дорожка} \times 1000 \text{ мс}/\text{с} \\ \approx 0.008 \text{ мс}$$

Если сложить все вместе, то общее вычисленное время доступа составляет

$$T_{\text{access}} = T_{\text{avg seek}} + T_{\text{avg rotation}} + T_{\text{avg transfer}} \\ = 8 \text{ мс} + 2 \text{ мс} + 0.008 \text{ мс} \\ \approx 10 \text{ мс}$$

РЕШЕНИЕ УПРАЖНЕНИЯ 6.4

Для создания комбинации обращения по шагу 1 необходимо переставить местами циклы так, чтобы показатели, находящиеся в крайнем положении справа, менялись наиболее быстро.

```

1     int sumarray3d (int a [N] [N] [N])
2     {
3         int i, j, k, sum = 0;
4
5         for (k = 0; k < N; k++) {
6             for (i = 0; i < N; i++) {
7                 for (j = 0; j < N; j++) {
8                     sum += a[k] [i] [j];
9                 }
10            }
11        }
12        return sum;
13    }
```

Это — очень важная идея. Необходимо понимать, почему результатом именно такой перестановки циклов становится комбинация по шагу 1.

РЕШЕНИЕ УПРАЖНЕНИЯ 6.5

Ключом к решению данного упражнения является отчетливое представление того, как массив расположен в памяти, и анализ комбинации обращения. Функция `clear1` осуществляет доступ к массиву, используя комбинацию обращения по шагу 1, следовательно, имеет наилучшую пространственную локальность. Функция `clear2` надлежащим образом просматривает каждую из N структур, что само по себе неплохо, однако она "перепрыгивает" комбинацию без шага 1 на следующих сдвигах с начала структуры: 0, 12, 4, 16, 8, 20. Поэтому `clear2` имеет более низкую пространственную локальность, чем `clear1`. Функция `clear3` "прыгает" не только в рамках каждой структуры, но и между структурами. Поэтому `clear3` демонстрирует меньшую пространственную локальность, нежели `clear1` и `clear2`.

РЕШЕНИЕ УПРАЖНЕНИЯ 6.6

Решение — непосредственное применение определений различных параметров кэша, показанных в табл. 6.14–6.15. Задача может показаться не очень приятной, но для истинного понимания принципов работы кэш сначала необходимо понять, как организация кэш влечет образование сегментов в адресных битах.

	m	C	B	E	S	t	s	b
1	32	1024	4	1	256	22	8	2
2	32	1024	8	4	32	24	5	3
3	32	1024	32	32	1	27	0	5

РЕШЕНИЕ УПРАЖНЕНИЯ 6.7

Заполнение устраняет конфликтные промахи. Таким образом, три четверти обращений являются результативными.

РЕШЕНИЕ УПРАЖНЕНИЯ 6.8

Иногда понимание того, почему та или иная идея себя не оправдывает, помогает понять, почему какая-либо альтернативная идея будет удачной. В данном случае рассматриваемой неудачной идеей является индексирование кэша битами старших разрядов, а не срединными.

- При индексировании битами старших разрядов каждый смежный массив состоит из 2^t блоков, где t — число теговых битов. Таким образом, первые 2^t смежных блоков массива отобразятся на множество 0, следующие 2^t блоков — на множество 1 и т. д.
- Для кэша прямого отображения, где $(S, E, B, m) = (512, 1, 32, 32)$, емкость кэша составит 512 32-байтовых блоков, где имеются $t = 18$ теговых битов в каждой

строке кэша. Таким образом, первые 2^{18} блоков в массиве отобразятся на множество 0, следующие 2^{18} блоков — на множество 1. Поскольку массив состоит только из $4096/32 = 512$ блоков, все блоки массива отображаются на множество 0. Следовательно, кэш будет удерживать максимум один блок массива в любой момент времени, несмотря на то, что массив слишком мал для того, чтобы целиком войти в кэш. Понятно, что использование индексирования битами старших разрядов ухудшает работу кэша.

РЕШЕНИЕ УПРАЖНЕНИЯ 6.9

Два бита младших разрядов являются сдвигом блока (CO), за которыми следуют три бита указателя множества (CI); оставшиеся биты выполняют роль тега (CT):

12	11	10	9	8	7	6	5	4	3	2	1	0
СТ	СТ	СТ	СТ	СТ	СТ	СІ	СІ	СІ	СО	СО	СО	СО

РЕШЕНИЕ УПРАЖНЕНИЯ 6.10

Адрес: 0x0E34.

1. Формат адреса:

12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	0	0	1	1	0	1	0	0
СТ	СІ	СІ	СІ	СО	СО	СО						

2. Обращение к ячейке памяти:

Параметр	Значение
Сдвиг блока кэша (CO)	0x0
Указатель множества кэша (CI)	0x5
Тег кэша (CT)	0x71
Результативное обращение в кэш (Да/Нет)	Да
Возвращенный байт кэша	0xB

РЕШЕНИЕ УПРАЖНЕНИЯ 6.11

Адрес: 0x0DD5.

1. Формат адреса:

12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	1	1	0	1	0	1	0	1
СТ	СІ	СІ	СІ	СО	СО	СО						

2. Обращение к ячейке памяти:

Параметр	Значение
Сдвиг блока кэша (CO)	0x1
Указатель множества кэша (CI)	0x5
Тег кэша (CT)	0x6E
Результативное обращение в кэш (Да/Нет)	Нет
Возвращенный байт кэша	—

РЕШЕНИЕ УПРАЖНЕНИЯ 6.12

Адрес: 0xFFFF4.

1. Формат адреса:

12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	0	1	0	0	0
СТ	СИ	СИ	СИ	СО	СО	СО						

2. Обращение к ячейке памяти:

Параметр	Значение
Сдвиг блока кэша (CO)	0x0
Указатель множества кэша (CI)	0x1
Тег кэша (CT)	0xFF
Результативное обращение в кэш (Да/Нет)	Нет
Возвращенный байт кэша	—

РЕШЕНИЕ УПРАЖНЕНИЯ 6.13

Данная задача является своего рода обратной версией упр. 6.9—6.12 и требует движения назад — от содержимого кэша к адресам, которые попадут в конкретное множество. В этом случае множество 3 содержит одну единственную строку с тегом 0x32. Поскольку во множестве имеется только одна единственная строка, четыре адреса будут результативными. Эти адреса двоичные: 0 0110 0100 11xx. Таким образом, четыре шестнадцатеричных адреса, которые будут результативными во множестве 3, следующие: 0x064C, 0x064D, 0x064E и 0x064F.

РЕШЕНИЕ УПРАЖНЕНИЯ 6.14

1. Ключ к решению данного упражнения — изучение схемы на рис. 6.34. Обратите внимание на то, что каждая строка кэша содержит только одну строку массива, что кэш достаточно большой для сохранения одного массива, и что для всех строка src и dst отображается в ту же строку кэша. По причине того, что кэш слишком мал для сохранения обоих массивов, обращения к одному массиву вытесняют полезные строки из другого массива. Так запись в dst [0] [0] вытесняет строку, загруженную при считывании src [0] [0]. Поэтому при следующем считывании src [0] [1] результатом является промах.

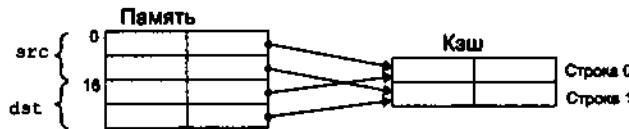


Рис. 6.34. Схема для упражнения 6.14

Массив dst		
	столбец 0	столбец 1
строка 0	m	m
строка 1	m	m

Массив src		
	столбец 0	столбец 1
строка 0	m	m
строка 1	m	h

2. Если кэш состоит из 32 байтов, то он достаточно большой для сохранения обоих массивов. Таким образом, единственными промахами будут первоначальные "холодные" промахи.

Массив dst		
	столбец 0	столбец 1
строка 0	m	h
строка 1	m	h

Массив src		
	столбец 0	столбец 1
строка 0	m	h
строка 1	m	h

РЕШЕНИЕ УПРАЖНЕНИЯ 6.15

Каждая 16-байтовая строка кэша содержит две смежные структуры `algae_position`. Каждый цикл входит в эти структуры в порядке их размещения в памяти, считывая за каждый раз один целый элемент. Поэтому комбинацией для каждого цикла будет промах — попадание и т. д. Обратите внимание, что для данного упражнения можно было предсказать коэффициент неудач фактически без нумерации общего числа считываний и промахов.

- Общее количество доступов при считывании — 512.
- Общее количество доступов при считывании с промахами кэша — 256.
- Коэффициент неудач — $256/512 = 50\%$.

РЕШЕНИЕ УПРАЖНЕНИЯ 6.16

Ключ к данной задаче: кэш может сохранить только половину массива. Поэтому просмотр (сканирование) второй половины массива по столбцам вытесняет строки, загруженные во время просмотра первой половины. Например, считывание первого элемента `grid [16] [0]` вытесняет строку, загруженную при считывании элементов `grid [0] [0]`. В этой строке также `grid [0] [1]`. Таким образом, при просмотре следующего столбца обращение к первому элементу `grid [0] [1]` является промахом.

- Общее количество доступов при считывании — 512.
- Общее количество доступов при считывании с промахами кэша — 256.
- Коэффициент неудач — $256/512 = 50\%$.
- Если бы кэш был в два раза больше, то он мог бы содержать весь массив `grid`. Единственными промахами были бы первоначальные "холодные" промахи, и коэффициент неудач составил бы 25%.

РЕШЕНИЕ УПРАЖНЕНИЯ 6.17

Данный цикл имеет комбинацию обращений по шагу 1. Следовательно, единственными промахами будут первоначальные "холодные" промахи.

- Общее количество доступов при считывании — 512.
- Общее количество доступов при считывании с промахами кэша — 128.
- Коэффициент неудач — $128/512 = 25\%$.
- Увеличение размера кэша на любое значение не изменило бы коэффициент неудач, поскольку избежать "холодных" промахов невозможно.

РЕШЕНИЕ УПРАЖНЕНИЯ 6.18

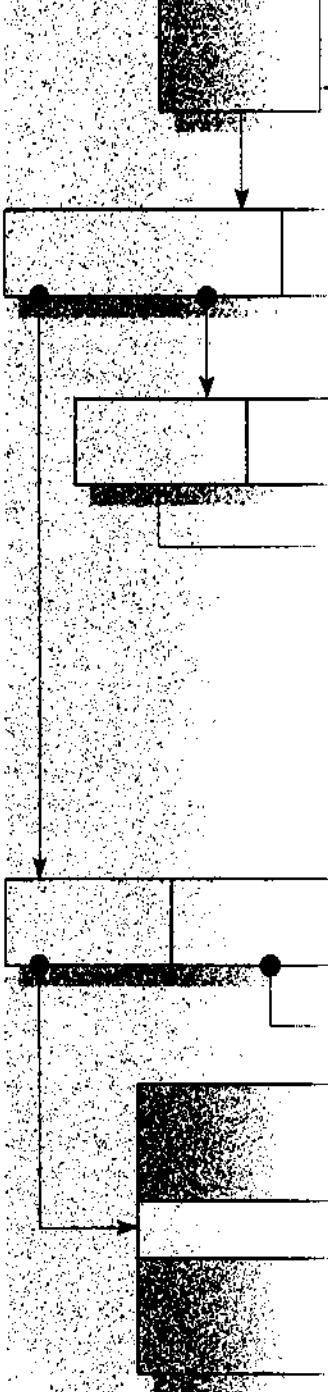
Данное упражнение представляет собой простую "санитарную" проверку понимания проведенного обсуждения. Шаг соответствует пространственной локальности. Рабочее множество соответствует временной локальности.

РЕШЕНИЕ УПРАЖНЕНИЯ 6.19

- Пиковая пропускная способность в L1 составляет порядка 1000 Мбайт/с, а тактовая частота — порядка 500 МГц. Таким образом, для доступа к слову в L1 требуется приблизительно $500/1000 \times 4 = 2$ цикла.
- Для оценки времени доступа в L2 необходимо идентифицировать область в горе памяти, где каждое обращение — промах в L1 и результативное обращение в L2. В частности, необходима область, где рабочее множество слишком большое для L1, но входит в L2 (например, 256 байтов), и шаг превышает размер строки (на-

пример, шаг из 16 слов). Обратите внимание на то, что эффективная пропускная способность в данной области (размер = 256, шаг = 16) составляет порядка 300 Мбайт/с. Таким образом, после проведения вычислений для считывания слова в L2 необходимо порядка $500/300 \times 4 \approx 7$ циклов синхронизации.

3. Для оценки времени доступа к основной памяти обратим внимание на точку горы с самым большим шагом и размером рабочего множества, где каждое обращение представляет собой промах как в L1, так и в L2. Пропускная способность считывания в области (размер = 8Мбайт, шаг = 16) составляет порядка 800 Мбайт/с. Таким образом, после проведения вычислений для считывания слова из основной памяти необходимо порядка $500/80 \times 4 \approx 25$ циклов синхронизации.

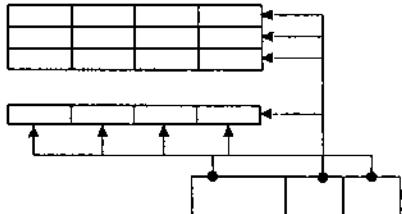


ЧАСТЬ II

Исполнение программ в системе

Изучение компьютерных систем мы продолжим, переходя к более подробному рассмотрению системного программного обеспечения, с помощью которого создаются и исполняются прикладные программы. Редактор связей компонует различные части наших программ в единый файл, который может быть загружен в память и запущен на выполнение. Современные операционные системы взаимодействуют с аппаратными средствами ЭВМ, создавая иллюзию, что каждая программа получает исключительный доступ к использованию процессора и оперативной памяти, тогда как в действительности в любой момент времени в системе исполняется несколько различных программ. Эффективное использование систем, таким образом, требует понимания проблемы и продуманного планирования.

В первой части этой книги вы получили четкое представление о сущности взаимодействия программ с аппаратными средствами ЭВМ. Вторая часть книги расширит знания о системе, даст вам правильное понимание взаимодействия ваших программ с операционной системой. Вы узнаете, как пользоваться услугами, предоставляемыми операционной системой, при создании программ системного уровня, таких как оболочки Unix и пакеты динамического распределения памяти.



ГЛАВА 7

Редактирование связей

- Драйверы компилятора.
 - Статическое связывание.
 - Объектные файлы.
 - Перемещаемые объектные файлы.
 - Идентификаторы и таблицы имен.
 - Разрешение ссылок.
 - Перемещение.
 - Исполняемые объектные файлы.
 - Загрузка исполняемых объектных файлов.
 - Динамическое связывание с совместно используемыми библиотеками.
 - Загрузка и связывание с совместно используемыми библиотеками из приложений.
 - Перемещаемый программный код.
 - Средства манипулирования объектными файлами.
 - Резюме.
-

Редактирование связей (*linking*) (компоновка) — это процесс сборки и объединения различных частей программного кода и данных в единый файл, который может быть загружен в память и запущен на исполнение. Редактирование связей может быть выполнено во время компиляции (*compile time*), после того как исходный текст будет транслирован в машинный код, во время загрузки, после того как программа будет загружена в память и обработана загрузчиком (*loader*) и даже во время исполнения (*run time*) прикладными программами. В ранних компьютерных системах редактирование связей выполнялось вручную. В современных системах это делается автоматически специальными программами, называемыми редакторами связей (*linker*) (компоновщиками).

Редакторы связей играют ключевую роль в разработке программного обеспечения, ввиду того, что они делают возможной *раздельную компиляцию* (*separate compilation*).

Вместо того чтобы создавать большое приложение в единственном монолитном исходном файле, мы можем расчленить его на меньшие, более управляемые модули, которые могут изменяться и компилироваться раздельно. После того, как один из этих модулей будет модифицирован, его следует просто перетранслировать и выполнить повторное редактирование связей всего приложения, без необходимости перетранслировать остальные файлы.

Компоновка обычно без проблем выполняется редактором связей и не волнует студентов, которые разрабатывают свои маленькие программы во вводных курсах по программированию. Итак, что нам дает понимание редактирования связей?

- Понимание сущности редактирования связей облегчит вам создание больших программ. Программисты, которые создают большие программы, часто сталкиваются с ошибками времени редактирования связей, вызванными отсутствием модулей, отсутствием библиотек или несовместимыми библиотечными версиями. Если вы не осознаете, каким образом редактор связей разрешает ссылки, что такое библиотека, как используется библиотека для разрешения ссылок, такие ошибки будут вас ставить в тупик.
- Понимание сущности редактирования связей поможет вам избегать опасных ошибок программирования. Решения, которые принимают редакторы связей Unix при разрешении ссылок на имена, могут незаметно для вас воздействовать на правильность программ. Программы, которые неправильно определяют различные глобальные переменные, по умолчанию проходят через редактор связей без каких-либо предупреждений. Получившиеся в результате программы могут проявить необъяснимое поведение во время исполнения, и их чрезвычайно трудно отлаживать. Мы покажем вам, каким образом это происходит, и как можно не допускать этого явления.
- Понимание сущности редактирования связей поможет вам усвоить, как реализованы в языке правила области видимости. Например, в чем состоит различие между глобальными и локальными переменными? Что фактически означает определение статических переменных или функций?
- Понимание сущности редактирования связей поможет вам освоить другие важные системные концепции. Исполняемые объектные файлы, образованные редакторами связей, играют ключевую роль важных системных функций, таких как загрузка и исполнение программ, управление виртуальной памятью, страничная организация памяти.
- Понимание сущности редактирования связей даст вам возможность пользоваться разделяемыми библиотеками. В течение многих лет редактирование связей считалось занятием примитивным и неинтересным. Однако вместе с увеличением значимости разделяемых библиотек и динамических связей в современных операционных системах редактирование связей стало представлять собой сложный творческий процесс, который предоставляет опытным программистам необычайно богатые возможности. Например, многие программные продукты используют разделяемые библиотеки для модернизации двоичных кодов архивированных прикладных программ во время их исполнения. Кроме того, работа значительной части Web-серверов основана на динамической связи с разделяемыми библиотеками, обслуживающими их динамически меняющееся содержимое.

В этой главе проведено исчерпывающее и всестороннее рассмотрение всех аспектов редактирования связей, от традиционного статического редактирования связей до динамической компоновки разделяемых библиотек во время загрузки и исполнения. При объяснении стандартных механизмов мы будем использовать реальные примеры и выделять ситуации, в которых решение вопросов редактирования связей может воздействовать на ход исполнения и правильность результатов. Чтобы объяснения оставались конкретными и понятными, мы будем формулировать наши рассуждения в контексте машины IA32, на которой инсталлирована одна из версий Unix, Linux или Solaris. Впрочем, для нас важно понять, что базовые концепции редактирования связей являются универсальными, независимо от операционной системы, стандартов ISA или формата объектного файла. Детали могут меняться, но концепции остаются теми же самыми.

7.1. Драйверы компилятора

Рассмотрим С-программу (листинги 7.1—7.2). Она содержит два исходных кода: main.c и swap.c. Функция main вызывает swap, которая переставляет два имеющихся элемента во внешнем глобальном массиве buf. Согласитесь, это довольно странный способ менять два числа, но на протяжении этой главы данная программа будет служить в качестве небольшого рабочего примера, позволяющего выполнять некоторые важные действия, показывающие, каким образом работает процедура редактирования связей.

Листинг 7.1. Программа инициализации

```
1 /* main.c */
2 void swap();
3
4 int buf[2] = {1, 2};
5
6 int main()
7 {
8     swap();
9     return 0;
10 }
```

Листинг 7.2. Программа перестановки

```
1 /* swap.c */
2 extern int buf[];
3
4 int *bufp0 = &buf[0];
5 int *bufp1;
6
```

```

7 void swap()
8 {
9     int temp;
10
11 bufp1 = &buf[1];
12 temp = *bufp0;
13 *bufp0 = *bufp1;
14 *bufp1 = temp;
15 }

```

Функция `main` (листинг 7.1) инициализирует массив с двумя элементами типа `int`, затем вызывает функцию `swap` (листинг 7.2), чтобы поменять местами содержащуюся в нем пару.

Большинство систем компиляции предусматривают наличие *диспетчера компилятора* (compiler driver), который по требованию пользователя вызывает препроцессор языка, компилятор, ассемблер и редактор связей. Например, чтобы построить программу примера, используя GNU-систему компиляции, можно вызвать диспетчер `gcc`, задав следующую команду:

```
unix> gcc -O2 -g -o p main.c swap.c
```

Рис. 7.1 иллюстрирует действия диспетчера при трансляции программы примера из исходного ASCII-файла в исполняемый объектный файл. Если вы хотите проследить эти шаги непосредственно, запустите `gcc` с опцией `-v`. Диспетчер сначала запустит препроцессор C (`cpp`), который транслирует исходный С-файл `main.c` в промежуточный ASCII-файл `main.i`:

```
cpr [другие аргументы] main.c /trap/main.i
```

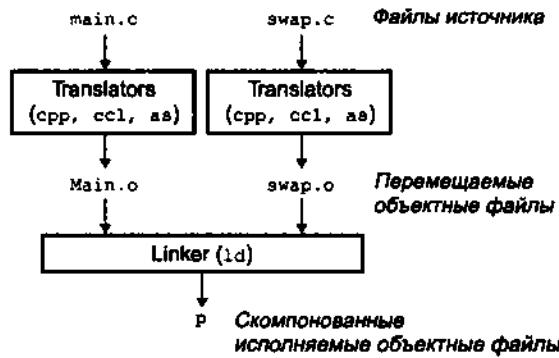


Рис. 7.1. Статическая компоновка

Затем диспетчер запускает С-компилятор (`ccl`), который транслирует `main.i` в ASCII-файл ассемблера `main.s`.

```
ccl /trap/main.i main.c -O2 [другие аргументы] -o /tmp/main.s
```

Далее диспетчер запускает ассемблер (as), который транслирует `main.s` в переместимый объектный файл `main.o`:

```
as [другие аргументы] -o /tmp/main.o /tmp/main.s
```

Такой же процесс происходит в диспетчере при генерировании `swap.o`. Наконец, он запускает программу `ld` редактора связей, который объединяет `main.o` и `swap.o` вместе с необходимыми системными объектными файлами, создавая исполняемый объектный файл `r`:

```
ld -o r [системные объектные файлы и аргументы] /tmp/main.o /tmp/swap.o
```

Для запуска исполняемой программы `r` мы набираем на клавиатуре ее имя в командной строке командного процессора Unix:

```
unix> ./r
```

Командный процессор вызывает функцию в операционной системе, называемую загрузчиком, которая копирует в память программный код и данные исполняемого файла `r` и затем передает управление в начало программы.

7.2. Статическое связывание

Статический редактор связей (static linker), такой как программа Unix `ld`, принимает на входе совокупность переместимых объектных файлов и параметры командной строки и генерирует на выходе полностью готовый связанный исполняемый объектный файл, который может быть загружен и запущен на исполнение. Входные переместимые объектные файлы содержат различные секции программного кода и данных. Команды находятся в одной секции, инициализированные глобальные переменные — в другой секции, а неинициализированные переменные — в третьей секции.

Чтобы построить исполняемый модуль, редактор связей должен проделать две основные процедуры.

- **Разрешение ссылок.** В объектных файлах содержатся определения имен и ссылки на имена. Цель разрешения ссылок — точно связать каждую ссылку на имя с уникальным определением имени.
- **Перемещение.** Компиляторы и ассемблеры генерируют секции программного кода и данных, начиная с нулевого адреса. Редактор связей перемещает эти секции, связывая каждое определение имени с адресом в памяти и изменяя все ссылки на эти имена так, чтобы они указывали на данный адрес в памяти.

В последующих разделах эти процедуры объясняются более подробно. Когда приступите к чтению этого материала, помните о некоторых основных фактах, связанных с редакторами связей: объектный файл — это просто совокупность блоков байтов. Одни из этих блоков содержат программный код, другие — данные программы, а трети — структуры данных, которые управляют редактором связей и загрузчиком. Редактор связей связывает блоки вместе, выбирает для этих блоков адреса времени исполнения и модифицирует различные адреса внутри блоков данных и программного кода. Редакторы связей имеют минимальное представление о целевой машине.

Компиляторы и ассемблеры, которые сгенерировали объектные файлы, уже продели значительную часть работы.

7.3. Объектные файлы

Объектные файлы появляются в одной из трех форм:

1. Переместимый объектный файл. Он содержит двоичный код и данные, которые для создания исполняемого объектного файла могут быть объединены во время компиляции с другими переместимыми объектными файлами.
2. Исполняемый (абсолютный) объектный файл — содержит двоичный код и данные в форме, которые могут быть скопированы непосредственно в память и запущены на исполнение.
3. Разделяемый объектный файл — особый вид переместимого объектного файла, который может быть загружен в память и связан динамически с другими модулями либо во время загрузки, либо во время исполнения.

Компиляторы и ассемблеры генерируют переместимые объектные файлы (включая разделяемые объектные файлы). Редакторы связей генерируют исполняемые объектные файлы. Технически, *объектный модуль* (*object module*) представляет собой последовательность байтов, а *объектный файл* (*object file*) — это объектный модуль, хранимый в файле на диске. Однако мы будем использовать эти термины как взаимозаменяемые.

Форматы объектного файла изменяются от системы к системе. Первые Unix-системы от Bell Labs использовали формат a.out. И по сей день еще исполняемые модули именуются a.out. Ранние версии Unix System V использовали формат COFF (Common Object File). Windows использует вариант COFF, называемый форматом PE (Portable Executable). Современные системы Unix, BSD-варианты Unix и Sun Solaris используют Unix ELF (Executable and Linkable Format). Хотя мы сосредоточимся на ELF, базовые концепции будут сходны, независимо от конкретного формата.

7.4. Переместимые объектные файлы

На рис. 7.2 показан формат типичного переместимого объектного файла ELF. Заголовок ELF начинается с 16-байтовой последовательности, которая описывает размер слова и упорядочение байтов для системы, сгенерировавшей данный файл. Остальная часть заголовка ELF содержит информацию, которая дает возможность редактору связей анализировать и интерпретировать объектный файл. Сюда относятся размер заголовка ELF, вид объектного файла (переместимый, исполняемый или разделяемый), вид машины (например, IA32), смещение таблицы заголовков секций (section header table) в файле, размер и количество входов в таблице заголовков секций. Адреса и размеры различных секций описаны в таблице заголовков секций, которая содержит вход фиксированного размера для каждой секции в объектном файле.



Рис. 7.2. Типичный переместимый объектный файл ELF

Сами секции находятся между заголовком ELF и таблицей заголовков секций. Типичный переместимый объектный файл ELF содержит следующие секции:

- **.text** — машинный код откомпилированной программы;
- **.rodata** — данные только для чтения, такие как строки формата в операторах `printf` и таблицах переходов для операторов `switch` (см. упр. 7.14);
- **.data** — инициализированные глобальные С-переменные. Локальные С-переменные размещаются в стеке во время исполнения и не появляются ни в секции `.data`, ни в секции `.bss`;
- **.bss** — неинициализированные глобальные С-переменные. Эта секция не занимает в объектном файле никакого реального пространства, это просто "держатель места". Форматы объектного файла различаются для инициализированных и неинициализированных переменных. Это сделано с целью экономии памяти: неинициализированные переменные не должны фактически занимать места в объектном файле на диске;
- **.symtab** — таблица имен (таблица идентификаторов) с информацией о функциях и глобальных переменных, которые определены в программе и на которые имеются ссылки. Некоторые программисты ошибочно считают, что для того чтобы получить информацию из таблицы имен, программа должна быть компилирована с опцией `-g`. На самом деле, каждый переместимый объектный файл имеет таблицу имен в `.symtab`. Однако, в отличие от таблицы идентификаторов в компиляторе, таблица имен `.symtab` не содержит входов для локальных переменных;
- **.rel.text** — список адресов в секции `.text`, которые должны быть изменены, когда редактор связей будет компоновать данный объектный файл с другими. Вообще, любая команда, которая вызывает внешнюю функцию или ссылается на глобальную переменную, должна быть модифицирована. С другой стороны, команды, которые вызывают локальные функции, не должны модифицироваться. Имейте

в виду, что информация о перемещении не требуется в исполняемых объектных файлах и обычно опускается, если пользователь не даст редактору связей указания включить ее явным образом;

- .ref.data** — информация о перемещении для каждой глобальной переменной, на которую имеется ссылка, или которая определяется в данном модуле. Вообще, каждая инициализированная глобальная переменная, начальное значение которой адрес глобальной переменной или внешняя функция, должна быть модифицирована;
- .debug** — таблица имен отладки с входами для локальных переменных и определений типа в данной программе, глобальных переменных, на которые имеются ссылки или которые определяются в данной программе, а также исходный С-файл. Они присутствуют только в том случае, если диспетчер компилятора вызван с опцией `-g`;
- .line** — таблица соответствия между номерами строк в исходной С-программе и командами машинного кода в секции `.text`. Она присутствует, если диспетчер компилятора вызван с опцией `-g`;
- .strtab** — таблица строк для таблиц имен в секциях `.symtab` и `.debug` и для секции имен в секции заголовков. Таблица строк представляет собой последовательность символьных строк, каждая из которых с нулевым символом в конце.

Имена неинициализируемых данных

Использование термина `.bss` для обозначения неинициализированных данных широко распространено. Первоначально (приблизительно в 1957 г.) это был акроним для команды ассемблера IBM 704 Block Storage Start (начало блока памяти), и этот акроним так и сохранился. Простой способ запомнить, чем различаются секции `.data` и `.bss` — это представлять себе "`bss`", как сокращение для "Better Save Space!" (экономь память, где только возможно).

7.5. Идентификаторы и таблицы имен

Каждый переместимый объектный модуль `m` имеет таблицу имен, которая содержит информацию обо всех определенных именах, на которые есть ссылки в `m`. С точки зрения редактора связей, имеются три различных вида имен:

- Глобальные имена (global symbol)**, которые определены в модуле `m` и на которые могут быть ссылки из других модулей. Глобальные имена редактора связей соответствуют **нестатическим** функциям языка С, а глобальные переменные — это те, которые определены без атрибута `static` языка С.
- Глобальные имена**, на которые имеются ссылки в модуле `m`, но определенные в некотором другом модуле, называют **внешними (external)**, и они принадлежат функциям и переменным языка С, которые определены в других модулях.
- Локальные имена (local symbol)**, которые определены и на которые имеются ссылки исключительно внутри модуля `m`. Некоторые локальные имена редактора

связей соответствуют функциям и глобальным переменным языка С, которые определены с атрибутом `static`. Эти имена видимы повсюду внутри модуля `m`, но на них не может быть ссылок из остальных модулей. Эти секции в объектном файле и имя исходного файла также получают локальное имя.

Важно понять, что локальные имена редактора связей — не то же самое, что локальные переменные программы. Таблица имен в `.symtab` не содержит никаких имен, которые соответствуют локальным нестатическим переменным программы. Последние размещаются в стеке во время исполнения и не представляют интереса для редактора связей.

Интересно, что локальные переменные процедур, определенные в языке С с атрибутом `static`, не размещаются в стеке. Для каждого такого определения компилятор выделяет место в `.data` или `.bss` и создает локальное имя редактора связей в таблице имен с уникальным идентификатором. Например, предположим, что пара функций в одном и том же модуле (листинг 7.3) определяет статическую локальную переменную `x`.

Листинг 7.3. Определение статической переменной

```
1 int f()
2 {
3     static int x = 0;
4     return x;
5 }
6
7 int g()
8 {
9     static int x = 1;
10    return x;
11 }
```

В данном случае компилятор выделяет в `.bss` место для двух целых чисел и передает ассемблеру два одинаковых нетождественных локальных имени редактора связей. Вместо этого можно было бы в функции `f` для определения переменной использовать `x.1`, а в функции `g` — `x.2`.

Скрытие имен переменных и функций

Чтобы скрыть объявления переменных и функций в модулях, программисты на языке С используют атрибут `static`, аналогично тому, как в Java и C++ использовали бы объявления `public` и `private`. Исходные файлы языка С играют роль модулей. Каждая глобальная переменная или функция, объявленная с атрибутом `static`, закрыта в этом модуле. Точно так же каждая глобальная переменная или функция, объявленная без атрибута `static`, является открытой и доступна из любых других модулей. Практика программирования показала целесообразность защищать переменные и функции с помощью атрибута `static` везде, где это возможно.

Таблицы имен, скомпонованные ассемблерами, используют имена, переданные компилятором в ассемблерный файл .e. Таблица имен ELF содержитется в секции .symtab. Она содержит массив входов. В листинге 7.4 показан формат каждого входа.

Листинг 7.4. Входы таблицы имен ELF

```

1 typedef struct {
2     int name;          /* смещение таблицы строк */
3     int value;         /* смещение секции, или VM-адрес */
4     int size;          /* размер объекта в байтах */
5     char type:4,       /* данные, функция, секция или имя исходного файла */
6         binding:4;    /* локальный или глобальный (4 бита) */
7     char reserved;    /* не используется */
8     char section;     /* индекс заголовка секции, ABS, UNDEF или COMMON */
9
10 } Elf_Symbol;

```

Переменная `name` представляет собой смещение (в байтах) в строковой таблице, которое указывает на строку имени с нулевым символом на конце, `value` — это адрес имени. Для переместимых модулей `value` — это смещение от начала секции, в которой определен данный объект. Для исполняемых объектных файлов `value` — это абсолютный адрес времени исполнения. Переменная `size` — это размер (в байтах) объекта, `type` — обычно либо данные, либо функция. Таблица имен может также содержать входы для отдельной секции и для имени пути к исходному файлу программы. Таким образом, имеются различные объекты такого типа. Поле `binding` указывает, является ли имя локальным или глобальным.

Каждое имя связано с некоторой секцией объектного файла, обозначенной полем `section`, которое представляет собой индекс в таблице заголовков секций. Имеются три специальные секции, которые не имеют входов в таблице заголовков секций:

1. ABS — для имен, которые не могут быть перемещены.
2. UNDEF — для неопределенных имен (на которые ссылаются в данном объектном модуле, но которые определены в другом месте).
3. COMMON — для неинициализированных объектов данных, которые еще не размещены. Для COMMON-имен поле `value` удовлетворяет требованию выравнивания, а `size` дает минимальный размер.

Возьмем, например, последние три входа в таблице имен для `main.o`, отображаемой сервисной программой GNU READELF. Первые восемь входов, которые не показаны, — это локальные имена, используемые редактором связей для внутренних целей.

В табл. 7.1 мы видим вход для определения глобального имени `buf`, 8-байтовый объект, расположенный с нулевым смещением (т. е. `value = 0`) в секции `.data`. За ним, по определению, следует глобальное имя `main`, 17-байтовая функция, расположенная с нулевым смещением в секции `.text`. Последний вход относится к ссылке для внешнего имени `swap`. Индекс, равный 1, обозначает секцию `.text`, а индекс, равный 3, — секцию `.data`.

Таблица 7.1. Входы глобальных имен основного модуля

Номер	Значение	Размер	Тип	Связь	Сдвиг	Индекс	Имя
8:	0	8	OBJECT	GLOBAL	0	3	buf
9:	0	17	FUNC	GLOBAL	0	1	main
10:	0	0	NOTYPE	GLOBAL	0	UND	swap

Вот аналогичные входы таблицы имен для swap.o (табл. 7.2).

Таблица 7.2. Входы глобальных имен объектного модуля

Номер	Значение	Размер	Тип	Связь	Сдвиг	Индекс	Имя
8:	0	4	OBJECT	GLOBAL	0	3	bufp0
9:	0	0	NOTYPE	GLOBAL	0	UND	buf
10:	0	39	FUNC	GLOBAL	0	1	swap
11:	4	4	OBJECT	GLOBAL	0	UND	bufp1

Прежде всего, мы видим вход для определения глобального имени bufp0, которое представляет 4-байтовый инициализированный объект, расположенный в секции .data со смещением 0. Следующее имя связано со ссылкой на внешнее имя buf в коде инициализации для bufp0. За ним следует глобальное имя swap, 39-байтовая функция в .text со смещением 0. Последний вход — это глобальное имя bufp1, 4-байтовый неинициализированный объект данных (с 4-байтовым выравниванием), который будет, в конечном счете, размещен как объект .bss при редактировании связей данного модуля.

УПРАЖНЕНИЕ 7.1

Это задание связано с модулем swap.o из листинга 7.2. Для каждого имени, которое определено или на которое есть ссылка в swap.o, укажите, действительно ли оно будет иметь вход в секции .symtab таблицы имен в модуле swap.o. Если да, то укажите тот модуль, в котором определяется имя (swap.o либо main.o), тип имени (локальное, глобальное либо внешнее) и секцию (.text, .data, либо .bss), которую он занимает в этом модуле.

Имя	Вход	Тип имени	В каком модуле определено	Секция
buf				
bufp0				
bufp1				
swap				
temp				

7.6. Разрешение ссылок

Редактор связей разрешает ссылки на имена, связывая каждую ссылку с одним определением имени из таблицы имен его входных переместимых объектных файлов. Разрешение ссылок на локальные имена, которые определены в том же самом модуле, что и сама ссылка, выполняется просто. Компилятор допускает лишь единственное определение каждого локального имени в модуле. Компилятор также гарантирует, что имена локальных статических переменных, которые получит редактор связей, будут уникальными.

Разрешение ссылок на глобальные имена, однако, является более сложной задачей. После того как компилятор сталкивается с именем (переменной или функции), которое не определено в текущем модуле, он предполагает, что это имя определено в некотором другом модуле, генерирует вход таблицы имен редактора связей и оставляет его для дальнейшей обработки редактором связей. Если ни в каком из его входных модулей редактор связей не способен найти определение для имени, на которое имеется ссылка, он печатает сообщение об ошибке (часто загадочное) и завершает свою работу. Например, попробуем компилировать и отредактировать связи для кода листинга 7.5 на Linux-машине:

Листинг 7.5. Образец для компиляции

```
1 void foo(void);
2
3 int main() {
4     foo();
5     return 0;
6 }
```

Компилятор отработает беспрекословно, но редактор связей завершит свою работу после того, как он не сможет разрешить ссылку на `foo`:

```
unix> gcc -Wall -O2 -o linkerror linkerror.c
/tmp/ccSz5uti.o: In function 'main': (в функции 'main')
/tmp/ccSz5uti.o(.text+0x7): undefined reference to 'foo' (не определена ссылка
на 'foo')
collect2: ld returned 1 exit status (возвращаемое состояние при выходе 1)
```

Разрешение ссылок для глобальных имен представляет собой также мудреную задачу, поскольку одно и то же имя может, в принципе, быть определено в нескольких различных объектных файлах. В таком случае редактор связей должен либо сигнализировать об ошибке, либо тем или иным образом выбрать одно из определений и отвергнуть остальные. Подход, принятый в Unix-системах, предполагает тесное сотрудничество между компилятором, ассемблером и редактором связей и при безалаберном программировании может привести к появлению некоторых "необъяснимых" ошибок.

Коверкание имен редактором связей в языках C++ и Java

И языки C++ и Java допускают использование перегруженных методов, которые в исходном коде имеют одинаковые идентификаторы, но отличаются списками параметров. Тогда возникает вопрос, каким же образом редактор связей сообщает о том, что эти перегруженные функции не идентичны? Механизм перегруженных функций работает в языках C++ и Java, поскольку компилятор перекодирует идентификатор каждого уникального метода, с учетом комбинации списка его параметров, в уникальное имя для редактора связей. Этот процесс кодировки представляет собой так называемое коверкание (*mangling*) идентификатора имени, а обратный процесс — восстановление (*demangling*) идентификатора имени.

Интересно, что языки C++ и Java используют совместимые схемы коверкания идентификаторов имен. Исковерканный идентификатор класса состоит из целого числа, обозначающего количество символов в идентификаторе, за которым следует оригинальный идентификатор. Например, класс Foo кодируется как 3Foo. Метод кодируется как исходный идентификатор метода, за которым идет подчеркивание, затем следует исковерканный идентификатор класса, далее однобуквенные кодировки типа каждого аргумента. Например, Foo::bar (int, long) кодируется как bar_3Fooil. Подобные же схемы используются и для коверкания глобальных переменных и идентификаторов шаблонов.

7.6.1. Многократно определенные глобальные ссылки

Во время компиляции компилятор передает в ассемблер каждое глобальное имя, либо как строго определенное (*strong*), либо как слабо определенное (*weak*), и ассемблер неявно кодирует данную информацию в таблице имен перемещаемого объектного файла. Функции и инициализированные глобальные переменные получают строго определенные имена. Неинициализированные глобальные переменные получают слабо определенные имена. Для программы из примера в листинге 7.2 buf, buf0, main и swap — строго определенные имена; bufp1 представляет собой слабо определенное имя.

Основываясь на этих понятиях строго определенных и слабо определенных имен, редакторы связей Unix используют следующие правила для обработки многократно определенных имен:

1. Не допускается использование нескольких строго определенных имен.
2. При наличии строго определенного имени и нескольких слабо определенных имен выбирается строго определенное имя.
3. При наличии нескольких слабо определенных имен выбирается любое из слабо определенных имен.

Предположим, что нам нужно компилировать и редактировать связи следующих двух модулей языка C (листинги 7.6 и 7.7).

Листинг 7.6. Пример 1 использования правила 1

```

1 /* fool.c */
2 int main()
3 {
4     return 0;
5 }

```

- 1 /* bar1.c */
- 2 int main()
- 3 {
- 4 return 0;
- 5 }

В данном случае редактор связей генерирует сообщение об ошибке ввиду того, что строго определенное имя `main` определено несколько раз (правило 1):

```

unix> gcc fool.c bar1.c
/tmp/cca015022.o: In function 'main': (в функции 'main')
/tmp/cca015022.o(.text+0x0): multiple definition of 'main' (повторное
определение 'main')
/tmp/cca015021.o(.text+0x0): first defined here (первое определение здесь)

```

Точно так же редактор связей генерирует сообщение об ошибке для следующих модулей, поскольку строго заданное имя `x` определено дважды (правило 1):

Листинг 7.7. Пример 2 использования правила 1

```

1 /* foo2.c */
2 int x = 15213;
3
4 int main()
5 {
6     return 0;
7 }

```

- 1 /* bar2.c */
- 2 int x = 15213;
- 3
- 4 void f()
- 5 {
- 6 }

Однако если переменная `x` не инициализирована в одном из модулей, то редактор связей без проблем выберет строго заданное имя, определенное в другом (правило 2):

Листинг 7.8. Пример использования правила 2

```

1 /* foo3.c */
2 #include <stdio.h>
3 void f(void);
4
5 int x = 15213;
6
7 int main()
8 {
9     f();
10    printf("x = %d\n", x);
11    return 0;
12 }

```

- 1 /* bar3.c */
- 2 int x;
- 3
- 4 void f()
- 5 {
- 6 x = 15212;
- 7 }

Во время исполнения функция `f` изменяет значение `x` с 15213 на 15212, что могло бы стать неприятной неожиданностью для автора функции `main`. Обратите внимание на то, что редактор связей обычно не реагирует, когда он обнаруживает несколько различных определений `x`:

```
unix> gcc -o foobar3 foo3.c bar3.c
unix> ./foobar3
x = 15212
```

То же самое может произойти, если имеются два слабых определения `x` (правило 3):

Листинг 7.9. Пример использования правила 3

```
1 /* foo4.c */
2 #include <stdio.h>
3 void f(void);
4
5 int x;
6
7 int main()
8 {
9     x = 15213;
10    f();
11    printf("x = %d\n", x);
12    return 0;
13 }
```

- 1 /* bar4.c */
- 2 int x;
- 3
- 4 void f()
- 5 {
- 6 x = 15212;
- 7 }

Применение правил 2 и 3 может привести к появлению некоторых коварных ошибок во время исполнения, которые мало понятны безалаберному программисту, особенно если повторные определения имени имеют различные типы. Рассмотрим следующий пример, где `x` определен как `int` в одном модуле, и как `double` — в другом:

Листинг 7.10. Пример использования правил 2 и 3

```
1 /* foo5.c */
2 #include <stdio.h>
3 void f(void);
4
5 int x = 15213;
6
7
8 int main()
9 {
10    f();
11    printf("x = 0x%x y = 0x%x\n",
12        x, y);
```

- 1 /* bar5.c */
- 2 double x;
- 3
- 4 void f()
- 5 {
- 6 int y = 15212; 6 x = -0.0;
- 7 }

```
13     return 0;
14 }
```

На машине IA32 под управлением Linux величинам с двойной точностью отводится 8 байтов, а целым — 4 байта. Таким образом, присваивание $x = -0.0$ в строке 6 модуля `bar5.c` изменит значения в адресах памяти для переменных `x` и `y` (строки 5 и 6 модуля `foo5.c`) при представлении отрицательной величины как данных двойной точности с плавающей запятой!

```
linux> gcc -o foobar5 foo5.c bar5.c
linux> ./foobar5
x = 0x0 y = 0x80000000
```

Эта — тонкая и опасная ошибка, особенно потому, что она проходит незаметно, без предупреждения со стороны системы компиляции. Она обычно проявляется несколько позже — при исполнении программы, далеко от того места, в котором ошибка произошла. В больших системах с сотнями модулей ошибки такого рода чрезвычайно трудно устранять, особенно ввиду того, что многие программисты не знают, каким образом работают редакторы связей. Если появляется сомнение, вызовите редактор связей с ключами, например `gcc -fcomptop`, чтобы указать, что при разрешении многократно определенных глобальных ссылок следует выводить предупреждающее сообщение.

УПРАЖНЕНИЕ 7.2

В этом упражнении будем предполагать, что запись `REF(x.i) --> DEF(x.k)` означает, что редактор связей ассоциирует произвольную ссылку на имя `x` в модуле `i` с определением `x` в модуле `k`. Для каждого последующего примера используйте данную нотацию, чтобы показать, каким образом редактор связей разрешил бы ссылки на многократно определенное имя в каждом модуле. Если имеется ошибка времени редактирования связей (правило 1), напишите "ERROR". Если редактор связей произвольно выбирает одно из определений (правило 3), напишите "UNKNOWN".

1.

```
/* Модуль 1 */                  /* Модуль 2 */
int main()                      int main;
{                                int p2();
}
                                         {
}
(a) REF(main.1) --> DEF( __. __)
(b) REF(main.2) --> DEF( __. __)
```

2.

```
/* Модуль 1 */                  /* Модуль 2 */
void main()                      int main = 1;
{                                int p2();
}
                                         {
}
```

- (a) REF(main.1) --> DEF(__. __)
 (b) REF(main.2) --> DEF(__. __)

3.

```
/* Модуль 1 */           /* Модуль 2 */
int x;                  double x = 1.0;
void main();             int p2()
{
}
(a) REF(x.1) --> DEF( __. __)
(b) REF(x.2) --> DEF( __. __)
```

7.6.2. Связывание со статическими библиотеками

До настоящего времени мы предполагали, что редактор связей читает совокупность перемещаемых объектных файлов и связывает их вместе в выходной исполняемый файл. Практически все системы компиляции обеспечивают механизм, позволяющий упаковывать связанные объектные модули в единый файл, называемый *статической библиотекой* (*static library*), который может затем быть передан в качестве входа редактору связей. Когда редактор связей компонует выходной исполняемый файл, он копирует из библиотеки только те объектные модули, на которые имеются ссылки из прикладной программы.

Почему концепция библиотек поддерживается системно? Рассмотрим язык ANSI C, в котором определен широкий набор функций стандартного ввода-вывода, операций над строками и целочисленных математических функций, таких как atoi, printf, scanf, strcpy и random. Они доступны в библиотеке libc.a каждой программе на языке C. ANSI C определяет также в библиотеке libm.a широкий набор математических функций для операций над данными с плавающей запятой, таких как sin, cos и sqrt.

Рассмотрим различные подходы, которые могли бы использовать разработчики компиляторов, чтобы обеспечить пользователям доступ к этим функциям, не прибегая к использованию статических библиотек. Первый возможный подход состоял бы в том, чтобы построить компилятор, распознающий запросы на стандартные функции и генерирующий соответствующий код непосредственно. В языке Pascal, который предусматривает наличие небольшого набора стандартных функций, принят именно такой подход, но он неприемлем для языка C, ввиду наличия большого количества стандартных функций, определенных по стандарту этого языка. Принятие такого подхода существенно усложнило бы компилятор и потребовало бы изменять версию компилятора каждый раз при добавлении, удалении, либо модификации функций. Прикладным программистам, однако, данный подход был бы весьма удобен, ввиду того, что стандартные функции постоянно были бы доступны.

Другой подход состоял бы в том, чтобы поместить все стандартные функции языка C в единственный перемещаемый объектный модуль, скажем, libc.o, который прикладные программисты могли бы связывать с их исполняемыми модулями:

```
unix> gcc main.c /usr/lib/libc.o
```

Данный подход имеет то преимущество, что он отделил бы реализацию стандартных функций от реализации компилятора, и все еще был бы достаточно удобен для программистов. Однако его большое неудобство состоит в том, что каждый исполняемый в системе файл теперь содержал бы полную копию набора стандартных функций, что было бы чрезвычайно расточительно с точки зрения использования места на диске. В типичной системе `libc.a` занимает приблизительно 8 Мбайт, а `libm.a` — приблизительно 1 Мбайт. Хуже того, каждая исполняющаяся программа теперь содержала бы свою собственную копию этих функций в памяти, что было бы чрезвычайно расточительно, с точки зрения использования памяти. Еще одно большое неудобство состоит в том, что любое изменение любой стандартной функции, сколь бы незначительным оно ни было, потребовало бы, чтобы разработчик библиотеки повторно компилировал весь исходный файл, а это — занимающая много времени операция, которая усложнила бы разработку и сопровождение стандартных функций.

Мы могли бы снять некоторые из этих проблем, создавая отдельный перемещаемый файл для каждой стандартной функции и храня эти файлы в известном каталоге. Однако данный подход потребовал бы, чтобы прикладные программисты явно редактировали связи соответствующих объектных модулей, образуя из них исполняемые файлы, а это процесс, подверженный ошибкам и довольно трудоемкий:

```
unix> gcc main.c /usr/lib/printf.o /usr/lib/scanf.o ...
```

Концепция статической библиотеки была разработана для того, чтобы устранить неудобства этих различных подходов. Связанные функции могут быть компилированы в отдельные объектные модули и затем собраны в единственный статический библиотечный файл. Прикладные программы могут впоследствии использовать любую из этих функций, определенных в библиотеке, задавая только имя файла в командной строке. Например, программа, которая использует функции стандартной библиотеки языка С и математической библиотеки, может быть компилирована и скомпонована с помощью команды следующего вида:

```
unix> gcc main.c /usr/lib/libm.a /usr/lib/libc.a
```

Во время компоновки редактор связей только копирует объектные модули, те, на которые имеются ссылки из программы, что уменьшает размер исполняемого файла на диске и в памяти. С другой стороны, прикладной программист должен всего лишь включить названия нескольких библиотечных файлов. Фактически, диспетчеры компилятора языка С всегда передают `libc.a` редактору связей, поэтому ссылка на `libc.a`, о которой говорилось ранее, не нужна.

В Unix-системах статические библиотеки хранятся на диске в файлах специального формата, известных как архивные. Архив представляет собой совокупность связанных перемещаемых объектных файлов с заголовком, который описывает размер и адрес каждого элемента объектного файла. Имена архивных файлов обозначаются с использованием расширения `.a`. Чтобы разговор о библиотеках был конкретным, предположим, что предметом нашего рассмотрения являются процедуры обработки векторов, представленные в листинге 7.11 в статической библиотеке `libvector.a`.

Листинг 7.11 Элементы объектных файлов

```

1 void addvec(int *x, int *y,
2             int *z, int n)
3 {
4     int i;
5
6     for (i = 0; i < n; i++)
7         z[i]=x[i]+y[i];
8 }
1 void multvec(int *x,   int *y,
2               int *z, int n)
3 {
4     int i;
5
6     for (i = 0; i < n; i++)
7         z[i]=x[i]*y[i];
8 }
```

Чтобы создать библиотеку, мы используем утилиту ar:

```
unix> gcc -c addvec.c multvec.c
unix> ar rcs libvector.a addvec.o multvec.o
```

Чтобы использовать эту библиотеку, можно написать такое приложение, как, например, main2.c в листинге 7.12, где вызывается библиотечная процедура addvec (файл заголовков vector.h определяет прототипы функций для процедур из libvector.a).

Листинг 7.12 Вызов элемента статической библиотеки

```

1 /* main2.c */
2 #include <stdio.h>
3 #include "vector.h"
4
5 int x[2] = {1, 2};
6 int y[2] = {3, 4};
7 int z[2];
8
9 int main()
10 {
11     addvec(x, y, z, 2);
12     printf("z = [%d %d]\n", z[0], z[1]);
13     return 0;
14 }
```

Чтобы построить исполняемый модуль, мы компилируем и редактируем связи для входных файлов main.o и libvector.a:

```
unix> gcc -O2 -c main2.c
unix> gcc -static -o p2 main2.o ./libvector.a
```

На рис. 7.3 представлен итог деятельности редактора связей. Параметр `-static` сообщает диспетчеру компилятора, что редактор связей должен построить полностью связанный объектный файл исполняемого модуля, который может быть загружен в память и запускаться на исполнение, не требуя никакой дальнейшей обработки свя-

зей во время загрузки. Когда редактор связей запускается на исполнение, он определяет, что имя `addvec`, определенное в `addvec.c`, имеет ссылки из `main.o`, поэтому он копирует `addvec.o` в исполняемый файл. Так как эта программа не ссылается на имена, определенные в `multvec.o`, редактор связей не копирует данный модуль в исполняемый файл. Редактор связей так же копирует модуль `printf.o` из `libc.a`, наряду с несколькими другими модулями.

Файлы источника

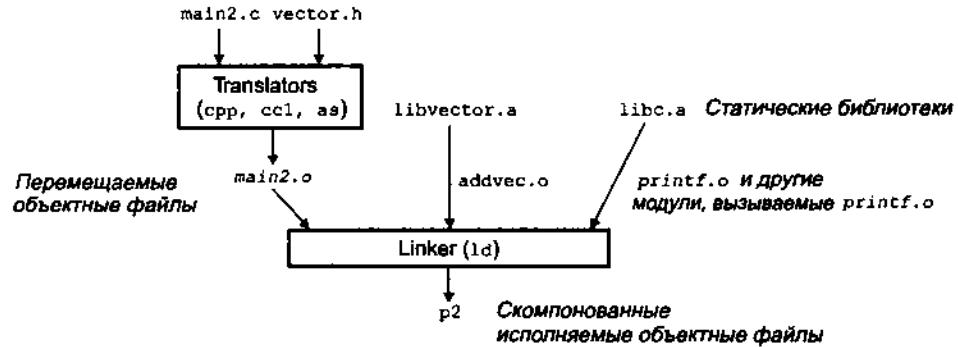


Рис. 7.3. Редактирование связей со статическими библиотеками

7.6.3. Использование статических библиотек

Несмотря на то, что статические библиотеки — это полезные и необходимые инструменты, они представляют собой также источник неуверенности для программистов, в связи с недостаточной осведомленностью относительно методов, используемых редактором связей Unix при разрешении внешних ссылок. На этапе разрешения ссылок редактор связей просматривает переместимые объектные файлы и архив слева направо, в той же самой последовательности, в которой они появляются в командной строке диспетчера компилятора. В течение текущего просмотра редактор связей поддерживает набор E переместимых объектных файлов, которые будут объединены при формировании исполняемого файла, набор U не разрешенных имен (на которые есть ссылки, но которые еще не определены) и набор имен D , которые были определены в предыдущих входных файлах. Первоначально E , U и D — пусты.

Для каждого входного файла f в командной строке редактор связей определяет, является ли f объектным файлом или архивным. Если окажется, что f — это объектный файл, то редактор связей добавляет f к E , корректирует U и D , чтобы отразить определения имени и ссылки в f , и переходит к следующему входному файлу.

Если окажется, что f — это архив, то редактор связей пытается подобрать имена, определенные как элементы архива, соответствующие не разрешенным именам в U . Если некоторый элемент архива, например m , определяет имя, которое разрешает ссылку в U , то m будет добавлено к E , и редактор связей подкорректирует U и D так, чтобы отразить определения имени и ссылки в m . Данный процесс последовательно

повторяется применительно к элементам объектных файлов архива, пока не будет достигнута точка, в которой U и D больше не изменяются. С этого момента все элементы объектного файла, не содержащиеся в E , просто отбрасываются, и редактор связей переходит к следующему входному файлу.

Если после того, как редактор связей заканчивает просматривать входные файлы в командной строке, окажется, что U не пусто, он выводит сообщение об ошибке и завершает свою работу. В противном случае, он объединяет и перемещает объектные файлы в E , чтобы построить выходной исполняемый файл.

К сожалению, данный алгоритм может привести к некоторым ставящим в тупик ошибкам времени редактирования связей, ввиду того, что порядок следования библиотек и объектных файлов в командной строке имеет существенное значение. Если библиотека, в которой определено некоторое имя, появляется в командной строке перед объектным файлом, который ссылается на данное имя, то ссылка не будет разрешена, и связывание завершится аварийно. Например, рассмотрим следующие строки:

```
unix> gcc -static ./libvector.a main2.c
/tmp/cc9XH6Rp.o: In function 'main': (в функции 'main')
/tmp/cc9XH6Rp.o(.text+0x18): undefined reference to 'addvec' (не определена
ссылка на 'addvec')
```

Что здесь произошло? Во время обработки `libvector.a` набор U будет пустым, поэтому к E не будет добавлено никаких элементов объектных файлов из `libvector.a`. Таким образом, ссылка на `addvec` не будет никогда разрешена, редактор связей выдаст сообщение об ошибке и завершает свою работу.

Общее правило для библиотек — это размещать их в конце командной строки. Если элементы различных библиотек независимы, в том смысле, что никакой элемент не ссылается на имя, определенное в другом элементе, то библиотеки могут быть помещены в конце командной строки в любом порядке.

Если, с другой стороны, библиотеки не являются независимыми, то они должны быть упорядочены таким образом, чтобы для каждого имени s , на которое имеется внешняя ссылка из элемента архива, по крайней мере, одно определение s следовало бы за ссылкой на s в командной строке. Например, предположим, что `foo.c` вызывает функции из `libx.a` и из `libz.a`, а те вызывают функции из `liby.a`. Тогда `libx.a` и `libz.a` должны в командной строке предшествовать `liby.a`:

```
unix> gcc foo.c libx.a libz.a liby.a
```

Библиотеки могут повторяться в командной строке, если это необходимо для удовлетворения требованиям зависимости. Например, предположим, что `foo.c` вызывает функцию из `libx.a`, та вызывает функцию из `liby.a`, которая вызывает функцию из `libx.a`. Тогда функция `libx.a` должна быть повторена в командной строке:

```
unix> gcc foo.c libx.a liby.a libx.a
```

В качестве альтернативы можно было бы объединить `libx.a` и `liby.a` в едином архиве.

УПРАЖНЕНИЕ 7.3

Пусть *a* и *b* обозначают объектные модули или статические библиотеки в текущем каталоге, и пусть *a->b* обозначает, что *a* зависит от *b* — в том смысле, что *b* определяет имя, на которое имеется ссылка из *a*. Для каждого из следующих сценариев укажите минимальную командную строку (с наименьшим количеством объектных файлов и библиотечных аргументов), которая позволит статическому редактору связей разрешить все ссылки на имена.

1. *p.o -> libx.a*
2. *p.o -> libx.a -> liby.a*
3. *p.o -> libx.a -> liby.a and liby.a -> libx.a -> p.o*

7.7. Перемещение

Как только редактор связей закончит этап разрешения ссылок, каждая ссылка на имя в программном коде будет связана с точно одним определением имени (входом таблицы имен в одном из своих входных объектных модулей). В этот момент редактор связей знает точные размеры секций программного кода и данных в его входных объектных модулях. Теперь он готов приступить к этапу перемещения, на протяжении которого он объединяет входные модули и назначает для каждого имени адреса времени исполнения. Перемещение выполняется за два шага.

1. Перемещение секций и определений имен. На этом шаге редактор связей объединяет все секции одного типа в новую составную секцию того же самого типа. Например, все секции *.data* из входных модулей объединены в единственную секцию, которая станет секцией *.data* для выходного исполняемого объектного файла. Затем редактор связей назначает адреса памяти времени исполнения новым составным секциям, для каждой секции, определенной входными модулями, и для каждого имени, определенного входными модулями. После того, как данный шаг будет закончен, каждая (машинная) команда и каждая глобальная переменная в программе будут иметь уникальный адрес памяти времени исполнения.
2. Перемещение ссылок на имена внутри секций. На этом шаге редактор связей модифицирует каждую ссылку на имя в теле программного кода и в секции данных так, чтобы они указывали на корректные адреса времени исполнения. Чтобы проделать этот шаг, редактор связей полагается на структуры данных в перемещимых объектных модулях, так называемые *входы перемещения* (*relocation entry*), которые мы рассмотрим позже.

7.7.1. Входы перемещения

Когда ассемблер генерирует объектный модуль, он не знает, где этот код и данные будут храниться в памяти. И при этом он также не знает никаких адресов внешне определенных функций или глобальных переменных, на которые ссылаются из данного модуля. Поэтому всякий раз, когда ассемблер сталкивается со ссылкой на объект, чье окончательное местоположение неизвестно, он генерирует *вход перемещения*.

ния, который сообщает редактору связей, каким образом следует изменить ссылку при объединении объектных файлов в исполняемый файл. Входы перемещения для программного кода расположены в `.relo.text`. Входы перемещения для инициализированных данных расположены в `.relo.data`.

В листинге 7.13 показан формат входа перемещения ELF. Переменная `offset` — это смещение внутри секции для ссылки, которая должна быть модифицирована. Переменная `symbol` идентифицирует имя, на которое должна указывать модифицируемая ссылка. Переменная `type` сообщает редактору связей, каким образом следует модифицировать новую ссылку.

Листинг 7.13. Вход перемещения ELF

```
1 typedef struct {
2     int offset; /* смещение ссылки относительно начала таблицы */
3     int symbol: 24, /* имя, на которое должна указывать ссылка */
4     type:8; /* тип перемещения */
5     Elf32_Rel;
```

ELF определяет 11 различных типов перемещения, некоторые из них — весьма загадочные. Мы остановимся только на двух самых главных типах.

1. `R_386_PC32` перемещает ссылку, которая использует 32-битовый адрес относительно счетчика команд (PC). Вспомните, в разд. 3.6.3 говорилось, что адрес относительно счетчика команд представляет собой смещение относительно текущего значения счетчика команд времени исполнения. Когда центральный процессор исполняет команду, используя относительную адресацию по счетчику команд, он формирует **эффективный адрес** (effective address) (например, адрес команды вызова) путем добавления 32-битового значения, закодированного в команде, к текущему значению PC времени исполнения, которое всегда является адресом следующей команды в памяти.
2. `R_386_32` перемещает ссылку, которая использует абсолютный 32-битовый адрес. При **абсолютной адресации** (absolute addressing) центральный процессор непосредственно использует 32-битовое значение, закодированное в команде как эффективный адрес, без дальнейших модификаций.

7.7.2. Перемещение ссылок на имя

В листинге 7.14 показан псевдокод редактора связей для алгоритма перемещения.

Листинг 7.14. Алгоритм перемещения

```
1  для каждой секции s {
2      для каждого входа перемещения r {
3          refptr = s + r.offset; /* указатель на перемещенную ссылку */
```

```

5      /* переместить ссылку относительно РС */
6      if (r.type == R_386_PC32) {
7          refaddr = ADDR(s) + r.offset; /* ссылки на исполнение */
8          *refptr = (unsigned) (ADDR(r.symbol) + *refptr - refaddr)
9      }
10
11     /* переместить абсолютную ссылку */
12     if (r.type == R_386_32)
13         *refptr = (unsigned) (ADDR(r.symbol) + *refptr);
14     }
15 }
```

Строки 1 и 2 исполняются в цикле для каждой секции *s* и каждого входа перемещения *r*, ассоциированного с каждой секцией. Для конкретности предположим, что каждая секция *s* представляет собой массив байтов, и что каждый вход перемещения *r* представляет собой структуру типа `Elf32_Rel`, определенную в листинге 7.13. Предположим также, что к тому времени, когда начнет исполняться алгоритм, редактор связей уже выбрал адреса времени исполнения для каждой секции (обозначено `ADDR (s)`) и каждого имени (обозначено `ADDR (r.symbol)`). В строке 3 вычисляется адрес в массиве *s* 4-байтовых ссылок, которые должны быть перемещены. Если эти ссылки используют относительную адресацию по РС, то перемещение будет выполнено в строках 5—9. Если ссылка использует абсолютную адресацию, то перемещение будет выполнено в строках 11—13.

Перемещение ссылок с относительной адресацией по РС

Вспомните, что в нашем действующем примере листинга 7.1 подпрограмма `main` в секции `.text` объектного модуля `main.o` вызывает подпрограмму `swap`, приютившуюся в `swap.o`. Вот дизассемблированный листинг команды вызова, сгенерированный утилитой GNU `objdump`:

```

6: e8 fc ff ff ff  call 7 <main+0x7> swap();
7: R_386_PC32 swap    вход перемещения
```

Из этого листинга видно, что команда вызова `call` начинается в секции со смещением `0x6` и состоит из однобайтового кода операции `0xe8`, за которым следует 32-битовая ссылка `0xfffffffffc` (десятичное число `-4`), которая хранится в памяти с прямым порядком байтов. Там также имеется вход перемещения для данной ссылки, показанный в следующей строке. (Вспомните, что входы перемещения и команды фактически хранятся в различных секциях объектного файла. Для удобства утилиты `objdump` показывает их вместе.) Вход перемещения *r* состоит из трех полей:

```

r.offset = 0x7
r.symbol = swap
r.type   = R_386_PC32
```

Эти области сообщают редактору связей, что 32-битовую ссылку относительно РС, расположенную со смещением `0x7`, во время исполнения следует изменить так, чтобы

она указывала на подпрограмму swap. Теперь предположим, что редактор связей определил, что

```
ADDR(s) = ADDR(.text) = 0x80483b4
```

и

```
ADDR(r.symbol) = ADDR(swap) = 0x80483c8
```

Применяя алгоритм листинга 7.14, редактор связей прежде всего вычисляет адрес времени исполнения данной ссылки (строка 7 листинга):

```
refaddr = ADDR(s) + r.offset
        = 0x80483b4 + 0x7
        = 0x80483bb
```

Затем он корректирует ссылку, изменяя ее текущее значение (-4) на 0x9 так, чтобы она указывала на подпрограмму swap во время исполнения (строка 8 листинга 7.14):

```
*refptr = (unsigned) (ADDR(r.symbol) + *refptr - refaddr)
        = (unsigned) (0x80483c8 + (-4) - 0x80483bb)
        = (unsigned) (0x9)
```

В получившемся в результате исполняемом объектном файле команда call имеет следующую переместимую форму:

```
80483ba: e8 09 00 00 00    call  80483c8 <swap>    swap();
```

Во время исполнения команда вызова будет сохранена по адресу 0x80483ba. Когда центральный процессор исполняет команду вызова, РС содержит значение 0x80483bf, представляющее собой адрес команды, непосредственно следующей за командой вызова. Чтобы исполнять эту команду, центральный процессор проделывает следующие шаги:

1. Затолкнуть РС в стек.
2. РС \leftarrow РС + 0x9 = 0x80483bf + 0x9 = 0x80483c8

Таким образом, следующая исполняемая команда — это первая команда процедуры swap, чего мы и добивались!

Вы можете задаться вопросом, а почему ассемблер образовал ссылку в команде call с начальным значением -4. Ассемблер использует это значение как постоянное смещение, чтобы учесть тот факт, что РС всегда указывает на команду, следующую за текущей командой. На различных машинах используются команды разных размеров и различные системы машинных кодов, и ассемблеры для разных конкретных машин использовать будут отличающееся смещение. Это эффективный прием, который позволяет редактору связей вслепую перемещать ссылки, будучи в счастливом неведении о системе машинных кодов команд для конкретной машины.

Перемещение абсолютных ссылок

Вспомните, что в листинге 7.2 модуль swap.o инициализирует глобальный указатель bufp0 адресом первого элемента глобального массива buf:

```
int *bufp0 = &buf[0];
```

Поскольку `bufp0` представляет собой объект инициализированных данных, он будет храниться в секции `.data` переместимого объектного модуля `swap.o`. Поскольку он будет инициализирован адресом глобального массива, он должен быть перемещаемым. Вот дизассемблированный листинг секции `.data` из `swap.o`:

```
00000000 <bufp0>:
0: 00 00 00 00      int *bufp0 = &buf[0];
0: R_386_32 buf    вход перемещения
```

Мы видим что секция `.data` содержит единственную 32-битовую ссылку, указатель `bufp0`, который имеет значение `0x0`. Вход перемещения сообщает редактору связей, что она представляет собой 32-битовую абсолютную ссылку, расположенную со смещением `0`, которая должна быть перемещена так, чтобы указывать на имя `buf`. Теперь предположим, что редактор связей определил, что

`ADDR(r.symbol) = ADDR(buf) = 0x8049454`

Редактор связей корректирует строку 13 использования ссылки из алгоритма листинга 7.14:

```
*refptr = (unsigned) (ADDRfr.symbol) + *refptr)
= (unsigned) (0x8049454        + 0)
= (unsigned) (0x8049454)
```

В получившемся в результате исполняемом объектном файле ссылка имеет следующую перемещаемую форму:

```
0804945c <bufp0>:
804945c: 54 94 04 08    перемещено!
```

Другими словами, редактор связей решил, что во время исполнения переменная `bufp0` будет расположена в памяти по адресу `0x804945c` и будет инициализироваться значением `0x8049454`, являющимся адресом времени исполнения массива `buf`.

Секция `.text` в модуле `swap.o` содержит пять абсолютных ссылок, которые перемещены аналогичным способом (см. упр. 7.12). В листингах 7.15 и 7.16 показаны перемещенные секции `.text` и `.data` в итоговом исполняемом объектном файле.

Листинг 7.15. Переместимая секция текста

```
1 080483b4 <main>:
2 80483b5: 55          push  %ebp
3 80483b7: 89 e5       mov   %esp, %ebp
4 80483ba: b3 ec 08    sub   $0x8, %esp
5 80483bf: e8 09 00 00 00 call  80483c8 <swap>      swap()
6 80483c1: 31 c0       xor   %eax, %eax
7 80483c3: 89 ec       mov   %ebp, %esp
8 80483c4: 5d          pop   %ebp
9 80483c5: c3          ret
```

```

10 80483c6: 90          pop
11 80483c7: 90          pop
12 80483b4: 90          pop

13 080483c8 <swap>:
14 80483c8: 55          push %ebp
15 80483c9: 8b 15 5c 94 04 08 mov 0x804945c,%edx Получить *bufp0
16 80483cf: a1 58 94 04 08 mov 0x8049458,%eax Получить buf[1]
17 80483d4: 89 e5        mov %esp,%ebp
18 80483d6: c7 05 48 95 04 08 58 movl $0x8049458,0x8049548
19 80483dd: 94 04 08
20 80483e0: 89 ec        mov %ebp,%esp
21 80483e2: 8b 0a        mov (%edx),%ecx
22 80483e4: 89 02        mov %eax,(%edx)
23 80483e6: a1 48 95 04 08 mov 0x8049548,%eax Получить *bufp1
24 80483eb: 89 08        mov %ecx,(%eax)
25 80483ed: 5d          pop %ebp
26 80483ee: c3          ret

```

Листинг 7.16. Переместимая секция данных

```

1 08049454 <buf>:
2 08049454: 01 00 00 00 02 00 00 00

3 0804945c <bufp0>:
4 0804945c 54 94 04 08     Перемещено!

```

УПРАЖНЕНИЕ 7.4

Это упражнение связано с перемещаемой программой в листинге 7.15.

- Каков шестнадцатеричный адрес перемещенной ссылки на swap в строке 5?
- Каково шестнадцатеричное значение перемещенной ссылки на swap в строке 5?
- Предположим, что редактор связей решил по некоторым причинам определить адрес секции .text как 0x80483b8 вместо 0x80483b4. Каково было бы шестнадцатеричное значение перемещенной ссылки в строке 5 в данном случае?

7.8. Исполняемые объектные файлы

Мы уже видели, каким образом редактор связей объединяет несколько объектных модулей в единый исполняемый объектный файл. Наша программа на языке C, которая начинала свое существование как простая совокупность текстовых ASCII-файлов, была преобразована в отдельный бинарный файл, содержащий всю информацию, необходимую для загрузки программы в память и запуска ее на исполнение. Рис. 7.4 подводит итог всем видам данных в типичном файле исполняемого модуля ELF.

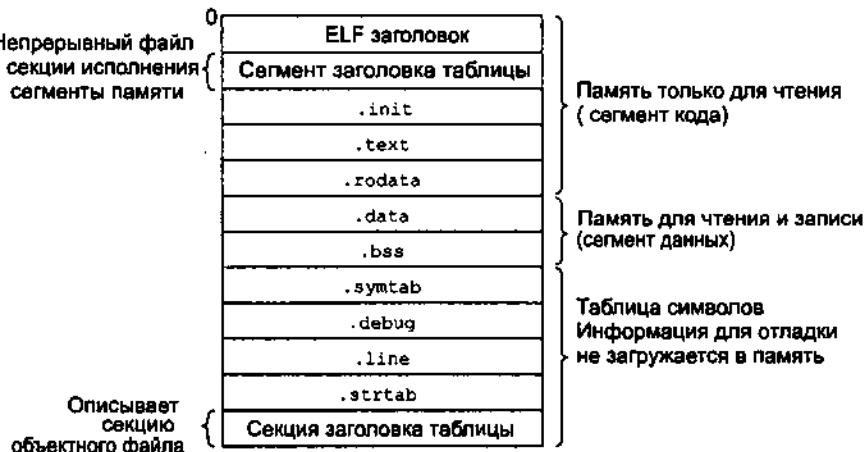


Рис. 7.4. Типичный исполняемый объектный файл ELF

Формат исполняемого объектного файла аналогичен формату переместимого объектного файла. Заголовок ELF описывает полный формат файла. Он также содержит точку входа (entry point) программы, которая представляет собой адрес первой исполняемой команды при вызове программы. Секции .text, .rodata и .data подобны аналогичным секциям переместимого объектного файла, за исключением того, что эти секции уже перемещены в позиции их окончательных адресов памяти времени исполнения. Секция .init определяет небольшую функцию, называемую `_init`, которая вызывается программным кодом инициализации программы. Поскольку исполняемый файл является полностью связанным (включая перемещения), он не нуждается в секции .relo.

Исполняемые модули ELF разработаны таким образом, чтобы они просто загружались в память, для чего смежные отрезки исполняемого файла отображались бы на смежные сегменты памяти. Это отображение описывается с помощью таблицы заголовков сегментов (segment header table). В листинге 7.17 задана таблица заголовков сегментов для нашего примера исполняемого модуля p, как она представляется утилитой objdump.

Листинг 7.17. Заголовки сегментов

Сегмент программного кода только для чтения

```
1 LOAD off 0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
2      filesz 0x00000448 memsz 0x00000448 flags r-x
```

Сегмент данных для чтения/записи

```
3 LOAD off 0x00000448 vaddr 0x08049448 paddr 0x08049448 align 2**12
4      filesz 0x000000e8 memsz 0x00000104 flags rw-
```

В листинге 7.17 принятые следующие условные обозначения: off — смещение файла, vaddr/paddr — виртуальный /физический адрес, align — выравнивание сегмента,

`filesize` — размер сегмента в объектном файле, `memsz` — размер сегмента в памяти, `flags` — права времени исполнения.

Из этой таблицы заголовков сегментов мы видим, что два сегмента памяти будут инициализированы содержимым исполняемого объектного файла. Строки 1 и 2 говорят нам, что первый сегмент (сегмент программного кода) выровнен по границе 4 Кбайт, имеет доступ для чтения и исполнения, стартует в памяти по адресу 0x08048000, имеет общий размер памяти 0x448 байтов и будет инициализирован первыми 0x448 байтами исполняемого объектного файла, которые включают заголовок ELF, таблицу заголовков сегментов и секций `.init`, `.text` и `.rodata`.

Строки 3 и 4 говорят нам, что второй сегмент (сегмент данных) будет выровнен по границе 4 Кбайт и имеет доступ для чтения и записи. Он начинается в памяти по адресу 0x08049448, занимает в памяти 0x104 байтов и будет инициализирован 0xe8 байтами, начинающимися в файле со смещением 0x448, что в данном случае соответствует началу секции `.data`. Остальные байты в этом сегменте соответствуют данным типа `.bss`, которые будут инициализированы нулевым значением во время исполнения.

7.9. Загрузка исполняемых объектных файлов

Чтобы запустить исполняемый объектный файл `r`, мы можем набрать на клавиатуре его имя в командной строке командного процессора Unix:

```
unix> ./r
```

Поскольку `r` не соответствует никакой встроенной команде командного процессора, командный процессор предполагает, что `r` — это исполняемый объектный файл, который он запускает путем вызова некоторого резидентного в памяти программного кода операционной системы, называемого загрузчиком. Каждая Unix-программа может запустить загрузчик путем вызова функции `execve`, которую мы опишем подробно в разд. 8.4.6. Загрузчик копирует программный код и данные из диска в исполняемый объектный файл в памяти и затем запускает программу, передавая управление ее первой команде, или в точку входа. Этот процесс копирования программы в память и последующего исполнения известен как загрузка.

Каждая Unix-программа имеет схему использования памяти времени исполнения, подобную изображенной на рис. 7.5. В Linux-системах сегмент программного кода всегда начинается с адреса 0x08048000. Сегмент данных следует за следующим после программного кода выровненным на 4 Кбайт адресом. Динамическая память (heap) следует за первым выровненным на 4 Кбайт адресом после сегмента для чтения и записи, она увеличивается по мере поступления вызовов библиотечной функции `malloc` (`malloc` и динамическую память мы подробно опишем в разд. 10.9). Сегмент, начинающийся с адреса 0x40000000, зарезервирован для разделяемых библиотек. Пользовательский стек всегда начинается с адреса 0xbfffffff и уменьшается (к более низким адресам памяти). Сегмент, начинающийся выше стека с адреса

0xc0000000, зарезервирован для программного кода и данных резидентной части операционной системы, известной как ядро (kernel).



Рис. 7.5. Схема использования памяти времени исполнения в Linux

После того как загрузчик приступает к работе, он создает образ памяти по схеме, показанной на рис. 7.5. Руководствуясь таблицей заголовков сегментов в исполняемом файле, он копирует порции исполняемого файла в сегменты программного кода и данных. Далее загрузчик передает управление на точку входа в программу, которая всегда является адресом под именем `_start`. Программный код запуска (`startup code`), начинающийся с адреса `_start`, определен в объектном файле `crt1.o`, и он один и тот же для всех программ на языке C. В листинге 7.18 показана характерная последовательность вызовов из кода запуска. После вызова процедуры инициализации из секций `.text` и `.init` код запуска вызывает процедуру `atexit`, которая дополняет список процедур, вызываемых при вызове приложением функции `exit`. Функция `exit` запускает функции, зарегистрированные как `atexit`, и затем возвращает управление в операционную систему, вызывая `_exit`. Далее программный код запуска вызывает процедуру `main` приложения, которая начинает выполнять наш программный код, написанный на языке C. Когда приложение доходит до исполнения оператора возврата, программный код запуска вызывает процедуру `_exit`, которая и возвращает управление операционной системе.

Листинг 7.18. Псевдокод процедуры запуска

```

1 0x080480c0 <_start>: /* точка входа в .text */
2  call libc_init_first /* код запуска в .text */

```

```

3 call init          /* код запуска в .init */
4 call atexit        /* код запуска в .text */
5 call main          /* главная процедура в приложении */
6 call exit          /* возвращает управление в OS */
7 /* управление никогда не достигает этой точки */

```

Как же в действительности работают загрузчики

Наше описание загрузки концептуально корректно, но мы преднамеренно сделали его неполным. Чтобы понять, каким образом в действительности происходит загрузка, следует освоить понятия процессов, виртуальной памяти и распределения памяти, которые мы еще не обсуждали. Позже, когда мы познакомимся с этими понятиями в главах 8 и 10, мы вновь возвратимся к вопросу о загрузке и постепенно откроем вам эту тайну.

Для нетерпеливого читателя здесь приведены предварительные сведения о том, каким образом работает реальный загрузчик. Каждая программа в системе Unix исполняется в контексте процесса в своем собственном виртуальном адресном пространстве. Когда командный процессор запускает программу, родительский процесс ветвится, порождая дочерний процесс, который представляет собой дубликат родителя. Порожденный дочерний процесс через системный вызов `execve` вызывает загрузчик. Загрузчик удаляет существующие виртуальные сегменты памяти дочернего процесса и создает новый комплект сегментов программного кода, данных, кучи и стека. Новые сегменты стека и кучи инициализируются нулевым значением. Новые сегменты программного кода и данных инициализируются значением содержимого исполняемого файла, путем отображения страниц в виртуальном адресном пространстве на участки исполняемого файла размером в страницу. Наконец, загрузчик передает управление на адрес `_start`, что, в конечном счете, вызывает процедуру `main` данного приложения. Кроме некоторой информации из заголовка, в течение загрузки не производится никакого копирования данных из диска в память. Копирование будет отсрочено до тех пор, когда центральный процессор сошлеется на отображенную виртуальную страницу, и в этот момент операционная система автоматически передаст страницу из диска в память, используя свой механизм подкачки страниц.

УПРАЖНЕНИЕ 7.5

1. Почему каждая С-программа требует наличия процедуры, называемой `main`?
2. Задавались ли вы когда-либо вопросом, почему в языке С процедура `main` может завершаться вызовом `exit`, оператором `return`, а в случае отсутствия и того, и другого все же завершает свою работу должным образом? Объясните.

7.10. Динамическое связывание с разделяемыми библиотеками

Статические библиотеки, которые мы изучали в разд. 7.6.2, вызвали большое количество вопросов, связанных с наличием большого набора функций, доступных прикладным программам. Однако статические библиотеки все же имеют некоторые су-

щественные неудобства. Статические библиотеки, как и все остальное программное обеспечение, должны периодически обслуживаться и модифицироваться. Если прикладной программист хочет использовать самую последнюю версию библиотеки, он должен так или иначе быть в курсе дела, какие библиотеки модифицированы, и затем явно выполнить повторное редактирование связей в своих программах, использующих модифицированные библиотеки.

Еще одна проблема состоит в том, что почти каждая C-программа использует стандартные функции ввода-вывода, такие как `printf` и `scanf`. Во время исполнения программный код этих функций будет дублироваться в текстовом сегменте каждого исполняющегося процесса. Типичной является ситуация, когда в системе будет исполняться 50—100 процессов, и это может представлять существенные затраты дефицитных системных ресурсов памяти. Интересное свойство памяти состоит в том, что она будет всегда дефицитным ресурсом, независимо от того, сколько ее имеется в системе. Это свойство является общим для дискового пространства и кухонных мусорных ведер.

Разделяемые библиотеки (shared library) — одна из последних новинок, которая создана с целью устранения недостатков статических библиотек. Разделяемая библиотека — это объектный модуль, который во время исполнения может быть загружен по произвольному адресу памяти и связан с программой в памяти. Этот процесс известен как *динамическое связывание (dynamic linking)*, и он выполняется программой, называемой *динамическим загрузчиком (dynamic linker)*.

Разделяемые библиотеки являются также совместно используемыми объектами (*shared objects*) и в Unix-системах обычно снабжаются расширением файла `.so`. Microsoft интенсивно используют разделяемые библиотеки, которые они называют *DLL (Dynamic Link Libraries, динамически связанные библиотеки)*.

Разделяемые библиотеки могут разделяться двумя различными способами. Во-первых, в любой заданной файловой системе имеется точно один файл `.so` для определенной библиотеки. Программный код и данные в этом файле разделены между всеми исполняемыми объектными файлами, которые ссылаются на данную библиотеку, в отличие от содержимого статических библиотек, которое скопировано и встроено в исполняемые файлы, ссылающиеся на них. Во вторых, единственная копия секции `.text` разделяемой библиотеки в памяти может быть разделена между различными исполняющимися процессами. Мы рассмотрим этот вопрос более подробно, после того, как в главе 10 изучим виртуальную память.

На рис. 7.6 подводится итог изучения процесса динамического связывания на примере листинга 7.12. Чтобы построить разделяемую библиотеку `libvector.so` векторных арифметических подпрограмм нашего примера (листинг 7.11), мы должны вызвать диспетчер компилятора с помощью следующей специальной директивы редактору связей:

```
unix> gcc -shared -fPIC -o libvector.so addvec.c multvec.c
```

Флаг `-fPIC` предписывает компилятору, чтобы тот сгенерировал позиционно-независимый программный код. Флаг `-shared` предписывает редактору связей, чтобы тот создал разделяемый объектный файл.

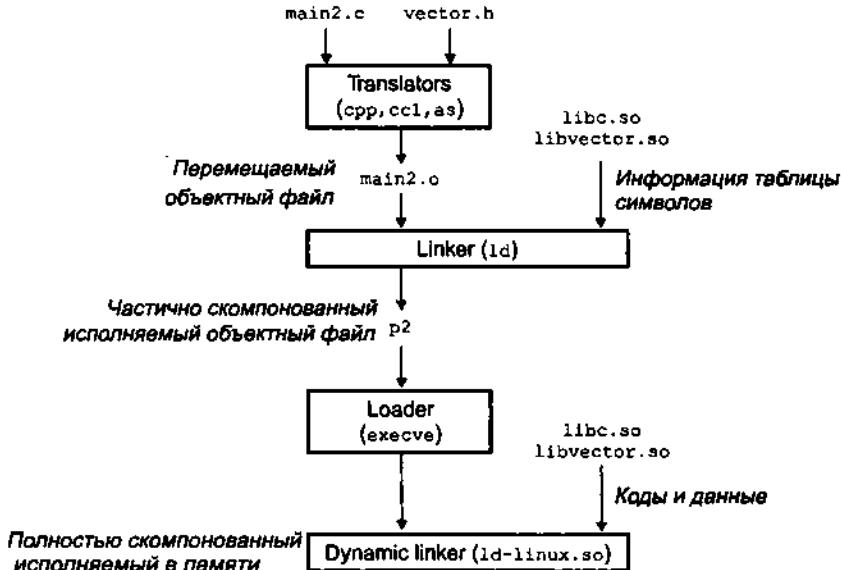


Рис. 7.6. Процесс динамического связывания

Как только мы создадим библиотеку, мы сможем компоновать ее с нашей программой из примера в листинге 7.12:

```
unix> gcc -o p2 main2.c ./libvector.so
```

Здесь создается исполняемый объектный файл p2 в форме, которая позволяет редактировать связи с libvector.so во время исполнения. Основная идея — это выполнять некоторую часть редактирования связей статически после того, как будет образован исполняемый файл, и затем завершать процесс связывания динамически после того, как программа будет загружена.

Важно понять, что в этот момент ни секции программного кода, ни секции данных из libvector.so фактически не скопированы в исполняемый файл p2. Вместо этого редактор связей копирует некоторую информацию из таблицы перемещений и таблицы имен, которая позволит разрешить ссылки на код и данные в libvector.so во время исполнения.

Когда загрузчик загружает и затем запускает исполняемый модуль p2, он загружает частично скомпонованный исполняемый модуль p2, используя методы, рассмотренные в разд. 7.9. Затем он предупреждает, что p2 содержит секцию .interp, где содержится составное имя файла динамического редактора связей, который представляет собой собственно разделяемый объект. Вместо того чтобы передавать управление приложению, как это делалось бы обычно, загрузчик загружает и запускает динамический редактор связей.

Затем динамический редактор связей завершает задачу компоновки, выполняя следующие действия:

- перемещение текста и данных из libc.so в некоторый сегмент памяти. В системах IA32 под управлением Linux разделяемые библиотеки загружены в область, начинаяющуюся с адреса 0x40000000 (см. рис. 7.5);
- перемещение текста и данных libvector.so в другой сегмент памяти;
- перемещение каждой ссылки в p2 на имена, определенные в libc.so и libvector.so.

Наконец, динамический редактор связей передает управление приложению. С этого момента местоположения разделяемых библиотек установлены и не изменяются в течение исполнения программы.

7.11. Загрузка и связывание с разделяемыми библиотеками из приложений

Вплоть до этого момента мы рассматривали сценарий, где динамический загрузчик загружает и связывает разделяемые библиотеки в процессе загрузки приложения, непосредственно перед тем, как оно станет исполняться. Однако, возможно также, чтобы приложение запросило динамический загрузчик загрузить и отредактировать связи с произвольными разделяемыми библиотеками во время исполнения этого приложения без того, чтобы редактировать связи в данном приложении с этими библиотеками во время компиляции.

Динамическое связывание представляет собой мощное и полезное средство. Вот только некоторые примеры из практики:

- Распространение программного обеспечения. Разработчики приложений Microsoft Windows часто используют разделяемые библиотеки для распространения программных модернизаций. Они генерируют новую копию разделяемой библиотеки, после чего пользователи могут загрузить и использовать ее как замену для текущей версии. Когда впоследствии они будут запускать свои приложения, эти приложения автоматически будут связаны и загружены с новой разделяемой библиотекой.
- Построение высокопроизводительных Web-серверов. Многие Web-серверы имеют динамическое содержимое, такое как персонифицированные Web-страницы, текущие остатки на счете и рекламные объявления. Ранние Web-серверы создавали динамическое содержимое с помощью функций fork и execve, используя порожденный процесс. Но современные высокопроизводительные Web-серверы используют намного более эффективный и совершенный подход, построенный на основе динамического связывания. Идея состоит в том, чтобы упаковывать в разделяемую библиотеку каждую функцию, которая генерирует динамическое содержимое. Когда запрос поступает в Web-браузер, сервер динамически загружает соответствующую функцию и редактирует связи, а затем вызывает ее непосредственно вместо того, чтобы использовать fork и execve для запуска на исполнение этой функции в контексте порожденного дочернего процесса. Функция остается в адресном пространстве сервера (помещена в кэш), поэтому последующие запросы

могут быть обработаны путем простого вызова данной функции. Это может оказать существенное влияние на эффективность работы сайта. Более того, существующие функции могут быть модифицированы, а новые функции можно добавлять во время исполнения, не останавливая сервера.

Системы класса Unix, например Linux и Solaris, предоставляют простой интерфейс с динамическим загрузчиком, который позволяет прикладным программам загружать и связывать разделяемые библиотеки во время исполнения.

```
#include <dlfcn.h>
void *dlopen(const char *filename, int flag);
```

Функция возвращает указатель на дескриптор, если все в порядке, и NULL — в случае ошибки.

Функция `dlopen` загружает и редактирует связи с разделяемой библиотекой `filename`. Внешние имена в `filename` разрешены с использованием библиотек, ранее открытых с флагом `RTLD_GLOBAL`. Если текущий исполняемый модуль был компилирован с флагом `-rdynamic`, то его глобальные имена также доступны для разрешения имен. Аргумент `flag` должен быть либо `RTLD_NOW`, что говорит редактору связей о том, что следует разрешать ссылки на внешние имена непосредственно, либо `RTLD_LAZY`, что дает указания редактору связей отсрочить разрешение ссылок до тех пор, когда будет исполняться программный код из данной библиотеки. Оба эти значения могут быть связаны логической операцией ИЛИ вместе с флагом `RTLD_GLOBAL`.

```
#include <dlfcn.h>
void *dlsym(void *handle, char *symbol);
```

Функция `dlsym` принимает в качестве аргумента дескриптор `handle` предварительно открытой разделяемой библиотеки и идентификатор `symbol`, и возвращает адрес, соответствующий этому идентификатору, если этот адрес существует, или NULL — в противном случае.

```
#include <dlfcn.h>
int dlclose (void *handle);
```

Функция возвращает 0, если все в порядке, и -1 — в случае ошибки. Функция `dlclose` выгружает разделяемую библиотеку, если никакие другие разделяемые библиотеки уже не используют ее.

```
#include <dlfcn.h>
const char *dlerror(void);
```

Функция `dlerror` возвращает строку, описывающую самую последнюю ошибку, которая произошла в результате вызова `dlopen`, `dlsym` или `dlclose`, либо NULL — если ошибки не было.

В листинге 7.19 показано, каким образом можно использовать этот интерфейс для того, чтобы динамически связать нашу разделяемую библиотеку `libvector.so` (лис-

тинг 7.11) и затем вызвать процедуру addvec. Чтобы откомпилировать программу, мы должны вызвать GCC следующим образом:

Листинг 7.19. Редактирование связей с разделяемой библиотекой

```
unix> gcc -rdynamic -O2 -o p3 main3.c -ldl
1 #include <stdio.h>
2 #include <dlfcn.h>
3
4 int x[2] = {1, 2};
5 int y[2] = {3, 4};
6 int z[2];
7
8 int main()
9 {
10     void *handle;
11     void (*addvec)(int *, int *, int *, int);
12     char *error;
13
14     /* динамически загружает разделяемую библиотеку, которая содержит
15        addvec() */
16     handle = dlopen("./libvector.so", RTLD_LAZY);
17     if (!handle) {
18         fprintf(stderr, "%s\n", dlerror());
19         exit(1);
20     }
21     /* получает указатель на функцию addvec(), которую мы только что
22        загрузили */
23     ddvec = dlsym(handle, "addvec");
24     if ((error = dlerror()) != NULL) {
25         fprintf(stderr, "%s\n", error);
26         exit(1);
27     }
28     /* Теперь мы можем вызывать addvec() точно так же, как любую другую
29        функцию */
30     addvec(x, y, z, 2);
31     printf("z = [%d %d]\n", z[0], z[1]);
32
33     /* выгрузить разделяемую библиотеку */
34     if (dlclose(handle) < 0) {
35         fprintf(stderr, "%s\n", dlerror());
36     }
37     return 0;
38 }
```

Разделяемые библиотеки и интерфейс языка Java

Язык Java определяет стандарт соглашения относительно вызовов, который называется *Java Native Interface* (JNI, собственный интерфейс Java). Этот стандарт допускает вызов "собственных" функций языка C и C++ из программ на Java. Основная идея JNI состоит в том, чтобы компилировать собственную функцию языка C, скажем `foo`, и поместить ее в разделяемую библиотеку, скажем `foo.so`. Когда в процессе исполнения Java-программа попытается вызвать функцию `foo`, интерпретатор Java использует интерфейс `dlopen` (или что-либо подобное), чтобы динамически отредактировать связи и загрузить `foo.so`, а затем вызвать `foo`.

7.12. Переместимый программный код

Основное назначение разделяемых библиотек состоит в том, чтобы позволить нескольким одновременно исполняющимся процессам совместно использовать один и тот же библиотечный код в памяти и таким образом экономить дорогостоящие ресурсы памяти. Но каким же образом несколько процессов могут совместно использовать единственную копию программы? Один из возможных подходов мог бы состоять в том, чтобы каждой разделяемой библиотеке априорно назначать участок адресного пространства и затем требовать, чтобы загрузчик всегда загружал разделяемую библиотеку по этому адресу. Хотя это и выглядит довольно просто, данный подход создает некоторые серьезные проблемы. Такое использование адресного пространства было бы неэффективным, ввиду того, что некоторые его части были бы выделены для использования даже в том случае, если процесс не использует библиотеку. Вторых, таким механизмом было бы трудно управлять. Мы не могли бы гарантировать, что никакие участки не перекрываются. Каждый раз, когда библиотека будет модифицирована, мы должны будем удостовериться, что она все еще находится в пределах назначенного ей участка памяти. И в случае отрицательного ответа мы должны были бы найти для нее новый участок. А если бы мы создавали новую библиотеку, мы должны были бы найти место и для нее. Через какое-то время мы получили бы сотни библиотек и их версий в системе, и было бы трудно предотвратить фрагментацию адресного пространства на множество маленьких и не пригодных для использования "дырок". Хуже того, прикрепление библиотек к памяти было бы различным для разных систем, что вызывало бы еще большую головную боль.

Лучший подход состоит в том, чтобы компилировать библиотечный код таким образом, чтобы он мог быть загружен и исполнен по любому адресу, без необходимости его модификации с помощью редактора связей. Такой программный код называется *позиционно-независимым программным кодом* (PIC, position-independent code). Пользователи предписывают системе компиляции GNU генерировать программный PIC-код путем задания опции `-fPIC` к команде `gcc`.

В системе IA32 обращения к процедурам в одном и том же объектном модуле не требует никакой специальной подготовки, поскольку ссылки отсчитываются относительно PC, при известных смещениях, и таким образом уже являются PIC (см. упр. 7.4). Но обращения к внешне определенным процедурам и ссылкам на глобальные переменные обычно не являются PIC, поскольку требуют перемещения во время редактирования связей.

7.12.1. Ссылки на данные PIC

Компиляторы генерируют ссылки PIC на глобальные переменные, используя следующий интересный факт: независимо от того, в какое место памяти мы загружаем объектный модуль (включая разделяемые объектные модули), сегмент данных всегда будет размещен непосредственно после сегмента программного кода. Таким образом, расстояние между любой командой в сегменте программного кода и любой переменной в сегменте данных во время исполнения постоянно, независимо от абсолютных адресов в памяти сегментов программного кода и данных.

Чтобы использовать этот факт, компилятор в начале сегмента данных создает таблицу, которая называется *глобальной таблицей смещений* (global offset table, GOT). GOT содержит входы для каждого глобального объекта данных, на который имеется ссылка из объектного модуля. Компилятор также генерирует запись перемещения для каждого входа в GOT. Во время загрузки динамический загрузчик перемещает каждый вход в GOT таким образом, чтобы он содержал соответствующий абсолютный адрес. Каждый объектный модуль, в котором есть ссылки на глобальные данные, имеет свою собственную GOT.

Во время исполнения ссылки на каждую глобальную переменную будут косвенными, через GOT, с использованием такого фрагмента программного кода:

```
call L1
L1: popl %ebx;           # ebx содержит текущий PC
    addl $VAROFF, %ebx   # ebx указывает на вход в GOT для переменной
    movl (%ebx), %eax   # косвенная ссылка через GOT
    movl (%eax), %eax
```

В этом симпатичном фрагменте программного кода обращение к L1 заталкивает в стек адрес возврата (который оказывается адресом команды popl). Затем команда popl выталкивает этот адрес в %ebx. Суммарный эффект выполнения этих двух команд состоит в том, что значение PC переносится в регистр %ebx.

Команда addl добавляет постоянное смещение к значению в %ebx таким образом, чтобы оно указывало на соответствующий вход в GOT, который содержит абсолютный адрес элемента данных. С этого момента на глобальную переменную можно ссылаться косвенно через вход GOT, содержащийся в %ebx. В этом примере две команды movl загружают содержимое глобальной переменной (косвенно через GOT) в регистр %eax.

Работа с программным кодом PIC имеет свои неудобства. Ссылка на каждую глобальную переменную теперь требует пяти команд вместо одной, да еще дополнительной ссылки на адрес памяти в GOT. Кроме того, программный код PIC использует дополнительный регистр для хранения адреса входа GOT. В машинах с большим набором регистров это не составляет главной проблемы. В бедной регистрами системе IA32, однако, потеря даже единственного регистра может вызвать необходимость переброски данных из регистра в стек.

7.12.2. Вызовы функций PIC

Конечно, для программного кода PIC можно было бы использовать тот же самый прием для разрешения вызовов внешних процедур:

```
call L1
L1: popl %ebx;           # ebx содержит текущий PC
    addl $PROCOFF,%ebx   # ebx указывает на вход в GOT для процедуры
    call (%ebx)          # косвенный вызов через GOT
```

Однако этот подход потребовал бы трех дополнительных команд для каждого вызова процедуры во время исполнения. Вместо этого, системы компиляции ELF применяют интересный прием, так называемое *позднее связывание* (*lazy binding*), которое отсрочивает связывание адресов процедуры до тех пор, когда процедура вызывается в первый раз. При этом имеется не очевидный перерасход времени исполнения при первом вызове процедуры, но каждый последующий вызов требует всего лишь единственной команды и ссылки на адрес памяти для косвенного обращения.

Позднее связывание реализовано с помощью компактного, но все же довольно сложного взаимодействия между двумя структурами данных: GOT и *таблицей связей процедуры* (*procedure linkage table*, PLT). Если объектный модуль вызывает какую-либо функцию, которая принадлежит разделяемой библиотеке, то он имеет свои собственные GOT и PLT. GOT — это часть секции .data, а PLT — это часть секции .text.

В табл. 7.3 показан формат GOT для программы примера main2.o из листинга 7.12. Первые три входа GOT — это специальные входы: GOT [0] содержит адрес сегмента .dynamic, в котором находится информация, использующаяся динамическим загрузчиком для связывания таких адресов процедуры, как адрес информации перемещения и таблицы имен. GOT [1] содержит некоторую информацию, которая определяет данный модуль. GOT [2] содержит точку входа в программный код позднего связывания динамического редактора связей (загрузчика).

Таблица 7.3. Глобальная таблица смещений

Адрес	Вход	Содержимое	Описание
08049674	GOT[0]	0804969c	Адрес секции .dynamic
08049678	GOT[1]	4000a9f8	Идентифицирующая информация для редактора связей
0804967c	GOT[2]	4000596f	Точка входа в динамический редактор связей
08049680	GOT[3]	0804845a	Адрес pushl в PLT [1] (printf)
08049684	GOT[4]	0804846a	Адрес pushl в PLT [2] (addvec)

Каждая процедура, которая определена в разделяемом объекте и вызывается из main2.o, получает вход в GOT, начиная с входа GOT[3]. Для программы нашего при-

мера мы показали входы GOT для printf, который определен в libc.so и addvec, который определен в libvector.so.

В листинге 7.20 находится PLT для программы p2 нашего примера. PLT — это массив 16-байтовых входов. Первый вход, PLT[0], представляет собой специальный вход, который передает управление динамическому загрузчику. Каждая вызываемая процедура имеет вход в PLT, начиная с PLT[1]. На рисунке PLT[1] соответствует printf, а PLT[2] — addvec.

Листинг 7.20. PLT для программы p2

```

PLT[0]
08048444: ff 35 78 96 04 08    pushl  0x8049678  # втолкнуть в стек &GOT(1)
0804844a: ff 25 7c 96 04 08    jmp     *0x804967c  # перейти на *GOT[2]
                                (linker)

08048450: 00 00                # заполнение пробелами
08048452: 00 00                # заполнение пробелами

PLT[1] <printf>
08048454: ff 25 80 96 04 08    jmp     *0x8049680  # перейти на *GOT(3)
0804845a: 68 00 00 00 00        pushl  $0x0      # ID для printf
0804845f: e9 e0 ff ff ff        jmp     8048444   # перейти на PLT[0]

PLT[2] <addvec>
08048464: ff 25 84 96 04 08    jmp     *0x8049684  # перейти на *GOT(4)
0804846a: 68 08 00 00 00        pushl  $0x8      # ID для addvec
0804846f: e9 d0 ff ff ff        jmp     8048444   # перейти на PLT[0]

< другие входы PLT >
```

В исходном положении, после того как программа будет динамически связана и начнет исполняться, процедуры printf и addvec связаны с первой командой соответствующего им входа в PLT. Например, вызов addvec имеет форму

```
80485bb: e8 a4 fe ff ff        call    8048464 <addvec>
```

Когда addvec вызывается в первый раз, управление передается первой команде в PLT[2], которая выполняет косвенный переход через GOT [4]. В исходном положении каждый вход GOT содержит адрес входа pushl в соответствующем входе PLT. Поэтому косвенный переход в PLT просто возвращает управление на следующую команду в PLT[2]. Эта команда заталкивает в стек ID для имени addvec. Последние команды вызывают переход к PLT[0], где в стек заталкивается другое слово идентифицирующей информации из GOT[1], и затем косвенно через GOT[2] управление передается динамическому редактору связей. Динамический редактор связей использует два входа стека для определения адреса addvec, записывает в GOT[4] этот адрес, и передает управление на addvec.

Когда в этой программе addvec вызывается в следующий раз, управление передается к PLT [2], как и ранее. Но в этот раз косвенный переход через GOT[4] передает управление на addvec. С этого момента единственные дополнительные издержки — это ссылка на адрес памяти для косвенного перехода.

7.13. Средства управления объектными файлами

Имеются несколько утилит, доступных в системе Unix, которые помогут вам освоить понятия, связанные с объектными файлами и научиться управлять ими. В частности, особенно полезным будет пакет GNU binutils, который исполняется на любой Unix-платформе.

- AR — создает статические библиотеки, вставляет, удаляет, составляет список элементов и извлекает их элементы.
- STRINGS — составляет список всех пригодных для вывода на печать строк, содержащихся в объектном файле.
- STRIP — удаляет таблицу имен из объектного файла.
- NM — составляет список имен, определенных в таблице имен объектного файла.
- SIZE — составляет список имен и перечисляет размеры секций в объектном файле.
- READELF — показывает полную структуру объектного файла, включая всю информацию, закодированную в заголовке ELF.
- OBJDUMP — основа всех бинарных утилит. Может отобразить всю информацию в объектном файле. Ее самая используемая функция — это деассемблирование бинарных кодов в секции .text.

Системы Unix, кроме того, предоставляют программу ldd для управления разделяемыми библиотеками.

- LDD — составляет список разделяемых библиотек, которые необходимы исполняемому файлу во время счета.

7.14. Резюме

Редактирование связей может быть выполнено во время компиляции с помощью статического редактора связей, во время загрузки и во время исполнения с помощью динамического загрузчика. Редакторы связей работают с бинарными файлами, называемыми *объектными файлами*, которые бывают трех видов: переместимые, исполняемые и разделяемые. Переместимые объектные файлы собираются с помощью статических редакторов связей в исполняемый объектный файл, который может быть загружен в память и запущен на исполнение. Разделяемые объектные файлы (разделяемые библиотеки) связываются и загружаются с помощью динамических загрузчиков во время исполнения, — либо неявно, если вызывающая программа загружается и начинает исполняться, либо по требованию, когда функцию вызывает программа из библиотеки dlopen.

Редакторы связей выполняют две основные задачи — разрешение ссылок, при котором каждое глобальное имя в объектном файле будет привязано к уникальному определению, и перемещение, при котором для каждого имени будет определен окончательный адрес памяти, а ссылки на эти объекты будут модифицированы.

Статические редакторы связей вызываются с помощью диспетчера компилятора, такого как gcc. Они компонуют несколько различных переместимых объектных файлов в единственный исполняемый объектный файл. В различных объектных файлах может определяться одно и то же имя, и правила, которые используются редакторами связей для разрешения этих различных определений по умолчанию, могут привести к коварным ошибкам в пользовательских программах.

Несколько объектных файлов могут быть соединены в единую статическую библиотеку. Библиотеки используются редакторами связей для разрешения ссылок на имя в других объектных модулях. Последовательный просмотр слева направо, который используют многие редакторы связей для разрешения ссылок на имя, является еще одним источником трудно объяснимых ошибок времени редактирования связей.

Загрузчики отображают содержимое исполняемых файлов в память и запускают на счет программы. Редакторы связей могут, кроме того, создавать частично скомпонованные исполняемые объектные файлы с неразрешенными ссылками на процедуры и данные, определенные в разделяемой библиотеке. Во время загрузки загрузчик отображает частично скомпонованный исполняемый файл в память и затем вызывает динамический загрузчик, который завершает задачу связывания путем загрузки разделяемой библиотеки и перемещения ссылок в программе.

Разделяемые библиотеки, которые скомпилированы как позиционно-независимый программный код, могут быть загружены в любое место адресного пространства и разделены во время исполнения между несколькими процессами. Приложения могут, кроме того, использовать динамический загрузчик во время исполнения, чтобы загрузить, отредактировать связи и обращаться к функциям и данным в разделяемых библиотеках.

Библиографические замечания

Редактирование связей — это довольно слабо документированная тема в литературе по компьютерным системам. Поскольку оно охватывает область знаний по компиляторам, компьютерной архитектуре и операционным системам, редактирование связей требует понимания сущности генерации объектного кода, программирования на машинном языке, реализации программ и виртуальной памяти. Оно не вписывается ни в одну из обычных специальностей в области компьютерных систем, и таким образом оказывается, что классические вопросы в этой области знаний не достаточно хорошо освещены в литературе. Впрочем, монография Левина (Levine) [47] предоставляет серьезное общее введение в предмет. Исходные спецификации для ELF и DWARF (спецификация для содержимого секций .debug и .line) описаны в [35].

Можно отметить некоторое оживление исследовательской и коммерческой активности, связанной с понятием *бинарной трансляции* (binary translation), где разбирается, анализируется и модифицируется содержимое объектного файла. Бинарная трансляция может быть использована для трех различных целей [46]: чтобы эмулировать одну систему на другой системе, чтобы наблюдать поведение программы или чтобы провести системно-зависимую оптимизацию, которая невозможна во время компиля-

ции. Коммерческие программы, такие как VTune, Purify и BoundsChecker, используют бинарную трансляцию, чтобы обеспечить программистов возможностью детального обследования своих программ.

Система Atom [74] предоставляет гибкий механизм для исследования исполняемых объектных файлов и разделяемых библиотек в системе Alpha с помощью случайных функций языка С. Atom использовался для построения несметного числа средств анализа, которые отслеживают вызовы процедур, представляют статистику использования команд и схемы ссылок в памяти (профилирование), моделируют поведение системы памяти и локализуют ошибки обращения к памяти. Etch [66] и EEL [46] позволяют ориентировочно оценить аналогичные возможности на различных платформах. Система Shade [15] использует бинарную трансляцию для профилирования команд. Dynamo [2] и Dyninst [8] предоставляют механизмы для исследования и оптимизации исполняемых модулей в памяти во время исполнения. Смит (Smith) [91] и его коллеги исследовали бинарную трансляцию для профилирования и оптимизации программ.

Задачи для домашнего решения

УПРАЖНЕНИЕ 7.6 ◆

Рассмотрите следующую версию функции swap.c, которая подсчитывает, сколько раз ее вызывали:

```
1 extern int buf[];
2
3 int *bufp0 = &buf[0];
4 static int *bufp1;
5
6 static void incr()
7 {
8     static int count = 0;
9
10    count++;
11 }
12
13 void swap()
14 {
15     int temp;
16
17     incr();
18     bufp1 = &buf[1];
19     temp = *bufp0;
20     *bufp0 = *bufp1;
21     *bufp1 = temp;
22 }
```

Для каждого имени, которое определено и на которое есть ссылка в swap.o, укажите, будет ли оно иметь вход в таблице имен в секции .symtab в модуле swap.o. Если да, то укажите тот модуль, в котором определено данное имя (swap.o или main.o), тип имени (локальное, глобальное или внешнее), а также секцию (.text, .data или .bss), которую оно занимает в этом модуле.

Имя	Вход в .symtab для swap.o?	Тип имени	Модуль, в котором определено имя	Секция

УПРАЖНЕНИЕ 7.7 ◆

Не изменения никаких идентификаторов переменных, модифицируйте bar5.c в листинге 7.10 таким образом, чтобы foo5.c выводила на печать правильные значения x и y (т. е. шестнадцатеричные представления целых чисел 15213 и 15212).

УПРАЖНЕНИЕ 7.8 ◆

В этом упражнении REF(x.i) --> DBF(x.k) обозначает, что редактор связей свяжет некоторую ссылку на имя x в модуле i с определением x в модуле k. Используйте эту нотацию для каждого примера, чтобы указать, каким образом редактор связей разрешал бы ссылки на многократно определенное имя в каждом модуле. Для ошибки времени редактирования связей (правило 1) пишите "ERROR". Если редактор связей произвольно выбирает одно из возможных из определений (правило 3), пишите "UNKNOWN".

1.

```
/* Модуль 1 */
int main()
{
}

(a) REF(main.1) --> DBF(    . )

/* Модуль 2 */
static int main=1;
int p2()
{
}

(b) REF(main.2) --> DBF(    . )
```

2.

```
/* Модуль 1 */          /* Модуль 2 */
int main()              double x;
{
}                      int p2()
{
}

(a) REF(x.1) --> DBF( . )
(b) REF(x.2) --> DBF( . )
```

3.

```
/* Модуль 1 */          /* Модуль 2 */
int main()              double x=1.0;
{
}                      int p2()
{
}

(a) REF(x.1) --> DBF( . )
(b) REF(x.2) --> DBF( . )
```

УПРАЖНЕНИЕ 7.9 •

Рассмотрите следующую программу, которая состоит из двух объектных модулей:

```
1 /* foo6.c */
2 void p2(void);
3
4 int main()
5 {
6     p2();
7     return 0;
8 }

1 /* bar6.c */
2 #include <stdio.h>
3
4 char main;
5
6 void p2()
7 {
8     printf("0x%x\n", main);
9 }
```

Если эта программа будет компилирована и запущена на исполнение в системе Linux, она выведет на печать строку "0x55\n" и нормально завершит свою работу, даже при том, что p2 никогда не инициализирует переменную main. Можете ли вы объяснить это?

УПРАЖНЕНИЕ 7.10 •

Пусть а и б обозначают объектные модули или статические библиотеки в текущем каталоге, и пусть а->б обозначает, что а зависит от б, в том смысле, что б определяет имя, на которое имеется ссылка из а. Для каждого из следующих сценариев покажите минимальную (содержащую наименьшее количество объектных файлов и библиотечных аргументов) командную строку, которая позволит статическому редактору связей разрешать все ссылки на имена:

p.o → libx.a → p.o

p.o → libx.a → liby.a and liby.a → libx.a

p.o → libx.a → liby.a → libz.a and liby.a → libx.a → libz.a

УПРАЖНЕНИЕ 7.11 •

Из заголовка сегмента в листинге 7.17 видно, что этот сегмент данных занимает 0x104 байтов в памяти. Однако только первые 0x68 байтов из них принадлежат секции исполняемого файла. Чем вызвано это несоответствие?

УПРАЖНЕНИЕ 7.12 ••

Процедура swap в листинге 7.15 содержит пять перемещенных ссылок. Для каждой перемещенной ссылки дайте номер ее строки в листинге 7.15, адрес памяти времени исполнения и ее значение. Исходный программный код и входы перемещения в модуле swap.o показаны в программном коде далее.

Номер строки	Адрес	Значение

```

1 00000000 <swap>:
2 0: 55          push  %ebp
3 1: 8b 15 00 00 00 00    mov    0x0,%edx      получить *bufp0=&buf[0]
4                                         3: R_386_32   bufp0 вход перемещения
5 7: a1 04 00 00 00    mov    0x4,%eax      получить buf[1]
6                                         8: R_386_32   buf   вход перемещения
7 c: 89 e5          mov    %esp,%ebp
8 e: c7 05 00 00 00 00 04  movl   $0x4,0x0      bufp1 = &buf[1];
9 15: 00 00 00
10                                         10: R_386_32  bufp1 вход перемещения
11                                         14: R_386_32  buf   вход перемещения
12 18: 89 ec          mov    %ebp,%esp

```

```

13 1a: 8b 0a          mov    (%edx),%escx   temp = buf[0];
14 1c: 89 02          mov    %eax,(%edx)  buf[0]=buf[1];*
15 1e: a1 00 00 00 00  mov    0x0,%eax      получить *bufpl=&buf[1]
16                                1f: R_386_32  bufpl  вход перемещения
17 23: 89 08          mov    %escx,(%eax)  buf[1] = temp;
18 25: 5d              pop    %ebp
19 26: c3              ret

```

УПРАЖНЕНИЕ 7.13 ***

Рассмотрите программный код на языке С и соответствующий переместимый объектный модуль далее.

1. Определите, какие команды в .text должны быть модифицированы редактором связей во время перемещения модуля. Для каждой такой команды перечислите информацию в ее входе перемещения: смещение секции, тип перемещения и идентификатор имени.
2. Определите, какие объекты данных в .data должны быть модифицированы редактором связей во время перемещения модуля. Для каждого такого объекта перечислите информацию в его входе перемещения: смещение секции, тип перемещения и идентификатор имени.

Использование свободно распространяемых утилит, таких как objdump, поможет вам выполнить это упражнение.

Фрагмент программного кода на языке С выглядит следующим образом:

```

1 extern int p3(void);
2 int x = 1;
3 int *xp = &x;
4
5 void p2(int y) {
6 }
7
8 void p1() {
9     p2(*xp + p3());
10 }

```

Секция .text переместимого объектного файла представлена далее:

```

1 00000000 <p2>:
2     0: 55          push   %ebp
3     1: 89 e5        mov    %esp,%ebp
4     3: 89 ec        mov    %ebp,%esp
5     5: 5d          pop    %ebp
6     6: c3          ret
7
8 00000008 <p1>:
9     8: 55          push   %ebp
10    9: 89 e5        mov    %esp,%ebp

```

```

10    b: 83 ec 08          sub   $0x8,%esp
11    e: 83 c4 f4          add    $0xffffffff4,%esp
12    11: e8 fc ff ff ff  call   12 <p1+0xa>
13    16: 89 c2             mov    %eax,%edx
14    18: a1 00 00 00 00    mov    0x0,%eax
15    1d: 03 10             add    (%eax),%edx
16    1f: 52                push   %edx
17    20: e8 fc ff ff ff  call   21 <p1+0x19>
18    25: 89 ec             mov    %ebp,%esp
19    27: 5d                pop    %ebp
20    28: c3                ret

```

Секция .data переместимого объектного файла выглядит следующим образом:

```

1 00000000 <x>:
2     0: 01 00 00 00
3 00000004 <xp>:
4     4: 00 00 00 00

```

Рассмотрите программный код на языке С и соответствующий переместимый объектный модуль.

1. Определите, какие команды в .text должны быть модифицированы редактором связей во время перемещения модуля. Для каждой такой команды перечислите информацию в ее входе перемещения: смещение секции, тип перемещения и идентификатор имени.
2. Определите, какие объекты данных в .rodata должны быть модифицированы редактором связей во время перемещения модуля. Для каждого такого объекта перечислите информацию в его входе перемещения: смещение секции, тип перемещения и идентификатор имени.

Использование свободно доступных утилит, таких как, например objdump, поможет вам выполнить это упражнение.

Фрагмент программного кода на языке С выглядит следующим образом:

```

1 int relo3(int val) {
2     switch (val) {
3         case 100:
4             return(val);
5         case 101:
6             return(val+1);
7         case 103: case 104:
8             return(val+3);
9         case 105:
10            return(val+5);
11        default:
12            return(val+6);
13    }
14 }

```

Секция .text переместимого объектного файла представлена далее:

```

1 00000000 <relo3>:
2   0: 55          push  %ebp
3   1: 89 e5       mov    %esp, %ebp
4   3: 8b 45 08   mov    0x8(%ebp), %eax
5   6: 8d 50 9c   lea    0xfffffff9c(%eax), %edx
6   9: 83 fa 05   cmp    $0x5, %edx
7   c: 77 17       ja    25 <relo3+0x25>
8   e: ff 24 95 00 00 00 00 jmp   *0x0(%edx, 4)
9  15: 40          inc    %eax
10 16: eb 10       jmp   28 <relo3+0x28>
11 18: 83 c0 03   add    $0x3, %eax
12 1b: eb 0b       jmp   28 <relo3+0x28>
13 1d: 8d 76 00   lea    0x0(%esi), %esi
14 20: 83 c0 05   add    $0x5, %eax
15 23: eb 03       jmp   28 <relo3+0x28>
16 25: 83 c0 06   add    $0x6, %eax
17 28: 89 ec       mov    %ebp, %esp
18 2a: 5d          pop    %ebp
19 2b: c3          ret

```

Секция .data переместимого объектного файла выглядит следующим образом:

1 Это таблица переходов для оператора switch

2 0000 28000000 15000000 25000000 18000000

4 слова со смещениями
0x0, 0x4, 0x8 и 0xc

3 0010 18000000 20000000

2 слова со смещениями 0x10
и 0x14

УПРАЖНЕНИЕ 7.15 +++

Выполнение следующих заданий поможет вам приобрести опыт использования различных средств для обработки объектных файлов.

1. Сколько объектных файлов содержится в библиотеках libc.a и libm.a в вашей системе?
2. Будут ли отличаться коды исполняемых программных модулей при компиляции в одном случае с командной строкой `gcc -O2`, а в другом со строкой `gcc -2 -g?`
3. Какие разделяемые библиотеки используются в вашей системе?

Решение упражнений

РЕШЕНИЕ УПРАЖНЕНИЯ 7.1

Назначение этого задания состоит в том, чтобы помочь вам ознакомиться с соотношением между именами редактора связей, с одной стороны, и переменными и функ-

циями языка С — с другой. Обратите внимание, что на языке С локальная переменная `temp` не имеет входа в таблице имен.

Имя	Вход <code>swap.o</code> или <code>main.o</code> ?	Тип имени	В каком модуле определено	Секция
<code>buf</code>	да	внешнее	<code>main.o</code>	<code>.data</code>
<code>bufp0</code>	да	глобальное	<code>swap.o</code>	<code>.data</code>
<code>bufp1</code>	да	глобальное	<code>swap.o</code>	<code>.bss</code>
<code>swap</code>	да	глобальное	<code>swap.o</code>	<code>.text</code>
<code>temp</code>	нет	—	—	—

РЕШЕНИЕ УПРАЖНЕНИЯ 7.2

Это задание представляет собой простую отработку приемов, при которой проверяется ваше понимание правил использования редактора связей Unix, когда он разрешает ссылки на глобальные имена, которые определяются в более чем одном модуле. Понимание этих правил может помочь вам избежать некоторых рискованных ошибок при программировании.

1. Редактор связей выбирает строго определенное имя из модуля 1 через слабо определенное имя, принадлежащее модулю 2 (правило 2):

```
REF(main.1) --> DEF(main.1)
REF(main.2) --> DEF(main.1)
```

2. Здесь будет "ERROR", ввиду того, что каждый модуль определяет строго определенное имя `main` (правило 1).

3. Редактор связей выбирает строго определенное имя из модуля 2 через слабо определенное имя, определенное в модуле 1 (правило 2):

```
REF(x.1) --> DEF(x.2)
REF(x.2) --> DEF(x.2)
```

РЕШЕНИЕ УПРАЖНЕНИЯ 7.3

Размещение статических библиотек в командной строке в неправильном порядке является распространенным источником ошибок времени редактирования связей, которые ставят в тупик многих программистов. Однако, как только вы разберетесь с тем, каким образом редактор связей использует статические библиотеки для разрешения ссылок, все станет на свои места. Это небольшое упражнение служит проверкой вашего понимания сущности этой идеи:

1. `gcc p.o libx.a`
2. `gcc p.o libx.a liby.a`
3. `gcc p.o libx.a liby.a libx.a`

РЕШЕНИЕ УПРАЖНЕНИЯ 7.4

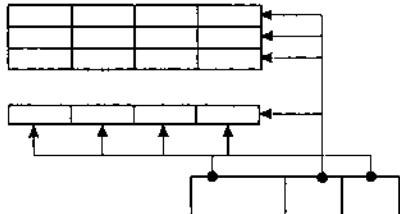
Это задание связано с листингом деассемблирования 7.15. Наша цель здесь состоит в том, чтобы дать вам некоторую практику чтения листингов деассемблирования и проконтролировать ваше представление об относительной адресации по РС.

1. Шестнадцатеричный адрес перемещенной ссылки в строке 5 есть 0x80483bb.
2. Шестнадцатеричное значение перемещенной ссылки в строке 5 есть 0x9. Вспомните, что листинг дизассемблирования показывает значение ссылки в прямом порядке следования байтов.
3. Ключевым моментом здесь является то, что независимо от того, в какое место редактор связей помещает секцию .text, расстояние между ссылкой и функцией swap будет всегда одним и тем же. Таким образом, ввиду того, что ссылка представляет собой адрес относительно РС, ее значение будет 0x9, независимо от того, в какое место редактор связей помещает секцию .text.

РЕШЕНИЕ УПРАЖНЕНИЯ 7.5

Каким образом фактически запускаются программы на языке С, является тайной для большинства программистов. Вопросы этого задания контролируют ваше понимание сущности процесса запуска. Вы можете ответить на них, используя представление кода запуска 7.18:

1. Каждая программа требует функции main, ввиду того, что код запуска, общий для всех программ на языке С, по соглашению передает управление функции main.
2. Если main завершает свою работу исполнением оператора return, то управление передается назад в процедуру запуска, которая возвращает управление операционной системе, путем вызова _exit. Тот же самый процесс происходит, если пользователь опускает оператор return. Если main завершает свою работу вызовом exit, то exit, в конечном счете, возвращает управление операционной системе, вызывая _exit. Суммарный результат один и тот же во всех трех случаях: когда main завершает свою работу, управление передается назад в операционную систему.



ГЛАВА 8

Управление исключениями

- Исключения.
 - Процессы.
 - Системные вызовы и обработка ошибок.
 - Управление процессами.
 - Сигналы.
 - Нелокальные передачи управления.
 - Организация управления процессами.
 - Резюме.
-

С того момента, как вы подадите питание на процессор, и до того момента, когда вы его выключите, счетчик команд будет принимать последовательность значений

$$a_0, a_1, \dots, a_{n-1}$$

где каждое a_k — это адрес, соответствующий некоторой команде I_k . Каждый переход из a_k в a_{k+1} называется передачей управления. Последовательность таких передач управления называется потоком управления, или управляющим потоком процессора.

Простейший поток управления представляет собой "линейную" последовательность, где каждые I_k и I_{k+1} , в памяти расположены рядом. Как правило, неожиданные изменения этого линейного потока, когда I_{k+1} уже не будет смежной с I_k , вызываются известными всем командами, такими как команда перехода, вызов и оператор возврата. Такие команды создают необходимые механизмы, позволяющие программам реагировать на изменения в их внутреннем состоянии, представленном программными переменными.

Но системы должны также реагировать на такие изменения в их состоянии, которые не определяются только внутренними переменными программы и не обязательно связаны с исполнением самой программы. Например, аппаратный таймер срабатывает через регулярные интервалы времени, и его сигналы должны быть обработаны. Пакеты поступают в сетевой адаптер и должны быть сохранены в памяти. Программа

запрашивает данные из диска и затем засыпает — до тех пор, когда она получит уведомление о том, что данные готовы. Родительский процесс, который порождает дочерние процессы, должен получать уведомления, когда его дочерние процессы завершают свою работу.

Современные системы реагируют на такие ситуации путем выполнения скачкообразных переходов в потоке управления. Вообще, мы называем такие скачкообразные переходы передачей управления по исключению (exceptional control flow, ECF). Передача управления по исключению может происходить на любом уровне компьютерной системы. Например, на аппаратном уровне события, зафиксированные аппаратными средствами, вызывают скачкообразные передачи управления к обработчикам исключительных ситуаций. На уровне операционной системы ядро передает управление из одного пользовательского процесса в другой через контекстные переключатели. На прикладном уровне один процесс может отправить сигнал другому, который скачкообразно передает управление обработчику сигнала в месте приема. Отдельная программа может реагировать на ошибки, путем обхода обычных стековых механизмов, и выполнения нелокальных переходов по произвольным адресам — в другие функции.

Можно привести несколько обоснованных доводов, почему программистам имеет смысл серьезно ознакомиться с ECF:

- Понимание сущности ECF поможет вам разобраться со многими важными понятиями организации систем. ECF — это базовый механизм, который используется операционными системами для организации ввода-вывода, процессов и виртуальной памяти. Перед тем как по-настоящему приступить к изучению таких важных понятий, следует ознакомиться с ECF.
- Понимание сущности ECF поможет вам разобраться с тем, как и каким образом приложения взаимодействуют с операционной системой. Приложения запрашивают службы операционной системы путем использования формы ECF, известной как системное прерывание (trap), или системный вызов. Например, запись данных на диск, чтение данных из сети, порождение нового процесса и завершение текущего процесса — все это достигается путем вызова прикладных программ, исполняющих системные вызовы. Понимание стандартных механизмов системного вызова поможет вам разобраться, каким образом такие службы могут быть использованы приложениями.
- Понимание сущности ECF поможет вам писать новые содержательные прикладные программы. Операционная система дает возможность прикладным программам использовать эффективные механизмы ECF для создания новых процессов, для ожидания завершения обработки, для уведомления других процессов о возникновении в системе исключительных ситуаций, а также для обнаружения этих событий и реакции на них. Если вы разберетесь в механизмах ECF, то сможете применять их при написании содержательных программ, таких как оболочки Unix и Web-серверы.
- Понимание сущности ECF поможет вам разобраться с тем, как и каким образом работают программные исключения. Такие языки, как C++ и Java, обеспечивают доступ к механизмам программных исключений через операторы try (попытать-

ся), `catch` (перехватить) и `throw` (вбросить). Программные исключения дают возможность делать нелокальные переходы в программе (т. е. скачки, которые выводят за пределы обычного стекового механизма вызова/возврата), в качестве реакции на возникшую ошибку. Нелокальный переходы — это форма ECF прикладного уровня, и в языке С они реализуются функциями `setjmp` и `longjmp`. Изучение этих функций низкого уровня поможет вам разобраться, каким образом можно реализовать высокоуровневые программные исключения.

До сих пор при изучении систем мы в основном интересовались, каким образом приложения взаимодействуют с аппаратными средствами. Эта глава будет базовой в том смысле, что мы приступим к изучению вопроса о том, каким образом приложения взаимодействуют с операционной системой. Интересно, что все эти взаимодействия тесно связаны с ECF. Мы опишем различные формы ECF, существующие на различных уровнях компьютерной системы. Мы начнем с использования исключений, которые находятся в области, общей для аппаратных средств и операционной системы. Мы также обсудим системные вызовы, которые являются исключениями, обеспечивающими приложениям возможность использовать точки входа в операционную систему. Затем мы повысим уровень абстракции с тем, чтобы описать процессы и сигналы, которые находятся в области, общей для прикладных задач и операционной системы. Наконец, мы обсудим нелокальные переходы, которые представляют собой одну из форм ECF прикладного уровня.

8.1. Исключения

Исключения как таковые представляют собой форму передачи управления по исключению. Их особенность состоит в том, что они реализованы частично с использованием аппаратных средств, а частично — за счет средств операционной системы. Поскольку их реализация связана с аппаратными средствами, детали меняются от системы к системе. Тем не менее основные концепции остаются одними и теми же для всех систем. В этом разделе наша цель будет состоять в том, чтобы дать общее представление об исключениях и об обработке исключений, а также снять ауру таинственности и помочь разобраться в часто трудно объяснимых аспектах современных компьютерных систем.

Исключение — это скачкообразный переход в потоке управления в ответ на некоторые изменения в состоянии процессора. Рис. 8.1 представляет основную идею этого явления.

На этом рисунке процессор исполняет некоторую текущую команду I_{curr} в момент, когда происходит существенное изменение его состояния. Это состояние кодируется набором значений различных битов и сигналов внутри процессора. Изменение состояния называется *событием*. Событие может быть непосредственно связано с исполнением текущей команды. Например, оно может произойти в результате сбоя при обращении к странице виртуальной памяти, в результате арифметического переполнения, или когда команда делает попытку деления на нуль. С другой стороны, событие может быть не связано с исполнением текущей команды. Например, оно может быть вызвано срабатыванием системного таймера или завершением запроса на ввод-вывод.

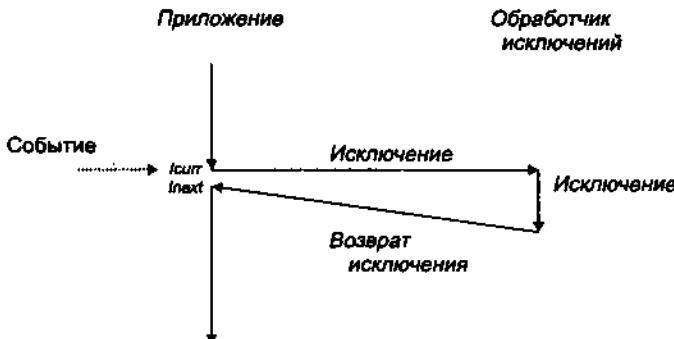


Рис. 8.1. Структура протекания процессов при возникновении исключения

В любом случае, когда процессор обнаруживает, что произошло событие, он через таблицу векторов прерываний, называемую *таблицей исключений*, исполняет косвенный процедурный вызов (исключение) некоторой подпрограммы операционной системы (обработчик исключительных ситуаций), которая специально предназначена для обработки этого конкретного события.

После того как обработчик исключительных ситуаций заканчивает свою работу, в зависимости от вида события, вызвавшего исключение, происходит одно из трех:

1. Обработчик возвращает управление текущей команде I_{curr} , т. е. той команде, которая исполнялась в тот момент, когда произошло данное событие.
2. Обработчик возвращает управление команде I_{next} , которая исполнялась бы следующей, если бы не произошло исключение.
3. Обработчик аварийно завершает исполнение прерванной программы.

Более подробно об этом говорится в разд. 8.1.2.

Сравнение аппаратных и программных исключений

Программирующим на C++ и Java следует иметь в виду, что термин "исключение" используется также для обозначения ECF-механизмов, обеспечиваемых в языках C++ и Java с помощью операторов `catch`, `throw` и `try`. Если быть предельно точным, то следовало бы делать различия между аппаратными и программными исключениями, но это обычно излишне, поскольку смысл и так ясен из контекста.

8.1.1. Обработка исключений

Разобраться с тем, как работают исключения, может оказаться нелегкой задачей, ввиду того, что их обработка предполагает тесное взаимодействие между аппаратными и программными средствами. Нетрудно запутаться в том, какой компонент какую выполняет задачу. Рассмотрим более подробно распределение работ между аппаратными и программными средствами.

Каждому типу возможного в системе исключения присваивается уникальный неотрицательный целочисленный *номер исключения*. Некоторые из этих номеров при-

сваиваются разработчиками процессора. Другие номера присваиваются разработчиками ядра операционной системы (резидентная в памяти часть операционной системы). Примерами первого являются деление на нуль, сбои при обращении к странице, нарушения доступа к памяти, контрольные точки и арифметические переполнения. Примеры последнего включают системные вызовы и сигналы от внешних устройств ввода-вывода.

Во время загрузки системы (в тот момент, когда компьютер либо перезагружается, либо включается) операционная система размещает и инициализирует таблицу векторов прерываний, называемую *таблицей исключений*. Это делается таким образом, чтобы вход k содержал адрес обработчика для исключения k . На рис. 8.2 представлен формат таблицы исключений.

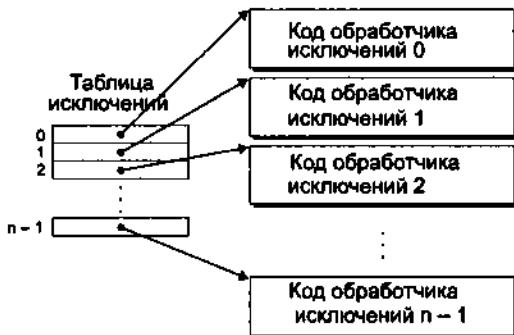


Рис. 8.2. Таблица исключений

Если во время прогона (время, когда система исполняет некоторую программу), процессор обнаруживает, что произошло событие, он определяет соответствующий номер исключения k . После этого процессор через вход k таблицы исключений выполняет косвенный процедурный вызов исключения с соответствующим обработчиком. Рис. 8.3 показывает, каким образом процессор использует таблицу исключений, чтобы сформировать адрес соответствующего обработчика исключительных ситуаций. Номер исключения — это индекс в таблице исключений, начальный адрес которой — это содержимое специального регистра центрального процессора, называемого *регистром базы таблицы исключений*.



Рис. 8.3. Получение адреса обработчика исключительных ситуаций

Как и в случае вызова подпрограммы, перед тем как выполнить переход на обработку, процессор заслуживает в стек адрес возврата. Тем не менее, в зависимости от

класса исключения, адрес возврата будет либо адресом текущей команды (команды, которая исполнялась в тот момент, когда произошло событие), либо адресом последующей команды (команды, которая исполнялась бы после текущей команды, если бы не произошло событие).

Процессор также заталкивает в стек некоторые дополнительные атрибуты состояния процессора, которые будут необходимы для возобновления исполнения прерванной программы, после того, как обработчик возвратит управление. Например, система IA32 заталкивает в стек содержимое регистра EFLAGS, где, среди прочего, имеются коды текущего состояния.

Если управление передается из пользовательской программы в ядро, то все эти элементы заталкиваются в стек ядра, а не в пользовательский стек.

Обработчики исключительных ситуаций исполняются в привилегированном режиме (см. разд. 8.2.3), что означает, что они имеют полный доступ к любым системным ресурсам.

Как только вызывается аппаратное исключение, остальная часть работы выполняется программным путем с помощью обработчика исключительных ситуаций. После того как обработчик обработает событие, он не обязательно возвращает управление в прерванную программу, путем исполнения специальной команды "возврат из прерывания". Эта команда выталкивает из стека соответствующее состояние назад в управляющие регистры и регистры данных процессора, восстанавливая состояние пользовательского режима (см. разд. 8.2.3), если данное исключение прервало пользовательскую программу. После этого управление возвращается в прерванную программу.

8.1.2. Классы исключений

Исключения можно разделить на четыре класса: аппаратные прерывания, системные прерывания, сбои и аварийное завершение. В табл. 8.1 представлены общие признаки этих классов.

Таблица 8.1. Классы исключений

Класс	Причина	Синхронное/ Асинхронное	Поведение при возврате
Аппаратное прерывание (interrupt)	Сигнал от устройства ввода-вывода	Асинхронное	Всегда возвращается к следующей команде
Системное прерывание (trap)	Предусмотренное исключение	Синхронное	Всегда возвращается к следующей команде
Сбой (fault)	Потенциально восстановимая ошибка	Синхронное	Может возвратиться к текущей команде
Аварийное завершение (abort)	Невосстановимая ошибка	Синхронное	Никогда не возвращается

Аппаратные прерывания

Аппаратные прерывания (*interrupt*) возникают асинхронно в результате поступления сигналов от устройств ввода-вывода, которые являются внешними по отношению к процессору. Аппаратные прерывания являются асинхронными, в том смысле, что они не вызваны исполнением какой-либо конкретной команды. Обработчики исключительных ситуаций для аппаратных прерываний часто называются *обработчиками прерываний*.

На рис. 8.4 представлена схема обработки прерывания. Устройства ввода-вывода, такие как сетевые адаптеры, контроллеры диска и чипы таймера, инициируют прерывания, посыпая сигнал на определенный вывод чипа процессора и передавая номер исключения на системную шину, которая идентифицирует устройство, вызвавшее прерывание.

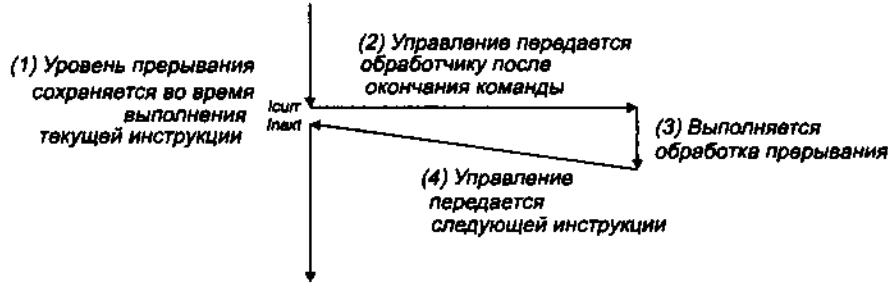


Рис. 8.4. Обработка прерывания

После того, как завершается исполнение текущей команды, и процессор обнаруживает, что на выводе прерывания высокий уровень, он читает из системной шины номер исключения, а затем вызывает соответствующий обработчик прерывания. После того, как обработчик выполнит операцию возврата, управление переходит на следующую команду (т. е. команду, которая в потоке управления следовала бы за текущей командой, если бы прерывание не произошло). В результате программа продолжает исполняться, как если бы не было никакого прерывания.

Остальные классы исключений (системные прерывания, сбои и аварийные завершения) происходят синхронно, в результате выполнения текущей команды. Такие команды мы будем называть *аварийными командами*.

Системные прерывания

Системные прерывания (*trap*) — это предопределенные исключения, которые возникают в результате исполнения команды. Подобно обработчикам аппаратного прерывания, обработчики системного прерывания возвращают управление в следующую команду. Наиболее важное назначение системных прерываний состоит в том, чтобы обеспечить процедурный интерфейс между пользовательскими программами и ядром, так называемый *системный вызов*.

Пользовательские программы часто должны запрашивать у ядра службы, такие как чтение файла (*read*), порождение нового процесса (*fork*), загрузка новой программы

(execve), или завершение текущего процесса (exit). Для организации управляемого доступа к таким службам ядра, процессоры предоставляют специальную команду "syscall *n*", которая может исполняться пользовательскими программами в случае, если необходимо запросить службу номер *n*. Исполнение команды *syscall* приводит к тому, что возникает системное прерывание с вызовом обработчика исключительных ситуаций, который расшифровывает аргумент, и вызывает соответствующую подпрограмму ядра. На рис. 8.5 представлена схема обработки системного вызова.

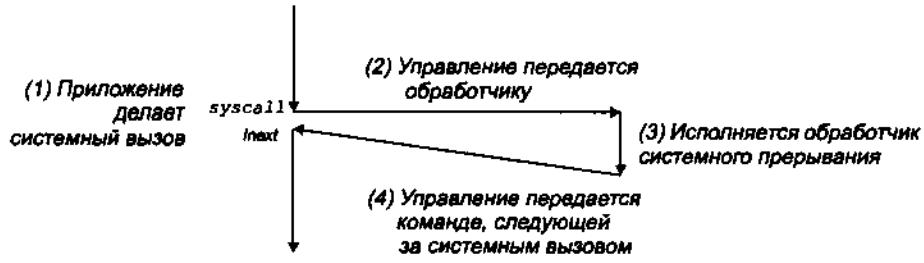


Рис. 8.5. Обработка системного прерывания

С точки зрения программиста, системный вызов — это обычный вызов функции. Тем не менее реализации этих вызовов довольно сильно отличаются. Стандартные функции исполняются в *пользовательском режиме*, при котором ограничен набор команд, допустимых для исполнения, и эти функции обращаются к тому же самому стеку, что и вызывающая функция. Системный вызов исполняется в *привилегированном режиме*, при котором допускается выполнять команды и обращаться к стеку, определенному в ядре. Более подробно пользовательский и привилегированный режимы обсуждаются в разд. 8.2.3.

Сбои

Сбои (fault) являются следствием таких ошибочных состояний, которые обработчик, возможно, мог бы исправить. Когда возникает сбой, процессор передает управление обработчику ошибок. Если обработчик имеет возможность исправить состояние ошибки, он возвращает управление команде, вызвавшей сбой, выполняя ее повторно. В противном случае, обработчик возвращает управление в ядро — подпрограмме аварийного завершения, которая завершает работу прикладной программы, вызвавшей данный сбой. На рис. 8.6 представлена схема обработки сбоев.

Классический пример сбоя — это исключение при ошибке обращения к странице, возникающее, когда команда ссылается на виртуальный адрес, которому соответствует физическая страница, не резидентная в памяти, которую поэтому следует восстановить с диска. Как мы увидим в главе 10, страница представляет собой непрерывный блок (обычно 4 Кбайт) виртуальной памяти. Обработчик ошибки обращения к странице загружает соответствующую страницу с диска и затем возвращает управление команде, которая вызвала данный сбой. Когда эта команда будет исполняться снова, соответствующая физическая страница уже будет резидентна в памяти, и данная команда сможет завершиться безаварийно.



Рис. 8.6. Обработка сбоя

Аварийные завершения

Аварийные завершения (*abort*) являются результатом невосстановимых фатальных ошибок. Обычно это аппаратные ошибки, такие как ошибки четности, которые возникают при потере значения отдельных битов DRAM или SRAM. Обработчики аварийного завершения никогда не возвращают управление в прикладную программу. Как показано на рис. 8.7, обработчик передает управление подпрограмме *abort*, которая завершает работу прикладной программы.

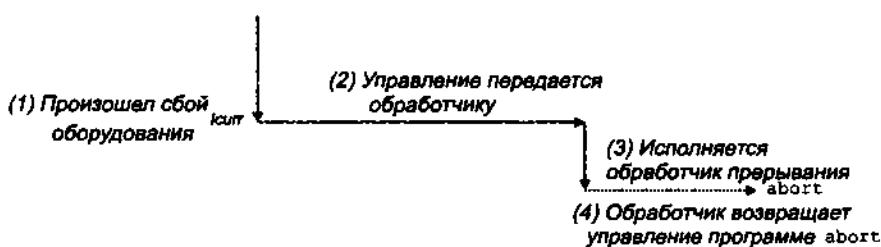


Рис. 8.7. Обработка аварийного завершения

8.1.3. Исключения в процессорах Intel

Легче рассуждать о конкретных вещах, поэтому рассмотрим некоторые исключения, определенные для платформы Intel. Системы Pentium могут иметь до 256 различных видов исключений. Номера в интервале от 0 до 31 соответствуют исключениям, определенным архитектурой Pentium, и таким образом идентичным в любых системах класса Pentium. Номера в интервале от 32 до 255 соответствуют аппаратным и системным прерываниям, принадлежащим операционной системе. В табл. 8.2 представлены некоторые примеры.

Ошибка деления (исключение 0) возникает, когда приложение делает попытку деления на нуль, или когда результат команды деления слишком велик для операнда назначения. Unix не пытается исправлять ошибки деления, предпочитая просто прервать программу. Командные процессоры Unix обычно сообщают об ошибках деления как "Ошибка операции с плавающей точкой".

Таблица 8.2. Примеры исключений

Номер исключения	Описание	Класс исключения
0	Ошибка деления	Сбой
13	Общее нарушение защиты	Сбой
14	Ошибка обращения к странице	Сбой
18	Ошибка аппаратного контроля	Аварийное завершение
32–127	Исключения, определенные OS	Аппаратное или системное прерывание
128 (0×80)	Системные вызовы	Системное прерывание
129–255	Исключения, определенные OS	Аппаратное или системное прерывание

Печально известное общее нарушение защиты (исключение 13) возникает по разным причинам, обычно ввиду того, что программа ссылается на не определенную область виртуальной памяти, или когда программа пытается сделать запись в процедурный сегмент, определенный как только-для-чтения. Unix не пытается исправлять такие сбои. Командные процессоры Unix обычно сообщают об общих нарушениях защиты как "Сбой сегментации".

Отсутствие страницы (исключение 14) — это пример исключения, в котором команда, вызвавшая ошибку, будет исполняться повторно. Обработчик отображает соответствующую страницу физической памяти на диске в страницу виртуальной памяти и затем повторно исполняет команду, вызвавшую ошибку. В главе 10 мы подробно рассмотрим, каким образом работает этот механизм.

Ошибка аппаратного контроля (исключение 18) возникает в результате фатальной ошибки аппаратных средств и будет обнаружена в течение исполнения команды, вызвавшей ошибку. Обработчики ошибок аппаратного контроля никогда не возвращают управление в прикладную программу.

Системные вызовы обеспечиваются в системе IA32 через команды перехвата INT и, где индекс *l* может быть любым из 256 входов в таблице исключений. Исторически системные вызовы производятся через исключение 128 (0×80).

О терминологии

Терминология в области классов исключений до сих пор не устоялась и изменяется от системы к системе. В спецификациях макроархитектуры процессоров часто различаются асинхронные "прерывания" и синхронные "исключения", но, тем не менее, не существует никакого обобщающего термина, обозначающего эти очень сходные понятия. Чтобы постоянно не прибегать к использованию словосочетаний "исключения и прерывания" и "исключения или прерывания", мы будем в этих случаях использовать, как общий термин, слово "исключение", и будем при этом делать различия между асинхронными исключениями (аппаратные прерывания) и синхронными исключениями (системные прерывания, сбои и аварийные завершения), но только в тех случаях, когда это будет иметь смысл. Как уже отмечалось,

основные положения остаются неизменными для любой системы, но следует иметь в виду, что некоторые разработчики в своих руководствах используют слово "исключение", обозначая им только те изменения в потоке управления, которые вызываются синхронными событиями.

8.2. Процессы

Исключения — это один из тех "кирпичиков", которые используются для построения и развития одной из самых плодотворных и успешных идей в информатике — концепции процесса в операционных системах.

Когда в современной вычислительной системе программа запускается на счет, создается иллюзия, что в текущий момент в этой системе единственной исполняющейся будет данная программа. Эта программа, казалось бы, монопольно использует как процессор, так и память. Процессор, казалось бы, исполняет команды программы последовательно одну за другой, без всяких прерываний. Наконец, создается иллюзия, что кроме программного кода и данных этой программы в памяти системы других объектов нет. Эти иллюзии создаются, благодаря концепции процесса.

Традиционно процесс определяется, как экземпляр исполняющейся программы. Каждая программа в системе исполняется в контексте некоторого процесса. Контекст представляет собой некоторую структуру, позволяющую программе исполняться правильно. Эта структура включает программный код и данные, хранящиеся в памяти, стек программы, содержимое ее регистров общего назначения, ее счетчика команд, переменные среды исполнения и набор дескрипторов открытых файлов.

Каждый раз, когда пользователь запускает программу на счет, вводя в командной строке имя исполнимого объектного файла, командный процессор порождает новый процесс и затем запускает на счет данный исполнимый объектный файл в контексте этого нового процесса. Прикладные программы также могут порождать новые процессы и в контексте этих новых процессов запускать на счет свой собственный программный код или другие приложения.

Подробное обсуждение того, каким образом в операционных системах реализуются процессы, выходит за пределы нашей темы. Вместо этого, мы сосредоточим свое внимание на следующих ключевых вопросах взаимосвязи процесса и приложения:

- автономный логический поток управления, который создает иллюзию того, что программа монопольно использует процессор;
- закрытое адресное пространство, которое создает иллюзию того, что программа монопольно использует систему памяти.

Рассмотрим эти вопросы более подробно.

8.2.1. Логический поток управления

Процесс создает иллюзию того, что каждая программа монопольно использует процессор, даже если в системе будут исполняться несколько других программ. Если бы мы использовали отладчик в пошаговом режиме, мы могли бы наблюдать последова-

тельность значений счетчика команд (PC), которая соответствовала бы исключительно командам, содержащимся в исполняемом объектном файле нашей программы или в совместно используемых объектах, связанных с нашей программой динамически во время исполнения. Эта последовательность значений PC называется логическим потоком управления.

Рассмотрим систему, в которой исполняется три процесса, как показано на рис. 8.8. Единый физический поток управления процессора делится на три логических потока, по одному для каждого процесса. Каждая вертикальная линия представляет часть логического потока для процесса. В данном примере некоторое время исполняется процесс A. Затем следует B, который исполняется до момента своего завершения. Затем некоторое время исполняется процесс C. Затем следует процесс A, который исполняется до момента своего завершения. Наконец, процесс C получает возможность исполняться до момента своего завершения.

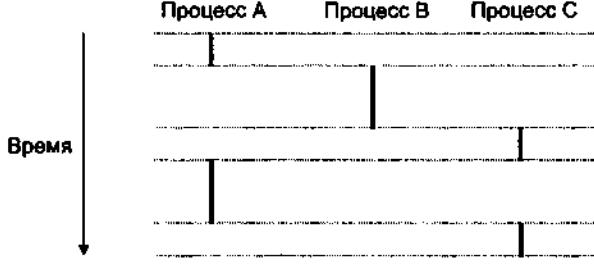


Рис. 8.8. Логические потоки управления

Ключевым моментом на рис. 8.8 является то, что процессы сменяют друг друга при использовании процессора. Каждый процесс исполняет часть своего потока и затем прерывается (временно приостанавливается) на то время, когда другие процессы заступают на свою смену. Программе, исполняющейся в контексте одного из этих процессов, кажется, что она монопольно использует процессор. Единственное, что не подлежит сомнению, что если бы мы точно измерили время, затраченное каждой командой (см. главу 9), мы обратили бы внимание на то, что центральный процессор как бы периодически останавливается в промежутках между исполнением некоторых команд программы. Тем не менее каждый раз, когда процессор "останавливается", он впоследствии возобновляет исполнение программы без всяких изменений в содержимом адресов памяти программы или регистров.

Вообще, каждый логический поток независим от всех остальных потоков в том смысле, что логические потоки, связанные с различными процессами, не затрагивают состояний друг друга. Единственное исключение из этого правила возникает тогда, когда процессы используют такие механизмы взаимодействия (interprocess communication, IPC), как каналы, гнезда, разделяемая память и семафоры, позволяющие явным образом взаимодействовать друг с другом.

Любой процесс, логический поток которого пересекается во времени с потоком другого, называется *параллельным процессом*, и мы говорим, что эти два процесса исполняются одновременно. Например, на рис. 8.8 A и B исполняются одновременно,

так же А и С. С другой стороны, В и С не исполняются одновременно, поскольку последняя команда В исполняется перед первой командой С.

Концепция процессов, сменяющих друг друга, лежит в основе *многозадачного режима*. Каждый интервал времени, в течение которого процесс исполняет свою часть потока, называется *квантом времени*. Таким образом, многозадачный режим можно также назвать режимом квантования времени.

8.2.2. Закрытое адресное пространство

Процесс также создает иллюзию того, что каждая программа монопольно использует адресное пространство системы. На машине с n -разрядными адресами, адресное пространство представляет собой 2^n возможных адресов, 0, 1, ..., $2^n - 1$. Процесс предоставляет каждой программе свое собственное *закрытое адресное пространство*. Это пространство называется закрытым в том смысле, что любой байт памяти, связанный с конкретным адресом в этом пространстве, обладает тем свойством, что к нему невозможно обратиться (прочитать или записать данные) из какого-либо другого процесса.

Хотя содержимое памяти, связанной с закрытым адресным пространством, вообще будет различным, общая структура каждого такого пространства будет одной и той же.



Рис. 8.9. Адресное пространство процесса

Например, на рис. 8.9 представлена структура организации адресного пространства для процессов в Linux. Нижние три четверти адресного пространства зарезервированы

ны для пользовательских программ с обычными сегментами текста, данных и стека. Верхняя четверть адресного пространства зарезервирована для ядра системы. Эта часть адресного пространства содержит программный код, данные и стек, которые ядро использует при исполнении команды по поручению процесса (например, когда прикладная программа исполняет системный вызов).

8.2.3. Пользовательский и привилегированный режимы

Для того чтобы ядро операционной системы поддерживало непроницаемую модель процесса, процессор должен обеспечить механизм ограничения набора исполняемых приложением команд, а также ограничения части доступного адресного пространства.

Процессоры обычно обеспечивают такую возможность, используя *бит режима* в управляющем регистре, определяющий привилегии, которыми обладает процесс в настоящее время. Когда бит режима будет установлен, процесс будет исполняться в привилегированном режиме (иногда называемом *режимом супервизора*). Процесс, исполняющийся в привилегированном режиме, может исполнять любую из имеющейся набора команд, а также обращаться к любому адресу памяти в системе.

Если бит режима не будет установлен, процесс будет исполняться в пользовательском режиме. В пользовательском режиме процесс не имеет возможности выполнять привилегированные команды, которые выполняют такие операции, как остановка процессора, изменение бита режима или инициация операции ввода-вывода. Он также не имеет возможности непосредственно ссылаться на программный код или данные в области адресного пространства ядра. Любая такая попытка кончается фатальной ошибкой защиты. Для выполнения таких операций пользовательские программы должны обращаться к программному коду ядра и к данным косвенно через интерфейс системного вызова.

Процесс, исполняющий прикладной программный код в исходном положении, находится в пользовательском режиме. Единственный способ изменить режим процесса на привилегированный — это создать исключительную ситуацию, такую как прерывание, сбой или системный вызов. Если возникает исключение, и управление передается обработчику исключительных ситуаций, процессор изменяет свой режим с пользовательского на привилегированный. Обработчик исполняется в привилегированном режиме. Когда управление возвращается в прикладной программный код, процессор изменяет режим с привилегированного назад, в пользовательский.

Linux и Solaris предоставляют хитроумный механизм, файловую систему /proc, позволяющий процессам пользовательского режима обращаться к содержимому структур данных ядра. Файловая система /proc экспортирует содержимое многих структур данных ядра, таких как иерархия файлов ASCII, которые могут быть прочитаны с помощью пользовательских программ. Например, можно использовать /proc файловой системы Linux, чтобы получить справку об общих характеристиках системы, таких как тип центрального процессора (/proc/cpuinfo), или о сегментах памяти, используемых конкретным процессом (/proc/<id процесса>/maps).

8.2.4. Контекстные переключатели

Ядро операционной системы реализует многозадачный режим, используя высоконивневую форму передачи управления по исключению, известную как *контекстный переключатель*. Механизм контекстного переключателя — это надстройка над механизмом исключения низкого уровня, который мы рассматривали в разд. 8.1.

Ядро поддерживает контекст для каждого процесса. Контекст — это структура состояния, которое ядро должно восстановить при возобновлении исполнения прерванного процесса. Она включает значения следующих объектов:

- регистры общего назначения;
- регистры с плавающей точкой;
- счетчик команд;
- пользовательский стек;
- регистры состояния;
- стек ядра;
- таблица страниц, которая задает адресное пространство;
- таблица процессов, которая содержит информацию о текущем процессе;
- таблица файлов, которая содержит информацию о файлах, открытых процессом.

В течение исполнения процесса ядро может принимать решение прерывать текущий процесс в некоторых точках, а затем возобновлять исполнение ранее прерванного процесса. Такое поведение называется *планированием активизации*, и оно обрабатывается с помощью программного кода в ядре, называемого *планировщиком*. Если ядро выбирает новый процесс, чтобы запустить его на исполнение, мы говорим, что ядро планирует активизацию некоторого процесса. После того как ядро запланирует активизацию нового процесса, оно прерывает текущий процесс и передает управление новому процессу, используя механизм, называемый *контекстным переключением*. При этом сохраняется контекст текущего процесса, восстанавливается сохраненный контекст некоторого ранее прерванного процесса, управление передается этому только что восстановленному процессу.

Контекстное переключение может произойти, когда ядро исполняет системный вызов по поручению пользователя. Если системный вызов блокирован по причине ожидания некоторого события, то ядро может отправить текущий процесс "поспать" и переключиться на другой процесс. Например, если системный вызов `read` требует доступа к диску, то ядро может выбрать контекстное переключение и запустить другой процесс вместо того, чтобы ожидать, когда данные будут получены с диска. Другим примером может служить системный вызов `sleep`, который явным образом переводит процесс в состояние сна. В общем случае, если даже системный вызов не блокирован, ядро может принять решение выполнить контекстное переключение вместо того, чтобы возвратить управление в вызывающий процесс.

Контекстное переключение может также произойти в результате прерывания. Например, некоторые системы имеют различные механизмы для генерирования периодических прерываний от таймера, обычно каждые 10 мс. Каждый раз, когда происхо-

дит прерывание от таймера, ядро может принять решение, что текущий процесс исполнялся уже достаточно долго, и переключить исполнение на новый процесс.

На рис. 8.10 представлен пример переключения контекста между парой процессов А и В. В этом примере в исходном положении процесс А исполняется в пользовательском режиме — до тех пор, пока он не перешлет в ядро запрос на исполнение системного вызова `read`. Обработчик системного прерывания в ядре запрашивает пересылку данных методом DMA в контроллер диска и принимает меры, чтобы контроллер диска вызвал прерывание процессора, после того как он закончит передавать данные из диска в память.



Рис. 8.10. Схема контекстного переключения процесса

Чтобы выбрать данные, диску потребуется относительно длительное время (порядка десятков миллисекунд), поэтому вместо того, чтобы ожидать в бездействии, ядро выполняет контекстное переключение от процесса А к процессу В. Имейте в виду, что перед тем как произвести переключение, ядро выполняет команды в пользовательском режиме по поручению процесса А. В течение первой фазы переключения ядро выполняет команды в привилегированном режиме по поручению процесса А. Затем в некоторый момент оно приступает к исполнению команд (все еще в привилегированном режиме) по поручению процесса В, и после того как переключение будет завершено, ядро начнет выполнять команды в пользовательском режиме по поручению процесса В.

Затем процесс В исполняется некоторое время в пользовательском режиме, до тех пор, когда диск пошлет сигнал прерывания, чтобы сообщить о том, что данные успешно переданы из диска в память. Ядро принимает решение, что процесс В исполнялся уже достаточно долго, и производит контекстное переключение от процесса В к процессу А, возвращая управление команде, непосредственно следующей за системным вызовом `read` в процессе А. Процесс А продолжает исполняться до тех пор, когда произойдет следующее исключение и т. д.

Засорение кэш и передача управления по исключению

Вообще, аппаратная кэш-память не приспособлена для взаимодействия с такими механизмами передачи управления по исключению, которые вызывают прерывания и контекстные переключения. Если текущий процесс будет прерван, то кэш становится недоступным для обработчика прерывания. Если обработчик взаимо-

действует с достаточно большим отрезком оперативной памяти, то кэш также становится недоступным и для прерванного процесса — в тот момент, когда он возобновляет свое исполнение. В этом случае мы говорим, что обработчик засорил (*pollute*) кэш. Аналогичное явление происходит при использовании контекстных переключений. В тот момент, когда процесс возобновляет свое исполнение после контекстного переключения, кэш становится недоступным для прикладной программы, и должен быть восстановлен заново.

8.3. Системные вызовы и обработка ошибок

Системы типа Unix предоставляют большой выбор системных вызовов, которые могут быть использованы прикладными программами — путем запроса сервисов от ядра, таких, например, как чтение файла или порождение нового процесса. Например, Linux предоставляет приблизительно 160 системных вызовов. Если ввести "man syscalls", то вы получите полный их список.

Из программы на языке C можно сделать любой системный вызов, непосредственно, путем использования макроса `_syscall`, описанного в "ман 2 intro". Тем не менее обычно делать системные вызовы непосредственно не представляется необходимым, более того, это даже и нежелательно. Стандартная C-библиотека предоставляет набор удобных функций-оболочек для наиболее часто используемых системных вызовов. Функции-оболочки пакета воспринимают аргументы, создают системное прерывание в ядре, инициируя соответствующий системный вызов, и затем передают статус возвращения системного вызова назад в вызывающую программу. В наших обсуждениях в последующих разделах мы иногда будем называть системные вызовы и связанные с ними функции-оболочки *функциями системного уровня*.

Если в Unix-функциях системного уровня возникают ошибки, они обычно возвращают `-1` и устанавливают глобальную целочисленную переменную `errno`, показывая, что вызов функции завершился неудачно. Программисты должны всегда принимать меры для отслеживания ошибок, но, к сожалению, многие обходят проверку ошибок, ввиду того, что программный код при этом разбухает и становится трудночитаемым. В качестве примера здесь показано, как можно было бы отслеживать ошибки при вызове Unix-функции `fork`:

```
1 if ((pid = fork()) < 0) {  
2     fprintf(stderr, "fork error: %s\n", strerror(errno));  
3     exit(0);  
4 }
```

Функция `strerror` возвращает строку текста с описанием ошибки, связанной с конкретным значением `errno`. Этот программный код можно несколько упростить путем определения следующей функции, выдающей сообщение об ошибке:

```
1 void unix_error (char *msg) /* функция в стиле unix */  
2 {  
3     fprintf(stderr, "%s: %s\n", msg, strerror(errno));  
4     exit(0);  
5 }
```

При наличии этой функции вызов `fork` сводится к двум строкам вместо четырех:

```
1 if ((pid = fork()) < 0)
2     unix_error("fork error");
```

Можно еще более упростить этот программный код, если использовать программу-оболочку с обработкой ошибок. Для заданной базовой функции `foo`, мы определим функцию `Foo` оболочки с идентичными параметрами, но первый символ имени которой — прописная буква. Программа-оболочка вызывает основную функцию, отслеживает ошибки и завершает свою работу, если имеются какие-либо проблемы. Например, вот как выглядит программа-оболочка обработки ошибок для функция `fork`:

```
1 pid_t Fork(void)
2 {
3     pid_t pid;
4
5     if ((pid = fork()) < 0)
6         unix_error("Fork error");
7     return pid;
8 }
```

При наличии такой программы-оболочки вызов функции `fork` сводится к единственной компактной строке:

```
1 pid = Fork();
```

Мы будем использовать программы-оболочки обработки ошибок на протяжении всей этой книги. Это даст нам возможность держать примеры программ немногословными, не создавая ошибочного впечатления, что допустимо игнорировать контроль ошибок. Имейте в виду, что когда в тексте мы будем обсуждать функции системного уровня, мы всегда будем ссылаться на них, указывая их базовые имена в нижнем регистре, но не имена их программ-оболочек.

*В приложении 2 можно найти обсуждение вопросов обработки ошибок в Unix, а также программы-оболочки обработки ошибок, используемые в этой книге. Программы-оболочки определены в файле `csapp.c`, а их прототипы — в файле заголовков `csapp.h`. Для справки, в *приложении 2* приведены исходные программы для этих файлов.*

8.4. Управление процессами

Unix предоставляет несколько системных вызовов для управления процессами из программы на C. В этом разделе описаны важные функции, а также даны примеры их использования.

8.4.1. Получение ID процесса

Каждый процесс имеет уникальный положительный (отличный от нуля) ID процесса (PID). Функция `getpid` возвращает PID вызывающего процесса. Функция `getppid` возвращает PID родительского процесса (породившего вызывающий процесс).

```
#include <unistd.h>
#include <sys/types.h>
pid_t getpid(void);
pid_t getppid(void);
```

Подпрограммы `getpid` и `getppid` возвращают целочисленное значение типа `pid_t`, который в системах Linux определен в `types.h` как `int`.

8.4.2. Порождение и завершение процессов

С точки зрения программиста, можно представить, что процесс находится в одном из трех состояний:

- Исполняющийся процесс либо исполняется в центральном процессоре, либо находится в состоянии ожидания наступления ситуации, когда он будет, в конечном счете, выбран для активизации.
- Приостановленный — его активизация не будет запланирована. Процесс останавливается в результате получения одного из сигналов `SIGSTOP`, `SIGTSTP`, `SIGTTIN` или `SIGTTOU`, и он остается приостановленным до тех пор, когда будет получен сигнал `SIGCONT`, после чего он продолжит исполняться с той же точки. Сигнал представляет собой форму программного прерывания, которая подробно описана в разд. 8.5.
- Завершенный процесс остановился навсегда. Процесс становится завершенным по одной из трех причин: получение сигнала, стандартное действие которого состоит в том, чтобы завершить работу процесса; возвращение из подпрограммы `main` или вызов функции `exit`.

```
#include <stdlib.h>
void exit(int status);
```

Функция `exit` завершает работу процесса, помещая в `status` значение статуса выхода. Другой способ установить статус выхода состоит в том, чтобы возвратить целочисленное значение из подпрограммы `main`.

Родительский процесс порождает новый исполняющийся дочерний процесс путем вызова функции `fork`.

```
#include <unistd.h>
#include <sys/types.h>
pid_t fork(void);
```

Функция возвращает 0 — дочернему процессу, PID дочернего процесса — родителю, и `-1` — в случае ошибки.

Вновь порожденный дочерний процесс почти, но не совсем, идентичен своему родителю. Дочерний процесс становится идентичной (но отдельной) копией родителя в виртуальном адресном пространстве пользовательского уровня, включая сегменты текста, данных и `bss`, динамическую память и пользовательский стек. Дочерний про-

цесс получает также идентичные копии всех описателей открытых файлов родителя, а это означает, что дочерний процесс может читать и писать во все файлы, которые были открыты в родительском процессе, когда он вызвал `fork`. Наиболее существенное различие между родителем и вновь порожденным дочерним процессом состоит в том, что они имеют различные PID.

Функция `fork` интересна (но часто и непонятна) тем, что она вызывается один раз, но возвращает значение дважды: один раз — в вызывающий (родительский) процесс, и один раз — во вновь порожденный дочерний процесс. В родительский процесс `fork` возвращает PID дочернего процесса, в дочерний процесс `fork` — значение 0. Поскольку PID дочернего процесса всегда отличен от нуля, возвращаемое значение однозначно указывает, кому оно предназначено: родительскому или дочернему процессу.

В листинге 8.1 представлен простой пример родительского процесса, который использует `fork`, чтобы породить дочерний процесс. В тот момент, когда вызов `fork` возвращает значение (строка 8), `x` имеет значение 1 как в родительском, так и в дочернем процессах. Дочерний процесс увеличивает значение и печатает свою копию `x` (строка 10). Точно так же родительский процесс уменьшает значение и печатает свою копию `x` (строка 15).

Листинг 8.1. Родительский и дочерний процессы

```

1 #include "csapp.h"
2
3 int main()
4 {
5     pid_t pid;
6     int x = 1;
7
8     pid = Fork();           /* дочерний процесс */
9     if (pid == 0) {         /* дочерний процесс */
10        printf("child : x=%d\n", ++x);
11        exit(0);
12    }
13
14    /* родительский процесс */
15    printf("parent : x=%d\n", --x);
16    exit(0);
17 }
```

При исполнении этой программы в системе Unix будет получен следующий результат:

```
unix> ./fork
parent: x=0
child : x=2
```

В этом простом примере можно отметить некоторые интересные особенности:

- Вызывается только один раз, но возвращается дважды. Функция `fork` вызывается один раз родителем, но она возвращается дважды, один раз — в порождающий, и один раз — во вновь порожденный дочерний процесс. Это вполне очевидно для программ, которые порождают единственный дочерний процесс. Но в программах с несколькими экземплярами `fork` можно запутаться, и их следует подробно разобрать.
- Параллельное исполнение. Родительский и дочерний процессы — это различные процессы, исполняющиеся одновременно. Ядро системы может произвольным образом чередовать выполнение команд в их логических потоках управления. При исполнении этой программы в нашей системе родительский процесс первым завершает отработку оператора `printf`, а за ним это делает дочерний процесс. Тем не менее в другой системе возможен обратный порядок. В общем, программисты никогда не должны делать предположений о порядке исполнения команд различными процессами.
- Идентичные, но отдельные адресные пространства. Если бы мы могли остановить родительский и дочерний процессы сразу же после того, как функция `fork` выполнит возврат в каждом из процессов, мы увидели бы, что адресные пространства этих процессов идентичны. Каждый из процессов имеет одинаковые пользовательские стеки, одинаковые значения локальных переменных, одинаковые значения глобальных переменных, а также одинаковые программные коды. Таким образом, в программе нашего примера локальная переменная `x` имеет значение 1 в как родительском, так и в дочернем процессе в тот момент, когда функция `fork` выполняет операцию возврата (строка 8). Тем не менее, поскольку родительский и дочерний процессы — это отдельные процессы, каждый из них имеет свои собственные закрытые адресные пространства. Любые последующие изменения переменной `x`, которые производятся в родительском или дочернем процессе, закрыты и не отображаются в памяти другого процесса. Вот почему переменная `x` имеет различные значения в родительском и дочернем процессах, когда они вызывают свои собственные операторы `printf`.
- Разделяемые файлы. Обратите внимание, что когда исполняется программа примера, как родительский, так и дочерний процессы выводят свои результаты на один и тот же экран. Это возможно потому, что дочерний процесс наследует все открытые файлы родителя. Когда родительский процесс вызывает `fork`, будет открыт файл `stdout`, и вывод будет направлен на экран. Дочерний процесс наследует этот файл и, таким образом, его вывод будет также направлен на экран.

В самом начале, при изучении функции `fork`, часто полезно делать набросок графа процесса, в котором каждая горизонтальная стрелка соответствует процессу, исполняющему команды слева направо, а каждая вертикальная стрелка соответствует исполнению функции `fork`.

Например, сколько строк вывода может сгенерировать программа в листинге 8.2, на рис. 8.11 представлен соответствующий граф процесса. Родительский процесс порождает дочерний процесс при первом (и единственном) исполнении `fork` в программе.

Каждый из них вызывает `printf` один раз, поэтому программа выводит две выходных строки.

Листинг 8.2. Один вызов fork()

```
1 #include "csapp.h"
2
3 int main()
4 {
5     Fork();
6     printf("hello!\n");
7     exit(0);
8 }
```

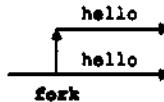


Рис. 8.11. Выводятся две выходных строки

А теперь допустим, что нам нужно вызвать `fork` дважды, как показано в листинге 8.3. Как видно из рис. 8.12, родительский объект вызывает `fork`, чтобы породить дочерний процесс, а затем как родительский, так и дочерний процесс вызывают `fork`, что дает на два процесса больше. Таким образом, имеются четыре процесса, каждый из которых вызывает `printf`, поэтому программа генерирует четыре выходных строки.

Листинг 8.3. Два вызова fork()

```
1 #include "csapp.h"
2
3 int main()
4 {
5     Fork();
6     Fork();
7     printf("hello!\n");
8     exit(0);
9 }
```

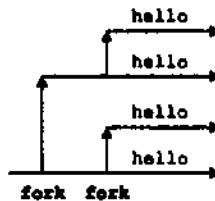


Рис. 8.12. Выводятся четыре выходных строки

Продолжая двигаться в том же направлении, зададимся вопросом, а что произойдет, если мы вызовем `fork` три раза, как в листинге 8.4? Как видно из графа процесса на рис. 8.13, теперь общее количество процессов достигло восьми. Каждый процесс вызывает `printf`, поэтому программа образует восемь выходных строк.

Листинг 8.4. Программа вызывает трижды `fork`

```

1 #include "csapp.h"
2
3 int main()
4 {
5     Fork();
6     Fork();
7     Fork();
8     printf("hello!\n");
9     exit(0);
10 }
```

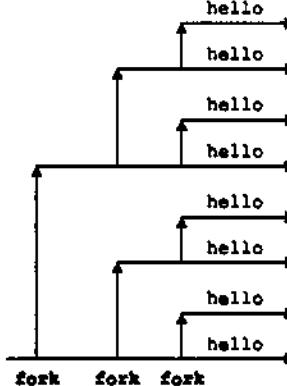


Рис. 8.13. Выводятся восемь выходных строк

УПРАЖНЕНИЕ 8.1

Рассмотрим следующую программу:

```

1 #include "csapp.h"
2
3 int main()
4 {
5     int X = 1;
6
7     if (Fork() == 0)
8         printf("printf1 : x=%d\n", ++x);
9     printf("printf2 : x=%d\n", --x);
10    exit(0);
11 }
```

1. Каков будет вывод дочернего процесса?
2. Каков будет вывод родительского процесса?

УПРАЖНЕНИЕ 8.2

Сколько строк "hello" будет выведено этой программой?

```

1 #include "csapp.h"
2
3 int main()
4 {
5     int i;
6
7     for (i = 0; i < 2; i++)
8         Fork();
9     printf("hello!\n");
10    exit(0);
11 }
```

УПРАЖНЕНИЕ 8.3

Сколько строк "hello" будет выведено этой программой?

```

1 #include "csapp.h"
2
3 void doit()
4 {
5     Fork();
6     Fork();
7     printf("hello\n");
8     return;
9 }
10
11 int main()
12 {
13     doit();
14     printf("hello\n");
15     exit(0);
16 }
```

8.4.3. Снятие дочерних процессов

После того как процесс завершается по той или иной причине, ядро не удаляет его из системы немедленно. Процесс остается в состоянии завершенности до тех пор, когда он будет снят (reap) своим родителем. После того как родитель снимет завершенный дочерний процесс, ядро передает родителю статус выхода дочернего процесса, а за-

тем удаляет завершенный процесс, и в этот момент тот прекращает свое существование. Завершенный, но еще не удаленный процесс называется зомби.

Почему завершенные дочерние процессы называются зомби

В обыденной речи зомби представляет собой как бы живой труп, существо, которое наполовину живо, а наполовину мертвое. Процесс-зомби имеет с ним сходство в том смысле, что, несмотря на то, что он уже завершился, ядро поддерживает некоторые его состояния до тех пор, когда он будет снят своим родителем.

Если родительский процесс завершается, не сняв свои дочерние процессы-зомби, ядро принимает меры, чтобы их снял процесс `init`. Процесс `init` имеет PID, равный 1, и порождается ядром во время инициализации системы. Такие долго исполняющиеся программы, как оболочки или серверы, должны всегда снимать свои дочерние процессы-зомби. Даже при том, что зомби не исполняются, они все еще потребляют ресурсы памяти системы.

Процесс ожидает завершения или остановки своих дочерних процессов, вызывая функцию `waitpid`.

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *status, int options);
```

Функция возвращает PID дочернего процесса, если все в порядке, 0 (если `WNOHANG`), иначе -1 — в случае ошибки. По умолчанию (когда `options = 0`) `waitpid` приостанавливает исполнение вызывающего процесса, до тех пор, когда завершится дочерний процесс в его наборе ожидания. Если процесс в наборе ожидания уже завершился к моменту запроса, то `waitpid` возвращается немедленно. И в том, и в другом случае `waitpid` возвращает PID завершенного дочернего процесса, и завершенный дочерний процесс будет удален из системы.

Члены набора ожидания определяются с помощью параметра `pid`:

- Если `pid > 0`, то набор ожидания — это одиночный дочерний процесс, чей ID равен `pid`.
- Если `pid = -1`, то набор ожидания включает все дочерние процессы родителя.

Наборы ожидающих процессов

Функция `waitpid` поддерживает также другие виды наборов ожидания, включая группы процессов Unix, которые мы здесь не будем рассматривать.

Изменение поведения по умолчанию

Поведение по умолчанию можно изменять, присваивая переменной `options` различные логические комбинации значений констант `WNOHANG` и `WUNTRACED`:

- `WNOHANG` — возвратится тотчас же (с возвращаемым значением 0), если ни один из дочерних процессов в наборе ожидания еще не завершился.

- **WUNTRACED** — приостановит исполнение вызывающего процесса до тех пор, когда процесс в наборе ожидания становится завершенным или остановившимся. Возвращает PID завершенного или остановленного дочернего процесса, который послужил причиной этого возвращения.
- **WNOHANG | WUNTRACED** — приостановит исполнение вызывающего процесса до тех пор, когда дочерний процесс в наборе ожидания завершится или остановится, и затем возвратить PID приостановленного или завершенного дочернего процесса, который послужил причиной этого возврата. Кроме того, возвратиться тотчас же (с возвращаемым значением 0), если ни один из процессов в наборе ожидания еще не завершен или не остановлен.

Проверка статуса выхода снятого дочернего процесса

Если окажется, что аргумент `status` не `NULL`, то `waitpid` кодирует в аргументе `status` информацию о статусе дочернего процесса, который послужил причиной возврата. Включаемый файл `wait.h` содержит несколько макроопределений для интерпретации аргумента `status`:

- **WIFEXITED** возвращает значение истины, если дочерний процесс завершился normally, посредством вызова `exit` или с помощью `return`.
- **WEXITSTATUS** возвращает статус выхода normally завершенного дочернего процесса. Этот статус определяется только в том случае, если **WIFEXITED** возвратил истину.
- **WIFSIGNALED** возвращает значение истины, если дочерний процесс завершился ввиду того, что сигнал не был перехвачен.
- **WTERMSIG** возвращает номер сигнала, который был причиной завершения дочернего процесса. Этот статус определяется только в том случае, если **WIFSIGNALED** возвратил истину.
- **WIFSTOPPED** возвращает значение истины, если дочерний процесс, который послужил причиной возврата, в настоящее время остановлен.
- **WSTOPSIG** возвращает номер сигнала, который был причиной остановки дочернего процесса. Этот статус определяется только в том случае, если **WIFSTOPPED** возвратил истину.

Состояния ошибки

Если вызывающий процесс не имеет никаких дочерних процессов, то `waitpid` возвращает `-1` и устанавливает в `errno` значение `ECHILD`. Если функция `waitpid` была прервана сигналом, то она возвращает `-1` и устанавливает в `errno` значение `EINTR`.

Константы, связанные с функциями Unix

Такие константы, как **WNOHANG** и **WUNTRACED**, определены в системном файле заголовков. **HANG** и **WUNTRACED** определены (косвенно) в файле заголовков `wait.h`:

```
/* биты в третьем параметре waitpid */
#define WNOHANG 1 /* не блокировать ожидание */
#define WUNTRACED 2 /* сообщает статус остановленных дочерних процессов */
```

Для того чтобы использовать эти константы, в программный код следует включить файл заголовков `wait.h`:

```
#include <sys/wait.h>
```

Страница тап для каждой функции Unix содержит список файлов заголовков, которые необходимо включать всякий раз, когда эта функция используется в программе. Кроме того, для того чтобы можно было проверять коды возврата, такие как `ECHILD` и `EINTR`, следует включать `errno.h`. Для упрощения программных примеров мы включаем единственный файл заголовков `csapp.h`, который включает файлы заголовков для всех функций, используемых в этой книге. Файл заголовков `csapp.h` содержится в *приложении 2*.

Примеры

В листинге 8.5 представлена программа, в которой порождается N дочерних процессов, используется `waitpid` для ожидания их завершения, и затем проверяется статус выхода каждого завершенного дочернего процесса.

Листинг 8.5. Снятие дочерних процессов

```
1 #include "csapp.h"
2 #define N 2
3
4 int main()
5 {
6     int status, i;
7     pid_t pid;
8
9     for (i = 0; i < N; i++)
10        if ((pid = Fork()) == 0) /* дочерний процесс */
11            exit(100+i);
12
13    /* родитель ожидает завершения всех своих дочерних процессов */
14    while ((pid = waitpid(-1, &status, 0)) > 0) {
15        if (WIFEXITED(status))
16            printf("child %d terminated normally with exit status=%d\n",
17                   pid, WEXITSTATUS(status));
18        else
19            printf("child %d terminated abnormally\n", pid);
20    }
21    if (errno != ECHILD)
22        unix_error("waitpid error");
23 }
```

```
24     exit(0);
25 }
```

Если мы запустим на счет эту программу в системе Unix, то на выходе появится следующий вывод:

```
unix> ./waitpid1
child 22966 terminated normally with exit status=100
child 22967 terminated normally with exit status=101
```

Обратите внимание, что в программе не снимаются дочерние процессы в каком-либо определенном порядке. В листинге 8.6 показано, каким образом можно было бы использовать waitpid, чтобы снять дочерние процессы из листинга 8.5 в том же самом порядке, в каком они были порождены родителем.

Листинг 8.6. Снятие дочерних процессов в порядке их порождения

```
1 #include "csapp.h"
2 #define N 2
3
4 int main()
5 {
6     int status, i;
7     pid_t pid[N+1], retpid;
8
9     for (i = 0; i < N; i++)
10        if ((pid[i] = Fork()) == 0)          /* child */
11            exit(100 + i);
12
13    /* родитель снимает N дочерних процессов по очереди */
14    i = 0;
15    while ((retpid = waitpid(pid[i++], &status, 0)) > 0) {
16        if (WIFEXITED(status))
17            printf("child %d terminated normally with exit status=%d\n",
18                   retpid, WEXITSTATUS(status));
19        else
20            printf("child %d terminated abnormally\n", retpid);
21    }
22
23    /* нормальное завершение возможно, если только больше не осталось
       никаких дочерних процессов */
24    if (errno != ECHILD)
25        unix_error("waitpid error");
26
27    exit(0);
28 }
```

УПРАЖНЕНИЕ 8.4

Рассмотрите следующую программу:

```

1 #include "csapp.h"
2
3 int main()
4 {
5     int status;
6     pid_t pid;
7
8     printf("Hello\n");
9     pid = Fork();
10    printf("%d\n", !pid);
11    if (pid != 0) {
12        if (waitpid(-1, &status, 0) > 0) {
13            if (WIFEXITED(status) != 0)
14                printf("%d\n", WEXITSTATUS(status));
15        }
16    }
17    printf("Bye\n");
18    exit(2);
19 }
```

- Сколько выходных строк будет сгенерировано этой программой?
- Укажите один из возможных порядков вывода этих строк.

8.4.4. Перевод процессов в состояние "сна"

Функция `sleep` приостанавливает процесс на некоторый период времени.

```
#include <unistd.h>
unsigned int sleep(unsigned int secs);
```

Функция `sleep` возвращает нуль, если запрошенное время истекло, и количество "недоспанных" секунд — в противном случае. Последний случай возможен, если функция `sleep` исполняет операцию возврата преждевременно. А это происходит в том случае, когда она прерывается сигналом. Подробно сигналы мы рассмотрим в разд. 8.5.

Еще одна полезная функция — это функция `pause`, которая помещает вызывающую функцию в состояние сна, до тех пор, когда процессом будет получен сигнал.

```
#include <unistd.h>
int pause(void);
```

Функция всегда возвращает `-1`.

УПРАЖНЕНИЕ 8.5

Напишите функцию-оболочку для sleep, с именем snooze, со следующим интерфейсом:

```
unsigned int snooze(unsigned int secs);
```

Функция snooze ведет себя в точности так же, как функция sleep, за исключением того, что она выдает сообщение, дающее представление о том, как долго процесс фактически пребывал в спящем состоянии: спал в течение 4 из 5 секунд.

8.4.5. Загрузка и запуск программ на исполнение

Функция execve загружает и запускает на счет новую программу в контексте текущего процесса.

```
#include <unistd.h>
int execve(char *filename, char *argv[], char *envp);
```

При успешном завершении она не возвращается в вызывающий процесс, а в случае ошибки возвращает `-1`.

Функция execve загружает и запускает на счет исполняемый объектный файл `filename` со списком параметров `argv` и списком переменных среды исполнения `envp`. Функция execve возвращает управление в вызывающую программу, только если произойдет ошибка, например, такая как отсутствие файла с заданным именем. Поэтому в отличие от функции fork, которая вызывается один раз, но возвращает значение дважды, execve вызывается один раз и никогда не возвращается.

Список параметров представлен структурой данных, показанной на рис. 8.14. Переменная `argv` указывает на завершающийся нулевым элементом массив указателей, каждый из которых указывает на строку аргумента. По соглашению, `argv[0]` — это имя исполняемого объектного файла. Список переменных среды исполнения представлен аналогичной структурой данных, показанной на рис. 8.15. Переменная `envp` указывает на завершающийся нулевым элементом массив указателей на строки переменных среды исполнения, каждая из которых представляет собой пару имени и значения в форме "name=value".

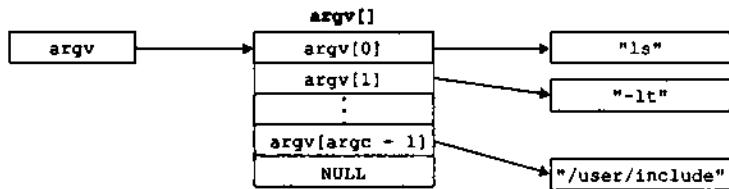


Рис. 8.14. Организация списка параметров

После того как execve загрузит `filename`, она вызывает программный код запуска, описанный в разд. 7.9. Код запуска устанавливает стек и передает управление в главную подпрограмму новой программы, прототип которой имеет форму

```
int main(int argc, char **argv, char **envp);
```

или, что эквивалентно,

```
int main(int argc, char *argv[], char *envp[]);
```

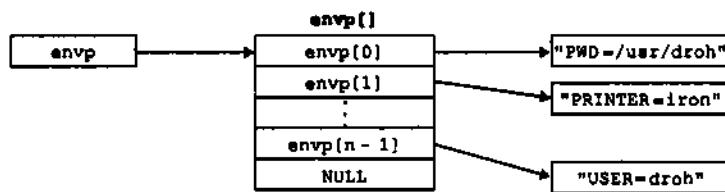


Рис. 8.15. Организация списка переменных среды исполнения

Когда main начнет исполняться в системе Linux, пользовательский стек будет иметь организацию, показанную на рис. 8.16. Пройдемся от основания стека (самый высокий адрес) к его вершине (самый низкий адрес). Первыми там будут строки аргументов и среды исполнения, которые хранятся рядом в стеке, последовательно одна за другой, без каких-либо промежутков. Далее в этом направлении следует завершающийся нулевым элементом массив указателей, каждый из которых указывает на одну из строковых переменных среды исполнения в стеке. На первый из этих указателей envp[0] указывает глобальная переменная environ. За массивом указателей среды исполнения непосредственно следует завершающийся нулевым элементом массив argv[], каждый элемент которого указывает на одну из строк аргументов в стеке.

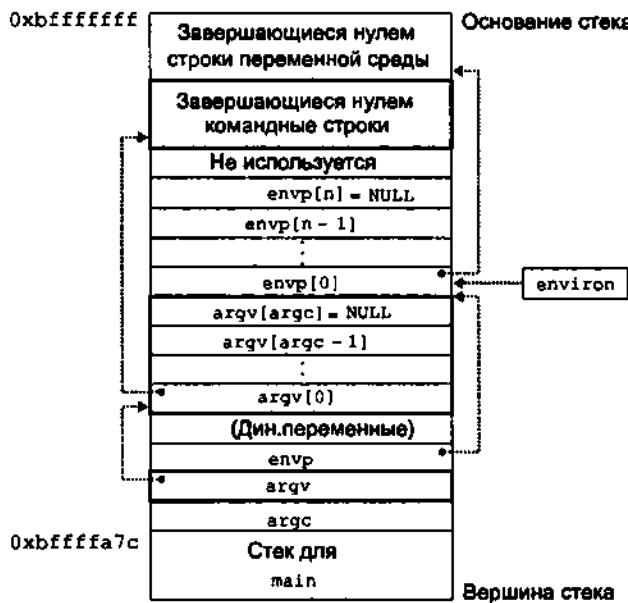


Рис. 8.16. Типичная организация пользовательского стека в тот момент, когда запускается новая программа

Наверху стека находятся три аргумента подпрограммы main:

1. envp, указывающий на массив envp[].
2. argv, указывающий на массив argv[].
3. argc, который дает количество ненулевых указателей в массиве argv[].

Unix предоставляет некоторые функции для управления массивом среды исполнения:

```
#include <stdlib.h>
char *getenv(const char *name);
```

Функция getenv просматривает в массиве среды исполнения строки вида "name =value". Если строка найдена, функция возвращает указатель на value, в противном случае она возвращает NULL.

```
#include <stdlib.h>
int setenv(const char *name, const char *newvalue, int overwrite);
```

Функция setenv возвращает 0 в случае успеха, -1 — в случае ошибки.

```
void unsetenv(const char *name);
```

Функция unsetenv не возвращает ничего.

Если массив среды исполнения содержит строку в форме "name=oldvalue", то unsetenv удаляет ее, и setenv заменяет oldvalue на newvalue, но только если overwrite отлично от нуля. Если name не существует, то setenv добавляет к массиву элемент "name=newvalue".

Установка переменных среды исполнения в системе Solaris

Вместо функции setenv в Solaris имеется функция putenv. Но в ней нет никакого аналога функции unsetenv.

Программы и процессы

Здесь как раз подходящее место, чтобы остановиться и проверить себя, достаточно ли хорошо вы усвоили различия между программой и процессом. Программа представляет собой совокупность программного кода и данных; программы могут существовать в виде объектных модулей на диске, или как сегменты в адресном пространстве. Процесс представляет собой конкретный экземпляр программы в стадии исполнения; программа всегда исполняется в контексте некоторого процесса. Понимание сущности этого различия важно, если вы хотите разобраться в функциях fork и execve. Функция fork исполняет ту же самую программу в новом дочернем процессе, представляющем собой дубликат родителя. Функция execve загружает и запускает на счет новую программу в контексте текущего процесса. Несмотря на то, что она переписывает адресное пространство текущего процесса, эта функция не порождает нового процесса. Новая программа по-прежнему имеет такой же самый PID, и она наследует все дескрипторы файлов, которые были открыты к моменту вызова функции execve .

УПРАЖНЕНИЕ 8.6

Напишите программу, называющуюся `myecho`, которая печатает аргументы своей командной строки и переменные среды исполнения. Например:

```
unix> ./myecho arg1 arg2
```

Аргументы командной строки:

```
argv[0]: myecho
argv[1]: arg1
argv[2]: arg2
```

Переменные среды исполнения:

```
envp[0] : PWD=/usr0/droh/ics/code/ecf
envp[1] : TERM=emacs
...
envp[25]: USER=droh
envp[26]: SHELL=/usr/local/bin/tcsh
envp[27]: HOME=/usr0/droh
```

8.4.6. Использование функций для запуска программ

Такие программы, как командные процессоры Unix и серверы Web (см. главу 12), интенсивно используют функции `fork` и `execve`. Командный процессор представляет собой прикладную диалоговую программу, которая запускает на счет другие программы по поручению пользователя. Самым первым командным процессором была программа `sh`, за которой последовали такие ее варианты, как `csh`, `tcsh`, `ksh` и `bash`. Командный процессор выполняет последовательность шагов чтения/обработки, после чего заканчивает свою работу. На шаге чтения происходит ввод пользовательской командной строки. На шаге обработки анализируется командная строка и исполняются программы по поручению пользователя.

В листинге 8.7 представлена главная подпрограмма простейшего командного процессора. Командный процессор выводит приглашение ввода командной строки, ожидает, когда пользователь введет свою командную строку через `stdin`, после чего обрабатывает эту командную строку.

Листинг 8.7. Главная подпрограмма простейшего командного процессора

```
1 #include "csapp.h"
2 #define MAXARGS 128
3
4 /* прототипы функций */
5 void eval(char *cmdline);
6 int parseline(char *buf, char **argv);
7 int builtin_command(char **argv);
8
```

```

9 int main()
10 {
11     char cmdline[MAXLINE]; /* командная строка */
12
13     while (1) {
14         /* чтение */
15         printf("> ");
16         fgets(cmdline, MAXLINE, stdin);
17         if (feof(stdin))
18             exit(0);
19
20         /* обработка */
21         eval(cmdline);
22     }
23 }
```

В листинге 8.8 представлен программный код, обрабатывающий командную строку. Его первая задача состоит в том, чтобы вызвать функцию `parseline` (листинг 8.9), которая производит синтаксический анализ разделенных пробелами аргументов командной строки и формирует вектор `argv`, который, в конечном счете, будет передан в `execve`. Предполагается, что первым аргументом будет либо имя встроенной команды командного процессора, которая будет интерпретироваться непосредственно, либо имя исполнимого объектного файла, который будет загружен и запущен на счет в контексте нового дочернего процесса.

Листинг 8.8. Обработка командной строки командного процессора

```

1 /* eval - обрабатывает командную строку */
2 void eval(char *cmdline)
3 {
4     char *argv[MAXARGS]; /* argv для execve () */
5     char buf [MAXLINE]; /* хранит модифицированную командную строку */
6     int bg; /* должно ли задание выполняться в bg или fg? */
7     pid_t pid; /* id процесса */
8
9     strcpy(buf, cmdline);
10    bg = parseline(buf, argv);
11    if (argv[0] == NULL)
12        return; /* игнорируем пустые строки */
13
14    if (!builtin_command(argv)) {
15        if ((pid = Fork()) == 0) { /* дочерний процесс выполняет
16                                  пользовательское задание */
17            if (execve(argv[0], argv, environ) < 0) {
18                printf("%s : Command not found.\n", argv[0]);
```

```

18             exit(0);
19         }
20     }
21
22     /* родительский процесс ожидает окончания приоритетного задания */
23     if (!bg) {
24         int status;
25         if (waitpid(pid, &status, 0) < 0)
26             unix_error("waitfg : waitpid error");
27     }
28     else
29         printf("%d %s", pid, cmdline);
30 }
31 return;
32 }
33
34 /* если первый аргумент является встроенной командой, то она исполняется
   и возвращается истина */
35 int builtin_command(char **argv)
36 {
37     if (!strcmp(argv[0], "quit"))      /* команда выхода */
38         exit(0);
39     if (!strcmp(argv[0], "&"))        /* игнорируем одиночный символ & */
40         return 1;
41     return 0;                         /* не встроенная команда */
42 }
```

Листинг 8.9 Синтаксический разбор строки ввода для командного процессора

```

1  /* parsesline - синтаксический разбор командной строки и формирование
   массива argv */
2  int parsesline(char    *buf,    char    **argv)
3  {
4      char *delim;                  /* указывает на первый пробел-разделитель */
5      int argc;                   /* количество аргументов */
6      int bg;                     /* фоновое задание? */
7
8      buf [strlen(buf)-1] = ' ';   /* заменяем конечный символ '\n'
   на пробел */
9      while (*buf && (*buf == ' ')) /* пропускаем ведущие пробелы */
10         buf++;
11
12     /* формируем список argv */
13     argc = 0;
14     while ((delim = strchr(buf, ' '))) {
15         argv[argc++] = buf;
16         *delim = '\0';
17     }
18     argv[argc] = NULL;
19 }
```

```

17     buf = delim + 1;
18     while (*buf && (*buf == ' ')) /* пропускаем пробелы */
19         buf++;
20     }
21     argv[argc] = NULL;
22
23     if (argc == 0)                  /* пропускаем пустую строку */
24         return 1;
25
26     /* задание исполняется в фоновом режиме? */
27     if ((bg = (*argv[argc-1] == '&')) != 0)
28         argv[--argc] = NULL;
29
30     return bg;
31 }

```

Если последним аргументом является символ "&", то `parseline` возвращает 1, показывая что программа должна быть выполнена в фоновом режиме (командный процессор не ожидает ее завершения). В противном случае она возвращает 0, показывая, что программа должна выполняться в приоритетном режиме (командный процессор ожидает ее завершения).

После синтаксического разбора командной строки `eval` вызывает функцию `builtin_command`, которая проверяет, является ли первый аргумент командной строки встроенной командой командного процессора. Если да, то она интерпретирует команду непосредственно и возвращает 1. В противном случае она возвращает 0. Наш простейший командный процессор имеет только одну встроенную команду, команду `quit`, которая завершает работу командного процессора. Реальные командные процессоры имеют довольно много команд, например `pwd`, `jobs` и `fg`.

Если `builtin_command` возвращает 0, то командный процессор порождает дочерний процесс и запускает на исполнение запрошенную программу в дочернем процессе. Если пользователь определил, что эта программа должна запускаться на исполнение в фоновом режиме, то командный процессор возвращается к началу цикла и ожидает ввода последующей командной строки. В противном случае командный процессор использует функцию `waitpid`, чтобы перейти к ожиданию завершения данного задания. В тот момент, когда задание завершается, командный процессор переходит к следующей итерации.

Имейте в виду, что этот простейший командный процессор — неполноценный, поскольку он не снимает никаких своих фоновых дочерних процессов. Исправление этого недостатка потребует использования сигналов, которые мы опишем в следующем разделе.

8.5. Сигналы

К этому моменту в процессе изучения потоков управления с исключениями мы познакомились с тем, каким образом взаимодействуют аппаратные и программные

средства, обеспечивая базовый механизм исключений низкого уровня. Мы также познакомились с тем, каким образом операционная система использует исключения, чтобы поддержать высокоуровневую форму передачи управления по исключению, известную как контекстное переключение. В этом разделе мы изучим высокоуровневую форму программных исключений (сигналы), которая дает возможность процессам прерывать другие процессы.

Сигнал представляет собой сообщение, что в системе произошло событие некоторого типа. Например, в табл. 8.3 представлены 30 различных типов сигналов, поддерживающихся на Linux-системах.

Таблица 8.3. Сигналы в Linux

Номер	Имя	Действие по умолчанию	Соответствует событию
1	SIGHUP	Завершить	Разрыв линии с терминалом
2	SIGINT	Завершить	Прерывание от клавиатуры
3	SIGQUIT	Завершить	Сигнал выхода от клавиатуры
4	SIGILL	Завершить	Незаконная машинная команда
5	SIGTRAP	Завершить и вывести дамп содержимого области памяти	Системное прерывание трассировки
6	SIGABRT	Завершить и вывести дамп содержимого области памяти	Сигнал аварийного завершения от функции <code>abort</code>
7	SIGBUS	Завершить	Ошибка шины
8	SIGFPE	Завершить и вывести дамп содержимого области памяти	Исключение от операции с плавающей точкой
9	SIGKILL	Завершить	Прекратить исполнение программы
10	SIGUSR1	Завершить	Пользовательский сигнал 1
11	SIGSEGV	Завершить и вывести дамп содержимого области памяти	Незаконная ссылка на адрес памяти (сбой при обращении к сегменту)
12	SIGUSR2	Завершить	Пользовательский сигнал 2
13	SIGPIPE	Завершить	Запись в программный канал при отсутствии адресата
14	SIGALRM	Завершить	Сигнал таймера от функции <code>alarm</code>
15	SIGTERM	Завершить	Программный сигнал завершения

Таблица 8.3 (окончание)

Номер	Имя	Действие по умолчанию	Соответствует событию
16	SIGSTKFLT	Завершить	Ошибка стека в сопроцессоре
17	SIGCHLD	Игнорировать	Дочерний процесс остановлен или завершен
18	SIGCONT	Игнорировать	Продолжение процесса, если он остановлен
19	SIGSTOP	Остановить до появления SIGCONT	Сигнал останова не от терминала
20	SIGTSTP	Остановить до появления SIGCONT	Сигнал останова от терминала
21	SIGTTIN	Остановить до появления SIGCONT	Фоновый процесс читает из терминала
22	SIGTTOU	Остановить до появления SIGCONT	Фоновый процесс пишет в терминал
23	SIGURG	Игнорировать	Срочное состояние в сокете
24	SIGXCPU	Завершить	Истекло время центрального процессора
25	SIGXFSZ	Завершить	Превышен предельный размер файла
26	SIGVTALRM	Завершить	Сигнал истечения времени от виртуального таймера
27	SIGPROF	Завершить	Сигнал истечения времени от таймера профилирования
28	SIGWINCH	Игнорировать	Размер окна изменился
29	SIGIO	Завершить	Ввод-вывод теперь возможен по дескриптору
30	SIGPWR	Завершить	Сбой в системе электропитания

Примечание к таблице

Много лет назад оперативная память реализовывалась с использованием технологии, известной как оперативное запоминающее устройство (ОЗУ) на магнитных сердечниках. "Дамп содержимого ОЗУ" — это исторический термин, который означает сброс образов программного кода и сегментов данных из памяти на диск.

Каждый тип сигнала соответствует некоторому событию в системе. Аппаратные исключения низкого уровня обрабатываются обработчиками исключительных ситуаций

ядра и обычно не видимы для пользовательских процессов. Сигналы обеспечивают механизм представления возникших исключений. Например, если процесс делает попытку деления на нуль, то ядро отправляет ему сигнал SIGFPE (сигнал 8). Если процесс исполняет незаконную команду, ядро отправляет ему сигнал SIGILL (сигнал 4). Если процесс пытается выполнить незаконную ссылку на адрес памяти, ядро отправляет ему сигнал SIGSEGV (сигнал 11). Другие сигналы соответствуют высокуюровневым программным событиям в ядре или в других пользовательских процессах. Например, если ввести с клавиатуры `<Ctrl>+<C>` (одновременно нажать клавиши), когда процесс исполняется в приоритетном режиме, то ядро отправляет этому приоритетному процессу SIGINT (сигнал 2). Процесс может принудительно закончить другой процесс путем посылки ему сигнала SIGKILL (сигнал 9). Сигналы 9 и 19 не могут быть ни перехвачены, ни проигнорированы. В тот момент, когда дочерний процесс завершается или останавливается, ядро отправляет родительскому процессу сигнал SIGCHLD (сигнал 17).

8.5.1. Терминология, связанная с сигналами

Передача сигнала процессу назначения происходит в два этапа:

- Постылка сигнала. Ядро посыпает (отправляет) сигнал процессу назначения путем корректировки некоторых состояний в контексте процесса назначения. Сигнал может отправляться по одной из двух причин:
 - ядро обнаружило событие — например, ошибка деления на нуль или завершение дочернего процесса;
 - процесс вызвал функцию `kill` (см. разд. 8.5.2), чтобы явно запросить ядро отправить сигнал процессу назначения. Процесс может отправить сигнал и самостоятельно.
- Получение сигнала. Процесс назначения получает сигнал в тот момент, когда он побуждается ядром к тому, чтобы отреагировать некоторым способом на посыпку сигнала. Процесс может либо игнорировать сигнал, либо завершаться, либо перехватить сигнал, исполнив функцию пользовательского уровня, называемую *обработчиком сигнала*.

Сигнал, который послан, но еще не получен, называется *задержанным сигналом*. В каждый момент времени может быть не более одного задержанного сигнала любого конкретного типа. Если процесс имеет задержанный сигнал типа k , то никакие последующие сигналы типа k , отправленные этому процессу, не ставятся в очередь; они просто теряются. Процесс может выборочно блокировать получение некоторых сигналов. Если сигнал блокирован, он может отправляться, но результирующий задержанный сигнал не будет получен до тех пор, пока процесс не разблокирует этот сигнал.

Задержанный сигнал может быть получен, самое большее, один раз. Для каждого процесса ядро поддерживает набор задержанных сигналов в векторе битов `pending` и набор блокированных сигналов в векторе битов `blocked`. Ядро устанавливает бит k в `pending` всякий раз, когда отправляется сигнал типа k и очищает бит k в `pending` всякий раз, когда сигнал типа k становится полученным.

8.5.2. Посылка сигналов

В системах Unix имеется несколько механизмов, обеспечивающих посылку сигналов процессам. Все эти механизмы базируются на понятии группы процессов.

Группы процессов

Каждый процесс принадлежит в точности одной группе процессов, которая идентифицируется положительным целочисленным *ID группы процессов*. Функция `getpgid` возвращает ID группы процессов для текущего процесса.

```
#include <unistd.h>
pid_t getpgid(void);
```

По умолчанию дочерний процесс принадлежит той же самой группе процессов, что и его родительский процесс. Путем использования функции `setpgid` процесс может изменить свою собственную группу процессов или группу процессов другого процесса:

```
#include <unistd.h>
pid_t setpgid(pid_t pid, pid_t pgid);
```

Функция `setpgid` возвращает 0 в случае успешного завершения, -1 — в случае ошибки и изменяет группу процессов процесса `pid` на `pgid`. Если `pid` равен нулю, то используется PID текущего процесса. Если `pgid` равен нулю, то PID процесса, заданный с помощью `pid`, используется для ID группы процессов. Например, если процесс 15213 — это вызывающий процесс, то

```
setpgid(0, 0);
```

создает новую группу процессов, для которой ID группы процессов равен 15213, и добавляет процесс 15213 к этой новой группе.

Программа `/bin/kill` отправляет произвольный сигнал другому процессу. Например, команда

```
unix> kill -9 15213
```

отправляет сигнал 9 (SIGKILL) процессу 15213. Отрицательное значение PID приводит к тому, что сигнал будет отправлен каждому процессу в группе процессов PID. Например, команда

```
unix> kill -9 -15213
```

посыпает сигнал SIGKILL каждому процессу в группе процессов 15213.

Посылка сигналов с клавиатуры

Командные процессоры Unix используют абстракцию задания (job) для представления процессов, порождаемых в результате обработки отдельной командной строки. В любой момент времени имеется, самое большое, одно приоритетное задание и, возможно, несколько фоновых заданий.

Например, введение

```
unix> ls | sort
```

создает приоритетное задание, состоящее из двух процессов, соединенных с помощью программного канала Unix: один выполняет программу `ls`, другой — программу `sort`.

Командный процессор для каждого задания образует отдельную группу процессов. Как правило, ID группы процессов берется от родительского процесса в этом задании. Например, на рис. 8.17 представлен командный процессор с одним приоритетным заданием и двумя фоновыми заданиями. PID родительского процесса в приоритетном задании равен 20, и ID группы процессов равен 20. Родительский процесс породил два дочерних процесса, каждый из которых является также членом группы процессов с ID, равным 20.

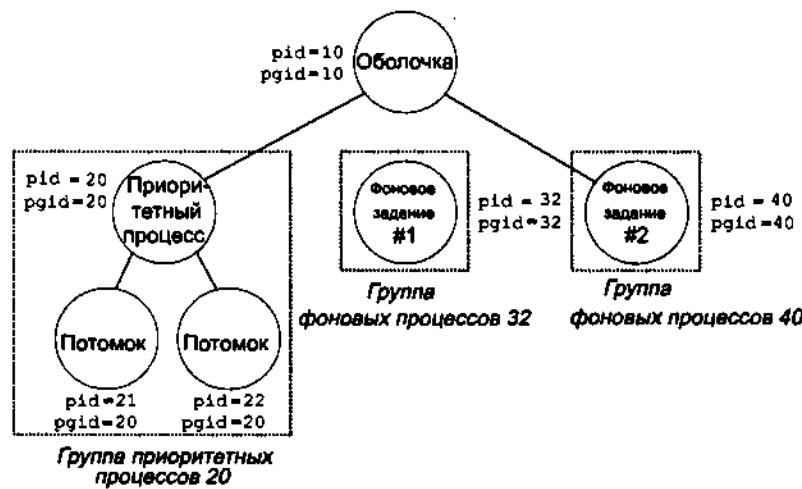


Рис. 8.17. Приоритетные и фоновые группы процессов

Ввод `<Ctrl>+<C>` с клавиатуры приводит к тому, что командному процессору будет отправлен сигнал `SIGINT`. Командный процессор перехватывает этот сигнал (см. раздел 8.5.3) и затем посыпает `SIGINT` каждому процессу в группе приоритетных процессов. Стандартное действие по умолчанию состоит в том, чтобы завершить приоритетное задание. Точно так же ввод `<Ctrl>+<Z>` посыпает командному процессору сигнал `SIGTSTP`, который тот перехватывает и посыпает сигнал `SIGTSTP` каждому процессу в группе приоритетных процессов. Стандартное действие по умолчанию состоит в том, чтобы остановить (приостановить) приоритетное задание.

Посылка сигналов с помощью функций

Процессы посыпают сигналы другим процессам (в том числе и себе) путем вызова функции `kill`:

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

Функция возвращает 0, если все в порядке, -1 — в случае ошибки. Если pid больше нуля, то функция kill посылает процессу pid номер сигнала sig. Если pid меньше нуля, то kill посылает сигнал sig каждому процессу в группе процессов abs(pid). В листинге 8.10 представлен пример родительского процесса, который использует функцию kill, чтобы послать сигнал SIGKILL его дочернему процессу.

Листинг 8.10. Использование функции kill для отправки сигнала дочернему процессу

```
1 #include "csapp.h"
2
3 int main()
4 {
5     pid_t pid;
6
7     /* дочерний процесс засыпает до тех пор, когда будет получен сигнал
       SIGKILL, затем умирает */
8     if ((pid = Fork()) == 0) {
9         Pause(); /* ожидает поступления сигнала */
10        printf("control should never reach here!\n");
11        exit(0);
12    }
13
14    /* родительский процесс отправляет сигнал SIGKILL дочернему процессу */
15    Kill(pid, SIGKILL);
16    exit(0);
17 }
```

Процесс может отправлять себе сигналы SIGALRM путем вызова функции alarm.

```
#include <unistd.h>
unsigned int alarm(unsigned int secs);
```

Функция alarm организует посылку ядром сигнала SIGALRM вызывающему процессу через secs секунд. Если secs равно нулю, то никакого нового сигнала тревоги не планируется. В любом случае, вызов alarm отменяет все задержанные сигналы тревоги и возвращает количество секунд, оставшееся до того момента, когда какой-либо задержанный сигнал тревоги должен был бы быть отправлен (если этот вызов alarm не отменяет его), или 0, если таких задержанных сигналов тревоги не было.

В листинге 8.11 представлена программа, вызываемая alarm, которая в течение пяти секунд каждую секунду организует прерывание сигналом SIGALRM. Когда посыпается шестой SIGALRM, она завершается. При исполнении программы каждую се-

кунду в течение пяти секунд будет слышен звуковой сигнал: "BEEP", после чего по-следует сигнал "BOOM!" — и программа завершится:

```
unix> ./alarm
BEEP
...
BEEP
BOOM!
```

Листинг 8.11. Использование функции signal

```
1 #include "csapp.h"
2
3 void handler(int sig)
4 {
5     static int beeps = 0;
6
7     printf("BEEP\n");
8     if (++beeps < 5)
9         Alarm(1); /* следующий SIGALRM будет послан через 1 сек */
10    else {
11        printf("BOOM!\n");
12        exit (0);
13    }
14 }
15
16 int main()
17 {
18     Signal(SIGALRM, handler); /* install SIGALRM handler */
19     Alarm(1); /* следующий SIGALRM будет послан через 1 сек */
20
21     while (1) {
22         ; /* сюда обработчик сигнала каждый раз возвращает
23           управление */
24     }
25 }
```

Обратите внимание, что программа в листинге 8.11 использует функцию `signal` для установки функции *обработчика сигнала* (`handler`), которая вызывается асинхронно, прерывая бесконечный цикл `while` в `main` всякий раз, когда процесс получает сигнал `SIGALRM`. Когда функция `handler` возвращается, управление передается назад в `main`, в ту точку, где она была прервана поступившим сигналом. Установка и использование обработчиков сигнала может оказаться довольно тонким делом, но это — тема последующих трех разделов.

8.5.3. Получение сигналов

Когда в ядро возвращается управление из обработчика исключительных ситуаций, и оно готово передать управление процессу p , оно проверяет набор разблокированных задержанных сигналов (pending & ~blocked). Если это множество пусто (обычная ситуация), то ядро передает управление на следующую команду (I_{next}) в логическом потоке управления p .

Тем не менее, если данное множество не пусто, то ядро выбирает некоторый сигнал k в этом множестве (обычно наименьшее значение k) и делает p получателем сигнала k . Получение сигнала вызывает некоторое ответное действие процесса. Как только процесс завершает это действие, управление передается назад на следующую команду (I_{next}) в логическом потоке управления p . Каждый тип сигнала имеет предопределеное заданное по умолчанию действие, одно из перечисленных:

- Процесс завершается.
- Процесс завершается, и выводится содержимое области памяти.
- Процесс останавливается до того момента, когда он будет перезапущен сигналом SIGCONT.
- Процесс игнорирует данный сигнал.

В табл. 8.3 представлены стандартные действия, связанные с каждым типом сигнала. Например, стандартное действие при получении SIGKILL состоит в том, чтобы завершить процесс получения. С другой стороны, стандартное действие при получении SIGCHLD состоит в том, чтобы игнорировать этот сигнал. Процесс может изменить стандартное действие, связанное с сигналом путем использования функции signal. Исключениями являются только SIGSTOP и SIGKILL, чьи стандартные действия не могут быть изменены.

```
#include <signal.h>
typedef void handler_t(int);
handler_t *signal(int signum, handler_t *handler);
```

Функция signal возвращает указатель на предыдущий обработчик, если все в порядке, и SIG_ERR — в случае ошибки (не устанавливает errno). Она может изменить действие, связанное с использованием сигнала signum, одним из трех способов:

- Если handler содержит SIG_JGN, то сигналы типа signum пропускаются.
- Если сработчик содержит SIG_DFL, то действиями для сигналов типа signum снова будут стандартные действия по умолчанию.
- В противном случае, handler — это адрес определенной пользователем функции, называемой обработчиком сигнала, которая будет вызываться всякий раз, когда процесс получает сигнал типа signum. Изменение стандартного действия путем передачи этого адреса обработчика функции signal называется установкой обработчика. Вызов обработчика называется перехватом сигнала. Исполнение обработчика называется обработкой сигнала.

Когда процесс перехватывает сигнал типа k , обработчик, установленный для сигнала k , вызывается с использованием единственного целочисленного аргумента, значение

которого равно k . Этот аргумент дает возможность одной и той же функции обработчика перехватывать различные типы сигналов.

Когда обработчик исполняет оператор возврата, управление (обычно) передается назад на ту команду в потоке управления, в которой процесс был прерван получением сигнала. Мы говорим "обычно" потому, что в некоторых системах прерванный системный вызов немедленно возвращает ошибку.

В листинге 8.12 представлена программа, которая перехватывает сигнал SIGINT, отправляемый командным процессором всякий раз, когда пользователь наберет на клавиатуре $<\text{Ctrl}>+<\text{C}>$. Стандартное действие для SIGINT состоит в том, чтобы немедленно завершить процесс. В этом примере мы изменяем стандартное поведение на перехват сигнала, печать сообщения и затем — завершение процесса.

Листинг 8.12. Программа перехвата сигнала SIGINT

```

1 #include "csapp.h"
2
3 void handler(int sig)      /* обработчик SIGINT */
4 {
5     printf("Caught SIGINT\n");
6     exit(0);
7 }
8 int main()
9 {
10    /* установка обработчика SIGINT */
11    if (signal(SIGINT, handler) == SIG_ERR)
12        unix_error("signal error");
13
14    pause();      /* ожидаем получения сигнала */
15
16    exit(0);
17 }
```

Функция обработчика определена в строках 3—7. Главная подпрограмма в строках 12—13 устанавливает обработчик, после чего процесс переходит в состояние сна до того момента, когда будет получен сигнал (строка 15). После того как сигнал SIGINT будет получен, исполняется обработчик, выводится сообщение (строка 5), после чего процесс завершается (строка 6).

УПРАЖНЕНИЕ 8.7

Напишите программу с именем snooze, которая принимает из командной строки единственный аргумент, вызывает функцию snooze из упр. 8.5 с этим аргументом, после чего завершается. Напишите эту программу таким образом, чтобы пользователь мог вызвать прерывание функции snooze, вводя $<\text{Ctrl}>+<\text{C}>$ с клавиатуры.

Например:

```
unix> ./snooze 5
Slept for 3 of 5 secs. Пользователь нажимает <Ctrl>+<C> через 3 секунды
unix>
```

8.5.4. Некоторые вопросы обработки сигналов

Для программ, в которых перехватывается единственный сигнал, обработка сигнала выполняется просто и заканчивается их завершением. Тем не менее, если программа перехватывает несколько сигналов, возникают некоторые проблемы.

- Задержанные сигналы заблокированы. Обработчики сигнала в Unix обычно блокируют задержанные сигналы того типа, который в настоящее время обрабатывается данным обработчиком. Например, предположим, что процесс перехватил сигнал SIGINT, и в настоящее время исполняется его обработчик SIGINT. Если другой сигнал SIGINT будет послан процессу, то этот SIGINT станет задержанным и не будет принят до тех пор, пока данный обработчик не возвратится.
- Задержанные сигналы не находятся в очереди. Допустимо существование, самое большое, одного задержанного сигнала любого конкретного типа. Таким образом, если два сигнала типа k отправлены одному процессу назначения, в то время как сигнал k блокирован, ввиду того, что этот процесс назначения в настоящее время исполняет обработчик для сигнала k , то второй сигнал просто теряется; он не становится в очередь. Ключевая идея состоит в том, что факт существования задержанного сигнала только показывает, что поступил, по крайней мере, один такой сигнал.
- Системные вызовы могут быть прерваны. Системные вызовы, такие как `read`, `wait` и `accept`, которые могут потенциально блокировать процесс в течение длительного времени, называются *медленными системными вызовами*. В некоторых системах медленные системные вызовы, которые прерываются, когда обработчик перехватывает сигнал, не восстанавливаются, когда управление возвращается из обработчика сигнала, но вместо этого управление возвращается непосредственно пользователю с указанием состояния ошибки, и в `errno` устанавливается значение `EINTR`.

Рассмотрим более подробно тонкости обработки сигнала, используя простейшее приложение, которое по существу аналогично реальным программам, таким как командные процессоры и Web-серверы. Суть дела состоит в том, что родительский процесс порождает несколько дочерних процессов, которые некоторое время исполняются независимо, после чего завершаются. Родительский процесс должен снять дочерние процессы, чтобы не оставлять в системе зомби. Но мы также хотим, чтобы родительский процесс был в состоянии делать другую работу в то время, когда исполняются дочерние процессы. Поэтому мы приняли решение снимать дочерние процессы с помощью обработчика `SIGCHLD` вместо того, чтобы явно ожидать завершения дочерних процессов. Вспомните, что ядро посылает сигнал `SIGCHLD` родительскому процессу всякий раз, когда один из его дочерних процессов завершается или останавливается.

Листинг 8.13 представляет нашу первую попытку. Родительский процесс устанавливает обработчик SIGCHLD и затем образует три дочерних процесса, каждый из которых исполняется в течение одной секунды, а затем завершается. Тем временем родительский процесс ожидает строку ввода с терминала и затем обрабатывает ее. Эта обработка организована с помощью бесконечного цикла. При завершении каждого дочернего процесса ядро уведомляет об этом родительский процесс, посыпая ему сигнал SIGCHLD. Родительский процесс перехватывает SIGCHLD, снимает один из дочерних процессов, выполняет некоторую дополнительную работу по очистке, представленную оператором sleep(2), и затем возвращает управление.

Программа signal1 на первый взгляд кажется необычайно простой. Но если мы запустим ее на счет в системе Linux, то получим следующий выход:

```
linux> ./signal1
Hello from child 10320
Hello from child 10321
Hello from child 10322
Handler reaped child 10320
Handler reaped child 10322
<cr>
Parent processing input
```

Листинг 8.13. Программа в первой попытке

```
1 #include "csapp.h"
2
3 void handler1(int sig)
4 {
5     pid_t pid;
6
7     if ((pid = waitpid(-1, NULL, 0)) < 0)
8         unix_error("waitpid error");
9     printf("Handler reaped child %d\n", (int)pid);
10    Sleep(2);
11    return;
12 }
13
14 int main()
15 {
16     int i, n;
17     char buf[MAXBUF];
18
19     if (signal(SIGCHLD, handler1) == SIG_ERR)
20         unix_error("signal error");
21 }
```

```

22     /* родительский процесс порождает дочерний процесс */
23     for (i = 0; i < 3; i++) {
24         if (Fork() == 0) {
25             printf("Hello from child %d\n", (int)getpid());
26             Sleep(1);
27             exit(0);
28         }
29     }
30
31     /* родительский процесс ожидает ввод с терминала, и затем обрабатывает
   его */
32     if ((n = read(STDIN_FILENO, buf, sizeof(buf))) < 0)
33         unix_error("read");
34
35     printf("Parent processing input\n");
36     while (1)
37         ;
38
39     exit(0);
40 }
```

Из этого выхода мы можем сделать заключение, что хотя родительскому процессу были отправлены три сигнала SIGCHLD, получены были только два из них, и, таким образом, родительский процесс снял только два дочерних процесса. Если мы приостановим родительский процесс, мы увидим, что действительно, дочерний процесс 10321 не был снят, и остается зомби:

```

<Ctrl>+<Z>
Suspended
linux> ps
PID TTY STAT TIME COMMAND
...
10319 p5 T 0:03 signall
10321 p5 Z 0:00 (signall <zombie>)
10323 p5 R 0:00 ps
```

В чем причина неудачного счета? Проблема состоит в том, что эта программа не в состоянии учесть того обстоятельства, что сигналы могут быть заблокированы, и что сигналы не ставятся в очередь. Вот что здесь произошло.

Первый сигнал получен и перехвачен родительским процессом. В то время, когда обработчик еще занят обработкой первого сигнала, второй сигнал поступает и добавляется к множеству ждущих обработки сигналов. Тем не менее, поскольку сигналы типа SIGCHLD блокированы обработчиком SIGCHLD, второй сигнал не будет получен. Вскоре после этого, когда обработчик все еще занят обработкой первого сигнала, поступает третий сигнал. Поскольку имеется уже задержанный SIGCHLD, этот третий сигнал SIGCHLD будет потерян. В дальнейшем, в какой-то момент времени,

после того как обработчик возвратит управление, ядро обнаруживает, что имеется задержанный сигнал SIGCHLD, и побуждает родительский процесс получить этот сигнал. Родительский процесс перехватывает сигнал и запускает обработчик во второй раз. После того как обработчик заканчивает обработку второго сигнала, задержанных сигналов SIGCHLD больше не имеется, и уже не будет, потому что все сведения о третьем SIGCHLD были потеряны. Из всего этого можно сделать вывод о том, что сообщения нельзя использовать для подсчета появления событий в других процессах.

Чтобы устранить эту проблему, необходимо вспомнить, что наличие ждущего обработки сигнала означает только то, что, по крайней мере, один сигнал поступил с тех пор, как в последний раз процесс получал сигнал данного типа. Поэтому необходимо модифицировать обработчик SIGCHLD так, чтобы при каждом вызове снимать по возможности больше дочерних процессов-зомби. Листинг 8.14 представляет модифицированный обработчик SIGCHLD.

Листинг 8.14. Улучшенная версия программы

```
1 #include "csapp.h"
2
3 void handler2(int sig)
4 {
5     pid_t pid;
6
7     while((pid = waitpid(-1, NULL, 0)) > 0)
8         printf("Handler reaped child %d\n", (int)pid);
9     if (errno != ECHILD)
10         unix_error("waitpid error");
11     Sleep(2);
12     return;
13 }
14
15 int main()
16 {
17     int i, n;
18     char buf[MAXBUF];
19
20     if (signal(SIGCHLD, handler2) == SIG_ERR)
21         unix_error("signal error");
22
23     /* родительский процесс порождает дочерний процесс */
24     for (i = 0; i < 3; i++) {
25         if (Fork() == 0) {
26             printf("Hello from child %d\n", (int)getpid());
27             Sleep(1);
```

```
28         exit(0);
29     }
30 }
31
32 /* родительский процесс ожидает ввод с терминала, и затем обрабатывает
его */
33 if ((n = read(STDIN_FILENO, buf, sizeof(buf))) < 0)
34     unix_error("read error");
35
36 printf("Parent processing input\n");
37 while (1)
38     ;
39
40 exit(0);
41 }
```

Если запустить на счет signal2 в системе Linux, то теперь правильно снимаются все дочерние процессы-зомби:

```
linux> ./signal2
Hello from child 10378
Hello from child 10379
Hello from child 10380
Handler reaped child 10379
Handler reaped child 10378
Handler reaped child 10380
<cr>
Parent processing input
```

Тем не менее мы еще не проделали всей работы. Если мы запустим на счет программы signal2 в системе Solaris, она правильно снимает все дочерние процессы-зомби. Однако теперь блокированный системный вызов read возвращается преждевременно с ошибкой, еще до того, как мы сможем ввести что-либо с клавиатуры:

```
solaris> ./signal2
Hello from child 18906
Hello from child 18907
Hello from child 18908
Handler reaped child 18906
Handler reaped child 18908
Handler reaped child 18907
read: Interrupted system call
```

В чем причина неудачного счета? Проблема здесь возникает потому, что в такой системе, как Solaris, медленные системные вызовы, например read, автоматически не возобновляют исполнения после того, как они были прерваны полученным сигналом. Вместо этого они преждевременно возвращаются в вызывающее приложение с со-

стоянием ошибки, в отличие от Linux-систем, которые автоматически возобновляют исполнение прерванных системных вызовов.

Для того чтобы написать переносимый программный код обработки сигнала, необходимо учитывать возможность преждевременного возврата из системного вызова, и возобновлять его исполнение программно, если это произойдет. В листинге 8.15 представлена модификация программы `signall`, в которой программно возобновляется исполнение аварийно завершенного вызова `read`. Код возврата `EINTR` в `errno` показывает, что системный вызов `read` преждевременно возвратил управление, после того как он был прерван.

Листинг 8.15. Версия программы, правильно учитывющая системные вызовы

```
1 #include "csapp.h"
2
3 void handler2(int sig)
4 {
5     pid_t pid;
6
7     while((pid = waitpid(-1, NULL, 0)) > 0)
8         printf("Handler reaped child %d\n", (int)pid);
9     if (errno != ECHILD)
10        unix_error("waitpid error");
11     Sleep(2);
12     return;
13 }
14
15 int main() {
16     int i, n;
17     char buf[MAXBUF];
18     pid_t pid;
19
20     if (signal(SIGCHLD, handler2) == SIG_ERR)
21         unix_error("signal error");
22
23     /* родительский процесс порождает дочерний процесс */
24     for (i = 0; i < 3; i++) {
25         pid = Fork();
26         if (pid == 0) {
27             printf("Hello from child %d\n", (int)getpid());
28             Sleep(1);
29             exit(0);
30         }
31     }
32 }
```

```

33     /* программное возобновление исполнения вызова read после его
34     прерывания */
35     while ((n = read(STDIN_FILENO, buf, sizeof(buf))) < 0)
36         if (errno != EINTR)
37             unix_error("read error");
38
39     printf("Parent processing input\n");
40     while (1)
41         ;
42     exit(0);
43 }

```

Если наша новая программа signal3 запускается на счет в системе Solaris, ее исполнение происходит правильно:

```

solaris> ./signal3
Hello from child 19571
Hello from child 19572
Hello from child 19573
Handler reaped child 19571
Handler reaped child 19572
Handler reaped child 19573
<cr>
Parent processing input

```

8.5.5. Переносимая обработка сигналов

Различные системы по-разному реализуют семантику обработки сигнала. Такой, например, аспект, как поведение в результате прерванного медленного системного вызова: возобновляется ли его исполнение, или он аварийно преждевременно завершается — представляет собой неприятную сторону обработки сигналов в Unix. Чтобы преодолеть эту проблему, разработан стандарт POSIX, который определяет функцию `sigaction`, дающую возможность пользователям в POSIX-совместимых системах, например Linux и Solaris, четко определить семантику обработки сигнала, которую они хотят соблюдать в своих разработках.

```
#include <signal.h>
int sigaction(int signum, struct sigaction *act, struct sigaction *oldact);
```

Функция `sigaction` возвращает 0, если все в порядке, -1 в случае ошибки, она довольно громоздка, поскольку требует, чтобы пользователь установил элементы структуры. Более ясный подход, впервые предложенный Стивенсоном (Stevens) [81], состоит в том, чтобы определить функцию-оболочку, называемую `Signal`, которая вызывает `sigaction`. В листинге 8.16 представлено определение функции `Signal`, ко-

торая вызывается точно так же, как функция `signal`. Функция-оболочка `Signal` устанавливает обработчик сигнала со следующей семантикой обработки сигнала:

- Блокируются только сигналы типа, обрабатываемого в настоящее время обработчиком.
- В любой реализации сигналы не ставятся в очередь.
- Прерванные системные вызовы автоматически возобновляют свое исполнение, как только появляется такая возможность.
- Как только обработчик сигнала будет установлен, он остается установленным до того момента, когда `Signal` будет вызван с аргументом `handler`, либо `SIG_IGN`, либо `SIG_DFL`. Некоторые прежние системы Unix восстанавливают действие сигнала к его стандартному, заданному по умолчанию значению, после того как сигнал будет обработан обработчиком.

Листинг 8.18: Функция-оболочка Signal

```

1 handler_t *Signal(int signum, handler_t *handler)
2 {
3     struct sigaction action, old_action;
4
5     action.sa_handler = handler;
6     sigemptyset(&action.sa_mask); /* блокируем сигналы обрабатываемого
7                             типа */
8     action.sa_flags = SA_RESTART; /* возобновляем исполнение системных
9                             вызовов, если это возможно */
10
11    if (sigaction(signum, &action, &old_action) < 0)
12        unix_error("Signal error");
13    return (old_action.sa_handler);
14 }
```

В листинге 8.17 представлена версия программы `signal2` из листинга 8.14, в которой используется функция-оболочка `Signal`, что дает предсказуемую семантику обработки сигнала в различных компьютерных системах. Единственное различие между ними состоит в том, что мы установили обработчик с вызовом `Signal`, вместо вызова `signal`. Программа теперь исполняется правильно как в системе Solaris, так и в Linux, и мы больше не должны самостоятельно возобновлять исполнение прерванных системных вызовов `read`.

Листинг 8.17: Функция-оболочка позволяет получить переносимую семантику

```

1 #include "csapp.h"
2
3 void handler2(int sig)
4 {
5     pid_t pid;
6 }
```

```

7     while((pid = waitpid(-1, NULL, 0)) > 0)
8         printf("Handler reaped child %d\n", (int)pid);
9     if (errno != ECHILD)
10        unix_error("waitpid error");
11     Sleep(2);
12     return;
13 }
14
15 int main()
16 {
17     int i, n;
18     char buf[MAXBUF];
19     pid_t pid;
20
21     Signal (SIGCHLD, handler2); /* функция-оболочка обработчика ошибок
22                                sigaction */
23
24     /* родительский процесс порождает дочерний процесс */
25     for (i = 0; i < 3; i++) {
26         pid = Fork();
27         if (pid == 0) {
28             printf("Hello from child %d\n", (int)getpid());
29             Sleep(1);
30             exit(0);
31         }
32     }
33
34     /* родительский процесс ожидает ввода с терминала, и затем обрабатывает
35     его */
36     if ((n = read(STDIN_FILENO, buf, sizeof(buf))) < 0)
37         unix_error("read error");
38
39     printf("Parent processing input\n");
40     while (1)
41         ;
42     exit(0);
43 }
```

8.5.6. Явная блокировка сигналов

Приложения могут явно блокировать и разблокировать избранные сигналы, используя функцию `sigprocmask`:

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
int sigemptyset(sigset_t *set);
```

```
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
Возвращает: 0, если все в порядке, -1 — в случае ошибки
int sigismember(const sigset_t *set, int signum);
```

Функция `sigprocmask` возвращает 1, если это элемент набора, 0 — если не принадлежит набору, и -1 — в случае ошибки. Она изменяет набор блокированных в настоящее время сигналов (вектор битов `blocked` описан в разд. 8.5.1). Конкретное поведение зависит от значения `how`:

- `SIG_BLOCK` добавляет сигналы, находящиеся в `set`, к `blocked` (`blocked = blocked | set`).
- `SIG_UNBLOCK` удаляет сигналы, находящиеся в `set`, из `blocked` (`blocked = blocked & ~set`).
- `SIG_SETMASK` есть `blocked = set`.

Если `oldset` не `NULL`, то предыдущее значение вектора битов `blocked` сохраняется в `oldset`.

Наборами сигналов, например `set`, можно управлять, используя следующие функции.

- Функция `sigemptyset` инициализирует `set`, очищая этот набор.
- Функция `sigfillset` добавляет каждый сигнал к `set`.
- Функция `sigaddset` добавляет `signum` к `set`, `sigdelset` удаляет `signum` из `set`.
- Функция `sigismember` возвращает 1, если `signum` — это элемент `set`, и 0, если это не так.
- Функция `sigprocmask` удобна для синхронизации родительских и дочерних процессов. Например, рассмотрим листинг 7.18, где представлены основные черты структуры типичного командного процессора Unix. Родительский процесс ведет учет своих дочерних процессов в списке заданий (job list). Когда родительский процесс порождает новый дочерний процесс, он добавляет этот дочерний процесс в список заданий. Когда родительский процесс снимает завершенный (зомби) дочерний процесс в обработчике `SIGCHLD`, он удаляет этот дочерний процесс из списка заданий.

Листинг 8.18. Синхронизация процессов

```
1 void handler(int sig)
2 {
3     pid_t pid;
4     while ((pid = waitpid(-1, NULL, 0)) > 0) /* снимаем дочерний
   процесс-зомби */
5         deletejob(pid); /* удаляем дочерний процесс из списка заданий */
6     if (errno!= ECHILD)
7         unix_error("waitpid error");
8 }
```

```

10 int main(int argc, char **argv)
11 {
12     int pid;
13     sigset(SIGCHLD, handler);
14
15     Signal(SIGCHLD, handler);
16     initjobs(); /* инициализируем список заданий */
17
18     while(1) {
19         Sigemptyset(&mask);
20         Sigaddset(&mask, SIGCHLD);
21         Sigprocmask(SIG_BLOCK, &mask, NULL); /* блокируем SIGCHLD */
22
23         /* дочерний процесс */
24         if((pid = Fork()) == 0) {
25             Sigprocmask(SIG_UNBLOCK, &mask, NULL); /* разблокируем SIGCHLD */
26             Execve("/bin/ls", argv, NULL);
27         }
28
29         /* родительский процесс */
30         addjob(pid); /* добавляем дочерний процесс в список заданий */
31         Sigprocmask(SIG_UNBLOCK, &mask, NULL); /* разблокируем SIGCHLD */
32     }
33     exit(0);
34 }
```

В этом примере родительский процесс обеспечивает, чтобы добавление задания выполнялось до того, как будет вызвана соответствующая функция `deletejob`. Если мы не синхронизируем родительский и дочерний процессы тем или иным образом, то возможна такая последовательность событий.

- Родительский процесс инициирует функцию `fork`, ядро выбирает вновь порожденный дочерний процесс и запускает его на исполнение вместо родительского процесса.
- Еще до того, как родительский процесс сможет снова запуститься на исполнение, дочерний процесс завершается и становится зомби, вызывая посылку ядром сигнала `SIGCHLD` в родительский процесс.
- В дальнейшем, когда родительский процесс снова становится способным к исполнению, но еще до того, как он будет запущен, ядро обнаруживает задержанный сигнал `SIGCHLD`, и это приводит к тому, что он будет получен исполняющимся обработчиком в родительском процессе.
- Обработчик снимает завершенный дочерний процесс, пытается удалить задание, но при этом ничего не происходит, ввиду того, что родительский процесс еще не успел добавить данный дочерний процесс в список.

- После того как обработчик завершает свою работу, ядро запускает на счет родительский процесс, который возвращается из fork, и путем вызова addjob добавляет несуществующий дочерний процесс в список заданий.

Дело в том, что если мы специально ничего не предпримем, то возможна ситуация, когда deletejob будет вызвана перед addjob. Листинг 7.18 показывает один из способов, как устранить эту проблему. Блокирование сигнала SIGCHLD перед вызовом fork и разблокирование его только после вызова addjob гарантирует, что дочерний процесс может быть снят только после того, как он будет добавлен в список заданий.

Обратите внимание, что дочерние процессы наследуют набор blocked своих родительских процессов, поэтому следует быть внимательным при разблокировании сигнала SIGCHLD в дочернем процессе, перед тем как вызывать execve.

8.6. Нелокальные передачи управления

Язык С предоставляет пользователям форму передачи управления по исключению, называемую нелокальным переходом, который передает управление непосредственно из одной функции в другую, исполняющуюся в настоящее время, без необходимости совершить обычную последовательность действий "вызов с возвратом". Нелокальные переходы реализованы функциями setjmp и longjmp.

```
#include <setjmp.h>
int setjmp(jmp_buf env);
int sigsetjmp(sigjmp_buf env, int savesigs);
```

Функция setjmp возвращает 0 из setjmp, ненулевое значение — из longjmp, сохраняет текущий контекст стека в буфере env для его последующего использования в longjmp.

```
#include <setjmp.h>
void longjmp(jmp_buf env, int retval);
void siglongjmp(sigjmp_buf env, int retval);
```

Функция longjmp восстанавливает контекст стека из буфера env и после этого инициирует возврат из самого последнего вызова setjmp, который инициализировал env. После этого setjmp возвращает отличное от нуля значение retval.

На первых порах взаимодействия между setjmp и longjmp могут показаться запутанными и непонятными. Функция setjmp вызывается один раз, но возвращает управление несколько раз: один раз, когда setjmp вызывается первый раз и контекст стека сохраняется в буфере env, и по одному разу — для каждого вызова longjmp. В то же время функция longjmp вызывается один раз, но никогда не возвращает управления.

Важное прикладное значение нелокальных переходов заключается в том, что они допускают непосредственный возврат из глубоко вложенных вызовов функций, обычно после обнаружения некоторых состояний ошибки. Если состояние ошибки обнаруживается в глубоко вложенном вызове функции, можно использовать нелокальную передачу управления.

кальный переход, чтобы возвратиться непосредственно в общедоступный обработчик ошибок вместо утомительного раскручивания стека вызовов.

В листинге 8.19 показан пример того, каким образом это можно осуществить. Подпрограмма `main` сначала вызывает `setjmp`, чтобы сохранить текущий контекст стека, и после этого вызывает функцию `foo`, которая в свою очередь вызывает функцию `bar`. Если в `foo` или в `bar` возникает ошибка, они возвращают управление непосредственно из `setjmp` через вызов `longjmp`. Отличное от нуля возвращаемое из `setjmp` значение показывает тип ошибки, которая после этого может быть расшифрована и обработана в одном месте программного кода.

Листинг 8.19 Пример на локального перехода

```
1 #include "csapp.h"
2
3 jmp_buf buf;
4
5 int error1 = 0;
6 int error2 = 1;
7
8 void foo(void), bar(void);
9
10 int main()
11 {
12     int rc;
13
14     rc = setjmp(buf);
15     if (rc == 0)
16         foo();
17     else if (rc == 1)
18         printf("Detected an error1 condition in foo\n");
19     else if (rc == 2)
20         printf("Detected an error2 condition in foo\n");
21     else
22         printf("Unknown error condition in foo\n");
23     exit(0);
24 }
25
26 /* глубоко вложенная функция foo */
27 void foo(void)
28 {
29     if (error1)
30         longjmp(buf, 1);
31     bar();
32 }
33
```

```
34 void bar(void)
35 {
36     if (error2)
37         longjmp(buf, 2);
38 }
```

Этот пример показывает структуру, использующую нелокальные переходы, с целью возвратиться из состояния ошибки в глубоко вложенной функции — без того, чтобы раскручивать полный стек.

Другое важное приложение нелокальных переходов состоит в том, чтобы вызывать переход из обработчика сигнала в заданный адрес памяти в программе вместо того, чтобы возвращаться к команде, в которой произошло прерывание от поступившего сигнала. В листинге 8.20 показана простая программа, которая иллюстрирует этот стандартный прием. Программа использует сигналы и нелокальные переходы, позволяющие делать мягкий рестарт всякий раз, когда пользователь вводит с клавиатуры <Ctrl>+<C>. Функции `sigsetjmp` и `siglongjmp` — версии `setjmp` и `longjmp`, которые могут быть использованы обработчиками сигналов.

Листинг 8.20. Нелокальные переходы для перезапуска

```
1 #include "csapp.h"
2
3 sigjmp_buf buf;
4
5 void handler(int sig)
6 {
7     siglongjmp(buf, 1);
8 }
9
10 int main()
11 {
12     Signal(SIGINT, handler);
13
14     if (!sigsetjmp(buf, 1))
15         printf("starting\n");
16     else
17         printf("restarting\n");
18
19     while(1) {
20         Sleep(1);
21         printf("processing...\n");
22     }
23     exit(0);
24 }
```

Самый первый вызов функции `sigsetjmp` сохраняет стек и контекст сигнала, когда программа только стартует. Затем программа `main` входит в бесконечный цикл обработки. Если пользователь наберет `<Ctrl>+<C>`, то командный процессор отправит данному процессу сигнал `SIGINT`, который тот перехватит. Вместо того чтобы возвратиться из обработчика сигнала, который передал бы управление обратно в прерванный цикл обработки, обработчик выполняет нелокальный переход обратно к началу программы `main`. Если мы запустим на счет эту программу в нашей системе, мы получим следующий вывод:

```
unix> ./restart
starting
processing...
processing...
restarting      пользователь нажал <Ctrl>+<C> -C
processing...
restarting      пользователь нажал <Ctrl>+<C> -C
processing...
```

Программные исключения в C++ и Java

Механизмы исключений, предоставляемые языками C++ и Java, это высокоуровневые, более структурированные версии функций `setjmp` и `longjmp` языка С. Можно рассматривать предложение `catch` внутри предложения `try`, как некий аналог функции `setjmp`. Точно так же оператор `throw` имеет некоторое сходство с функцией `longjmp`.

8.7. Организация управления процессами

Системы Unix предоставляют несколько удобных средств для контроля и управления процессами:

- `strace` выводит трассировку каждого системного вызова, произведенного программой, и его дочерние процессы. Увлекательное средство для любопытных студентов. Компилируйте вашу программу с опцией `-static`, чтобы получить более чистую трассировку без громоздкого вывода, связанного с разделяемыми библиотеками;
- `ps` регистрирует процессы (включая зомби), находящиеся в настоящее время в системе;
- `top` выводит информацию об использовании ресурсов текущими процессами;
- `kill` посылает сигнал процессу. Удобен для отладки программ с обработчиками сигнала и очищает от непредсказуемых процессов;
- `/proc` (Linux and Solaris) — виртуальная файловая система, которая выводит содержимое многочисленных структур данных ядра в форме ASCII-текста и может читаться пользовательскими программами. Например, введите "cat /proc/loadavg", чтобы посмотреть текущую среднюю загрузку на вашей системе Linux.

8.8. Резюме

Передача управления по исключению может происходить на всех уровнях компьютерной системы. На аппаратном уровне исключения — это непредвиденные переходы в потоке управления, которые вызваны событиями в процессоре. Поток управления передается программному обработчику, который выполняет некоторые действия по обработке и после этого возвращает управление в прерванный поток управления.

Всего имеется четыре различных типа исключений: аппаратные прерывания, ошибки, аварийные завершения и системные прерывания. Аппаратные прерывания происходят асинхронно (относительно всех команд), если внешнее устройство ввода-вывода, например чип таймера или контроллер диска, подают сигнал прерывания на вывод в чипе процессора. Управление возвращается к команде, следующей за командой, вызвавшей ошибку. Ошибки и аварийные завершения происходят синхронно, как результат исполнения команды. Обработчики ошибок возобновляют исполнение команды, вызвавшей ошибку во время прерывания, тогда как обработчики никогда не возвращают управления в прерванный поток. Наконец, системные прерывания подобны вызовам функций, которые используются для реализации системных вызовов, обеспечивающих приложения управляемыми точками входа в программный код операционной системы.

На уровне операционной системы ядро обеспечивает реализацию фундаментальной концепции процесса. Процесс предоставляет приложениям две важные абстракции:

- логических потоков управления, которые создают для каждой программы иллюзию того, что она монопольно использует процессор;
- закрытых адресных пространств, обеспечивающих иллюзию того, что каждая программа монопольно использует оперативную память.

В среде интерфейса с операционной системой приложения могут порождать дочерние процессы, ожидать остановки или завершения дочерних процессов, запускать на счет новые программы, а также перехватывать сигналы от других процессов. Семантика обработки сигналов — это довольно тонкая материя, и она может изменяться от системы к системе. Тем не менее на Posix-совместимых системах имеются механизмы, которые дают возможность программам четко определять ожидаемую семантику обработки сигнала.

Наконец, на прикладном уровне C-программы могут использовать нелокальные переходы, позволяющие обойти нормальный стековый механизм вызовов, и совершать передачи управления непосредственно из одной функции в другую.

Библиографические замечания

Спецификация макроархитектуры Intel содержит подробное описание исключений и прерываний в процессорах Intel [18]. Тексты операционных систем [70, 75, 83] содержат дополнительную информацию об исключениях, процессах и сигналах. Классическая работа Стивенса (Stevens) [76], несмотря на то, что она отчасти устарела, остается ценным и очень подробным руководством по работе с процессами и сигналами из прикладных программ.

Задачи для домашнего решения

УПРАЖНЕНИЕ 8.8 +

В этой главе мы ввели некоторые функции с необычным поведением вызова и возврата управления: `setjmp`, `longjmp`, `execve` и `fork`. Сопоставьте каждой из этих функций одно из соответствующих ей поведений:

- вызывается один раз, возвращает управление дважды;
- вызывается один раз, но никогда не возвращает управления;
- вызывается один раз, возвращает управление один или несколько раз.

УПРАЖНЕНИЕ 8.9 +

Приведите один из возможных выводов следующей программы:

```

1 #include "csapp.h"
2
3 int main()
4 {
5     int x = 3;
6
7     if (Fork() != 0)
8         printf("x=%d\n", ++x);
9
10    printf("x=%d\n", --x);
11    exit(0);
12 }
```

УПРАЖНЕНИЕ 8.10 +

Сколько раз эта программа выведет строку "hello"?

```

1 #include "csapp.h"
2
3 void doit()
4 {
5     if (Fork() == 0) {
6         Fork();
7         printf("hello\n");
8         exit(0);
9     }
10    return;
11 }
12
13 int main()
14 {
15     doit();
```

```

16     printf("hello\n");
17     exit(0);
18 }
```

УПРАЖНЕНИЕ 8.11 ◆

Сколько раз эта программа выведет строку "hello"?

```

1 #include "csapp.h"
2
3 void doit()
4 {
5     if (Fork() == 0) {
6         Fork();
7         printf("hello\n");
8         return;
9     }
10    return;
11 }
12
13 int main()
14 {
15     doit();
16     printf("hello\n");
17     exit(0);
18 }
```

УПРАЖНЕНИЕ 8.12 ◆

Каков будет вывод следующей программы?

```

1 #include "csapp.h"
2 int counter = 1;
3
4 int main()
5 {
6     if (fork() == 0) {
7         counter--;
8         exit(0);
9     }
10    else {
11        Wait(NULL);
12        printf("counter = %d\n", ++counter);
13    }
14    exit(0);
15 }
```

УПРАЖНЕНИЕ 8.13 ♦

Перечислите все возможные выходы программы в упр. 8.4.

УПРАЖНЕНИЕ 8.14 ♦

Рассмотрите следующую программу:

```

1 #include "csapp.h"
2
3 void end(void)
4 {
5     printf("2");
6 }
7
8 int main()
9 {
10    if (Fork() == 0)
11        atexit(end);
12    if (Fork() == 0)
13        printf("0");
14    else
15        printf("1");
16    exit(0);
17 }
```

Определите, который из следующих выводов может иметь место. Обратите внимание: функция `atexit` принимает указатель на функцию и добавляет его к списку функций (в исходном положении список пуст), которые будут вызываться при вызове функции `exit`.

1. 112002;
2. 211020;
3. 102120;
4. 122001;
5. 100212.

УПРАЖНЕНИЕ 8.15 ++

Используя `execve`, напишите программу `myls`, поведение которой идентично поведению программы `/bin/ls`. Ваша программа должна принимать те же самые аргументы командной строки, интерпретировать те же самые переменные среды исполнения и производить идентичный выход.

Программа `ls` из переменной среды исполнения `COLUMNS` получает значение ширины экрана. Если `COLUMNS` не установлено, то `ls` предполагает, что экран имеет 80 столбцов в ширину. Таким образом, путем установки в `COLUMNS` значения меньше 80 можно контролировать переменные среды исполнения:

```
unix> setenv COLUMNS 40
unix> ./myls
... выход составляет 40 позиций в ширину
unix> unsetenv COLUMNS
unix> ./myls
... теперь выход составляет 80 позиций в ширину
```

УПРАЖНЕНИЕ 8.16 ***

Модифицируйте программу в листинге 8.5 таким образом, чтобы были удовлетворены следующие два условия:

1. Каждый дочерний процесс завершается ненормально после попытки произвести запись по адресу в процедурный сегмент только-для-чтения.
2. Родительский процесс производит вывод, идентичный (за исключением PID) следующему:

```
child 12255 terminated by signal 11: Segmentation fault
child 12254 terminated by signal 11: Segmentation fault
```

Совет: почитайте страницы man, для разделов wait(2) и psignal(3).

УПРАЖНЕНИЕ 8.17 ***

Напишите вашу собственную версию Unix-функции system:

```
int mysystem(char *command);
```

Функция mysystem исполняет command, делая вызов "/bin/sh -c command", и затем, как только command завершится, возвращает управление. Если command завершается обычным образом (путем вызова функции exit или исполнения оператора return), то mysystem возвращает состояние выхода из command. Например, если command завершается вызовом exit(8), то system возвращает значение 8. В противном случае, если command завершается неправильно, то mysystem возвращает состояние, возвращаемое командным процессором.

УПРАЖНЕНИЕ 8.18 *

Один из ваших коллег задумал использовать сигналы, передаваемые родительскому процессу, чтобы подсчитывать события, происходящие в дочернем процессе. Идея его состоит в том, чтобы уведомлять родительский процесс всякий раз, когда происходит событие, посыпая ему сигнал, с тем, чтобы обработчик сигнала родительского процесса увеличивал глобальную переменную counter, к которой родительский процесс впоследствии может обратиться, когда завершится дочерний процесс. Тем не менее, когда он на своей системе запускает на счет тестовую программу следующего листинга, он обнаруживает, что когда родительский процесс вызывает printf, счетчик всегда содержит значение 2, несмотря на то, что дочерний процесс послал пять сигналов родительскому процессу. Озадаченный, он приходит к вам за помощью. Можете ли вы объяснить, в чем состоит его ошибка?

```

1 #include "csapp.h"
2
3 int counter = 0;
4
5 void handler(int sig)
6 {
7     counter++;
8     sleep(1); /* даем возможность поработать обработчику */
9     return;
10 }
11
12 int main()
13 {
14     int i;
15
16     Signal(SIGUSR2, handler);
17
18     if (Fork() == 0) /* дочерний процесс */
19         for (i = 0; i < 5; i++) {
20             Kill(getpid(), SIGUSR2);
21             printf("sent SIGUSR2 to parent\n");
22         }
23         exit(0);
24     }
25
26     Wait(NULL);
27     printf("counter=%d\n", counter);
28     exit(0);
29 }
```

УПРАЖНЕНИЕ 8.19 ***

Напишите версию функции fgets, называющуюся tfgets, которая выдерживает время 5 секунд. Функция tfgets принимает те же самые входные данные, что и fgets. Если пользователь не ввел входную строку в течение 5 секунд, tfgets возвращает NULL. В противном случае она возвращает указатель на введенную строку.

УПРАЖНЕНИЕ 8.20 ****

Используя в качестве отправной точки пример в листинге 8.17, напишите программу командного процессора, которая поддерживает управление заданиями. Ваш командный процессор должен обладать нижеприведенными свойствами.

- Командная строка, набранная пользователем, представляет собой имя и возможные аргументы, разделенные одним или более пробелами. Если имя — это встроенная команда, командный процессор сразу же обрабатывает ее и ожидает появления последующей командной строки. В противном случае командный процессор предполагает, что имя — это исполняемый файл, который он загружает и

запускает на исполнение в контексте начального дочернего процесса (задание). ID группы процессов для задания — это аналог PID дочернего процесса.

- Каждое задание идентифицируется либо ID процесса (PID), либо ID задания (JID). Этот ID является произвольным небольшим положительным целым числом, назначенным командным процессором. JID обозначаются в командной строке префиксом %. Например, %5 обозначает JID 5, а 5 — PID 5.
- Если командная строка заканчивается амперсандом &, то командный процессор выполняет данное задание в фоновом режиме. В противном случае командный процессор выполняет задание как приоритетное.
- Если набрать на клавиатуре <Ctrl>+<C> (<Ctrl>+<Z>), то это приведет к тому, что командный процессор посыпает сигнал SIGINT (SIGTSTP) каждому процессу в группе приоритетных процессов.
- Встроенная команда jobs регистрирует все фоновые задания.
- Встроенная команда bg <job> возобновляет исполнение <job>, посыпая этому заданию сигнал SIGCONT, затем запускает его на счет в фоновом режиме. Аргумент <job> может быть либо PID, либо JID.
- Встроенная команда fg <job> возобновляет исполнение <job>, посыпая этому заданию сигнал SIGCONT, затем запускает его на счет как приоритетное.
- Командный процессор снимает все свои дочерние процессы зомби. Если какая-либо работа завершается из-за того, что она получает сигнал, который не был перехвачен, то командный процессор выводит сообщение на терминал с PID задания и описанием данного сигнала.

В следующем листинге представлен пример протокола сеанса работы командного процессора.

```
unix> ./shell                                Запускаем программу командного процессора
> bogus
bogus: Command not found.          Execve не может найти исполнимый модуль
> foo 10
Job 5035 terminated by signal: Interrupt      Пользователь вводит <Ctrl>+<C>
> foo 100 &
[1] 5036 foo 100 &
> foo 200 &
[2] 5037 foo 200 &
> jobs
[1] 5036 Running    foo 100 &
[2] 5037 Running    foo 200 &
> fg $1
Job [1] 5036 stopped by signal: Stopped      Пользователь вводит <Ctrl>+<Z>
> jobs
[1] 5036 Stopped    foo 100 &
[2] 5037 Running    foo 200 &
> bg 5035
5035: No such process
```

```

> bg 5036
[1] 5036 foo 100 &
> /bin/kill 5036
Job 5036 terminated by signal: Terminated
> fg %2                         Ожидает, когда закончится задание fg.
> quit
unix>                                Назад к командному процессору Unix

```

Решение упражнений

РЕШЕНИЕ УПРАЖНЕНИЯ 8.1

В программе нашего примера в листинге 8.1 родительский процесс и дочерний процесс исполняют непересекающиеся наборы команд. Однако в данной программе родительский процесс и дочерний процесс имеют идентичные сегменты программного кода. Концептуально это может оказаться трудным для понимания, поэтому постараитесь разобраться в решении этого задания.

1. Каков будет выход дочернего процесса? Ключевая идея здесь в том, что дочерний процесс исполняет оба оператора `printf`. После того как `fork` возвращает управление, он исполняет `printf` в строке 8. После этого происходит выход из оператора `if` и исполняется `printf` в строке 9. Вот выход, полученный от дочернего процесса:

```

printf1: x=2
printf2: x=1

```

2. Каков будет выход родительского процесса? Родительский процесс исполняет только `printf` в строке 9:

```
printf2: x=0
```

РЕШЕНИЕ УПРАЖНЕНИЯ 8.2

Эта программа имеет тот же самый граф процесса, что и программа в листинге 8.3. Всего имеется четыре процесса, каждый из которых выводит свою собственную строку "hello". Таким образом, программа выводит четыре строки "hello".

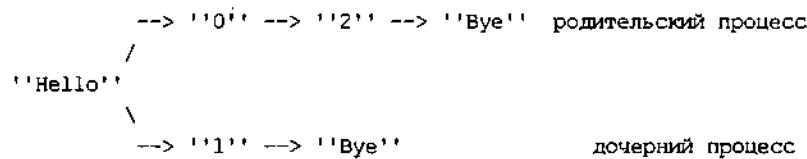
РЕШЕНИЕ УПРАЖНЕНИЯ 8.3

Эта программа имеет тот же самый график процесса, что и программа в листинге 8.4. Всего имеется четыре процесса, каждый из которых выводит свою собственную строку "hello" в `doit` и одну строку "hello" в `main`, после того как управление возвращается из `doit`. Таким образом, программа выводит всего восемь строк "hello".

РЕШЕНИЕ УПРАЖНЕНИЯ 8.4

1. всякий раз, когда мы запускаем на счет эту программу, она генерирует шесть выходных строк.

2. Порядок расположения выходных строк будет меняться от системы к системе, в зависимости от того, каким образом ядро чередует команды родительского и дочернего процессов. Вообще говоря, допустима любая топологически неизменная перетасовка вершин следующего графа:



Например, если мы запустим программу в нашей системе, мы получим следующий выход:

```
unix> ./waitprob1
Hello
0
1
Bye
2
Bye
```

В этом случае родительский процесс исполняется первым, выводя "Hello" в строке 8, а также "0" — в строке 10. Вызов wait блокируется ввиду того, что дочерний процесс еще не завершился, поэтому ядро делает контекстное переключение, а также передает управление на дочерний процесс, который выводит "1" в строке 10, а также "Bye" — в строке 16, и после этого завершается с состоянием выхода 2 в строке 17. После того как дочерний процесс завершается, возобновляется исполнение родительского процесса. При этом выводится состояние выхода дочернего процесса в строке 14, а также "Bye" — в строке 16.

РЕШЕНИЕ УПРАЖНЕНИЯ 8.5

```
1 unsigned int snooze(unsigned int secs) {
2     unsigned int rc = sleep(secs);
3     printf("Slept for %u of %u secs.\n", secs - rc, secs);
4     return rc;
5 }
```

РЕШЕНИЕ УПРАЖНЕНИЯ 8.6

```
1 #include "csapp.h"
2
3 int main(int argc, char *argv[], char *envp[])
4 {
5     int i;
```

```

7     printf("Command line arguments:\n");
8     for (i=0; argv[i] != NULL; i++)
9         printf("    argv[%d]: %s\n", i, argv[i]);
10
11    printf("\n");
12    printf("Environment variables:\n");
13    for (i=0; envp[i] != NULL; i++)
14        printf("    envp[%d]: %s\n", i, envp[i]);
15
16    exit(0);
17 }

```

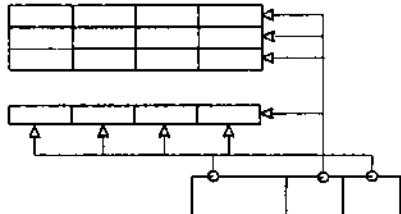
РЕШЕНИЕ УПРАЖНЕНИЯ 8.7

Функция sleep временно возвращает управление всякий раз, когда спящий процесс получает не игнорируемый сигнал. Но поскольку стандартное действие по SIGINT состоит в том, чтобы завершить процесс (см. табл. 8.3), следует установить обработчик SIGINT, чтобы дать возможность возвратиться из функции sleep. Обработчик просто перехватывает SIGNAL и возвращает управление функции sleep, которая возвращает его незамедлительно.

```

1 #include "csapp.h"
2
3 /* обработчик SIGINT */
4 void handler(int sig)
5 {
6     return; /* перехватываем сигнал и возвращаем управление */
7 }
8
9 unsigned int snooze(unsigned int secs) {
10    unsigned int rc = sleep(secs);
11    printf("Slept for %u of %u secs.\n", secs - rc, secs);
12    return rc;
13 }
14
15 int main(int argc, char **argv) {
16
17    if (argc != 2) {
18        fprintf(stderr, "usage: %s <secs>\n", argv[0]);
19        exit (0);
20    }
21
22    if (signal(SIGINT, handler) == SIG_ERR) /* устанавливаем обработчик
23                                              SIGINT */
24        unix_error("signal error\n");
25    (void)snooze(atoi(argv[1]));
26    exit(0);
27 }

```



ГЛАВА 9

Измерение времени исполнения программы

- Течение времени в компьютерной системе.
 - Измерение времени подсчетом количества интервалов.
 - Счетчики тактов.
 - Измерение времени исполнения программы с помощью счетчиков тактов.
 - Измерение по часам реального времени.
 - Протокол эксперимента.
 - Взгляд в будущее.
 - Реализация схемы измерения K-best.
 - Задания по пройденному материалу.
 - Резюме.
-

Вот один из наиболее часто задаваемых вопросов: как долго считается программа X на машине Y. Такой вопрос может возникнуть у программиста, желающего повысить эффективность программы, или у пользователя, принимающего решение купить машину. В предшествующих главах (см. главу 5), рассуждая об оптимизации производительности компьютерной системы, мы предполагали, что вопрос этот может быть разрешен с исчерпывающей точностью. Мы пытались ввести меру производительности в тактах на элемент (СРЕ) с точностью до двух десятичных знаков. Это требовало точности 0.1% для процедур, имеющих СРЕ порядка 10. В этой главе мы вернемся к данной проблеме и покажем, что эта отнюдь не простая задача. Можно было бы ожидать, что проведение предельно точных временных измерений на компьютерной системе будет довольно простым. В конце концов, для каждой конкретной комбинации программы и данных машина должна будет выполнить фиксированную последовательность команд. Исполнение каждой команды управляется внутренним прецизионным тактовым генератором процессора. Имеется немало факторов, влияющих на исполнение программы, которые могут изменяться от одного исполнения программы к другому. Компьютеры просто не исполняют одну программу за один прием. Они не-

прерывно переключаются с одного процесса на другой, исполняя некоторый программный код "по поручению" одного процесса, перед тем как перейти к следующему. Точное планирование ресурсов процессора для одной программы зависит от таких факторов, как количество пользователей, совместно находящихся в системе, сетевой трафик и согласование во времени дисковых операций. Модели реализации доступа к кэш-памяти зависят не только от текущих ссылок из программы, в которой мы производим измерения, но также и от других процессов, исполняющихся одновременно. Наконец, логика предсказаний условных переходов требует учитывать, будет ли условный переход зависеть от предыстории развития событий в программе. Эта предыстория развития может изменяться от одного исполнения программы к другому.

В этой главе мы описываем два основных механизма, используемых компьютерными системами для регистрации времени счета, один из них основан на использовании низкочастотного таймера, который периодически прерывает процессор, другой — на использовании счетчика, который увеличивает свое значение с каждым тактом задающего генератора. Прикладные программисты могут получить доступ к первому механизму измерения времени, вызывая библиотечные функции. В некоторых системах к таймерам можно обратиться с помощью библиотечных функций, но на других потребуется писать программы на языке ассемблера. Мы отложили обсуждение программы измерения времени до настоящего момента, потому что это требует понимания некоторых аспектов функционирования и аппаратных средств, и центрального процессора, а также того, каким образом операционная система управляет исполнением процесса.

Используя эти два механизма измерения времени, мы будем исследовать методы получения надежных измерений производительности программы. Мы увидим, что разброс измеренных временных величин, вызванный переключениями контекста, имеет тенденцию к увеличению и, следовательно, должен быть устранен. Отклонения, вызванные другими факторами, например, обращениями к кэш-памяти и прогнозированием условного перехода, управляемы, если оценивать поведение программы при тщательно оговоренных условиях. Вообще, мы можем получить точные измерения для временных величин, которые являются или очень короткими (менее 10 мс), или очень долгими (более 1 с), даже на предельно загруженных машинах. Интервалы от 10 мс до 1 с требуют, чтобы специальное внимание уделялось точности измерений.

В значительной степени вопросы измерения эффективности исполнения программ решаются самими разработчиками компьютерных систем. Различные группы и отдельные специалисты разрабатывают свои собственные методы таких измерений, но широко доступная литература по этому важному вопросу отсутствует. Отдельные компании и исследовательские группы, занимаясь измерениями эффективности исполнения программ с повышенной точностью, часто устанавливали специально конфигурированные машины, которые способны минимизировать любые возможные причины нарушения согласования во времени, например, ограничивая доступ или отключая большинство служб ОС и сетевых служб. Нам нужны методы, которые прикладные программисты могли бы использовать на обычных машинах, но у нас нет доступных инструментальных средств, пригодных для этих целей. Исходя из этого, мы разработаем свои собственные средства.

В данной главе мы будем последовательно рассматривать эти вопросы. Мы опишем схему проведения и оценки для нескольких экспериментов, которые обеспечат возможность получить точные измерения на некотором ограниченном множестве систем. Не всегда удается найти подробное описание экспериментальных исследований в книгах подобного уровня. Вообще, многие надеются прочитать заключительные рекомендации, а не описание того, на чем основываются эти рекомендации. В нашем случае, однако, мы лучше воздержимся от категорических рекомендаций по измерению времени исполнения произвольной программы в произвольной системе. Тем не менее мы надеемся, что вы сами будете экспериментировать в этой области и писать собственные программы для измерения эффективности исполнения. Мы надеемся, что наши учебные примеры помогут вам в этом деле. Мы подведем итог наших исследований в форме протокола, который поможет организовать ход проведения эксперимента.

9.1. Течение времени в компьютерной системе

Компьютеры функционируют в двух различных временных масштабах. На микроравнении компьютеры исполняют команды за один такт цикла генератора, а каждый такт цикла занимает всего лишь около одной наносекунды (нс). На макроуровне процессор должен реагировать на внешние события, которые делятся в течение времени, измеряемого в миллисекундах (мс). Например, в течение воспроизведения видеосигнала графический дисплей для большинства компьютеров должен обновляться каждые 33 мс. Диски обычно требуют приблизительно 10 мс для перемещения головок. Процессор непрерывно переключается между несколькими задачами на временном макроуровне, уделяя каждой задаче приблизительно 5–20 мс. В этих обстоятельствах пользователю кажется, что задачи выполняются одновременно, т. к. человек не может различить длительности времени короче, чем 100 мс. За это время процессор может исполнить миллионы команд.

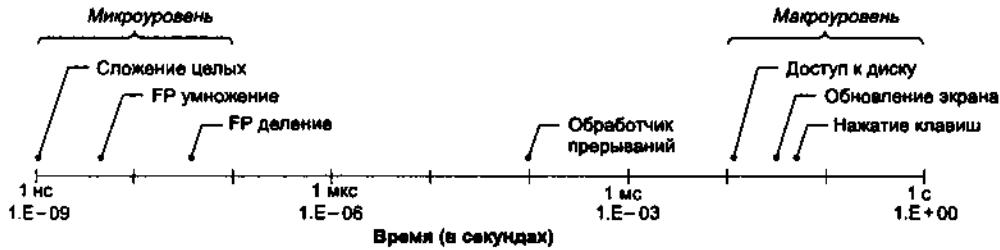


Рис. 9.1. Шкала времени для событий в компьютерной системе

На рис. 9.1 в логарифмическом масштабе представлены различные типы событий, где продолжительность события на микроуровне измеряется в наносекундах, а продолжительность события на макроуровне — в миллисекундах. События на микроуровне управляются подпрограммами операционной системы, которые занимают приблизи-

тельно 5—200 тыс. тактов цикла генератора. Эти интервалы времени измеряются в мс. Хотя может показаться, что это связано с гигантскими вычислениями, вся эта обработка происходит настолько быстрее, чем обработка события на макроуровне, что эти подпрограммы составляют лишь незначительную долю загрузки процессора.

УПРАЖНЕНИЕ 9.1

Когда пользователь редактирует файлы в редакторе реального времени типа EMACS, каждое нажатие клавиши генерирует сигнал прерывания. Операционная система должна запланировать обслуживание процесса редактирования для каждого нажатия клавиши. Предположим, что мы имеем систему с генератором, тактовая частота которого 1 ГГц, и пусть в системе имеется 100 пользователей, использующих EMACS и печатающих со скоростью 100 слов в минуту. Допустим, что среднее число символов в слове равно 6. Предположим также, что подпрограмма ОС обработки нажатий клавиш требует в среднем 100 тыс. тактов цикла генератора на одно нажатие клавиши. Какова доля от всей загрузки процессора, вызванная обработкой нажатия клавиш?

9.1.1. Планирование процессов и прерывания от таймера

Такие внешние события, как нажатия клавищ, дисковые операции и сетевая активность, генерируют прерывания для планировщика операционной системы, связанные с возможным переключением на другой процесс. Даже в отсутствие таких событий иногда необходимо переключать процессор с одного процесса на другой так, чтобы у пользователя создалось впечатление, как будто процессор выполняет несколько программ одновременно. Для этого компьютеры имеют внешний таймер, который периодически генерирует для процессора сигнал прерывания. Промежуток времени между этими сигналами прерывания называют *интервалом таймера* (interval time). Когда происходит прерывание от таймера, планировщик операционной системы может либо продолжить выполнение текущего процесса, либо переключить на другой процесс. Этот промежуток времени должен быть установлен достаточно коротким, чтобы гарантировать, что процессор будет переключаться между задачами достаточно часто, обеспечивая иллюзию одновременного исполнения нескольких задач. С другой стороны, переключение с одного процесса на другой требует тысяч тактов цикла генератора для сохранения состояния текущего процесса и установки состояния следующего процесса, и таким образом установка слишком короткого интервала вызвала бы значительное снижение производительности. Типичные интервалы таймера находятся в диапазоне между 1 и 10 мс, в зависимости от процессора и его конфигурации.

Соотношение временных уровней в компьютерах

Интересно сравнить компьютер VAX-11/780 корпорации DEC с современными процессорами. Эта машина была представлена в 1977 г. с начальной ценой приблизительно 200 тыс. долларов. Она стала первой широко используемой машиной, управляемой операционной системой Unix. Обратите внимание, что интервал таймера на этой машине, как обычно, составлял 10 мс, даже притом, что его цен-

тральный процессор был более чем в 1000 раз медленнее, чем процессор современной машины. На макроуровне шкала времени не изменилась, несмотря на то, что шкала времени на микроуровне стала намного быстрее.

На рис. 9.2 показана последовательность операций, выполняющихся на протяжении гипотетических 250 мс в системе с интервалом прерываний от таймера 10 мс. В течение этого периода существуют два активных процесса: А и В. Процессор поочередно исполняет часть процесса А, затем часть В и т. д. При исполнении этих процессов он действует либо в пользовательском режиме (user mode), исполняя команды прикладной программы, либо в привилегированном режиме (kernel mode), исполняя функции операционной системы по поручению программы, такие как обработка обращений к отсутствующей странице, ввод или вывод. Вспомните, что операции ядра операционной системы считаются частью каждого регулярного процесса, а не отдельного процесса. Планировщик операционной системы вызывается каждый раз, когда происходит внешнее событие или прерывание от таймера. Моменты прерываний от таймера на рисунке обозначены временными метками. Это означает, что фактически в каждой временной метке ядро проявляет некоторую активность, но для упрощения на рисунке мы этого не показываем.

Когда планировщик переключает обработку с процесса А на процесс В, он должен перейти в привилегированный режим, чтобы сохранить состояние процесса (по-прежнему оставаясь внутри процесса А) и затем восстановить состояние процесса В (оставаясь внутри процесса В). Таким образом, ядро проявляет активность в течение каждого перехода от одного процесса к другому. В другие моменты времени ядро может активизироваться без переключения процессов, например, когда обращение к отсутствующей странице может быть выполнено путем использования страницы, которая уже находится в памяти.

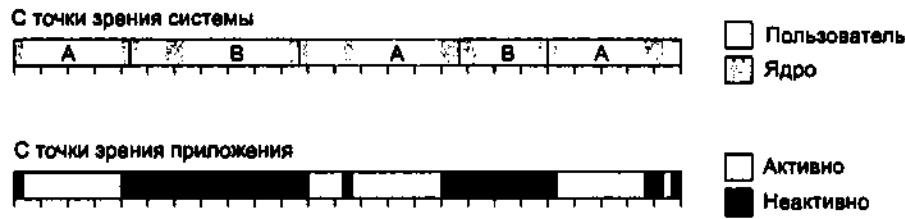


Рис. 9.2. Отличия в представлении времени системой и приложением

9.1.2. Течение времени

с точки зрения прикладной программы

С точки зрения прикладной программы, течение времени может рассматриваться как чередование периодов, когда программа активна и неактивна (ожидание, когда операционная система запланирует ее к исполнению). На рис. 9.2 показано, как программа А видела бы течение времени. Она активна в течение промежутков времени, отмеченных светлым цветом; в остальное время она неактивна.

Приложение производит полезные вычисления только тогда, когда его процесс исполняется в пользовательском режиме.

В качестве средства регистрации чередования между активными и неактивными периодами времени мы написали программу, которая непрерывно отслеживает и фиксирует длительность периодов пассивности. Затем она генерирует протокол *трассировки* (trace), показывающий чередование активности и пассивности. Подробности этой программы будут описаны далее в этой главе. Пример такой трассировки, сгенерированной при исполнении на Linux-машине с тактовой частотой приблизительно 550 МГц, показан в листинге 9.1. Каждый период помечен либо как активный (A), либо как неактивный (I). Для идентификации периоды пронумерованы от 0 до 9. Для каждого периода обозначено начальное время (относительно начала трассировки) и его продолжительность. Длительность промежутков времени выражена как в количестве тактов цикла генератора, так и в мс. Эта трассировка охватывает в общем 20 периодов времени (10 активных и 10 неактивных) при общей продолжительности 66.9 мс. В этом примере периоды пассивности довольно коротки, самый большой составляет 0.50 мс. Большинство этих периодов пассивности было вызвано прерываниями от таймера. Процесс был активен в течение приблизительно 95.1% общего времени наблюдения. Обратите внимание на то, что промежуток времени между периодами активности сохраняет постоянное значение. Эти границы вызваны прерываниями от таймера.

Листинг 9.1. Пример измерения трассировки

A0	время 0	(0.00 мс),	продолжительность	3726508	(6.776448 мс)
I0	время 3726508	(6.78 мс),	продолжительность	275025	(0.500118 мс)
A1	время 4001533	(7.28 мс),	продолжительность	0	(0.000000 мс)
I1	время 4001533	(7.28 мс),	продолжительность	7598	(0.013817 мс)
A2	время 4009131	(7.29 мс),	продолжительность	5189247	(9.436358 мс)
I2	время 9198378	(16.73 мс),	продолжительность	251609	(0.457537 мс)
A3	время 9449987	(17.18 мс),	продолжительность	2250102	(4.091686 мс)
I3	время 11700089	(21.28 мс),	продолжительность	14116	(0.025669 мс)
A4	время 11714205	(21.30 мс),	продолжительность	2955974	(5.375275 мс)
I4	время 14670179	(26.68 мс),	продолжительность	248500	(0.451883 мс)
A5	время 14918679	(27.13 мс),	продолжительность	5223342	(9.498358 мс)
I5	время 20142021	(36.63 мс),	продолжительность	247113	(0.449361 мс)
A6	время 20389134	(37.08 мс),	продолжительность	5224777	(9.500967 мс)
I6	время 25613911	(46.58 мс),	продолжительность	254340	(0.462503 мс)
A7	время 25868251	(47.04 мс),	продолжительность	3678102	(6.688425 мс)
I7	время 29546353	(53.73 мс),	продолжительность	8139	(0.014800 мс)
A8	время 29554492	(53.74 мс),	продолжительность	1531187	(2.784379 мс)
I8	время 31085679	(56.53 мс),	продолжительность	248360	(0.451629 мс)
A9	время 31334039	(56.98 мс),	продолжительность	5223581	(9.498792 мс)
I9	время 36557620	(66.48 мс),	продолжительность	247395	(0.449874 мс)

С точки зрения прикладной программы, функционирование процессора состоит из периодов, когда программа активно исполняется, и когда она неактивна. В данной

трассировке показан файл регистрации этих периодов для программы общей продолжительностью 66.9 мс. Программа была активна в течение 95.1% этого промежутка времени.

В листинге 9.2 показан фрагмент трассировки, когда имеется еще один активный процесс, использующий тот же процессор. Графическое представление этой трассировки показано на рис. 9.3. Обратите внимание, что шкалы времени не представляют собой непрерывного отрезка, поскольку фрагмент этой трассировки мы начинаем с момента времени 349.40 мс. Можно заметить, что в этом примере при обработке некоторых из прерываний от таймера ОС производит также переключение контекстов процессов. В результате каждый процесс активен в течение только приблизительно 50% времени.

Листинг 9.2. Другой пример трассировки

```
A48 время 191514104 (349.40 мс), продолжительность 5224961 (9.532449 мс)
I48 время 196739065 (358.93 мс), продолжительность 247557 (0.451644 мс)
A49 время 196986622 (359.38 мс), продолжительность 858571 (1.566382 мс)
I49 время 197845193 (360.95 мс), продолжительность 8297 (0.015137 мс)
A50 время 197853490 (360.97 мс), продолжительность 4357437 (7.949733 мс)
I50 время 202210927 (368.91 мс), продолжительность 5718758 (10.433335 мс)
A51 время 207929685 (379.35 мс), продолжительность 2047118 (3.734774 мс)
I51 время 209976803 (383.08 мс), продолжительность 7153 (0.013050 мс)
A52 время 209983956 (383.10 мс), продолжительность 3170650 (5.784552 мс)
I52 время 213154606 (388.88 мс), продолжительность 5726129 (10.446783 мс)
A53 время 218880735 (399.33 мс), продолжительность 5217543 (9.518916 мс)
I53 время 224098278 (408.85 мс), продолжительность 5718135 (10.432199 мс)
A54 время 229816413 (419.28 мс), продолжительность 2359281 (4.304286 мс)
I54 время 232175694 (423.58 мс), продолжительность 7096 (0.012946 мс)
A55 время 232182790 (423.60 мс), продолжительность 2859227 (5.216390 мс)
I55 время 235042017 (428.81 мс), продолжительность 5718793 (10.433399 мс)
```

На данной трассировке показан файл регистрации периодов для программы с общей продолжительностью 89.8 мс. Процесс был активен в течение 53.0% этого промежутка времени.

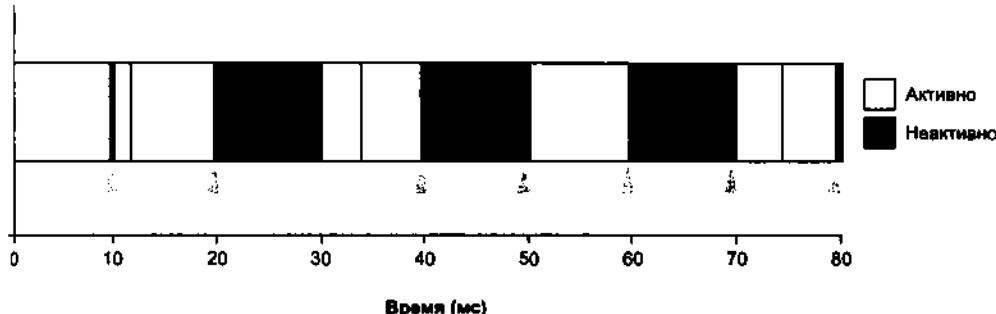


Рис. 9.3. Графическое представление периодов активности

УПРАЖНЕНИЕ 9.2

Это упражнение связано с интерпретацией участка трассировки в листинге 9.2.

1. В какие моменты времени в течение этого фрагмента трассировки происходили прерывания от таймера?
2. Какие из этих прерываний произошли в то время, когда трассируемый процесс был активен, и какие — в то время, когда он был неактивен?
3. Почему самые продолжительные периоды пассивности дольше, чем самые продолжительные периоды активности?
4. Исходя из модели активных и неактивных периодов, принятой в этой трассировке, какой процент времени, по вашему мнению, процесс трассировки будет неактивным, если рассмотреть более длинный отрезок времени?

9.2. Измерение времени подсчетом количества интервалов

Операционная система использует таймер также для того, чтобы регистрировать суммарное время, занимаемое каждым процессом. Такой подсчет дает несколько неточное измерение времени исполнения программы. На рис. 9.4 в графическом виде показано, как этот подсчет работает на примере системы, приведенной на рис. 9.2. Отрезок времени, в течение которого исполняется только один процесс, здесь мы будем называть *квантом времени* (*time segment*).

(а) Временные интервалы

A	B	F	A	C	B	A
Au Au Au As Bu Bs Bu Bu Bu As Au Au Au Au Au Bs Bu Bs Au Au Au As						

A 110u + 40s
B 70u + 30s

(б) Фактическое время

A	A	A		A		B		B
10	20	30	40	50	60	70	80	90

A 120.0u + 33.3s
B 73.3u + 23.3s

Рис. 9.4. Измерение времени процесса подсчетом интервалов:
а — измерение времени подсчетом интервалов; б — фактическое время

9.2.1. Роль операционной системы

Когда происходит прерывание от таймера, операционная система определяет, какой процесс был активен, и увеличивает счетчик этого процесса на один интервал таймера. Она будет увеличивать системное время, если система в привилегированном режиме, и пользовательское время — в противном случае. На рис. 9.4, а показан такой подсчет для двух процессов. Временные метки показывают моменты наступления прерываний от таймера. В каждом отмечен счетчик, который получает приращение: Au или As — для пользовательского процесса A или системное время; Bs или Bs —

для пользовательского процесса В или системное время. Каждая временная отметка имеет надпись в соответствии с ее активностью. Результат подсчета показывает, что процесс А использовал суммарно 150 мс: 110 из них — это пользовательское время, а 40 — системное время. Он показывает также, что В использовал в общем 100 мс: 70 из них — это пользовательское время, а 30 — системное время.

9.2.2. Считывание данных от таймеров

При исполнении команды из оболочки Unix пользователь может добавить в качестве префикса команды слово `time`, чтобы измерить время исполнения команды. Например, чтобы определить время исполнения программы `prog` с помощью параметра командной строки `-n 17`, пользователь может просто ввести команду

```
unix> time prog -n 17
```

После завершения исполнения программы оболочка выведет строку итоговой статистики времени исполнения, например, в таком виде:

```
2.230u 0.260s 0:06.52 38.1% 0+0k 0+0io 80pf+0w
```

Первые три числа в этой строке — это промежутки времени. Первые два показывают количество секунд пользовательского и системного времени. Обратите внимание на то, что в обоих числах в третьем десятичном разряде стоит 0. При интервале таймера 10 мс все измерения времени умножаются на сто секунд. Третье число — общее затраченное время, данное в минутах и секундах. Заметьте, что системное и пользовательское время в сумме составляет 2.49 с: меньше, чем половина прошедшего времени (6.52 с), а это указывает на то, что процессор исполнял одновременно и другие процессы. Процент показывает, какую долю общего прошедшего времени пользователь и система использовали совместно, например $(2.23 + 0.26)/6.52 = 0.381$. Остальные статистические данные характеризуют поведение ввода-вывода и подкачки страниц.

Программисты могут также прочитать данные таймеров процесса, вызывая библиотечную функцию `times`, объявленную следующим образом:

```
#include <sys/types.h>
struct tms {
    clock_t tms_utime; /* пользовательское время */
    clock_t tms_stime; /* системное время */
    clock_t tms_cutime; /* пользовательское время снятого дочернего процесса */
    clock_t tms_cstime; /* системное время снятого дочернего процесса */
};
clock_t times(struct tms *buf);
```

Для измерения времени используются единицы измерения, которые называются тактами системных часов (`clock ticks`). Константа `CLK_TCK` определяет количество тактов системных часов в секунду. Тип данных переменной `clock_t` обычно определяется как `long integer`. Поля для времен дочерних процессов дают суммарное время тех дочерних процессов, которые закончились и были сняты. Таким образом, `times` не может использоваться для отслеживания времени, использованного любыми дейст-

вующими (не снятыми) дочерними процессами. Значение времени, возвращаемое `times`, измеряется общим количеством тактов системных часов, которые были отсчитаны с момента запуска системы. Поэтому можно вычислить полное время (в тактах системных часов), прошедшее между двумя точками в исполняемой программе, делая два запроса к `times` и вычисляя разность возвращаемых значений.

Стандарт ANSI C определяет также функцию `clock`, которая измеряет полное время, использованное текущим процессом:

```
#include <time.h>
clock_t clock(void);
```

Хотя возвращаемое значение в объявлении имеет тот же самый тип `clock_t`, используемый функцией `times`, эти две функции не выражают время в одних и тех же единицах измерения. Чтобы перевести в секунды время, о котором сообщает `clock`, его следует разделить на определенную константу `CLOCKS_PER_SEC`. Это значение не обязательно должно быть тем же самым, что и у константы `CLK_TCK`.

9.2.3. Точность таймеров процесса

В качестве примера можно сослаться на рис. 9.4, где показано, что этот механизм измерения времени довольно приблизителен. На рис. 9.4, б показано фактическое время, использованное этими двумя процессами. Процесс А исполнялся всего 153.3 мс, из которых 120.0 в пользовательском режиме и 33.3 — в привилегированном режиме. Процесс В исполнялся всего 96.7 мс, из которых 73.3 в пользовательском режиме и 23.3 — в привилегированном режиме. Схема подсчета интервалов не делает никакой попытки определить время с точностью большей, чем интервал таймера.

УПРАЖНЕНИЕ 9.3

Что должна сообщить операционная система, если пользовательское и системное время для последовательности исполнения будет таким, как показано на рис. 9.5? Примите интервал таймера равным 10 мс.



Рис. 9.5. Последовательность исполнения

УПРАЖНЕНИЕ 9.4

На системе с интервалом таймера 10 мс некоторый отрезок процесса зарегистрирован, как требующий 70 мс суммарно системного и пользовательского времени. Какими могут быть минимальное и максимальное фактическое время, использованное на этом отрезке?

УПРАЖНЕНИЕ 9.5

Что должны зарегистрировать счетчики системного и пользовательского времени для трассировки в листинге 9.1? Насколько это соответствует фактическому времени, в течение которого каждый процесс был активен?

Для программ, которые работают достаточно долго (не менее нескольких секунд), погрешности в этой схеме имеют тенденцию компенсировать друг друга. Усредненная по нескольким отрезкам ожидаемая ошибка приближается к нулю. С теоретической точки зрения, однако, нет никакого обоснования того, насколько эти измерения могут отличаться от истинных значений.

Чтобы проверить точность этого метода измерения времени, мы провели ряд экспериментов по сравнению промежутка времени T_m , измеренного операционной системой для типового вычисления, с нашей оценкой того, каким будет этот промежуток времени T_c , если системные ресурсы будут выделены исключительно выполнению этого вычисления. Вообще T_c будет отличаться от T_m по некоторым причинам:

1. Собственные погрешности, свойственные схеме подсчета интервала, могут привести к тому, что T_m будет либо меньше, либо больше, чем T_c .
2. Активность ядра, вызванная прерываниями от таймера, потребляет 4—5% от общего количества тактов центрального процессора, но эти такты не учитываются должным образом. Как можно заметить из трассировки, эта активность заканчивается перед следующим прерыванием от таймера и, следовательно, не поддается явному учету. Тем не менее при этом уменьшается количество тактов, доступных для процесса, исполняющегося в течение следующего интервала времени. Это вызовет тенденцию возрастания T_m относительно T_c .
3. Когда процессор переключается с одной задачи на другую, кэш в течение переходного периода перестает выполнять свои функции. Таким образом, процессор не работает так же эффективно при переключении между нашей программой и другими, как это было бы, если бы наша программа исполнялась непрерывно. Этот фактор вызовет тенденцию возрастания T_m относительно T_c .

Вопрос о том, как можно определить значение T_c для нашего примера вычисления, мы обсудим позже в этой главе.

На рис. 9.6 показаны результаты этого эксперимента, выполнявшегося при двух различных режимах загрузки. На графиках показаны измерения нормированной ошибки, определяемой отношением $(T_m - T_c)/T_c$ как функция от T_c . Эта ошибка имеет отрицательное значение, если T_m меньше, чем T_c , и положительное, если T_m больше, чем T_c . Эти две последовательности показывают результаты измерений, полученные при двух различных режимах загрузки. Последовательность, отмеченная как "загрузка 1", соответствует случаю, когда процесс, выполняющий типовое вычисление — единственный активный процесс. Последовательность, отмеченная как "загрузка 11", соответствует случаю, когда еще 10 других процессов стремятся выполнить аналогичные вычисления. Вторая последовательность соответствует режиму повышенной загрузки: система заметно замедляет реакцию на нажатия клавиш и другие сервисные запросы. Обратите внимание на широкий разброс значений ошибки, заметный на этом графике. Вообще могут считаться приемлемыми только те измерения, которые попадают в интервал 10% от истинного значения, следовательно, можно рассматривать только ошибки в пределах от -0.1 до +0.1.

Ниже приблизительно 100 мс (10 интервалов таймера) измерения не точны из-за грубости метода измерения времени. Метод подсчета интервалов пригоден только для

того, чтобы делать измерения относительно продолжительных вычислений: 100 млн тактов цикла генератора или больше. Помимо того, мы видим, что эта ошибка вообще находится в интервале между 0.0 и 0.1, т. е. в пределах 10%. Нет никакого значимого отличия между этими двумя разными режимами загрузки. Обратите внимание также, что ошибки имеют положительное смещение: средняя ошибка для всех измерений при $T_m \geq 100$ мс равна приблизительно 1.04, вследствие того, что прерывания от таймера потребляют приблизительно 4% процессорного времени.

Intel Pentium III, Linux, таймер процессора

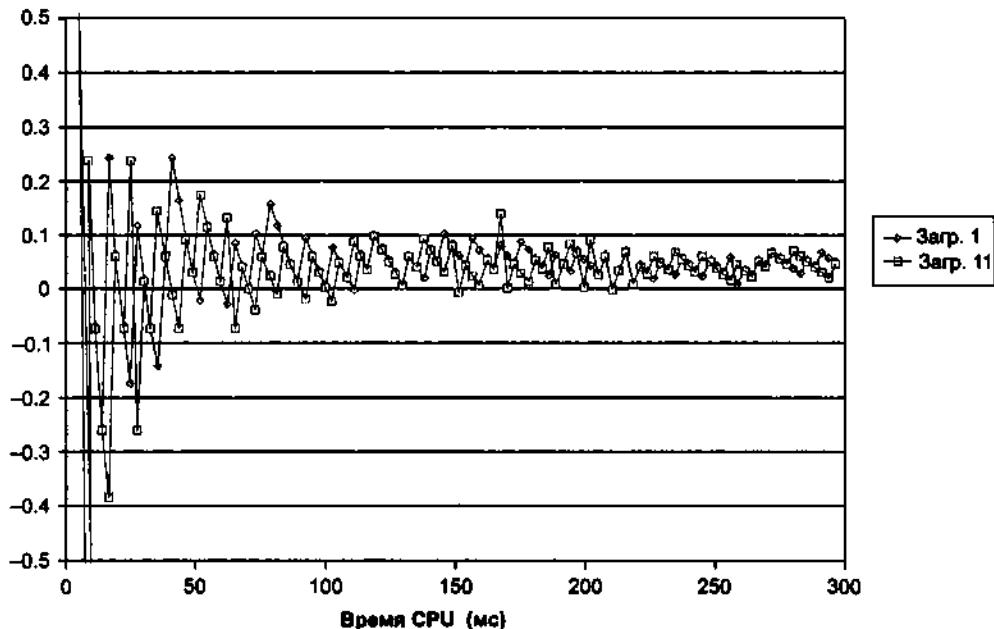


Рис. 9.6. Вычисление точности измерения интервала

Этот эксперимент показывает, что таймеры процесса пригодны только для того, чтобы получить приблизительные оценки эффективности исполнения программы. Они являются слишком грубыми, чтобы использовать для измерения, имеющего продолжительность меньше, чем 100 мс. На этой машине они имеют систематическое смещение приблизительно на 4%. Основное достоинство этого механизма измерения времени состоит в том, что его точность не сильно зависит от системной загрузки.

9.3. Счетчики тактов

Чтобы обеспечить большую точность измерения временных интервалов, многие процессоры также содержат таймер, который работает на уровне тактов цикла генератора. Этот таймер представляет собой специальный регистр, который получает прира-

щение с каждым тактом цикла. При этом могут использоваться специальные машинные команды, считывающие значение этого счетчика. Не все процессоры имеют такой счетчик, а те, которые его имеют, значительно различаются в деталях его реализации. В результате, не существует никакого стандартного, независимого от платформы интерфейса, с помощью которого программисты могут использовать эти счетчики. С другой стороны, с помощью совсем небольшого по объему ассемблерного кода, как правило, не трудно создать интерфейс программы для любой заданной машины.

9.3.1. Счетчики тактов в IA32

Все измерения времени, о которых мы говорили до сих пор, были проведены с использованием счетчика тактов цикла генератора в IA32. В архитектуре IA32 счетчики тактов были введены вместе с микроархитектурой P6 (Pentium Pro и последующие процессоры). Счетчик тактов — это 64-битовое, число без знака. Для процессора, работающего с тактовым генератором частотой 1 ГГц, этот счетчик совершил полный оборот (т. е. пересчитает все числа от $2^{64} - 1$ до 0) за время 1.8×10^{10} секунд, что составляет 570 лет. С другой стороны, если посчитать только биты 32 младших разрядов этого счетчика (как целое число без знака), то полный оборот потребует всего 4.3 секунды. Можно поэтому понять, почему разработчики IA32 решили реализовать счетчик на 64 бита.

К счетчику IA32 обращаются с помощью команды `rdtsc`. Эта команда не требует никаких параметров. Она устанавливает в регистре `%edx` значение старших 32 битов счетчика, и в регистре `%eax` — младшие 32 бита. Чтобы обеспечить интерфейс с С-программами, желательно заключить эту команду в тело процедуры:

```
void access_counter(unsigned *hi, unsigned *lo);
```

Эта процедура должна установить в области памяти `hi` значение старших 32 битов счетчика и в `lo` — младших 32 битов. Реализация доступа к счетчику представляет собой простое упражнение на использование встроенного в компилятор GCC средства ассемблирования, описанного в разд. 3.15, программный код показан в листинге 9.3.

Листинг 9.3. Реализация программного интерфейса к счетчику тактов в IA32

```
1 /* инициализирует счетчик тактов */
2 static unsigned cyc_hi = 0;
3 static unsigned cyc_lo = 0;
4
5
6 /* устанавливаем в *hi и *lo старшие и младшие биты счетчика тактов;
7 реализация требует, чтобы ассемблерный код использовал команду rdtsc */
8 void access_counter(unsigned *hi, unsigned *lo)
9 {
10     asm("rdtsc; movl %edx,%0; movl %eax,%1" /* счетчик тактов */
```

```

11      : "=r" (*hi), "=r" (*lo)          /* и передаем результат в */
12      : /* вывода нет */              /* эти две входные переменные */
13      : "%edx", "%eax");
14 }
15
16 /* регистрирует текущее значение счетчика тактов */
17 void start_counter()
18 {
19     access_counter(&cyc_hi, &cyc_lo);
20 }
21
22 /* возвращает количество тактов, отсчитанных, начиная с последнего вызова
   start_counter */
23 double get_counter()
24 {
25     unsigned ncyc_hi, ncyc_lo;
26     unsigned hi, lo, borrow;
27     double result;
28
29     /* получаем значение счетчика тактов */
30     access_counter(&ncyc_hi, &ncyc_lo);
31
32     /* выполняем вычитание с удвоенной точностью */
33     lo = ncyc_lo - cyc_lo;
34     borrow = lo > ncyc_lo;
35     hi = ncyc_hi - cyc_hi - borrow;
36     result = (double) hi * (1 << 30) * 4 + lo;
37     if (result < 0) {
38         fprintf(stderr, "Error: counter returns neg value: %.0f\n", result);
39     }
40     return result;
41 }

```

Имея эту подпрограмму, мы можем теперь реализовать несколько функций для измерения общего количества тактов, прошедших между любыми двумя временными отметками:

```
#include "clock.h"
void start_counter();
double get_counter();
```

Чтобы избежать возможных осложнений, связанных с переполнением при использовании просто 32-разрядного целого числа, значение времени, возвращаемое этой функцией, имеет тип `double`. Программный код этих двух подпрограмм показан в листинге 9.3. Особенности беззнаковой арифметики вынуждают выполнять операции вычитания с двойной точностью и преобразовывать результат к `double`.

9.4. Измерение времени исполнения программы с помощью счетчиков тактов

Счетчики тактов генератора предоставляют очень точное средство для измерения времени, прошедшего между двумя точками в исполняемой программе. Как правило, мы интересуемся измерением времени, необходимого для выполнения некоторой заданной части программного кода. Наши подпрограммы счетчика тактов вычисляют общее количество тактов, прошедших между обращением к `start_counter` и обращением к `get_counter`. Они не следят за тем, какие именно процессы используют эти такты, а также, работает ли процессор в режиме ядра или в пользовательском режиме. Следует быть внимательным при использовании такого средства измерения при определении времени исполнения. Мы рассмотрим некоторые из возможных трудностей, а также способы их преодоления.

В качестве примера программного кода, использующего счетчик тактов задающего генератора, может служить программа, показанная в листинге 9.4, которая дает возможность определить тактовую частоту процессора. Испытание этой функции на нескольких системах с параметром `sleeptime`, равным 1, показывает, что она выдает значение тактовой частоты с ошибкой в пределах 1.0% для данного процессора. Этот пример ясно показывает, что данная подпрограмма измеряет затраченное время, а не время, использованное отдельным процессом. Когда эта программа вызывает `sleep`, операционная система не будет возобновлять исполнение процесса, пока не истечет время "сна", равное одной секунде. Такты, которые будут отсчитаны в течение этого времени, будут потрачены на исполнение других процессов.

Листинг 9.4. Определение тактовой частоты процессора

```
1 /* вычисляет тактовую частоту путем измерения количества отсчитанных тактов */
2 /* при засыпании на sleeptime секунд */
3 double mhz(int verbose, int sleeptime)
4 {
5     double rate;
6
7     start_counter();
8     sleep(sleeptime);
9     rate = get_counter() / (le6*sleeptime);
10    if (verbose)
11        printf("Processor clock rate ^= %.1f MHz\n", rate);
12    return rate;
13 }
```

9.4.1. Эффект переключения контекста

Простейший способ измерять время исполнения некоторой процедуры Р состоит в том, чтобы просто использовать счетчик тактов за время однократного исполнения Р, как в следующем примере:

```

1 double time_P()
2 {
3     start_counter();
4     P();
5     return get_counter();
6 }

```

Если между двумя обращениями к подпрограмме счетчика исполняется также другой процесс, то легко можно получить вводящие в заблуждение результаты. Дополнительные трудности возникают, если или машина предельно загружена, или время исполнения для P особенно продолжительно. Этот эффект иллюстрируется на рис. 9.7. На этом рисунке показаны результаты повторяющегося измерения в программе, которая вычисляет сумму массива 131 072 целых чисел. Обратите внимание, что все значения больше, чем интервал таймера. Были проведены два испытания, и в каждом измерении 18 раз исполнялась одна и та же процедура. Последовательность, отмеченная как "загрузка 1", показывает время исполнения на слабо загруженной машине, где обрабатывается единственный активный процесс. Все измерения укладываются в пределы 3.4% минимального времени исполнения. Последовательность, отмеченная как "загрузка 4", показывает время исполнения, когда исполняются еще три других процесса, использующих сильно загруженный центральный процессор и общую систему памяти. Первые семь из этих выборок имеют время в пределах 2% от самой быстрой выборки из загрузки 1, но разброс других в 4.3 раза больше.

Примеры измерений: большой массив

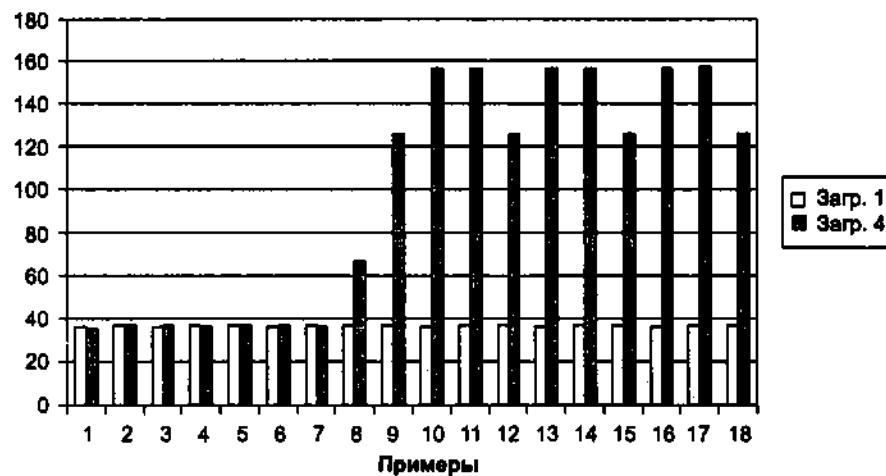


Рис. 9.7. Измерения для продолжительной процедуры при различных режимах загрузки

Как показывает этот пример, переключение контекста вызывает критические отключения во времени исполнения. Если процесс выгружается на диск, это вызовет задержку полного интервала времени на миллионы команд. Ясно, что любая схема,

которую мы изобретаем для измерения времени исполнения программы, должна не допускать появления таких больших отклонений.

9.4.2. Кэширование и другие эффекты

Эффекты кэширования и прогнозирование условного перехода вызывают меньшие отклонения при измерении времени, чем при переключении контекста. В качестве примера на рис. 9.8 показана последовательность измерений, подобная той, которую мы видели на рис. 9.7, отличие лишь в том, что массив здесь в 4 раза меньше, и как следствие, время исполнения приблизительно равно 8 мс. Это время исполнения короче, чем интервал таймера, и поэтому менее вероятно, что во время исполнения произойдет переключение контекста. Можно оценить величину отклонения при измерении: наименьшее в 1.1 раза меньше, чем самое большое, но ни одно из этих отклонений не достигает такой же величины, как если бы оно было вызвано переключением контекста.

Примеры измерений: небольшой массив

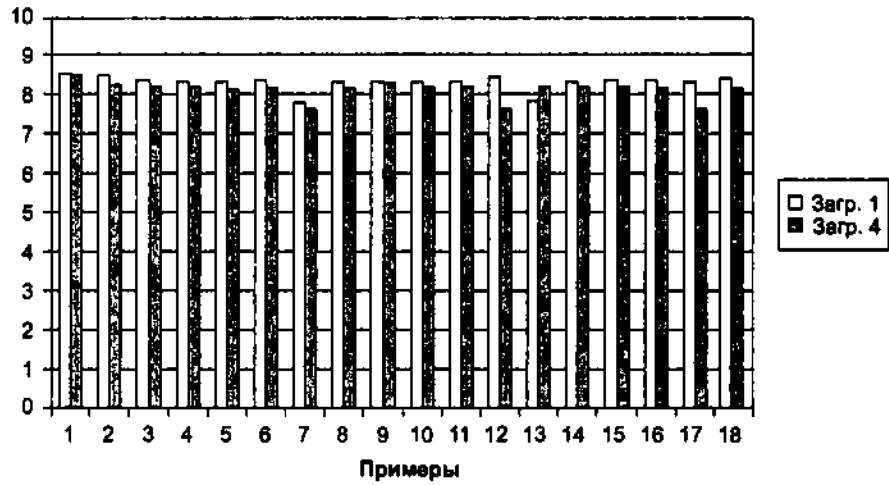


Рис. 9.8. Измерения для процедуры небольшой продолжительности при различных режимах загрузки

Отклонения, показанные на рис. 9.8, вызваны, главным образом, эффектом кэширования. Время исполнения блока программного кода может очень сильно зависеть от того, действительно ли данные и команды, используемые этим программным кодом, присутствуют в начале исполнения в кэше данных и команд.

В качестве примера мы написали две одинаковые процедуры, procA и procB, и установили значение 0.0 в восьми последовательных элементах, адресуемых указателем. Мы измеряли количество тактовых импульсов для различных обращений к этим процедурам с тремя различными указателями: b1, b2 и b3. Последовательность вызовов и результаты измерений показаны в табл. 9.1. Разброс измеренных значений времени

лежит в пределах коэффициента 4, несмотря даже на то, что при обращениях выполняются одинаковые вычисления. Поскольку в этом программном коде не было никаких условных переходов, мы можем заключить, что отклонения должны происходить из-за эффекта кэширования.

Таблица 9.1. Последовательность измерений

Измерение	Вызов	Тактов
1	procA(b1)	399
2	procA(b2)	132
3	procA(b3)	134
4	procA(b1)	100
5	procB(b1)	317
6	procB(b2)	100

УПРАЖНЕНИЕ 9.6

Обозначим через c количество тактов, которые потребовались бы для вызова procA или procB, если бы не было неудачных обращений к кэшу. Для каждого вычисления такты, израсходованные впустую из-за неудачных обращений к кэшу, могут быть отнесены и распределены между следующими действиями:

- команды, реализующие программный код измерения (например, start_counter, get_counter и т. д.). Обозначим через m количество необходимых для этого тактов;
- команды реализации процедуры измерения (procA или procB). Обозначим через p количество необходимых для этого тактов;
- адреса области памяти для данных, подлежащих обновлению. Обозначим через d количество необходимых для этого тактов.

Учитывая результаты измерений, показанные в табл. 9.1, дайте оценки значений c , m , p и d .

Рассматривая отклонения, полученные в этих измерениях, естественно задать вопрос: какое из них является правильным? К сожалению, простого ответа на этот вопрос нет. Все зависит как от условий, при которых будет фактически функционировать наш программный код, так и от условий, при которых мы можем получить надежные измерения. Одна из сложностей состоит в том, что измерения даже не повторяются от одного запуска на счет к следующему. Табл. 9.1 показывает данные только для одного выполненного испытания. В повторных испытаниях мы наблюдали для измерения 1 разброс в диапазоне от 317 до 606, а для измерения 5 — разброс от 301 до 326. С другой стороны, остальные четыре измерения изменяются не больше, чем на несколько тактов от одного запуска к другому.

Ясно, что измерение 1 дает завышенную оценку, потому что оно включает затраты на загрузку в кэш программного кода измерения и структур данных. Кроме того, ему

наиболее присущи значительные вариации. Измерение 5 включает затраты на загрузку procB в кэш. Ему также присущи существенные отклонения. В реальных приложениях один и тот же программный код исполняется многократно. В результате время для загрузки программного кода команды в кэш будет относительно незначительным. Наши измерения в этом примере носят несколько искусственный характер, эффект неудачного обращения в кэш команд был пропорционально больше, чем то, что будет происходить в реальном приложении.

Чтобы измерить время, необходимое процедуре P, где эффект неудачного обращения к кэшу команд минимизирован, мы можем вызвать следующую программу (листинг 9.5).

Листинг 9.5. Измерение

```
1 double time_P_warm()
2 {
3     P(); /* прогреваем кэш*/
4     start_counter();
5     P();
6     return get_counter();
7 }
```

После первого вызова P перед стартом измерения в кэш команд будет помещен программный код, используемый P.

Этот программный код также минимизирует эффект неудачного обращения к кэшу данных, т. к. в результате первого вызова P в кэш данных будут также перенесены данные, к которым обращается P. Для процедур procA или procB измерение с помощью time_P_warm заняло бы 100 тактов. Такую методику можно считать оправданной, если мы ожидаем, что наш программный код будет многократно обращаться к одним и тем же данным. Для некоторых приложений, однако, мы, скорее всего, при каждом новом исполнении будем обращаться к новым данным. Например, процедура, которая копирует данные из одной области памяти в другую, вероятнее всего, будет вызвана в ситуации, где никакой блок не кэширован. Процедура time_P_warm имела бы тенденцию приуменьшать время исполнения для такой процедуры. Для procA или procB это заняло бы 100, а не те 132—134 измеренных тактов.

Чтобы добиться того, чтобы программный код для измерения времени измерял эффективность процедуры, где никакие данные заранее не кэшируются, мы можем перед выполнением фактического измерения очистить кэш от всех полезных данных. Следующая процедура делает это для системы с размером кэша не больше, чем 512 Кбайт (листинг 9.6).

Листинг 9.6. Очистка кэша

```
1 /* количество байтов в наибольшем кэше, подлежащем очистке */
2 #define CBYTES (1<<19)
3 #define CINTS (CBYTES/sizeof(int))
4
```

```

5 /* большой массив для переноса в кэш */
6 static int dummy[CINTS];
7 volatile int sink;
8
9 /* удаляет существующие блоки из кэшей данных */
10 void clear_cache()
11 {
12     int i;
13     int sum = 0;
14
15     for (i = 0; i < CINTS; i++)
16         dummy[i] = 3;
17     for (i = 0; i < CINTS; i++)
18         sum += dummy[i];
19     sink = sum;
20 }

```

Эта процедура просто выполняет вычисление над очень большим искусственным (*dummy*) массивом, фактически удаляя все, что есть в кэше. Программный код имеет несколько характерных особенностей, назначение которых — избежать обычных ловушек. Он и сохраняет значения в *dummy*, и считывает их так, чтобы они кэшировались независимо от политики управления кэшем. Эта процедура выполняет вычисление, используя значения массива, и сохраняет результат в глобальном целом *volatile* так, чтобы высококлассный оптимизирующий транслятор не дошел до оптимизации этой части программного кода.

При наличии этой процедуры мы можем получить измерение для P в режиме, когда команды P кэшируются, но данные — нет, путем использования следующей процедуры (листинг 9.7).

Листинг 9.7. Метод кэширования

```

1 double time_P_cold()
2 {
3     P();           /* прогреваем кэш команд */
4     clear_cache(); /* очищаем кэш данных */
5     start_counter();
6     P();
7     return get_counter();
8 }

```

Конечно, даже этот метод имеет недостатки. На машине с объединенным кэшем второго уровня (L2) процедура *clear_cache* приведет к тому, что все команды P будут удалены. Хорошо хотя бы то, что команды останутся в кэше команд уровня L1. Процедура *clear_cache* также удаляет из кэша большую часть стека времени исполнения,

что приводит к завышенной оценке времени, необходимого Р при более реалистических условиях.

Как показывает анализ этих вопросов, эффект кэширования является причиной отдельных трудностей при измерении эффективности исполнения. Программисты ограничены в своих возможностях решать, какие команды и данные следует загружать в кэш, и что из него следует удалять, когда возникает необходимость загружать новые значения. В лучшем случае мы можем установить режим измерения, который в какой-то мере соответствует ожидаемому режиму нашего приложения в отношении комбинаций сбрасывания и загрузки кэша.

Как упоминалось ранее, логика прогнозирования условного перехода также влияет на эффективность исполнения программы, т. к. временные издержки, вызванные командами перехода, намного меньше, если направление перехода и адресат предсказаны правильно. Эта логика делает свои предсказания, руководствуясь предысторией команд перехода, исполненных ранее. Когда система переключается от одного процесса к другому, она заранее делает предсказание о возможном переходе в новом процессе исходя из того, что выполнялось в предыдущем процессе. Практически, однако, эти эффекты оказывают лишь незначительное влияние на эффективность исполнения. Больше всего предсказания зависят от недавних условных переходов, следовательно, влияние одних процессов на другие незначительно.

9.4.3. Схема измерения K-best

Хотя наши измерения, использующие такты таймера, не защищены от ошибок из-за переключения контекста, операций кэширования и прогнозирования условного перехода, одна важная их особенность состоит в том, что эти ошибки будут всегда приводить к завышению оценки истинного времени выполнения. Никакие операции, выполненные процессором, не могут искусственно ускорить программу. Мы можем воспользоваться этим свойством, чтобы получить надежные измерения времени исполнения, даже когда есть отклонение из-за переключения контекста и других эффектов.

Предположим, что мы многократно вызываем некоторую процедуру и измеряем количество тактов, используя либо процедуру `time_P_warm` (время разогретого Р), либо процедуру `time_P_cold` (время непротретого Р). Мы регистрируем K (например, три) самых быстрых времени. Если мы находим, что эти измерения укладываются в пределы некоторого небольшого допуска ϵ (например, 0.1%), то кажется разумным, что наиболее быстрый из них представляет истинное время исполнения процедуры. В качестве примера предположим, что для исполнения программы, показанной на рис. 9.7, мы устанавливаем допуск в 1.0%. Тогда самые быстрые шесть измерений для загрузки 1 находятся в пределах этого допуска, как и самые быстрые три — для загрузки 4. Поэтому можно сделать вывод, что для них времена исполнения — 35.98 мс и 35.89 мс соответственно. Для случая загрузки 4 мы также видим, что измерения сгруппированы приблизительно в районе значения 125.3 мс, и шесть — приблизительно в районе значения 155.8 мс, но мы можем смело отказаться от них, т. к. их оценка дана с превышением.

Мы будем называть этот подход к измерению схемой K-best (K лучших). Она требует установки трех параметров:

- K — количество измерений, которое должно попасть в пределы некоторой области окружения самых быстрых;
- ϵ — как близко должны располагаться измерения. То есть, если измерения в порядке возрастания отметить как $v_1, v_2, \dots, v_n, \dots$, то требуется, чтобы выполнялось $(1 + \epsilon) v_1 \geq v_K$;
- M — максимальное количество измерений в эксперименте.

В нашей реализации выполняется последовательность испытаний и корректировка сортировки в массиве K самых быстрых значений. При каждом новом измерении проверяется, быстрее ли его результат, чем текущий в позиции массива. Если это так, то заменяется элемент массива K и затем выполняется последовательность перестановок смежных позиций массива. Этот процесс продолжается до тех пор, пока либо критерий ошибки будет удовлетворен, и это будет свидетельствовать о том, что процесс измерения "сошелся", либо будет превышен предел M , и мы считаем, что измерения не сходятся.

Экспериментальная оценка

Мы провели ряд экспериментов с целью проверки точности схемы измерения K-best. Среди вопросов, на которые мы хотели получить ответ, были следующие:

1. Какую точность обеспечивает эта схема измерений?
2. При каких условиях и насколько быстро сходятся измерения?
3. Может ли данная схема определять точность ее собственных измерений?

Одна из сложностей в планировании такого эксперимента — это необходимость знать фактическое время исполнения тех программ, которые мы подвергаем тестированию. Лишь только после этого мы можем определять точность наших измерений. Мы знаем, что счетчик тактов дает точные результаты до тех пор, пока вычисление не оказывается прерванным. Вероятность прерывания невелика для вычислений, которые намного короче, чем интервал таймера, и при исполнении на слабо загруженной машине. Мы воспользуемся этим свойством, чтобы получить надежные оценки истинного времени исполнения.

В качестве объекта тестирования мы использовали процедуру, которая многократно записывает значения в массив из 2048 целых чисел и затем читает их назад, подобно программному коду для очистки кэша. Установливая количество r повторений, мы смогли создать выкладки для требуемых интервалов времени. Сначала мы определили ожидаемое время исполнения процедуры, как функцию от r , обозначив ее через $T(r)$ при изменении r в пределах от 1 до 10 (что соответствует изменению времени в пределах от 0.09 до 0.9 мс), и рассчитали отклонение методом наименьших квадратов по формуле, имеющей вид $T(r) = mr + b$. При небольших значениях r , выполнив по 100 измерений для каждого значения r на слабо загруженной системе, мы получили очень точную зависимость $T(r)$. Анализ методом наименьших квадратов показал, что формула $T(r) = 49273.4r + 166$ (в единицах измерения количества тактовых импуль-

сов) соответствует этим данным с максимальной ошибкой не более, чем 0.04%. Это дало нам уверенность в возможности точного предсказания фактического времени вычисления для процедуры в виде функции от r .

Мы провели измерения эффективности исполнения, используя схему K-best с параметрами $K = 3$, $\epsilon = 0.001$ и $M = 30$. Измерения были проведены для множества значений r , чтобы получить ожидаемое время исполнения в интервале от 0.27 до 50 мс. Для каждого из результатов измерения $M(r)$ мы вычисляли ошибку измерения $E_m(r)$ по формуле

$$E_m(r) = (M(r) - T(r))/T(r).$$

На рис. 9.9 показаны результаты экспериментальной проверки правильности схемы K-best на Intel Pentium III под управлением Linux. На этом рисунке мы показали ошибку измерения $E_m(r)$ как функцию от $T(r)$ в мс. Обратите внимание на то, что $E_m(r)$ представлено в логарифмическом масштабе: каждая горизонтальная линия представляет разность порядков величин ошибок измерения. Чтобы достичь точности в пределах 1%, ошибка не должна превышать 0.01. На рисунке не показаны ошибки меньше 0.001 (т. е. 0.1%), т. к. постановка нашего эксперимента не обеспечивает такой высокой точности.

Intel Pentium III, Linux

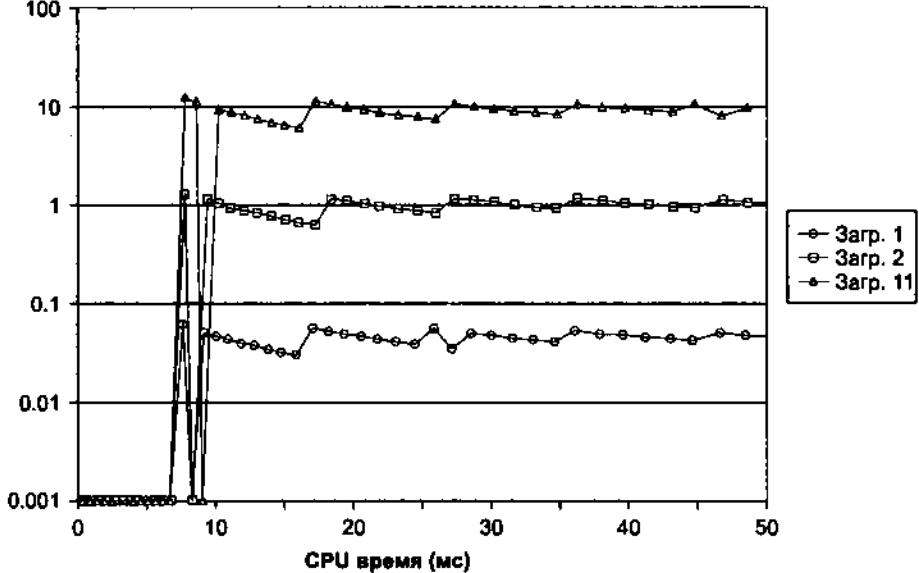


Рис. 9.9. Экспериментальная проверка правильности схемы измерения K-best на Linux-системе

Эти три последовательности представляют ошибки при трех различных режимах загрузки. Таким образом, наша схема может использоваться для измерения относительно коротких времен исполнения даже на предельно загруженной машине. Послед-

довательность "загрузка 1" соответствует случаю, когда имеется только один активный процесс. Для времени исполнения больше, чем 10 мс, измерения T_m систематически завышают время вычисления T_c приблизительно на 4–6%. Эти завышения оценок происходят из-за затрат времени на обработку прерываний от таймера. Они согласуются с трассировкой (см. листинг 9.1), из которой видно, что даже на слабо загруженной машине прикладная программа может исполняться в течение всего лишь 95–96% времени. Последовательности "загрузка 2" и "загрузка 11" показывают эффективности исполнения, когда активно исполняются другие процессы. В обоих случаях измерения становятся безнадежно неточными для времени исполнения более чем приблизительно 7 мс. Обратите внимание, что ошибка 1.0 означает, что T_m вдвое превышает T_c , в то время как ошибка 10.0 означает, что T_m в 11 раз больше, чем T_c . Очевидно, операционная система планирует к исполнению каждый активный процесс на отдельный интервал времени. Когда активны n процессов, каждый из них получает свою долю, равную $1/n$ от общего процессорного времени.

На основании этих результатов, мы приходим к выводу, что схема K-best обеспечивает точные результаты только для очень коротких вычислений. Это по существу не дает возможности измерять время исполнения, превышающее приблизительно 7 мс, особенно при наличии других активных процессов.

В дополнение ко всему, мы обнаружили, что наша программа измерения не способна надежно определить, действительно ли данное измерение было точным. Наша процедура измерения вычисляет прогнозируемую ошибку по формуле

$$E_p(r) = (v_k - v_1)/v_1,$$

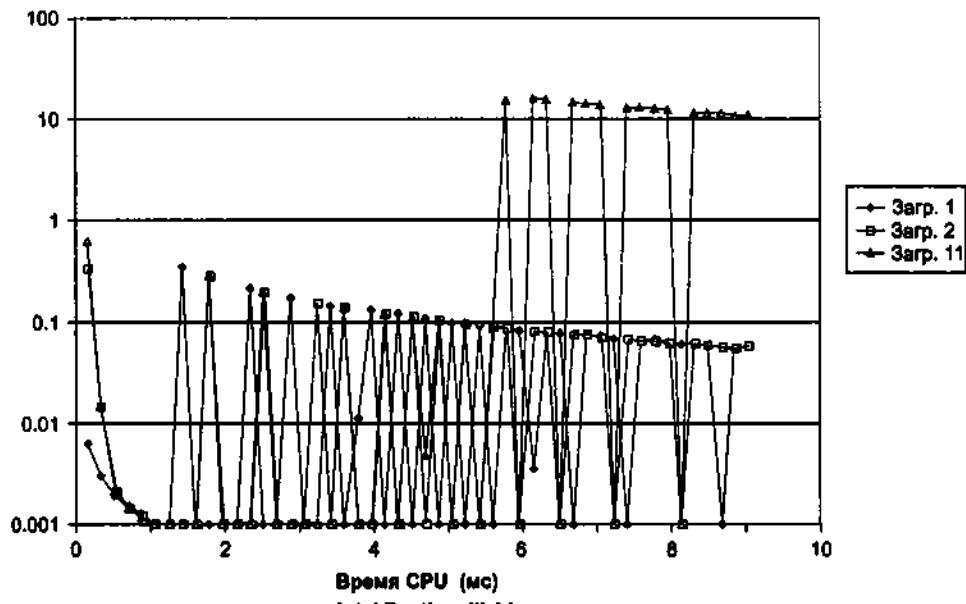
где v_i — это i -е наименьшее измерение. Эта формула вычисляет, насколько быстро достигается наш критерий сходимости. Мы обнаружили, что эти оценки были слишком оптимистическими. Даже для случая загрузки 11, где измерения были выключены коэффициентом 10, программа постоянно оценивала свою ошибку величиной меньше, чем 0.001.

Установка значения K

В наших ранних экспериментах мы для параметра K произвольно устанавливали значение 3, определяя количество измерений, которое требуется для завершения теста, при условии, что показания находились в пределах узкого диапазона, определяющего разброс наиболее быстрых. Чтобы более основательно оценить эффект этого фактора, мы выполнили ряд измерений, используя значения K в пределах от 1 до 5, как показано на рис. 9.10. Мы провели эти измерения для времени исполнения в диапазоне до 9 мс, т. к. это верхний предел достоверности нашей схемы.

Если мы задаем $K = 1$, то процедура возвращает управление после выполнения единственного измерения (рис. 9.10). Это может привести к весьма непредсказуемым результатам, особенно когда машина предельно загружена. Если происходит прерывание от таймера, то результат будет крайне неточен. Даже и без такого катастрофического события измерения будут зависеть от многих источников побочных воздействий. Установка $K = 2$ значительно повышает точность (рис. 9.10). Для времени исполнения меньше 5 мс мы устойчиво получаем точность лучше, чем 0.1%.

Intel Pentium III, Linux
K = 1



Intel Pentium III, Linux
K = 2

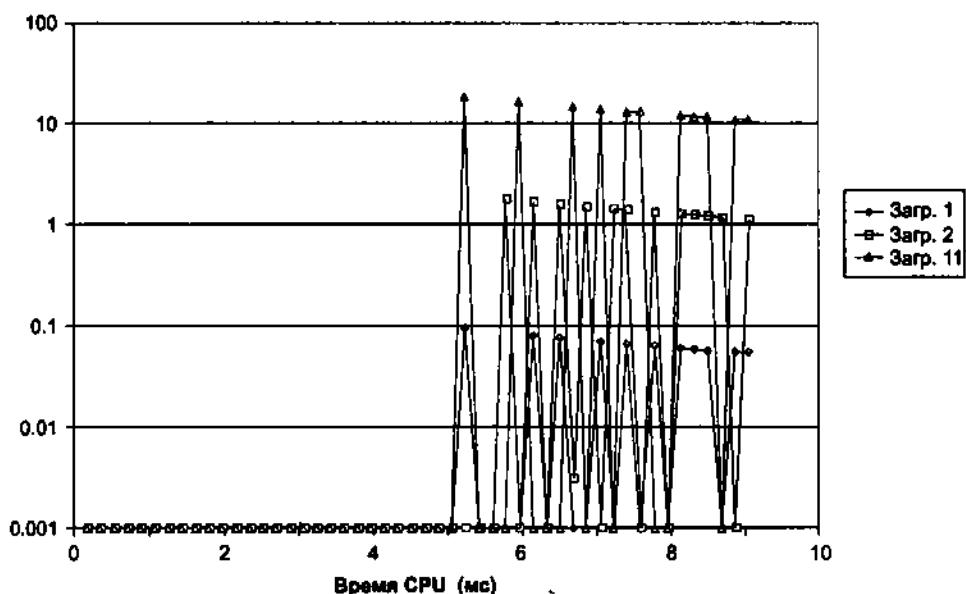
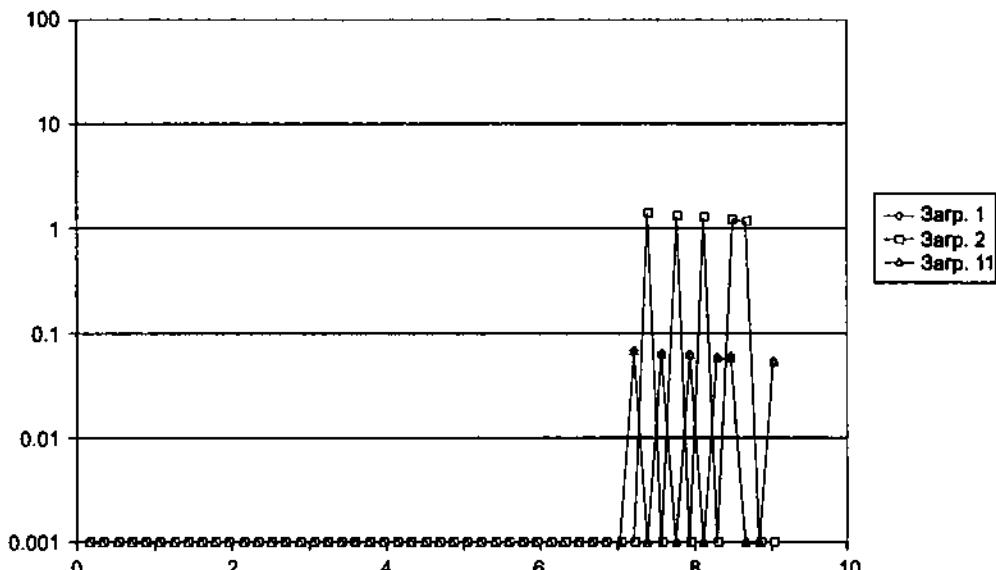


Рис. 9.10. Эффективность схемы K-best для различных значений K (см. продолжение)

Intel Pentium III, Linux
K = 3



Pentium III, Linux
K = 5

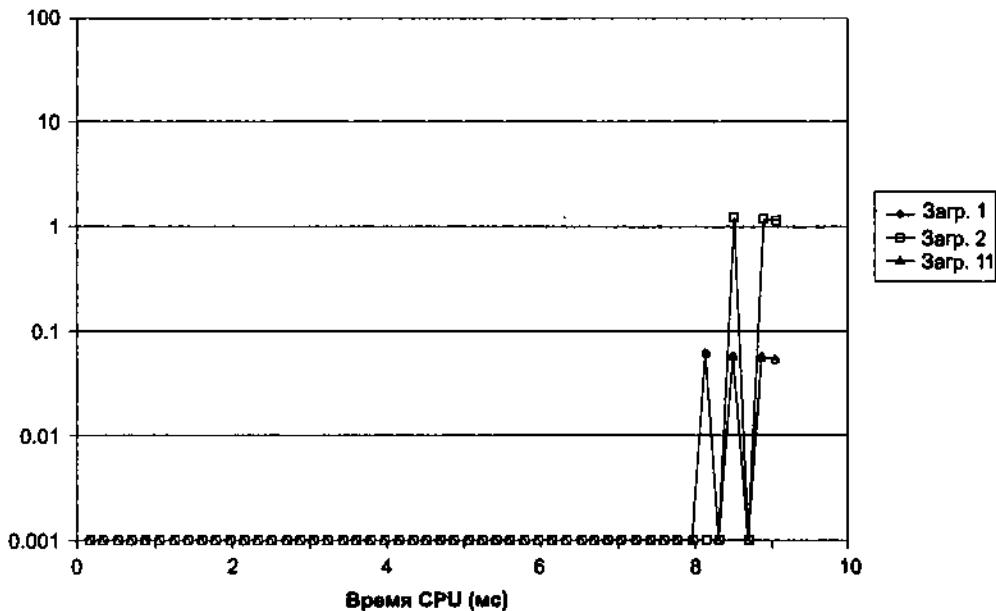


Рис. 9.10. Окончание

Установка еще более высоких значений K дает лучшие результаты и в устойчивости, и в точности, до предела приблизительно 8 мс (см. рис. 9.10). Этот эксперимент показывает, что наша начальная приблизительная оценка $K = 3$ вполне обоснована.

Компенсация обработки прерываний от таймера

Прерывания от таймера предсказуемы и являются источником большой систематической ошибки в наших измерениях для времени исполнения свыше приблизительно 7 мс. Было бы хорошо избавиться от этой систематической ошибки, вычитая из измеренного времени исполнения программы оценку времени, потраченного на обработку прерываний от таймера. Это потребует определения двух факторов:

1. Следует определить, сколько времени требуется для обработки отдельного прерывания от таймера. Мы должны определить минимальное количество тактовых импульсов, необходимых для обслуживания прерывания от таймера. Это обеспечит возможность избежать излишней компенсации.
2. Следует определить, сколько прерываний от таймера происходит за время измерения.

Используя метод, подобный этому, можно сгенерировать трассировку, подобную показанной в листингах 9.1 и 9.2, на ней мы можем выделить периоды пассивности и определить их продолжительность. Некоторые из них будут вызваны прерываниями

Intel Pentium III, Linux
Компенсация накладных расходов

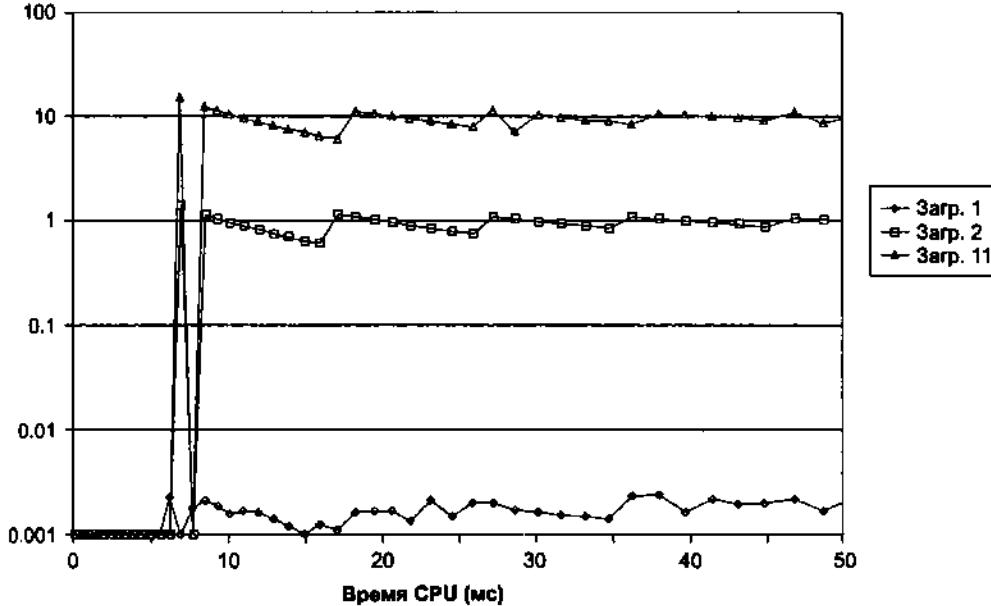


Рис. 9.11. Измерения с компенсацией накладных расходов на прерывание от таймера

от таймера, тогда как причиной других будут иные системные события. Произошло ли прерывание от таймера, мы можем определить путем использования процедуры `times`, поскольку возвращаемое значение будет увеличиваться на единицу с каждым тактом, когда происходит прерывание от таймера. Мы провели такую оценку для 100 отрезков периодов пассивности и обнаружили, что минимальная обработка прерывания от таймера требует 251 466 тактов. Чтобы определить количество прерываний от таймера, которые происходят в течение исполнения измеряемой программы, мы просто вызываем функцию `times` дважды: один раз при входе, и один раз — после завершения измеряемой программы, затем вычисляем разность возвращаемых значений.

На рис. 9.11 показаны результаты, полученные в соответствии с этой пересмотренной схемой измерения. Как показано на рисунке, мы можем теперь получить очень точные (в пределах 1.0%) измерения на слабо загруженной машине, даже для программ, которые исполняются в течение нескольких (с прерываниями) интервалов времени. Устранив систематическую ошибку от прерываний таймера, мы теперь имеем очень надежную схему измерения. С другой стороны, мы можем видеть, что эта компенсация не действует на предельно загруженных машинах.

Вычисление на других машинах

Так как наша схема в большой степени зависит от политики планирования времени конкретной операционной системы, мы также провели эксперименты на трех других системных конфигурациях:

1. Intel Pentium III под управлением более старой версии (2.0.36, а не 2.2.16) ядра Linux;
2. Intel Pentium II под управлением Windows NT. Хотя эта система использует процессор IA32, операционная система существенно отличается от Linux;
3. Alpha Compaq под управлением Tru64 Unix. Она использует совершенно другой процессор, но операционная система подобна Linux.

Как показано на рис. 9.12, характеристики эффективности исполнения под управлением более старой версии Linux сильно отличаются. На слабо загруженной машине точность измерения находится в пределах 0.2% для программ почти любой продолжительности. Мы обнаруживаем, что в этой версии Linux процессор тратит всего лишь 3500 тактов на обработку прерывания от таймера. Даже на интенсивно загруженной машине это позволяет процессам исполняться непрерывно в течение приблизительно 180 мс за один раз. Этот эксперимент показывает, что внутренние особенности операционной системы могут в значительной степени влиять на производительность системы и на наши возможности получить точные измерения.

На рис. 9.13 показаны результаты, полученные в системе Windows NT. В целом, эти результаты подобны тем, что были получены для более старой системы Linux. Для коротких вычислений или на слабо загруженной машине мы могли достичь большой точности измерений. В данном случае погрешность была приблизительно 0.01 (т. е. 1.0%), но не 0.001. Однако этого достаточно для большинства приложений. Кроме того, мы выяснили, что порог между надежными и ненадежными измерениями на

Intel Pentium III, Linux

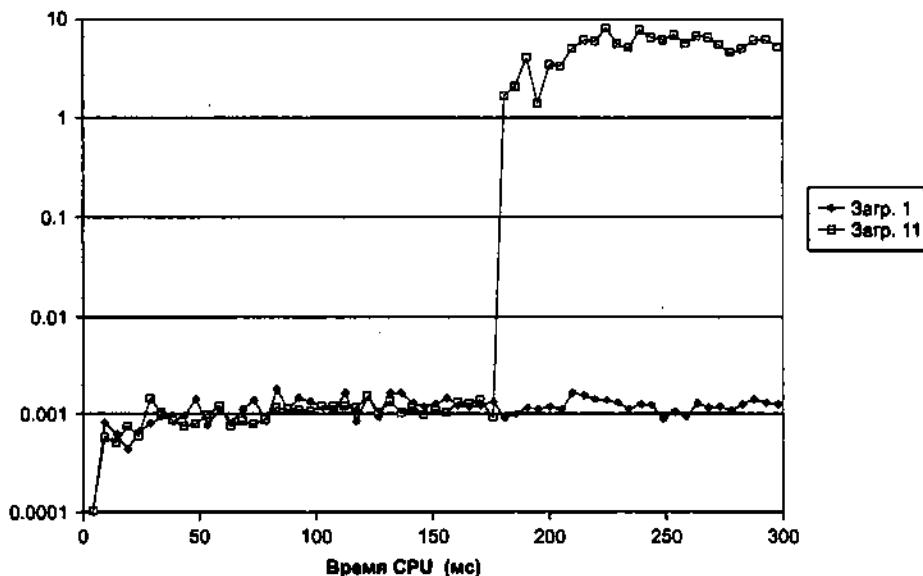


Рис. 9.12. Экспериментальная проверка правильности схемы измерения K-best в системе IA32/Linux

Pentium II, Windows NT

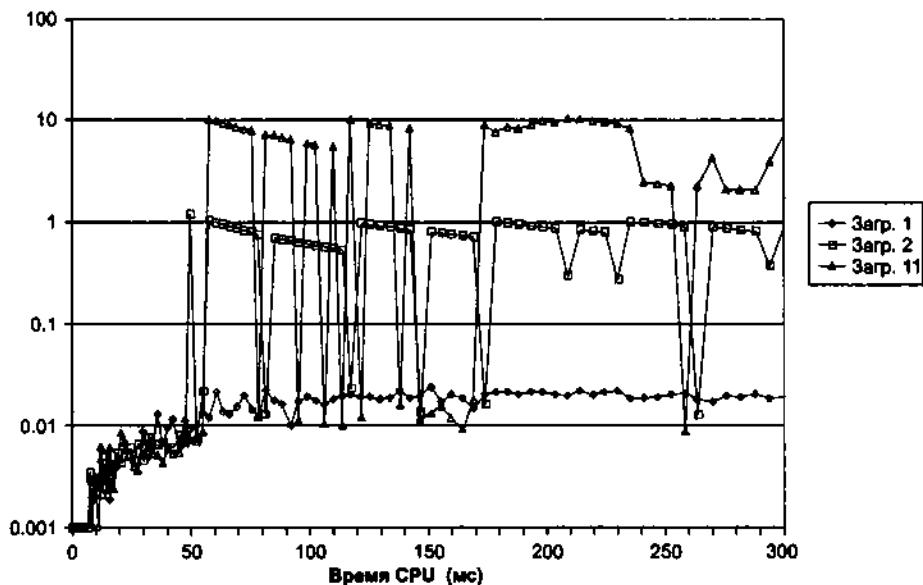


Рис. 9.13. Экспериментальная проверка правильности схемы измерения K-best в системе Windows NT

сильно загруженной машине приблизительно равен 48 мс. Одна интересная особенность состоит в том, что иногда точные измерения получаются на сильно загруженной машине даже для вычислений, дляящихся до 245 мс. Очевидно, планировщик NT иногда позволяет процессам оставаться активным в течение более продолжительного промежутка времени, но мы не можем полагаться на это свойство.

Результаты для Alpha Compaq показаны на рис. 9.14. Как и раньше, мы видим, что на слабо загруженной машине программы почти любой продолжительности могут быть измерены с ошибкой менее, чем 1%. На сильно загруженной машине точно могут быть измерены только программы с продолжительностью исполнения порядка 10 мс и меньше.

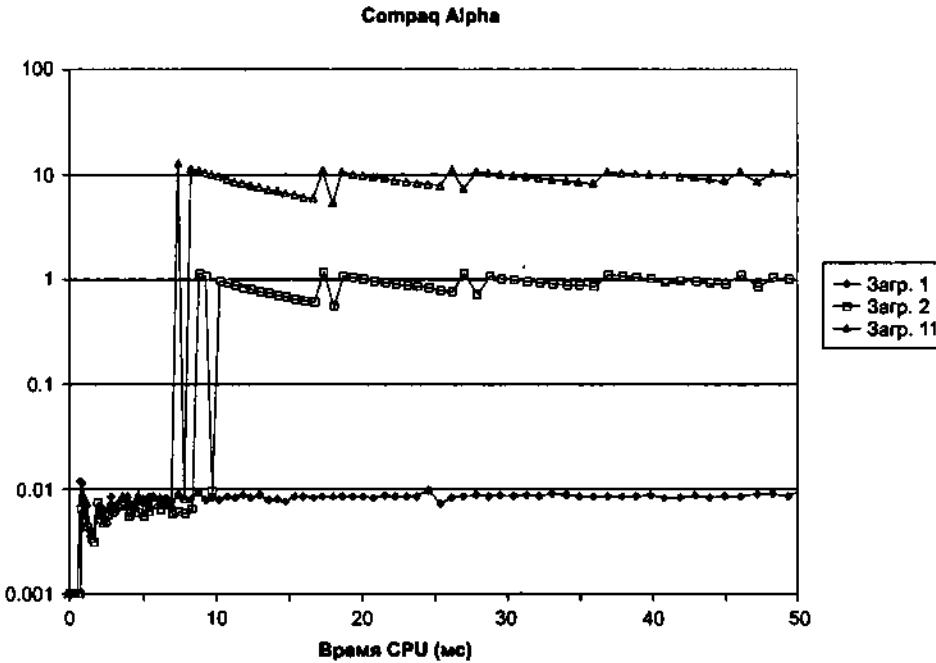


Рис. 9.14. Экспериментальная проверка правильности схемы измерения K-best на системе Alpha Compaq

УПРАЖНЕНИЕ 9.7

Предположим, что нам необходимо провести измерения для процедур, которые требуют t мс. Машина интенсивно загружена, процесс измерения не будет продолжаться более 50 мс за один сеанс.

- Каждое испытание представляет собой процесс измерения для однократного исполнения процедуры. Какова вероятность того, что это испытание произойдет от начала до конца (не прервано), в предположении, что оно начинается в некоторой (произвольной) точке в пределах 50 мс кванта времени? Представьте ваш ответ как функцию от t , рассматривая все возможные значения t .

2. Каким будет ожидаемое число испытаний, чтобы три из них давали надежные измерения процедуры (т. е. каждое должно выполняться в пределах отдельного кванта времени)? Представьте ваш ответ в виде функции от t . Какими, по вашему мнению, должны быть значения для $t = 20$ и $t = 40$?

Результаты наблюдений

Эти эксперименты показывают, что схема измерения K-best работает достаточно хорошо на ряде различных машин. В условиях слабо загруженных процессоров она устойчиво выдает точные результаты на большинстве машин даже для продолжительных интервалов вычислений. Только новая версия Linux несет достаточно высокие непроизводительные затраты ресурсов по обработке прерываний от таймера, которые оказывают серьезное влияние на точность измерения. Для этой системы компенсация упомянутых непроизводительных расходов значительно повышает точность измерения.

На интенсивно загруженных машинах получение точных измерений затруднено. Для большинства систем характерно некоторое максимальное время выполнения, сверх которого точность измерения становится крайне низкой. Точное значение этого порога сильно зависит от системы, но обычно оно находится в пределах от 10 до 200 мс.

9.5. Измерение по часам реального времени

Используемые нами счетчики тактов в IA32 обеспечивают высокую точность измерения времени, но недостаток их состоит в том, что они работают только на системах IA32. Было бы хорошо иметь более универсальное в смысле переносимости средство. Мы видели, что библиотечные функции `times` и `clock` реализованы с использованием счетчиков интервалов и поэтому не очень точны.

Еще одна возможность состоит в том, чтобы использовать библиотечную функцию `gettimeofday`. Эта функция обращается к *системным часам* (*system clock*), чтобы определить текущую дату и время.

```
#include "time.h"
struct timeval {
long tv_sec;    /* секунды */
long tv_usec;   /* мкс */
}
int gettimeofday(struct timeval *tv, NULL);
```

Эта функция записывает время в структуру, переданную ей из вызывающей программы. Структура состоит из двух полей: одно поле содержит секунды, а другое — микросекунды. Первое поле содержит код, представляющий общее количество секунд, прошедших с 1 января 1970 г. (Это — стандартная опорная точка для всех систем Unix.) Обратите внимание, что вторым аргументом функции `gettimeofday` при работе в системах Linux должен быть просто `NULL`, т. к. этот параметр относится к нереализованной функции коррекции показаний времени от часового пояса.

УПРАЖНЕНИЕ 9.8

При какой дате поле `tv_sec`, значение которого определяет функция `gettimeofday`, становится отрицательным на 32-разрядной машине?

Как показано в листинге 9.8, мы можем использовать функцию `gettimeofday` для создания пары функций таймера `start_timer` и `get_timer`, которые подобны используемым нами функциям, подсчитывающим такты, но отличающихся тем, что они измеряют время в секундах, а не в тактовых импульсах.

Листинг 9.8: Процедуры измерения времени

```

1 #include <sys/time.h>
2 #include <unistd.h>
3
4 static struct timeval tstart;
5
6 /* записывает текущее время */
7 void start_timer()
8 {
9     gettimeofday(&tstart, NULL);
10 }
11
12 /* возвращает время в секундах, прошедшее после последнего обращения
   к start_timer */
13 double get_timer()
14 {
15     struct timeval tfinish;
16     long sec, usec;
17
18     gettimeofday(&tfinish, NULL);
19     sec = tfinish.tv_sec - tstart.tv_sec;
20     usec = tfinish.tv_usec - tstart.tv_usec;
21     return sec + 1e-6*usec;
22 }
```

Применимость этого механизма измерения зависит от того, как реализована функция `gettimeofday`, и эта реализация различна для разных систем. Хотя тот факт, что функция генерирует результаты измерений в микросекундах, выглядит довольно многообещающим, оказывается, что эти измерения не всегда настолько точны, как хотелось бы. В табл. 9.2 показан результат испытания рассматриваемой функции в нескольких различных системах. Мы определяем *разрешающую способность* (*resolution*) функции, как минимальное время, различаемое таймером. Мы вычислили ее путем многократного вызова функции `gettimeofday` до тех пор, пока не изменится значение, заданное первым параметром. Таким образом, разрешающая способность — это количество микросекунд, которое привело к этому изменению. Некото-

рые системы могут фактически различать временные величины с точностью до микросекунды, в то время как другие обладают значительно меньшей точностью. Такие расхождения объясняются тем, что некоторые системы для реализации этой функции используют счетчики тактов задающего генератора, в то время как другие используют подсчет интервалов. В первом случае разрешающая способность может быть очень высока — потенциально выше, чем 1 мкс, выше, чем нижний предел, обеспечивающий представлением данных. В последнем случае разрешающая способность будет низкой — не лучше той, что может быть обеспечена функциями `times` и `clock`.

Таблица 9.2. Свойства реализаций функции

Система	Разрешение (мкс)	Задержка (мкс)
Pentium II, Windows NT	10 000	5.4
Compaq Alpha	977	0.9
Pentium III Linux	1	0.9
Sun UltraSparc	2	1.1

В табл. 9.2 также показана *задержка* (*latency*), вызванная обращением к функции `get_time` на различных системах. Эта характеристика означает минимальное время, необходимое для обращения к функции. Мы вычислили это значение путем многократного вызова указанной функции в течение 1 с и последующим делением на число вызовов. Легко видеть, что для вычисления функции требуется приблизительно 1 мкс в большинстве систем. Для сравнения, наша процедура `get_counter` требует всего лишь приблизительно 0.2 мкс на одно обращение. Вообще, системные вызовы связаны с большими накладными расходами, чем обычные вызовы функций. Эта задержка также ограничивает точность наших измерений. Даже если структура данных позволяет выражать время в единицах с более высокой разрешающей способностью, неясно, насколько большую точность мы могли бы получить при измерении времени, когда каждое измерение вносит такую длительную задержку.

На рис. 9.15 показана эффективность исполнения, полученная в результате реализации схемы измерения K-best с использованием функции `gettimeofday` вместо нашей собственной функции обращения к счетчику тактов. Мы показываем здесь результаты, полученные на двух различных машинах, чтобы проиллюстрировать влияние разрешающей способности по времени на точность измерений. Измерения на системе Windows NT показывают характеристики, подобные тем, какие мы получили для системы Linux при использовании функции `times` (см. рис. 9.6). Поскольку функция `gettimeofday` реализована с использованием таймеров процесса, ошибка может быть отрицательной или положительной, и она очень неустойчива в случае измерений небольших промежутков времени. Точность повышается при измерении более продолжительных промежутков времени, по существу, эта ошибка не превышает 2.0% при измерении промежутков, больших, чем 200 мс. Измерения в системе Linux дают результаты, подобные наблюдаемым при непосредственном использовании счетчиков

тактов. Это легко видеть, если сравнить измерения для случая загрузки 1, приведенные, соответственно, на рис. 9.9 (без компенсации) и на рис. 9.11 (с компенсацией). Используя компенсацию, мы можем получить точность, превосходящую 0.04%, даже при измерении промежутков времени порядка 300 мс. Таким образом, `gettimeofday` действует, как если бы она имела непосредственный доступ к счетчику тактов на этой машине.

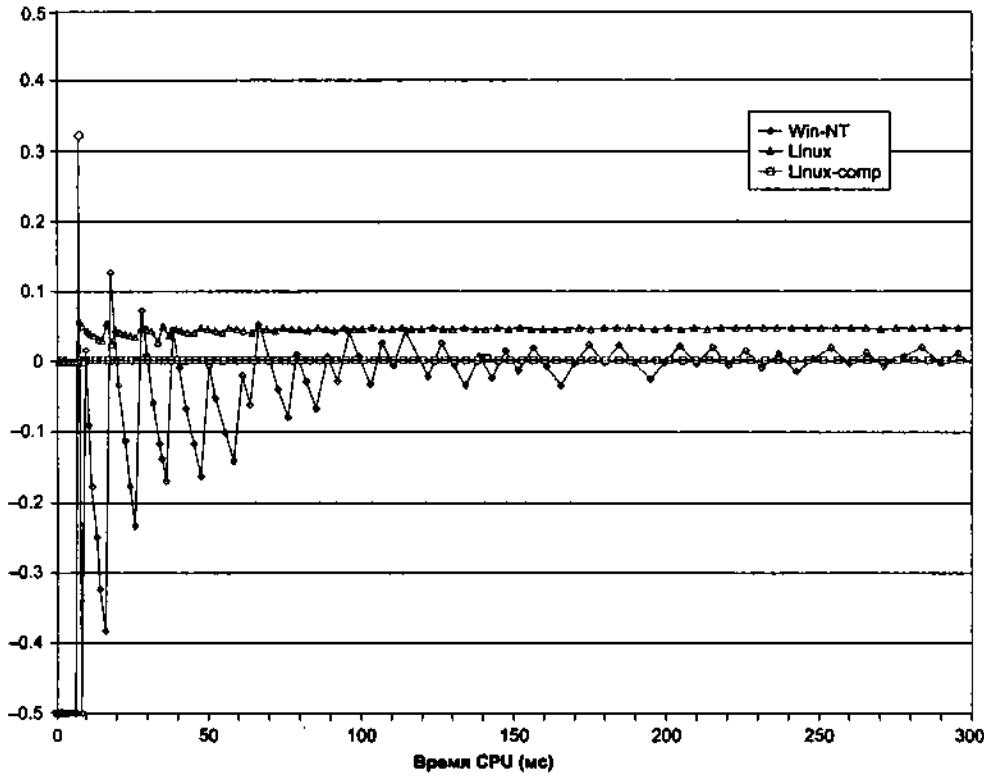


Рис. 9.15. Экспериментальная проверка правильности схемы измерения

Linux реализует эту функцию, используя счетчики тактов, и достигает такой же точности, что и наши собственные подпрограммы измерения времени. Windows NT реализует эту функцию, используя подсчет интервалов, и, следовательно, в этом случае точность низка, особенно при измерениях небольших промежутков времени.

9.6. Протокол эксперимента

Теперь мы можем подвести итог результатов наших экспериментов в форме протокола, чтобы дать ответ на вопрос: как быстро исполняется программа X на машине Y.

- Если ожидаемое время исполнения принимает большое значение (например, более 1.0 с), то подсчет должен работать достаточно хорошо и быть менее чувствителен к загрузке процессора.
- Если ожидаемое время исполнения находится в интервале приблизительно от 0.01 до 1.0 с, то измерения необходимо выполнять на слабо загруженной системе и использовать точное измерение времени с учетом количества тактов. Следует протестировать библиотечную функцию `gettimeofday`, чтобы определить, подсчитывает ли ее реализация на машине такты или интервалы:
 - если функция подсчитывает такты, то можно использовать ее в качестве базовой функции для схемы измерения K-best;
 - если функция подсчитывает интервалы, то следует найти какой-либо подходящий метод, использующий счетчик тактов задающего генератора машины. Это может потребовать написания ассемблерной программы.
- Если ожидаемое время исполнения меньше, чем примерно 0.01 с, то точные измерения могут быть выполнены даже на интенсивно загруженной системе, возможно, путем использования измерения времени посредством подсчета тактов. При этом также будет реализована схема измерения K-best, использующая либо функцию `gettimeofday`, либо прямой доступ к счетчику тактов задающего генератора машины.

9.7. Взгляд в будущее

Существует несколько особенностей, присущих системам и оказывающих существенное воздействие на измерение эффективности исполнения.

- Процесс-ориентированный отсчет тактов. Операционная система может относительно легко управлять счетчиком тактов задающего генератора таким образом, чтобы фиксировалось число тактов на протяжении конкретного процесса. Для этого просто необходимо сохранять результат подсчета как часть состояния процесса. После того как процесс возобновит свою активность, счетчик тактов будет установлен на значение, которое он имел, когда процесс последний раз был переведен в неактивное состояние, по сути дела, счетчик приостанавливается на то время, когда процесс неактивен. Конечно, показание счетчика будет по-прежнему зависеть от "накладных расходов", связанных с операциями ядра операционной системы и от эффектов кэширования, но, по крайней мере, другие процессы не будут оказывать серьезное влияние. Некоторые системы уже поддерживают такую функцию. В рамках используемого нами протокола это позволит производить измерение временных промежутков, используя такты и получая при этом точные значения временных интервалов, превосходящих 0.01 с, даже на интенсивно загруженных системах.
- Часы с изменяющейся скоростью отсчета времени. В стремлении уменьшить потребляемую энергию современные системы изменяют тактовую частоту, поскольку потребляемая энергия непосредственно пропорциональна тактовой частоте. В этом случае зависимость между тактовыми импульсами и наносекундами становится достаточно сложной. Трудно даже сказать, какими единицами измерения

следует пользоваться для выражения эффективности исполнения программы. Для оптимизатора программного кода более целесообразно подсчитывать такты, но для других приложений, например, с ограничениями, накладываемыми режимом реального времени, более важным является фактическое время исполнения.

9.8. Реализация схемы измерения K-best

Мы создали библиотечную функцию `fcyc`, которая использует схему K-best для измерения количества тактовых импульсов, необходимых для функции `f`:

```
#include "clock.h"
#include "fcyc.h"
typedef void (*test_funct)(int *);
double fcyc(test_funct f, int *params);
```

Параметр `params` указывает на целое число. Вообще, он может указывать на массив целых чисел, которые представляют собой параметры измеряемой функции. Например, при измерении функций `lower1` и `lower2`, преобразующие символы заданной строки в символы нижнего регистра, мы передаем в качестве параметра указатель на скалярное целое число, которое является длиной преобразуемой строки. При построении горы памяти (см. главу 6) мы могли бы передать указатель на массив из двух элементов, содержащий размер и шаг индекса.

Есть некоторое количество параметров, которые управляют измерением, например значения K , ϵ и M , а также указание на то, следует ли очищать кэш перед каждым измерением. Эти параметры могут быть установлены функциями, которые также можно найти в библиотеке.

9.9. Задания по пройденному материалу

В попытках построить схему точного измерения времени и получить оценку эффективности этих схем на нескольких различных системах, мы познакомились с некоторыми важными их особенностями:

- Каждая система имеет свои особенности. Различные свойства, определяемые используемыми аппаратными средствами, операционной системой и реализацией библиотечных функций, могут оказать существенное влияние на то, какие виды программ могут быть измерены и с какой точностью.
- Эксперимент может сказать о многом. Мы извлекли большую пользу, ознакомившись с работой планировщика операционной системы, выполняя простые эксперименты по генерированию трассировки активности процессов. Это привело к схеме компенсации, которая значительно повышает точность на слабо загруженной системе Linux. Зная, насколько отличаются результаты для одной системы от результатов для другой системы и даже для различных версий ядра одной и той же ОС, важно уметь анализировать и правильно понимать многие аспекты функционирования системы, которые влияют на ее производительность.

- Получение точных измерений времени на сильно загруженных системах вызывает дополнительные трудности. Большинство исследователей систем делают все свои измерения на специально выделенных для этих целей эталонных системах. Они часто запускают систему на нескольких операционных системах при заблокированных сетевых функциях, чтобы уменьшить влияние источников непредсказуемого воздействия. К сожалению, рядовые программисты лишены такой роскоши. Они должны использовать систему совместно с другими пользователями. Даже на интенсивно загруженных системах наша схема K-best достаточно устойчива для того, чтобы измерять промежутки времени короче, чем интервал таймера.
- В схеме эксперимента должна быть предусмотрена возможность управления некоторыми источниками нестабильности измерения эффективности исполнения. Эффекты кэширования могут в значительной степени повлиять на время исполнения программы. Общепринятая методика предполагает, что кэш должен быть освобожден от любых пригодных для вычисления данных еще до того, как начнется измерение времени, а в другом случае, что он загружен какими-либо данными, которые обычно должны быть в кэше в исходном его состоянии.

9.10. Резюме

Эта глава начиналась, казалось бы, с простого вопроса: как быстро исполняется программа X на машине Y. К сожалению, механизмы компьютерных систем, используемые для запуска на счет нескольких процессов одновременно, затрудняют получение надежных измерений эффективности исполнения программы. Действия операционной системы фактически развиваются в пространстве двух различных масштабов времени. На микроуровне отдельные команды исполняются в моменты времени, измеряемые в наносекундах. На макроуровне операции ввода-вывода выполняются с задержками, измеряемыми в миллисекундах. Компьютерные системы компенсируют это различие, непрерывно переключаясь с одной задачи на другую, теряя каждый раз несколько миллисекунд.

В компьютерных системах используются по существу два различных метода регистрации течения времени. Прерывания от таймера происходят со скоростью, которая кажется очень большой, если события рассматриваются на макроуровне, но очень маленькой, если события рассматриваются на микроуровне. Подсчитывая интервалы, система может использовать очень грубую меру времени исполнения программы. Этот метод пригоден только для долговременного (по меньшей мере, одна секунда) измерения. Счетчики тактов задающего генератора обладают высоким быстродействием, что обеспечивает хорошую точность измерения на микроуровне. Для счетчиков тактов задающего генератора, которые измеряют абсолютное время, переключение контекста может вызвать ошибку в пределах от небольшой неточности (в слабо загруженной системе) до очень большой ошибки (в интенсивно загруженной системе). Таким образом, идеальных схем не существует. Важно понимать, какая точность достижима на каждой из систем.

Эффекты кэширования и прогнозирования условного перехода могут приводить к тому, что время, необходимое для исполнения того или иного отрезка программного

кода, меняется от одного прогона к другому, в зависимости от предыстории обращений к памяти и условных переходов. Мы можем частично управлять этим источником неоднозначности, выполняя заблаговременно некоторый программный код, который переведет кэш в заранее предусмотренное состояние, но эти попытки могут оказаться неудачными в условиях контекстных переключений. Таким образом, мы должны произвести несколько измерений, а затем проанализировать результаты, прежде чем определять истинное время исполнения. К счастью, все источники неоднозначности вызывают возрастание времени исполнения, так что анализ сводится к определению того, является минимальное число измерений точным значением.

Путем выполнения ряда экспериментов нам удалось разработать и проверить правильность схемы измерения времени K-best, в которой производится последовательность повторных измерений до тех пор, пока самые быстрые K не попадут в пределы некоторой области, в которой они достаточно близки друг к другу. В некоторых системах мы можем производить измерения, используя библиотечные функции, возвращающие время дня. В других системах мы должны обращаться к счетчикам тактов задающего генератора через ассемблерный код.

Библиографические замечания

Удивительно мало издавалось литературы по измерению времени исполнения программ. В книге Стивенса (Stevens) [81] задокументированы все библиотечные функции для измерения времени исполнения программ. Книга Уодли (Wadleigh) и Кроуфорда (Crawford) по оптимизации программ [85] описывает, как получить статистические данные по использованию программного кода, и даны описания стандартных функций измерения временных значений.

Задачи для домашнего решения

УПРАЖНЕНИЕ 9.9 ◆◆

Ответьте на вопросы, связанные с использованием трассировки в листинге 9.1. Наша программа оценила тактовую частоту в 549.9 МГц. После этого она вычислила количество миллисекунд измеренного времени в трассировке, исходя из подсчитанных тактов. То есть время, выраженное в тактах как s , программа перевела в миллисекунды по формуле $s/549900$. К сожалению, программный метод вычисления тактовой частоты несовершенен, и следовательно, некоторые из измерений времени в миллисекундах не совсем точны.

1. Интервал таймера для этой машины — 10 мс. Какие из этих периодов времени были инициированы прерыванием от таймера?
2. Исходя из этой трассировки, ответьте на вопрос, каково минимальное количество тактовых импульсов, необходимых операционной системе, чтобы обслужить прерывание от таймера.
3. Исходя из данных трассировки, и предположив, что интервал таймера — точно 10.0 мс, ответьте, каково истинное значение тактовой частоты.

УПРАЖНЕНИЕ 9.10 ◆◆

Напишите программу, использующую библиотечные функции `sleep` и `times`, которая приблизительно определяла бы число тактов системных часов в секунду. Попробуйте компилировать эту программу и запустить ее в нескольких системах. Попробуйте найти две различные системы, которые выдают результаты, различающиеся не менее, чем в два раза.

УПРАЖНЕНИЕ 9.11 ◆

Для того чтобы сгенерировать трассировку активности, мы можем использовать счетчик тактов генератора, например, в листингах 9.1 и 9.2. Используйте функции `start_counter` и `get_counter`, чтобы написать такую функцию:

```
#include "clock.h"
int inactiveduration(int thresh);
```

Эта функция непрерывно проверяет счетчик тактов и обнаруживает, когда два последовательно считанных значения отличаются больше, чем на один такт, и это свидетельствует о том, что процесс был неактивен. Возвращает функция продолжительность (в тактах) этого неактивного отрезка времени.

УПРАЖНЕНИЕ 9.12 ◆

Предположим, что мы вызываем функцию `thz` (см. листинг 9.4) с параметром `sleeptime = 2`. Системный таймер имеет интервал 10 мс. Предположим, что функция `sleep` реализована следующим образом. Процессор поддерживает счетчик, который увеличивает свое значение на единицу каждый раз, когда происходит прерывание от таймера. Когда система исполняет `sleep(x)`, она планирует к исполнению процесс, который будет перезапущен, когда значение счетчика достигает $t + 100x$, где t текущее значение счетчика.

1. Обозначим через w промежуток времени, когда наш процесс неактивен из-за вызова функции `sleep`. Если игнорировать различные накладные расходы из-за вызова функции, прерываний от таймера и т. д., то какой может быть разброс значений w ?
2. Предположим, что вызов функции `thz` возвращает 1000.0. И снова, игнорируя различные непроизводительные затраты ресурсов, ответьте, каков возможный разброс истинной тактовой частоты?

Решение упражнений**РЕШЕНИЕ УПРАЖНЕНИЯ 9.1**

На первый взгляд кажется нецелесообразным прерывать центральный процессор и выполнять при этом 100 000 тактов только для того, чтобы обработать единственное нажатие клавиши. Однако полная загрузка центрального процессора будет при этом совсем небольшой.

100 слов в минуту соответствует 10 нажатиям клавиш в секунду. Общее количество тактов, используемых в секунду этими 100 нажатиями, будет $10 \times 10^2 \times 10^3 = 10^8$, т. е. 10% от общего количества тактов процессора.

РЕШЕНИЕ УПРАЖНЕНИЯ 9.2

Эта задача требует внимательного изучения трассировки и прогнозирования типа модели реализации. События происходят с интервалом 9.98—9.99 мс:

358.93, 368.91, 378.89, 388.88, 398.86, 408.85, 418.83, 428.81

1. Обратите внимание на точки, которые были определены путем добавления 9.98 к предыдущему моменту времени.
2. Эти моменты времени начинают каждый новый интервал пассивного состояния.
3. Неактивные промежутки времени состоят из промежутков времени, потраченного на обслуживание двух прерываний, и времени, когда исполнялся другой процесс.
4. Наш процесс активен в течение приблизительно 9.5 мс из каждого 20 мс, т. е. 47.5% времени.

РЕШЕНИЕ УПРАЖНЕНИЯ 9.3

Эта задача представляет собой просто маркирование последовательности исполнения процесса и определение, находится ли процесс в пользовательском или в привилегированном режиме (рис. 9.16).

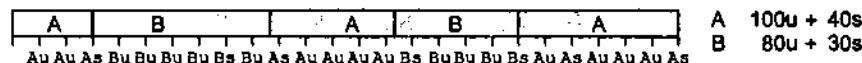


Рис. 9.16. Маркирование последовательности

РЕШЕНИЕ УПРАЖНЕНИЯ 9.4

Это интересная задача на сообразительность. Она заставит вас задуматься о диапазоне возможных значений, которые могли бы привести к данному числу интервалов.

На следующем чертеже представлены такие два случая (рис. 9.17):

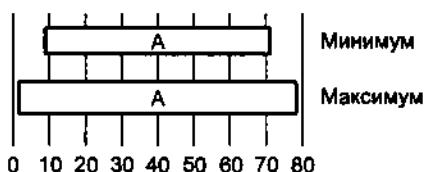


Рис. 9.17. Диапазон значений

Для случая, обозначенного как "минимум", запуск произошел непосредственно перед прерыванием в момент времени 10, и выполнение завершилось сразу же за прерыванием в момент времени 70, что дает чуть более 60 мс. Для случая "максимум" запуск

произошел сразу же после прерывания в момент времени 0, и исполнение продолжалось до момента времени непосредственно предшествующего отметке времени 80, что дает полное время чуть менее 80 мс.

РЕШЕНИЕ УПРАЖНЕНИЯ 9.5

Эта задача требует поразмышлять о том, насколько хорошо работает схема подсчета. Эти семь прерываний от таймера происходят во время, когда процесс активен, поэтому можно было бы подумать, что пользовательское время составит 70 мс, а системное время 0 мс. В фактической трассировке процесс исполнялся в течение 63.7 мс в пользовательском режиме и 3.3 мс в привилегированном. Счетчик дал завышенные показания истинного времени исполнения с коэффициентом $70 / (63.7 + 3.3) = 1.04$.

РЕШЕНИЕ УПРАЖНЕНИЯ 9.6

Эта задача требует поразмышлять о различных источниках задержки в программах и о том, при каких условиях эти источники возникают.

По результатам измерений мы получаем:

$$\begin{aligned}c + m + p + d &= 399; \\c + d &= 133 \pm 1; \\c + p &= 317.\end{aligned}$$

Из этого мы приходим к выводу, что $c = 100$, $d \approx 33$, $p = 217$ и $m = 49$.

РЕШЕНИЕ УПРАЖНЕНИЯ 9.7

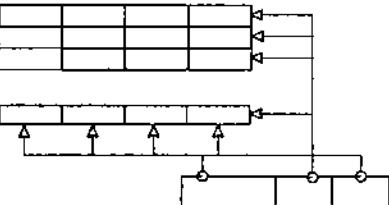
Эта задача требует обращения к простой вероятностной модели планирования процесса. Она показывает, что получение точных измерений становится затруднительным, как только время приближается к предельному для данного процесса.

- Для $t \leq 50$ вероятность исполнения в одном кванте времени составляет $1 - t/50$. Для $t > 50$ вероятность равна 0.
- Для $t \geq 50$ мы не получим ни одного испытания, которое выполняется в пределах одного кванта времени процесса. Для $t < 50$ вероятность положительного исхода составляет $p = (50 - t)/50$, и следовательно, мы можем ожидать $3/p = 150/(50 - t)$ испытаний. Для $t = 20$ мы ожидаем получить 5 испытаний, в то время как для $t = 40$ мы ожидаем 15.

РЕШЕНИЕ УПРАЖНЕНИЯ 9.8

Это версия проблемы двухтысячного года (Y2K) для Unix. Некоторые предсказывали всеобщее бедствие, когда стрелка часов завершит свой последний оборот. Так же, как и для Y2K, мы полагаем, что эти опасения необоснованны.

Это произойдет через 2^{31} секунд после 1 января 1970. На табло будет высвечено 19 января 2038, 3:14 после полуночи.



ГЛАВА 10

Виртуальная память

- Физическая и виртуальная адресация.
 - Пространства адресов.
 - Виртуальная память как средство кэширования.
 - Инструментальное средство манипулирования памятью.
 - Виртуальная память как средство защиты памяти.
 - Преобразование адресов.
 - Система памяти Pentium/Linux.
 - Отображение в памяти.
 - Динамическое распределение памяти.
 - Сборка мусора.
 - Часто встречающиеся ошибки.
 - Сводка некоторых ключевых понятий, связанных с виртуальной памятью.
 - Резюме.
-

Процессы в системе используют CPU (центральный процессор) и оперативную память совместно с другими процессами. Однако совместное использование оперативной памяти различными процессами порождает некоторые специфические проблемы. По мере возрастания запросов на ресурс CPU, скорость исполнения процессов постепенно уменьшается. Но если слишком много процессов потребуют слишком большого пространства памяти, то некоторые из них просто не смогут исполняться. Если программа выйдет за пределы выделенного пространства памяти, то это может привести за собой катастрофические последствия.

Информация в памяти подвержена искажениям. Если некоторый процесс по недосмотру запишет свои данные в память, используемую другим процессом, то исполнение того процесса может сорваться самым непостижимым способом, совершенно не связанным с логикой программы.

Чтобы управлять памятью более эффективно и с минимально возможным количеством ошибок, современные системы используют абстракцию оперативной памяти, известную как *виртуальная память* (VM, virtual Memory). Виртуальная память — это элегантная композиция взаимодействующих между собой исключительных ситуаций аппаратного уровня, трансляции адресов; оперативной памяти, файлов на диске и программ ядра системы, которые обеспечивают каждый процесс большим, однородным и закрытым адресным пространством. С помощью хорошо продуманного механизма виртуальная память обеспечивает три важных свойства:

- она эффективно использует оперативную память, обрабатывая ее как кэш для адресного пространства, сохраненного на диске, оставляя в оперативной памяти только активно используемые области и передавая, по мере необходимости, данные в обоих направлениях между диском и памятью;
- она упрощает управление памятью, обеспечивая каждый процесс однородным адресным пространством;
- она защищает адресное пространство каждого процесса от разрушения его другими процессами.

Виртуальная память — это одна из прекрасных идей, реализованных в компьютерных системах. Главная причина ее успеха состоит в том, что она работает автоматически и без какого-либо вмешательства со стороны прикладного программиста. И если виртуальная память работает так хорошо, оставаясь в тени, так зачем тогда программисту разбираться в ее устройстве? Для этого есть несколько причин.

- Виртуальная память находится в центре событий. Виртуальная память пронизывает все уровни компьютерных систем, играя ключевую роль в схеме аппаратных исключений, в работе ассемблеров, компоновщиков и редакторов связей, загрузчиков, в совместном использовании объектов, файлов и процессов. Знание устройства виртуальной памяти поможет вам лучше понимать, как вообще работают такие системы.
- Виртуальная память обладает большими возможностями. Виртуальная память предоставляет приложениям широкие возможности создавать и уничтожать участки памяти, отображать участки памяти на разделы файлов на диске, и обеспечивать совместное использование памяти несколькими процессами. Например, знаете ли вы о том, что можно читать или изменять содержимое файла на диске, читая и производя записи в области памяти? Или о том, что можно загрузить содержимое файла в память, не делая никакого явного копирования? Понимание устройства виртуальной памяти поможет вам использовать ее мощные возможности в ваших приложениях.
- Виртуальная память таит в себе опасности. Приложения взаимодействуют с виртуальной памятью каждый раз, когда в них встречается ссылка на переменную, разыменование указателя или производится запрос к одному из пакетов программ динамического распределения памяти, например к пакету `malloc`. Если виртуальная память не используется надлежащим образом, приложения могут попасть в тупиковую ситуацию, по причине появления изошренных, труднообъяснимых ошибок в памяти. Например, программа с неправильным указателем может сразу прекратить выполнение с диагнозом "ошибка сегментации" или "ошибка защиты".

памяти", но может молча выполнятся в течение многих часов, прежде чем произойдет ее аварийное завершение, либо, что хуже всего, завершит свое исполнение нормально, но с неправильными результатами. Знание основ виртуальной памяти и механизмов распределения памяти, например пакета malloc, могут помочь вам избежать таких ошибок.

В этой главе виртуальная память рассматривается с двух точек зрения. В первой половине главы мы покажем, как работает виртуальная память. Во второй половине мы опишем, как виртуальную память используют приложения, и как они управляют ею. Мы не можем игнорировать тот факт, что VM — это сложный механизм, и повсюду при обсуждении будем учитывать этот факт. Положительная сторона такого подхода заключается в том, что знание деталей этого механизма поможет вам самостоятельно смоделировать механизм виртуальной памяти небольшой системы, а сама идея виртуальной памяти навсегда потеряет для вас свою мистическую ауру.

Во второй части главы, предполагая, что читатель хорошо усвоил основные понятия, мы покажем вам, как использовать и управлять виртуальной памятью в ваших программах. Вы узнаете, как управлять виртуальной памятью, используя явное отображение в памяти и вызовы программных средств динамического распределения памяти, таких как malloc. Вы также столкнетесь со множеством ошибок, вызываемых неправильным использованием памяти в программах на языке C, и узнаете, как не допустить их возникновения.

10.1. Физическая и виртуальная адресация

Оперативная память компьютерной системы организована как массив последовательно расположенных ячеек размером в один байт. Каждый байт имеет уникальный физический адрес (PA, Physical Address). Первый байт имеет адрес 0, следующий байт — адрес 1, следующий за ним байт — адрес 2 и т. д. При такой простой организации памяти самый естественный способ доступа центрального процессора к памяти состоит в использовании физических адресов. Мы называем такой подход *физическими адресацией*. На рис. 10.1 показан пример физической адресации в контексте команды загрузки, которая считает слово, начинающееся по физическому адресу 4.

Когда центральный процессор (CPU) исполняет команду загрузки, он генерирует эффективный физический адрес и передает его в оперативную память по шине памяти. Оперативная память выбирает четырехбайтовое слово, начинающееся по физическому адресу 4, и возвращает это слово центральному процессору, который сохраняет его в регистре.

Самые первые ПК использовали физическую адресацию, и такие системы, как процессоры цифровых сигналов, встроенные микроконтроллеры, и супер-ЭВМ фирмы Cray продолжают использовать этот вид адресации. Однако современные процессоры, разработанные для универсальных вычислений, используют форму адресации известную как *виртуальная адресация* (VA, Virtual Addressing) (рис. 10.2).

Используя виртуальную адресацию, центральный процессор осуществляет доступ к оперативной памяти, генерируя *виртуальный адрес*, который преобразуется в соот-

ветствующий физический адрес, прежде чем произойдет обращение к памяти. Задача преобразования виртуального адреса в физический известна как трансляция адреса, или *переадресация* (address translation). Подобно обработке исключительных ситуаций, трансляция адреса требует тесного взаимодействия между аппаратными средствами центрального процессора и операционной системой. Специализированное аппаратное средство на плате центрального процессора, которое называется *модулем управления памятью* (MMU, Memory Management Unit), по ходу дела транслирует виртуальные адреса в физические, используя с этой целью таблицу преобразования, хранящуюся в оперативной памяти, содержимым которой управляет операционная система.

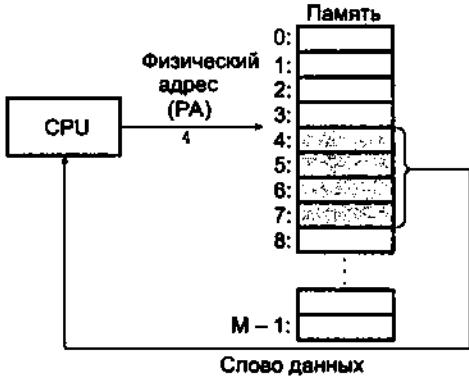


Рис. 10.1. Система, использующая физическую адресацию

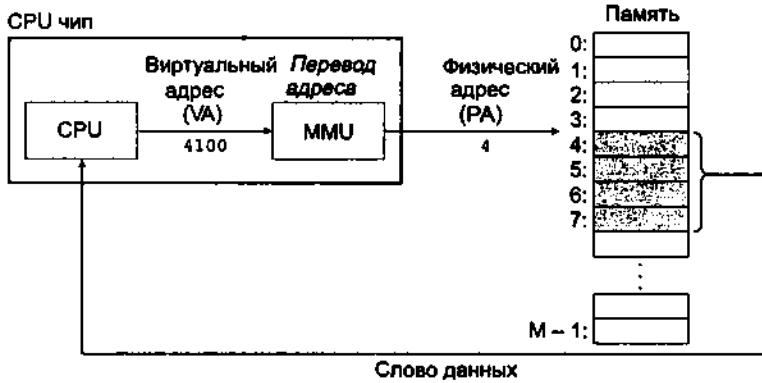


Рис. 10.2. Система, использующая виртуальную адресацию

10.2. Пространство адресов

Адресное пространство есть упорядоченное множество неотрицательных целочисленных адресов $\{0, 1, 2, \dots\}$. Если целочисленные значения адресного пространства линейно упорядочены, то мы говорим, что это *линейное адресное пространство*.

(linear address space). Чтобы упростить наше рассуждение, мы будем всегда полагать, что всегда используются линейные адресные пространства. В системе с виртуальной памятью центральный процессор генерирует виртуальные адреса из адресного пространства $N = 2^n$ адресов, называемого *виртуальным адресным пространством* (virtual address space) $\{0, 1, 2, \dots, N - 1\}$. Размер адресного пространства определяется количеством двоичных разрядов (битов), необходимых для представления наибольшего адреса. Например, виртуальное адресное пространство с $N = 2^n$ адресами называют n -разрядным адресным пространством. Современные системы обычно поддерживают 32- или 64-разрядные виртуальные адресные пространства.

В системе также имеется *физическое адресное пространство* (physical address space), которое соответствует M байтам физической памяти в системе $\{0, 1, 2, \dots, M - 1\}$.

M не обязательно должно быть степенью двойки, но для простоты рассуждений мы примем, что $M = 2^m$. Важность понятия адресного пространства объясняется тем, что оно проводит четкое различие между объектами данных (байтами) и их атрибутами (адресами). Как только мы осознаем это различие, мы сможем сделать обобщение и позволить каждому объекту данных иметь несколько независимых адресов, каждый из которых выбирается из своего адресного пространства. Такова основная идея виртуальной памяти. Каждому байту оперативной памяти ставится в соответствие виртуальный адрес из виртуального адресного пространства и физический адрес, выбранный из физического адресного пространства.

УПРАЖНЕНИЕ 10.1

Дополните приводимую ниже таблицу, вписав значения в незаполненные ячейки, и замените каждый вопросительный знак соответствующим целым числом. Используйте следующие единицы: $K = 2^{10}$ (Кило), $M = 2^{20}$ (Мега), $G = 2^{30}$ (Гига), $T = 2^{40}$ (Тера), $P = 2^{50}$ (Пета) или $E = 2^{60}$ (Эзба).

Количество битов виртуального адреса	Количество виртуальных адресов	Максимально возможный виртуальный адрес
8		
	$2^7 = 64K$	
		$2^{32} - 1 = ?G - 1$
	$2^7 = 256T$	
64		

10.3. Инструментальное средство кэширования

Концептуально виртуальная память организована как массив N последовательно расположенных ячеек размера 1 байт, хранящихся на диске. Каждый байт имеет уни-

кальный виртуальный адрес, который служит индексом в этом массиве. Содержимое массива на диске кэшируется в оперативной памяти. Как и в случае любого другого кэша в иерархии памяти, данные на диске (более низкий уровень) разделены на блоки, которые служат единицами обмена между диском и оперативной памятью (верхний уровень). VM-системы манипулируют ими, разбивая виртуальную память на блоки фиксированного размера, называемые *виртуальными страницами* (VP, Virtual Pages). Каждая виртуальная страница имеет размер $P = 2^p$ байтов. Точно так же физическая память разделена на *физические страницы* (PP, Physical Pages), также размером P байтов. Физические страницы называются также *страницами блоками* (page frames). В любой момент времени множество виртуальных страниц разбито на три непересекающихся подмножества:

- незанятые страницы, которые еще не распределены (или не были созданы) системой виртуальной памяти. Незанятые блоки не содержат никаких данных и таким образом не занимают никакого пространства на диске;
- кэшированные страницы, которые на текущий момент отведены под кэш в физической памяти;
- некэшированные — распределенные страницы, на текущий момент не кэшируются в физической памяти.

В примере, представленном на рис. 10.3, показана виртуальная память, состоящая из 8 виртуальных страниц. Виртуальные страницы 0 и 3 еще не были распределены, они не существуют на диске. Виртуальные страницы 1, 4 и 6 кэшированы в физической памяти. Страницы 2, 5 и 7 распределены, но на текущий момент не кэшированы в оперативной памяти.

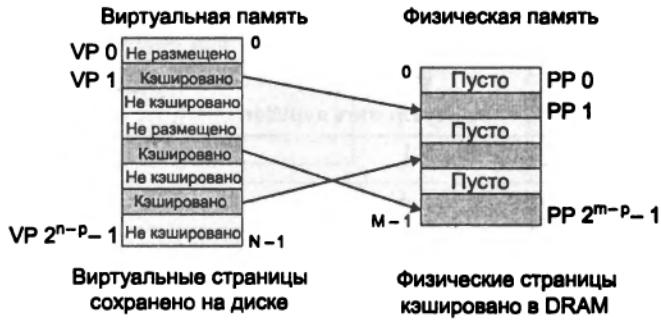


Рис. 10.3. Как система VM использует оперативную память в качестве кэш

10.3.1. Организация кэш в DRAM

Чтобы легче было ссылаться на различный кэш в иерархии памяти, мы будем использовать термин кэш SRAM (Static Random Access Memory, статическое запоминающее устройство с произвольной выборкой) для обозначения кэш-памяти первого (L1) и второго (L2) уровня между центральным процессором и оперативной памятью и тер-

мин кэш DRAM (Dynamic Random Access Memory, динамическое запоминающее устройство с произвольной выборкой) для обозначения кэш VM-системы для виртуальных страниц в оперативной памяти.

Место, занимаемое DRAM в иерархии памяти, имеет определяющее значение при выборе способа его организации. Еще раз напомним, что DRAM приблизительно в 10 раз медленнее, чем SRAM, и что диск приблизительно в 100 тыс. раз медленнее, чем DRAM. Таким образом, неудача при обращении к DRAM обходится очень дорого (в смысле времени), по сравнению с неудачным обращением к SRAM, поскольку обращения к DRAM требуют доступа к диску, в то время как обращения к SRAM обычно требуют доступа к оперативной памяти, построенной на базе DRAM. Более того, чтение первого байта из дискового сектора производится приблизительно в 100 тыс. раз медленнее, чем чтение последующих байтов сектора. Стремление сократить такого рода затраты явилось причиной организации DRAM.

В силу больших непроизводительных затрат, обусловленных отсутствием страниц и большим временем доступа к первому байту, существует тенденция делать виртуальные страницы большими, обычно их размеры выбираются в пределах от четырех до восьми килобайт. Из-за больших непроизводительных затрат, связанных с отсутствием нужной страницы, DRAM делают полностью ассоциативными, т. е. любая виртуальная страница может быть помещена в физическую страницу. Стратегия замен при отсутствии нужной страницы также имеет большое значение, поскольку накладные расходы, связанные с заменой отсутствующей виртуальной страницы, слишком высоки. Следовательно, операционные системы используют намного более сложные алгоритмы замены для DRAM, чем алгоритм аппаратных средств для SRAM. (Алгоритмы замены выходят за пределы интересующих нас вопросов.) Наконец, из-за большого времени доступа к диску, DRAM всегда используют алгоритм обратной записи, а не алгоритм сквозной записи, когда данные заносятся одновременно в кэш и в ОЗУ.

10.3.2. Таблицы страниц

Как и в случае любого кэша, система виртуальной памяти должна быть способной определить, кэширована ли виртуальная страница в DRAM. Если да, то система должна определить, в какой физической странице она кэширована. Если нужная страница отсутствует, система должна определить, в каком месте диска хранится эта виртуальная страница, выбрать в физической памяти страницу, которую можно удалить, и скопировать виртуальную страницу с диска в DRAM на место удаляемой страницы.

Эти возможности обеспечиваются некоторым сочетанием программного обеспечения операционной системы, аппаратных средств трансляции адресов блока MMU (блок управления памятью) и структуры данных, хранящейся в физической памяти, известной как *таблица страниц* (page table), которая отображает виртуальные страницы на физические страницы. Аппаратные средства трансляции адресов обращаются к таблице страниц каждый раз, когда нужно преобразовать виртуальный адрес в физиче-

ский адрес. Операционная система отвечает за поддержку содержимого таблицы страниц и за передачи страниц в обоих направлениях между диском и DRAM.

Рис. 10.4 служит иллюстрацией основных принципов построения таблицы страниц. Таблица страниц представляет собой массив элементов таблицы страниц (PTE, Page Table Entry). Каждая страница в виртуальном адресном пространстве имеет свой элемент PTE, смещение которого в таблице страниц фиксировано. Для наших целей предположим, что каждый элемент PTE состоит из разряда достоверности (valid bit) и n -разрядного поля адреса. Разряд достоверности показывает, кэширована ли в настоящее время данная виртуальная страница в DRAM. Если разряд достоверности установлен (т. е. равен 1), то поле адреса указывает на начало соответствующей физической страницы в DRAM, в которой кэширована виртуальная страница. Если разряд достоверности не установлен (т. е. равен 0), то нулевой адрес свидетельствует о том, что виртуальная страница еще не была закреплена за какой-либо физической страницей. В противном случае адрес указывает на начало виртуальной страницы на диске.

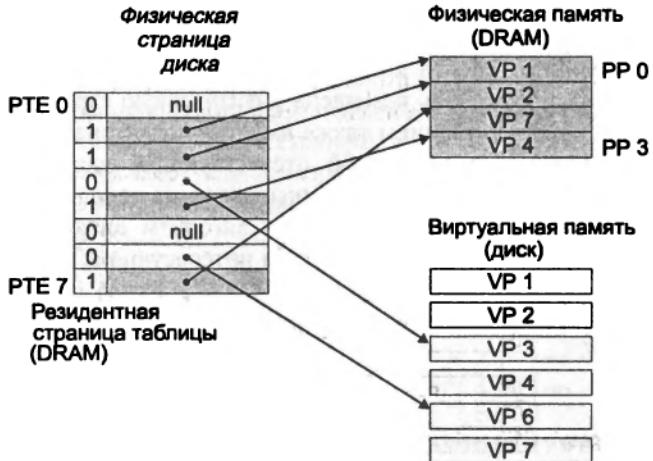


Рис. 10.4. Таблица страниц

В примере на рис. 10.4 показана таблица страниц для системы с 8 виртуальными страницами и 4 физическими страницами. Четыре виртуальных страницы (VP 1, VP 2, VP 4 и VP 7) в рассматриваемый момент времени кэшированы в DRAM. Две страницы (VP 0 и VP 5) еще не были распределены, а остальные (VP 3 и VP 6) размещены и не кэшированы. Важно обратить здесь внимание на то, что поскольку DRAM полностью ассоциативен, любая физическая страница может содержать любую виртуальную страницу.

УПРАЖНЕНИЕ 10.2

Определите количество элементов PTE таблицы страниц, которые необходимы для следующих комбинаций разрядности виртуального адреса n и размера страницы P :

n	$P = 2^n$	Количество PTE
16	4К	
16	8К	
32	4К	
32	8К	

10.3.3. Страница находится в DRAM

Рассмотрим, что происходит, когда центральный процессор читает слово из страницы виртуальной памяти, содержащейся в VP 2, которая кэширована в DRAM (рис. 10.5). В соответствии с методикой, которую мы опишем подробно в разд. 10.6, аппаратные средства трансляции адреса используют виртуальный адрес как индекс, чтобы определить местонахождение PTE 2 и прочитать слово из памяти. Поскольку разряд достоверности установлен в 1, аппаратным средствам трансляции адреса известно, что страница VP 2 кэширована в памяти. Следовательно, для получения физического адреса слова центральный процессор воспользуется физическим адресом памяти в PTE (который указывает на начало страницы, кэшированной в PP 0).

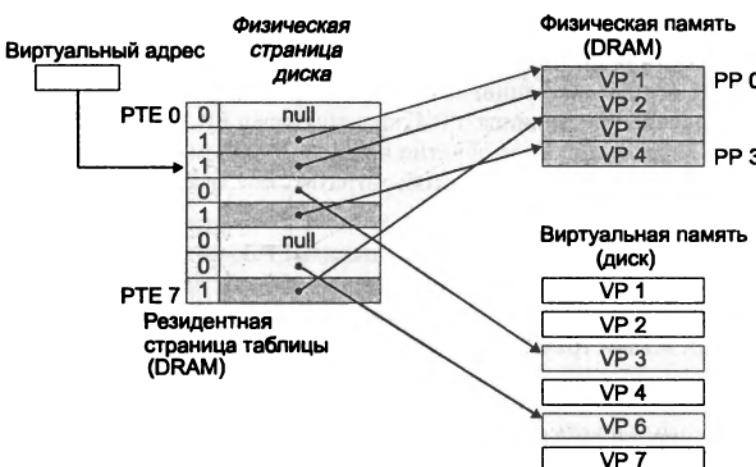


Рис. 10.5. Страница VM находится в DRAM

10.3.4. Обращение к отсутствующей странице

Применительно к виртуальной памяти отсутствие нужной страницы в DRAM называется ошибкой из-за отсутствия страницы (page fault). На рис. 10.6 показано состояние таблицы страниц, используемой в нашем примере, до того, как произошел сбой подобного рода. Центральный процессор сослался на слово в странице VP 3, которая

не кэширована в DRAM. Аппаратные средства модуля трансляции адреса предпринимают попытку читать PTE 3 из памяти, но по состоянию разряда достоверности приходят к выводу, что VP 3 не кэширована, и выдают сообщение об исключительной ситуации, возникшей по причине отсутствия нужной страницы.

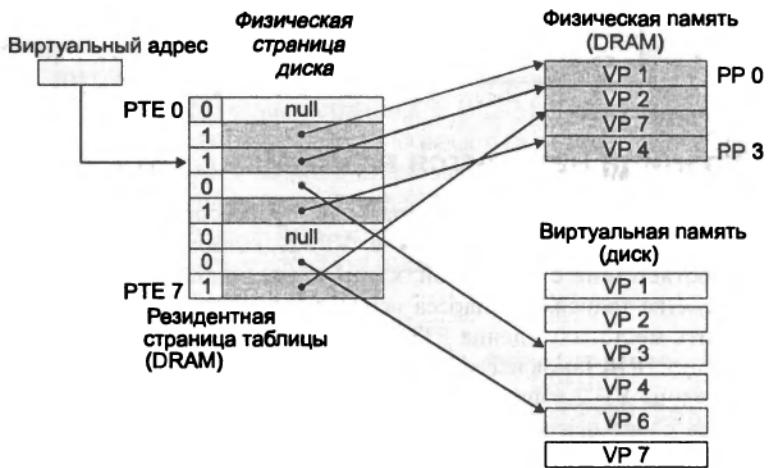


Рис. 10.6. Ошибка из-за отсутствия нужной страницы виртуальной памяти (до сбоя)

Исключительная ситуация при обращении к отсутствующей странице вызывает из ядра обработчик особых ситуаций, который выбирает страницу, подлежащую удалению, в данном случае это страница VP 4, хранящуюся в PP 3. Если VP 4 подверглась изменениям, то ядро копирует ее обратно на диск. В любом случае, ядро корректирует элемент таблицы страницы VP 4, чтобы отразить тот факт, что VP 4 теперь уже не кэширована в оперативной памяти.

После этого ядро копирует VP 3 из диска в память PP 3, модифицирует PTE 3 и затем возвращает управление. После возвращения из обработчика перезапускается вызвавшая сбой команда, которая вновь пересыпает вызвавший сбой виртуальный адрес к аппаратным средствам трансляции адреса. Но теперь VP 3 уже кэширована в оперативной памяти, и искомая страница будет обработана аппаратными средствами трансляции адреса обычным образом, как мы видели это на рис. 10.5. На рис. 10.7 показано состояние таблицы страниц нашего примера после обращения к отсутствующей странице.

Механизмы виртуальной памяти были придуманы в начале шестидесятых годов прошлого столетия, задолго до того, как непрерывно расширяющийся разрыв между оперативной памятью и центральным процессором был ликвидирован с помощью SRAM. В результате виртуальные системы памяти используют терминологию, отличающуюся от терминологии SRAM, несмотря на то, что в обоих случаях используются одни и те же идеи. В терминологии языка виртуальной памяти блоки называются страницами. Действия по пересылке страниц между диском и памятью называются подкачкой (swapping) или подкачкой страниц (paging). Страницы загружаются (swapped in, paged in) из диска в DRAM и выгружаются (swapped out, paged out) из

DRAM на диск. Стратегия выжидания загрузки страницы до последнего момента, когда происходит сбой по отсутствию страницы, называется замещением страниц по требованию (demand paging). Возможны и другие подходы, такие, например, как попытка предсказания отсутствия страницы и предварительная подкачка страницы до фактической ссылки на нее. Тем не менее все современные системы используют замещение страниц по требованию.

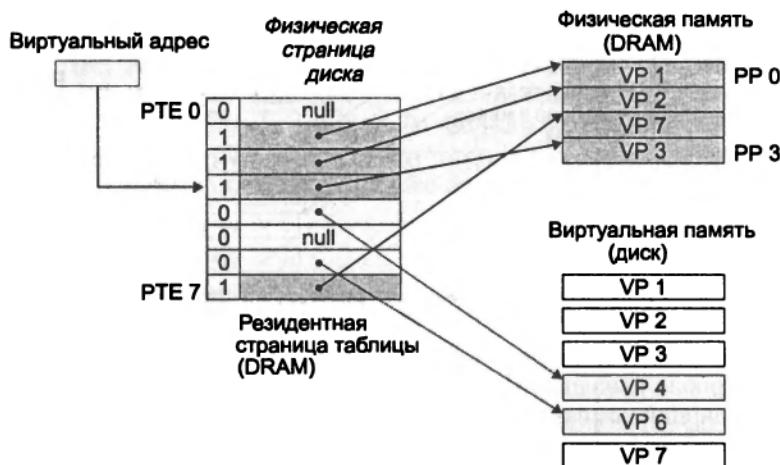


Рис. 10.7. Ошибка из-за отсутствия нужной страницы виртуальной памяти (после сбоя)

10.3.5. Размещение страниц

На рис. 10.8 показаны изменения в таблице страниц нашего примера, когда операционная система выделяет новую страницу виртуальной памяти, например, в результате вызова программы `malloc`. В этом примере на диске выделен участок памяти для страницы `VP 5`, а элемент `PTE 5` изменен так, чтобы указывал на вновь созданную страницу на диске.

Ядро размещает страницу `VP 5` на диске и устанавливает значение элемента `PTE 5`, указывающее на ее новое место в памяти (на диске).

10.3.6. Снова о компактности размещения

У многих из нас, когда мы узнаем об идеях, лежащих в основе виртуальной памяти, часто создается первое впечатление, что эти механизмы должны быть исключительно неэффективными. При таких больших непроизводительных затратах ресурсов, вызываемых отсутствием страницы, нас одолевает беспокойство, что подкачка страниц сведет на нет показатель эффективности исполнения программы. Тем не менее на практике виртуальная память работает совсем неплохо, главным образом благодаря нашей старой знакомой компактности размещения (locality).

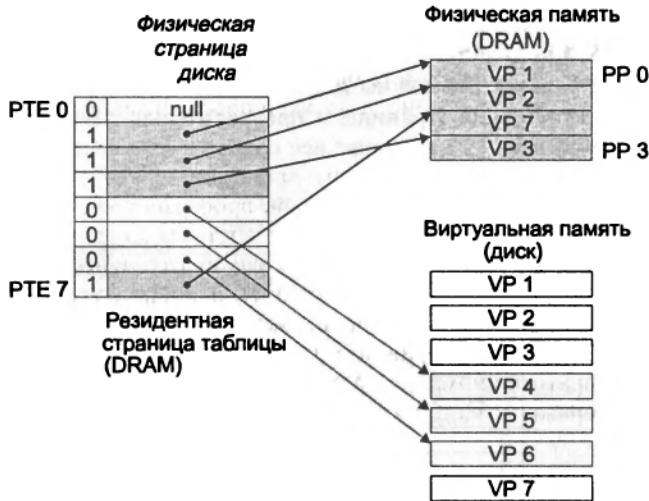


Рис. 10.8. Размещение новой виртуальной страницы

Хотя общее количество различных страниц, на которые ссылаются программы в течение всего периода исполнения, может превысить весь объем физической памяти, тем не менее принцип компактности размещения предполагает, что в каждый момент времени они будут работать с меньшим набором активных страниц (active page), называемым рабочим набором (working set), или *резидентным набором* (resident set) страниц. После начальных затрат на размещение рабочего набора в памяти последующие объекты ссылок находятся в рабочем наборе и не будут сопровождаться дополнительным обменом с диском.

До тех пор, пока наши программы расположены в памяти компактно, система виртуальной памяти функционирует без сбоев. В то же время, не для каждой программы можно обеспечить оптимальное размещение в памяти. Если размер рабочего набора превышает размер физической памяти, то работа программы может привести к такой нежелательной ситуации, которая называется *пробуксовкой* (thrashing), в условиях которой страницы непрерывно перекачиваются с диска в память и обратно. Обычно виртуальная память работает эффективно, но если скорость исполнения программы заметно уменьшается, опытный программист сразу же заподозрит, что возможна пробуксовка памяти.

Подсчет обращений к отсутствующей странице

Отслеживать количество обращений к отсутствующей странице (и получить много другой полезной информации) можно с помощью функции `getrusage` системы Unix.

10.4. Манипулирование памятью

В последнем разделе мы видели, как виртуальная память обеспечивает механизмы, позволяющие использовать DRAM с целью кэширования страниц, как правило, из значительно большего виртуального адресного пространства. Интересно, что некото-

рые ранние системы, такие как, например, PDP-11/70 компании DEC, поддерживали виртуальное адресное пространство, которое было меньше, чем вся физическая память. Тем не менее виртуальная память была и в этом случае полезным механизмом, значительно упрощавшим управление памятью и обеспечивавшим естественный способ защиты памяти.

До сих пор мы предполагали существование единственной таблицы страниц, которая отображает единственное виртуальное адресное пространство на физическое адресное пространство. На самом деле операционные системы каждому процессу предоставляют отдельную таблицу страниц и, следовательно, отдельное виртуальное адресное пространство. Рис. 10.9 служит иллюстрацией основной идеи этого механизма. В нашем примере таблица страниц для процесса *i* отображает VP 1 на PP 2 и VP 2 — на PP 7. Точно так же таблица страниц для процесса *j* отображает VP 1 на PP 7 и VP 2 — на PP 10. Обратите внимание на тот факт, что несколько виртуальных страниц могут быть отображены на одну и ту же совместно используемую физическую страницу.

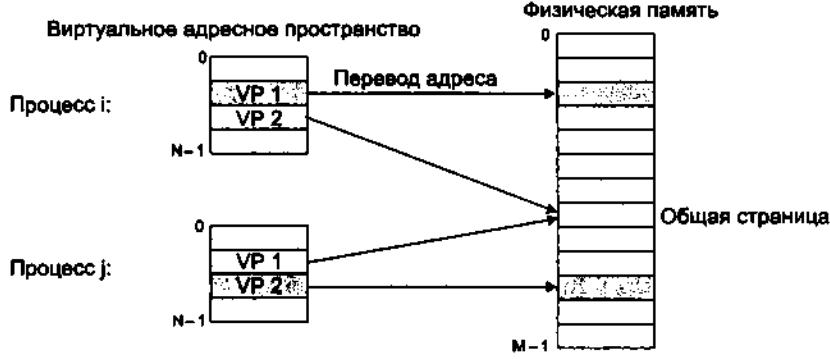


Рис. 10.9. Как виртуальная память предоставляет различным процессам отдельные адресные пространства

Сочетание принципа замещения страниц по требованию с использованием отдельных виртуальных адресных пространств оказывает глубокое влияние на способ использования памяти и алгоритмы управления ею в системе. В частности, виртуальная память упрощает редактирование связей и загрузку, совместное использование программных кодов и данных, а также выделение памяти приложениям.

10.4.1. Упрощение компоновки

Собственное адресное пространство позволяет каждому процессу использовать один и тот же основной формат для его образа в памяти, независимо от того, где на текущий момент находятся программные коды и данные в физической памяти. Например, каждый процесс Linux использует формат, представленный на рис. 10.10.

Раздел текста всегда начинается по виртуальному адресу 0x08048000, стек всегда располагается от адреса 0xbfffffff, программный код совместно используемой библиотеки всегда начинается по адресу 0x40000000, а программный код операционной системы

мы и данные всегда начинаются по адресу 0x00000000. Такой жесткий порядок очень упрощает разработку и реализацию редакторов связей, позволяя получать полностью связанные исполняемые файлы, не зависящие от того, в каком месте в области физической памяти расположены программный код и данные.



Рис. 10.10. Образ памяти процесса, исполняемого под управлением Linux

10.4.2. Упрощение совместного использования

Разделенные адресные пространства предоставляют операционной системе согласованный механизм для управления совместным использованием памяти пользовательскими процессами и самой операционной системой. В общем случае, каждый процесс имеет свой собственный закрытый программный код, данные, динамическую память и области стеков, которые используются исключительно этим процессом. При этом операционная система создает таблицы страниц, которые отображают соответствующие виртуальные страницы на не перекрывающиеся физические страницы. В то же время в некоторых случаях желательно, чтобы процессы могли совместно использовать некоторый программный код и данные. Возьмем, например, случай, когда каждый процесс должен вызывать один и тот же программный код ядра операционной системы, и каждая С-программа осуществляет вызовы подпрограмм из стандартной С-библиотеки, например, программы printf. Вместо того, чтобы включать отдельные копии ядра и стандартной С-библиотеки в каждый процесс, операционная система может устроить дело таким образом, чтобы несколько процессов совместно использовали единственную копию этого программного кода, отображая соответствующие виртуальные страницы различных процессов на одни и те же физические страницы.

10.4.3. Упрощение выделения памяти

Виртуальная память реализует простой механизм выделения пользовательским процессам дополнительной памяти. Когда программа, выполняющаяся в пользовательском процессе, запрашивает дополнительное пространство из "динамической памяти" (например, в результате вызова функции `malloc`), операционная система выделяет соответствующее число, скажем k , последовательно расположенных страниц виртуальной памяти и отображает их на k произвольных физических страниц, расположенных в произвольном месте в физической памяти. В силу особенностей функционирования таблиц страниц, нет необходимости в том, чтобы операционная система определяла место этих последовательно расположенных k страниц физической памяти. Страницы могут быть беспорядочно рассеяны в физической памяти.

10.4.4. Упрощение загрузки

Помимо прочего виртуальная память облегчает загрузку исполняемых и совместно используемых объектных файлов в память. Напомним, что разделы `.text` и `.data` в исполняемых модулях ELF расположены последовательно. Чтобы загрузить эти разделы во вновь порожденный процесс, загрузчик системы Linux выделяет набор последовательно расположенных виртуальных страниц, начинающихся по адресу `0x08048000`, отмечает их как неперемещаемые (т. е. не кэшируемые), и записывает в элементы их таблиц страниц указатели на соответствующие области памяти объектного файла.

Интерес представляет тот факт, что загрузчик на самом деле никогда не копирует никаких данных с диска в память. Данные помещаются системой виртуальной памяти в страницы автоматически и по требованию при каждой ссылке на соответствующую страницу, по требованию центрального процессора при выборе очередной команды или исполняемой командой, когда она ссылается на конкретную область памяти.

Такая концепция отображения множества последовательно расположенных виртуальных страниц на произвольное место того или иного файла называется *отображением в памяти* (*memory mapping*). Операционная система Unix реализует команду системного вызова, получившую название `mmap`, позволяющую прикладным программам получать свое собственное отображение в памяти. Более подробно отображение в памяти на уровне приложения мы опишем в разд. 10.8.

10.5. VM как средство защиты памяти

Любая современная компьютерная система должна обладать средствами, с помощью которых операционная система могла бы управлять доступом к системе памяти. Пользовательскому процессу нельзя разрешать изменять свой раздел текста только для чтения. Нельзя также разрешать читать или изменять какой-либо программный код и структуры данных ядра системы. Нельзя разрешать чтение или производить записи в закрытую память других процессов, нельзя также разрешать внесение изменений в какие-либо виртуальные страницы, которые используются совместно с други-

гими процессами, пока все заинтересованные стороны однозначно не дадут на это своего согласия (путем явных системных вызовов функций взаимодействия между процессами).

Как мы уже видели, наличие раздельных виртуальных адресных пространств облегчает изоляцию закрытых областей памяти различных процессов. Но механизм трансляции адресов может быть расширен естественным образом с тем, чтобы обеспечить еще более тонкое управление доступом. Поскольку аппаратные средства трансляции адреса читают элементы PTE каждый раз, когда центральный процессор генерирует адрес, они непосредственно управляют доступом к содержимому виртуальной страницы, добавляя к PTE некоторые дополнительные разряды разрешения (permission bits). Рис. 10.11 служит иллюстрацией общей идеи такого подхода.

Таблицы страниц с битами разрешения

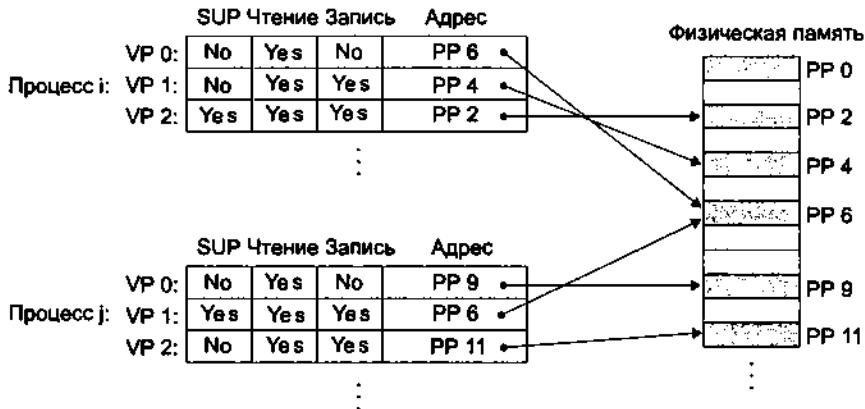


Рис. 10.11. Использование виртуальной памяти для обеспечения защиты памяти на уровне страниц

В этом примере к каждому элементу PTE мы добавили по три управляющих разряда. Разряд SUP показывает, должен ли данный процесс исполняться в режиме ядра (привилегированный режим), чтобы обратиться к заданной странице. Процессы, исполняющиеся в привилегированном режиме, могут обратиться к любой странице, но процессам, исполняющимся в непривилегированном режиме, разрешено обращаться только к страницам, для которых SUP содержит 0. Биты READ и WRITE управляют доступом по чтению и по записи к заданной странице. Например, если процесс *i* исполняется в непривилегированном режиме, он может читать страницу VP 0 и читать или производить запись в страницу VP 1. В то же время ему не позволено обращаться к странице VP 2.

Если команда пытается нарушить эти ограничения доступа, центральный процессор генерирует сигнал о нарушении общей защиты памяти и передает управление обработчику особых ситуаций в ядре. Командный процессор операционной системы Unix, как правило, сообщает об этой исключительной ситуации как об ошибке сегментации (segmentation fault).

10.6. Преобразование адресов

Этот раздел посвящен вопросам преобразования адресов. Наша задача состоит в том, чтобы достаточно подробно изучить механизм поддержки виртуальной памяти аппаратными средствами с тем, чтобы можно было самостоятельно выполнять некоторые конкретные примеры. В то же время следует заметить, что мы опускаем некоторые подробности, в частности связанные с измерением времени, которые являются немаловажными для разработчиков аппаратных средств, но выходят за пределы наших интересов. В справочных целях в табл. 10.1 приводятся символы, которыми мы будем пользоваться на всем протяжении этого раздела.

Таблица 10.1. Сводка символов трансляции адреса

Основные параметры	
Символ	Описание
$N = 2^n$	Количество адресов в виртуальном адресном пространстве
$M = 2^m$	Количество адресов в физическом адресном пространстве
$P = 2^p$	Размер страницы (байты)
Компоненты VA (Virtual Address)	
Символ	Описание
VPO	Смещение виртуальной страницы (Virtual Page Offset) в байтах
VPN	Номер виртуальной страницы (Virtual Page Number)
TLBI	Индекс в TLB (Translation Lookaside Buffer, буфер быстрого преобразования адреса)
TLBT	Тег TLB
Компоненты физического адреса (PA)	
Символ	Описание
PPO	Смещение физической страницы в байтах (Physical Page Offset)
PPN	Номер физической страницы (Physical Page Number)
CO	Смещение в пределах блока кэша в байтах
CI	Индекс в кэше (Cashe Index)
CT	Тег кэша (Cashe Tag)

С формальной точки зрения, трансляция адреса — это отображение между N элементами виртуального адресного пространства (VAS, Virtual Address Space) и M элементами физического адресного пространства (PAS, Physical Address Space):

$$\text{MAP : VAS} \rightarrow \text{PAS} \cup \emptyset$$

где:

- $\text{MAP}(A) = A'$, если данные с виртуальным адресом A хранятся по физическому адресу A' в пространстве PAS;
- $\text{MAP}(A) = \emptyset$, если данные с виртуальным адресом A отсутствуют в физической памяти.

На рис. 10.12 показано, каким образом блок MMU использует таблицу страниц, чтобы выполнить это отображение. Управляющий регистр центрального процессора, базовый регистр таблицы страниц (PTBR, Page Table Base Register), указывает на текущую таблицу страниц. n -битовый виртуальный адрес имеет два компонента: p -разрядное смещение виртуальной страницы (VPO, Virtual Page Offset) и $(n-p)$ -разрядный номер виртуальной страницы (VPN, Virtual Page Number). Блок MMU использует VPN, чтобы выбрать соответствующий элемент PTE. Например, VPN 0 выбирает PTE 0, VPN 1 выбирает PTE 1 и т. д. Соответствующий физический адрес — это конкатенация номера физической страницы (PPN, Physical Page Number), считая от входа таблицы страниц, и смещения VPO, считая от виртуального адреса. Обратите внимание на то, что поскольку как физические, так и виртуальные страницы имеют размер P байтов, смещение физической страницы (PPO, Physical Page Offset) идентично VPO.

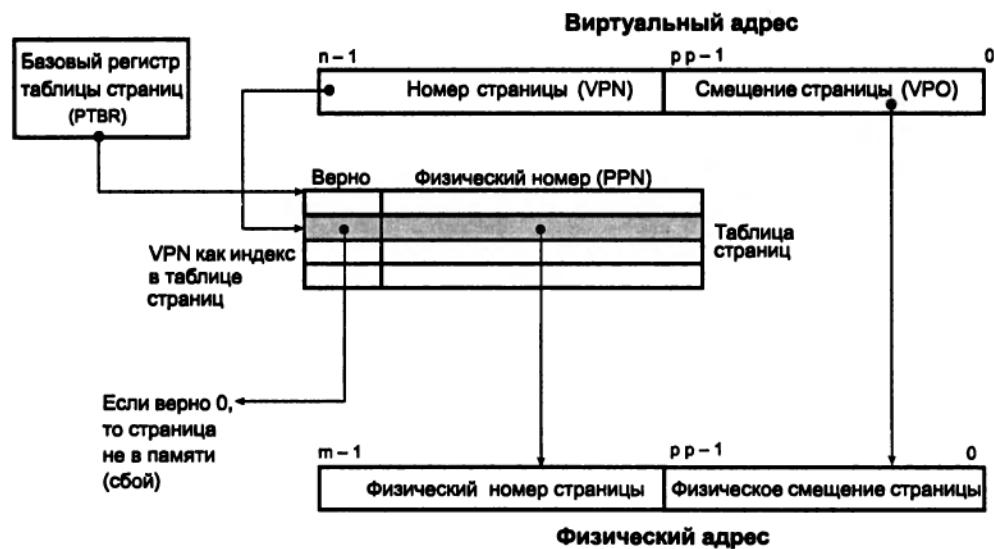


Рис. 10.12. Трансляция адресов с помощью таблицы страниц

На рис. 10.13 показана последовательность действий, выполняемых аппаратными средствами центрального процессора, когда искомая страница находится в памяти:

1. Процессор генерирует виртуальный адрес и отсылает его в MMU.
2. Блок MMU генерирует адрес PTE и запрашивает его из кэша или из оперативной памяти.

3. Кэш/оперативная память возвращает РТЕ в ММУ.
4. Блок ММУ образует физический адрес и отсылает его в кэш/оперативную память.
5. Кэш/оперативная память возвращает в процессор запрошенное слово данных.

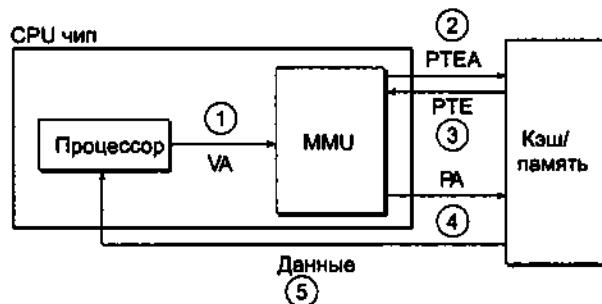


Рис. 10.13. Представление операций, выполняемых при наличии искомых страниц

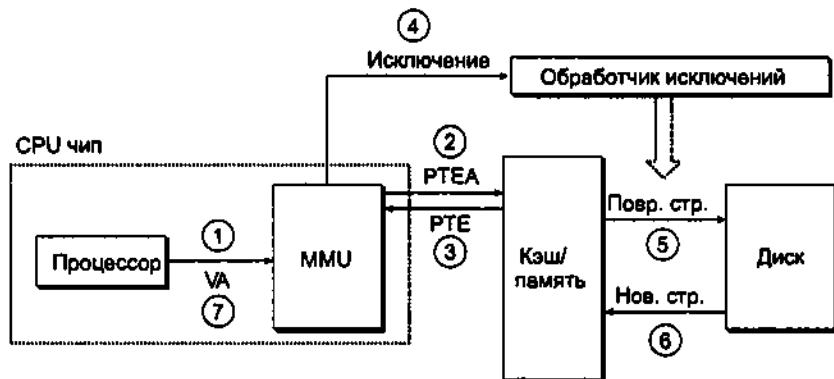


Рис. 10.14. Представление операций, выполняемых при отсутствии искомых страниц

В отличие от случая, когда искомая страница находится в месте, полностью обрабатываемом аппаратными средствами, обработка обращения к отсутствующей странице требует взаимодействия между аппаратными средствами и ядром операционной системы (рис. 10.14).

- (Шаги 1—3.) Первые три действия — те же самые, что и на рис. 10.13.
- (Шаг 4.) Разряд достоверности в РТЕ в этом случае равен нулю, так что блок ММУ возбуждает исключительную ситуацию, которая передает управление в центральный процессор обработчику исключительных ситуаций при обращении к отсутствующей странице в ядре операционной системы.
- (Шаг 5.) Обработчик ошибки идентифицирует подлежащую удалению страницу в физической памяти, и если эта страница подвергалась изменениям, отправляет ее на диск.

- (Шаг 6.) Обработчик ошибки помещает новую страницу на освободившееся место и модифицирует элемент PTE в памяти.
- (Шаг 7.) Обработчик ошибки возвращает управление исходному процессу, и это приводит к тому, что команда, вызвавшая сбой, будет исполнена повторно. Центральный процессор снова посыпает виртуальный адрес в блок MMU. Поскольку искомая виртуальная страница теперь кэшируется в физической памяти, после того, как блок MMU выполнит действия, представленные на рис. 10.14, оперативная память возвращает запрошенное слово процессору.

УПРАЖНЕНИЕ 10.3

Заданы 32-разрядное виртуальное адресное пространство и 24-разрядный физический адрес, определите число битов в VPN, VPO, PPN и PPO для следующих страниц размера P :

P	Число разрядов в VPN	Число разрядов в VPO	Число разрядов в PPN	Число разрядов в PPO
1 Кбайт				
2 Кбайт				
4 Кбайт				
8 Кбайт				

10.6.1. Интегрирование кэш в виртуальную память

Для любой системы, использующей как виртуальную память, так и кэш SRAM, можно поставить вопрос, следует ли при обращении к кэш использовать виртуальные или физические адреса. Хотя детальное обсуждение вопросов обмена не является предметом нашего рассмотрения, заметим, что большинство систем выбирают физическую адресацию. При использовании физической адресации естественным является

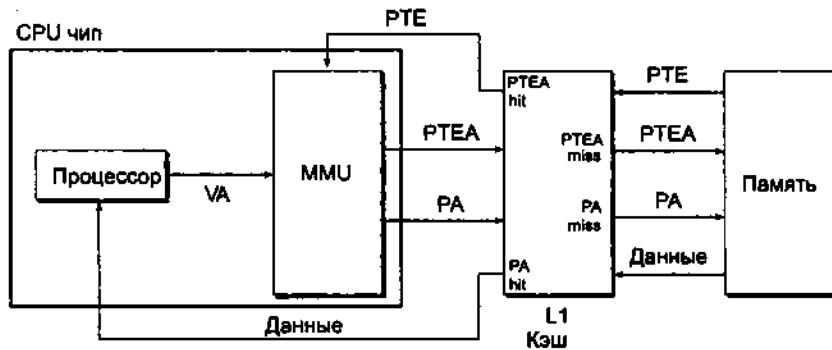


Рис. 10.15. Объединение виртуальной памяти с физически адресуемым кэш

положение, когда несколько процессов в одно и то же время имеют блоки в кэш и совместно используют блоки одних и тех же виртуальных страниц. Далее, кэш-память не связана с механизмами защиты, поскольку права доступа проверяются в процессе трансляции адреса.

На рис. 10.15 показано, как физически адресуемый кэш может быть объединен с виртуальной памятью. Основная идея заключается в том, что трансляция адреса производится еще до того, как осуществляется поиск в кэш. Обратите внимание на то, что элементы таблицы страниц могут кэшироваться точно так же, как и любые другие слова данных.

10.6.2. Ускорение трансляции адресов с помощью TLB

Как мы могли убедиться, каждый раз, когда центральный процессор генерирует виртуальный адрес, блок MMU должен обратиться к PTE, чтобы преобразовать виртуальный адрес в физический адрес. В наихудшем случае, это потребует дополнительной выборки из памяти, на что придется затратить от нескольких десятков до нескольких сотен тактов. Если окажется, что элемент PTE кэширован на уровне L1, то затраты снижаются до одного или двух тактов. Однако многие системы пытаются устранить даже эти затраты, включая в блок MMU небольшой кэш элементов PTE, так называемый *буфер быстрого преобразования адреса* (TLB, Translation Lookaside Buffer).

Буфер TLB представляет собой небольшой, виртуально адресуемый кэш, в котором каждая строка — блок, состоящий из единственного элемента PTE. Буфер TLB обычно имеет высокую степень ассоциативности. Как показано на рис. 10.16, поля индекса и тега, используемые для выбора набора и сравнения строк, извлекаются из номера виртуальной страницы в виртуальном адресе. Если буфер TLB может содержать $T = 2^l$ наборов, то *индекс TLB* (index TLBI) состоит из l младших битов VPN, а под *тег TLB* (tag TLBT) отводятся оставшиеся биты VPN.

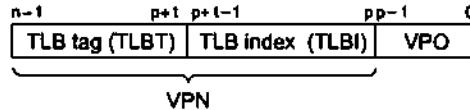


Рис. 10.16. Компоненты виртуального адреса, используемые для обращения к TLB

На рис. 10.17 показана последовательность действий, выполняемых, когда буфер TLB находится в памяти (обычная ситуация). Здесь важно то, что все этапы трансляции адреса выполняются в блоке MMU, благодаря чему обеспечивается необходимое быстродействие.

- (Шаг 1.) Центральный процессор генерирует виртуальный адрес.
- (Шаги 2—3.) Блок MMU выбирает из буфера TLB соответствующий PTE.
- (Шаг 4.) Блок MMU транслирует виртуальный адрес в физический адрес и отсылает его в кэш или оперативную память.

- (Шаг 5.) Кэш/оперативная память возвращает запрошенное слово данных в центральный процессор.

Если буфер TLB отсутствует, то блок MMU должен выбрать элемент PTE из кэша уровня L1, как показано в нижней части рис. 10.17. Новый выбранный PTE сохраняется в TLB, при этом, возможно, перезаписывается существующая запись.

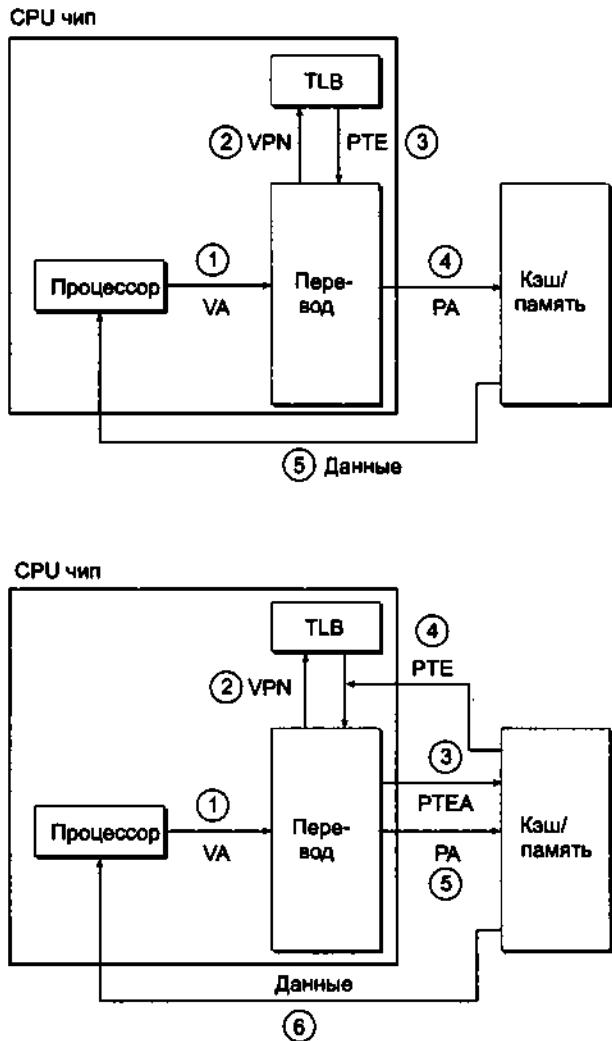


Рис. 10.17. Представление операций, выполняемых при наличии или отсутствии блока TLB

10.6.3. Многоуровневые таблицы страниц

До сих пор мы полагали, что для трансляции адреса система использует единственную таблицу страниц. Но если бы у нас было 32-разрядное адресное пространство, страницы размером 4 Кбайт и 4-байтовые элементы PTE, то необходимо было бы постоянно держать в памяти таблицу страниц размером 4 Мбайт, даже если приложение ссылалось бы только на небольшой участок виртуального адресного пространства. Задача существенно усложняется для систем с 64-разрядным адресным пространством.

Общепринятый подход, заключающийся в уплотнении таблицы страниц, предполагает использование иерархии таблиц страниц. Сама по себе эта идея чрезвычайно проста, и мы поясним ее на конкретном примере. Предположим, что 32-разрядное виртуальное адресное пространство разбито на страницы по 4 Кбайт, и каждый элемент таблицы страниц занимает четыре байта. Предположим также, что в некоторый момент времени виртуальное адресное пространство имеет следующую форму: первые 2К (1К соответствует числу 1024) страниц памяти отведены для программных кодов и данных, следующие 6K страниц свободны, дальнейшие 1023 страницы также свободны, а следующая за ними страница отведена для пользовательского стека. На рис. 10.18 представлен вариант построения двухуровневой иерархии для этого виртуального адресного пространства.

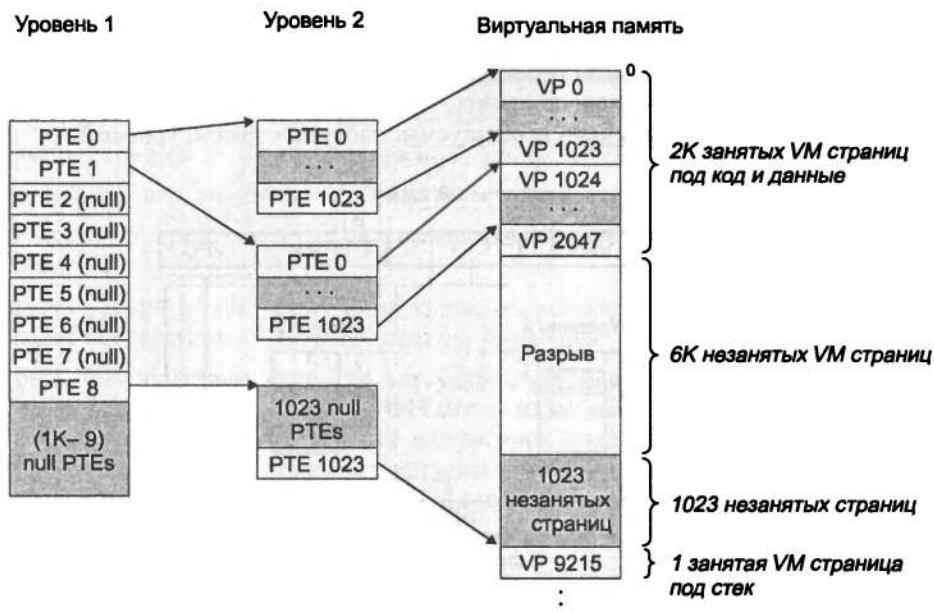


Рис. 10.18. Двухуровневая иерархия таблицы страниц

На каждый элемент PTE таблицы уровня 1 возлагается задача отображения участка памяти виртуального адресного пространства размером 4 Мбайт, при этом каждый

такой участок состоит из 1024 последовательно расположенных страниц. Например, PTE 0 отображает первый участок, PTE 1 — следующий участок и т. д. Если известно, что адресное пространство составляет 4 Гбайт, то достаточно 1024 PTE, чтобы перекрыть все это пространство.

Если ни одна из страниц на участке i не занята, то элемент PTE i уровня 1 пуст. Например, на рис. 10.18 участки 2—7 свободны. Однако если, по крайней мере, хотя бы одна страница в участке i распределена, то PTE i уровня 1 указывает на начало (базу) одной из таблиц страниц уровня 2. Например, на рис. 10.18 все или некоторая часть участков 0, 1 и 8 заняты, поэтому их PTE уровня 1 указывают на базу таблиц страниц уровня 2.

Каждый PTE в таблице страниц уровня 2, как и раньше, когда мы рассматривали одноранговые таблицы страниц, используется для отображения страницы размером 4 Кбайт виртуальной памяти. Обратите внимание на тот факт, что при использовании 4-байтовых элементов PTE каждая таблица страниц уровня 1 и уровня 2 имеет размер 4 Кбайт, который удобен тем, что совпадает с размером страницы.

Эта схема снижает требования к объему памяти по двум причинам. Во-первых, если какой-либо элемент PTE в таблице уровня 1 пуст, то соответствующие таблицы страниц уровня 2 просто не могут существовать. Потенциально это обстоятельство обеспечивает существенную экономию памяти, поскольку большая часть 4-гигабайтного виртуального адресного пространства, выделенного типичной программе, остается невостребованной. Во-вторых, только таблица уровня 1 должна постоянно находиться в оперативной памяти. Таблицы страниц уровня 2 могут быть созданы и использованы системой виртуальной памяти для перекачки содержимого в обоих направлениях, что уменьшает нагрузку на оперативную память. В оперативной памяти должны кэшироваться только наиболее интенсивно используемые таблицы страниц уровня 2.

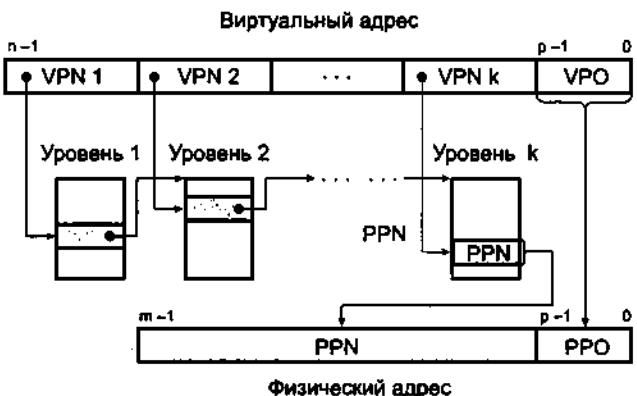


Рис. 10.19. Трансляция адреса с помощью k -уровневой таблицы страниц

Рис. 10.19 подводит итог описанию механизма трансляции адресов с использованием иерархии таблиц страниц до k -го уровня. Виртуальные адреса разбиты на k номеров VPN плюс смещение VPO.

Каждый номер

i , где $1 \leq i \leq k$,

является индексом в таблице страниц уровня i . Каждый вход PTE в таблице уровня

j , где $1 \leq j \leq k - 1$,

указывает на базу некоторой таблицы страниц на уровне $j + 1$. Каждый элемент PTE в таблицы уровня k содержит либо номер PPN некоторой физической страницы, либо адрес дискового блока. Чтобы построить физический адрес, блок MMU должен обратиться к k элементам PTE, прежде чем он сможет определить этот PPN. Как и в случае одноуровневой иерархии, смещение PPO идентично VPO.

На первый взгляд процедура доступа к k элементам PTE может показаться дорогостоящей и непрактичной. Однако здесь на помощь приходит буфер TLB, в котором кэшируются элементы PTE таблиц страниц различных уровней. На практике трансляция адреса с помощью многоуровневых таблиц страниц по быстродействию не намного отличается от использования одноуровневых таблиц страниц.

10.6.4. Непрерывная трансляция адреса

В этом разделе мы применяем все полученные знания на конкретном примере разработки механизма сквозной трансляции адресов в небольшой системе с буфером TLB и d -кэш уровня L1. Для определенности, сделаем следующие предположения:

- память адресуется побайтно;
- из памяти выбираются 1-байтовые слова (но не 4-байтовые);
- используются 14-разрядные виртуальные адреса ($n = 14$);
- физические адреса имеют 12-разрядную длину ($m = 12$);
- размер страницы — 64 байта ($P = 64$);
- буфер TLB представляет собой четырехканальный набор, связанный с 16 входами;
- d -кэш уровня L1 допускает физическую адресацию и непосредственное отображение посредством 4-байтовой строки и 16 наборов.

На рис. 10.20 показаны форматы виртуальных и физических адресов. Поскольку каждая страница имеет размер $2^6 = 64$ байтов, младшие шесть разрядов виртуальных и физических адресов используются для задания смещений VPO и PPO соответственно. Старшие восемь разрядов виртуального адреса отводятся под номер VPN. Старшие шесть битов физического адреса отводятся под номер PPN.

На рис. 10.21 показан моментальный снимок разрабатываемой нами небольшой системы памяти, который включает буфер TLB, часть таблицы страниц и кэш уровня L1. Над изображениями буферов TLB и кэша показано, как распределены разряды виртуальных и физических адресов с помощью аппаратных средств, посредством которых осуществляется доступ к этим устройствам.

- Здесь буфер TLB допускает виртуальную адресацию с использованием разрядов номера VPN. Поскольку TLB имеет четыре набора, два младших разряда VPN ис-

пользуются как индекс набора TLB1. Остальные шесть старших разрядов используются как тег TLB_T, их отличают друг от друга различные номера VPN, которые могут отображаться на один и тот же набор в буфере TLB.

- Здесь таблица страниц представлена собой одноуровневую конструкцию с $2^8 = 256$ элементами таблицы страниц (PTE). В то же время нас будут интересовать только первые шестнадцать элементов. Для удобства мы пометили каждый PTE с помощью номера VPN, который его индексирует; тем не менее, имейте в виду, что эти VPN не являются частью таблицы страниц и не хранятся в памяти. Обратите также внимание на то, что номер PPN каждого отсутствующего элемента PTE отмечен чертой, чтобы подчеркнуть тот факт, что содержимое отдельных разрядов не имеет никакого значения.
- Адрес кэша прямого отображения содержится в полях физического адреса. Поскольку каждый блок состоит из 4 байтов, младшие 2 разряда физического адреса используются как смещение блока CO. Поскольку всего имеется 16 наборов, следующие 4 разряда используются как индекс набора CI. Остальные 6 разрядов используются как тег кэша CT.



Рис. 10.20. Механизм адресации небольшой системы памяти

После того, как мы оговорили начальные условия, посмотрим, что произойдет, когда центральный процессор исполняет команду загрузки, читающую байт по адресу 0x03d4. Напоминаем, что наш гипотетический центральный процессор читает однобайтовые, но не четырехбайтовые слова. Приступая к подобного рода ручному моделированию, мы находим, что полезно будет записывать биты в виртуальном адресе, определить поля, которые далее нам понадобятся, и вычислить их шестнадцатеричные значения. Аппаратные средства решают аналогичную задачу, выполняя декодирование адреса (рис. 10.22).

Сначала блок MMU извлекает из виртуального адреса 0x0F номер VPN и сверяет его с соответствующим значением в буфере TLB, чтобы проверить, является ли это кшированной копией элемента PTE 0x0F некоторого предыдущего обращения в память. TLB извлекает из номера VPN индекс TLB 0x03 и тег TLB 0x3, обнаруживает подходящее соответствие во втором элементе набора 0x3, и возвращает кшированный номер PPN ох0D в блок MMU.

Если бы буфер TLB отсутствовал, то блок MMU должен был бы выбрать элемент PTE из оперативной памяти. Однако в этом случае нам повезло, и TLB был найден

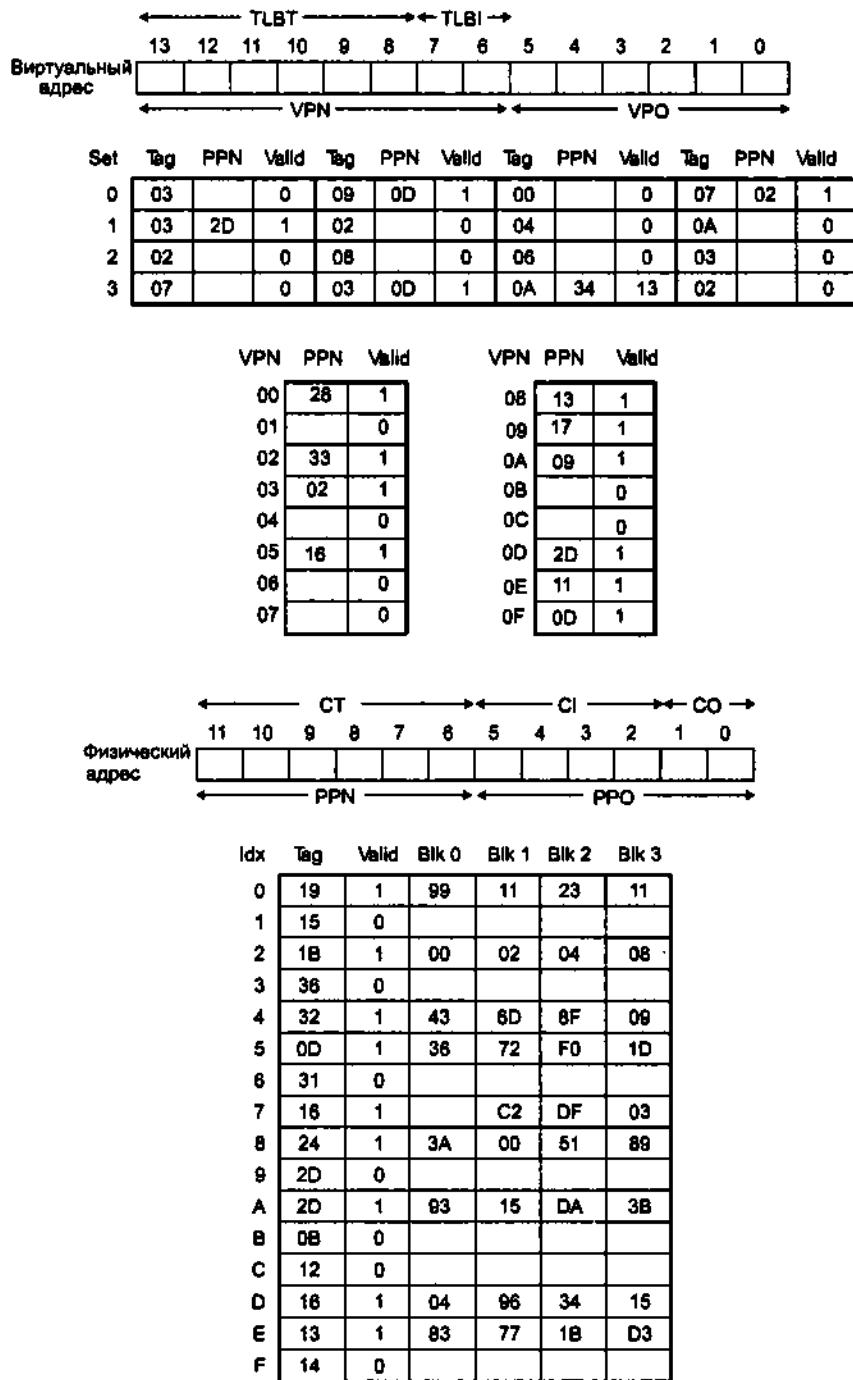


Рис. 10.21. Буфер TLB, таблица страниц и кэш для небольшой системы памяти

Позиция бита	TLBT						TLBI							
	0x03						0x03							
VA = 0x03d4	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	0	0	0	1	1	1	1	0	1	0	1	0	0
	VPN						VPO						0x14	
	0x0f						0x14							

Рис. 10.22. Индекс TLB

там, где и должен быть. Теперь блок MMU имеет все необходимое, чтобы сформировать физический адрес. Осуществляя сцепление номера PPN 0x0f из PTE со смещением VPO 0x14 из виртуального адреса, он получает физический адрес 0x354.

Далее блок MMU посыпает в кэш физический адрес, из которого извлекается смещение в кэш CO (0x0), индекс кэш CI (0x5) и тег кэш CT (0x0d) (рис. 10.23).

Позиция бита	CT						CI				CO	
	0x0d						0x05				0x0	
PA = 0x354	11	10	9	8	7	6	5	4	3	2	1	0
	0	0	1	1	0	1	0	1	0	1	0	0
	PPN						PPO					
	0x0f						0x14					

Рис. 10.23. Индекс кэш

Поскольку тег в наборе 0x5 соответствует CT, кэш обнаруживает наличие, читает байт данных 0x36 со смещением CO и возвращает его в блок MMU, который после этого передает его назад в центральный процессор.

Возможны также и другие варианты процесса трансляции адресов. Например, если буфер TLB отсутствует, то блок MMU должен выбрать номер PPN из соответствующего элемента PTE в таблице страниц. Если полученный при этом элемент PTE недопустим, то это означает попытку обращения к отсутствующей странице, и ядро должно считать (с диска) соответствующую страницу и повторно выполнить команду загрузки. Возможен другой случай, когда PTE является допустимым, но необходимый блок памяти отсутствует в кэше.

УПРАЖНЕНИЕ 10.4

Покажите, как система памяти в примере из разд. 10.6.4 транслирует виртуальный адрес в физический адрес и осуществляет доступ к кэшу. Для заданного виртуального адреса укажите, к какому элементу буфера TLB осуществлялся доступ, физический адрес и возвращаемое значение байта кэш. Укажите, имеется ли нужный буфер TLB, происходит ли обращение к отсутствующей странице, и имеет ли место неудачное обращение к кэшу. Если имеет место неудачное обращение, поставьте "—" в графе "возвращен байт кэш". Если имеет место обращение к отсутствующей странице, поставьте "—" в графе "PPN" и не заполняйте части 3 и 4.

Виртуальный адрес: 0x03d7:

- формат виртуального адреса;
- трансляция адреса;

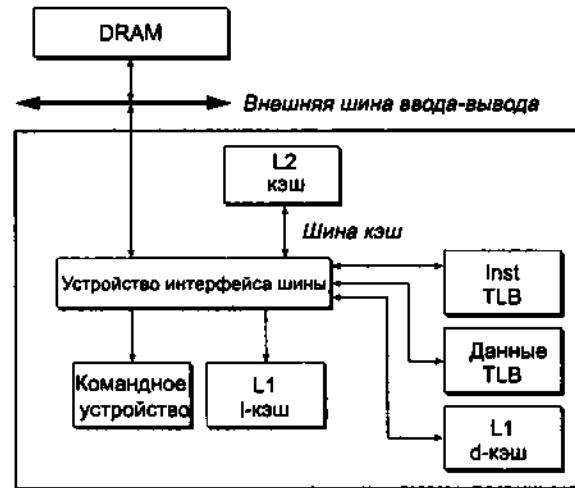
Параметр	Значение
VPN	
Индекс TLB	
Тег TLB	
TLB присутствует? (Да/Нет)	
Обращение к отсутствующей странице? (Да/Нет)	
PPN	

- формат физического адреса;
- обращение к физической памяти.

Параметр	Значение
Смещение байта	
Индекс кэш	
Тег кэш	
Удачное обращение в кэш? (Да/Нет)	
Возвращаемый байт кэш	

10.7. Система памяти Pentium/Linux

Мы завершаем наше обсуждение кэш и виртуальной памяти учебным изучением примера реальной системы: системы класса Pentium, работающей под управлением операционной системы Linux. На рис. 10.24 представлены характеристики системы памяти Pentium. Pentium имеет 32-разрядное адресное пространство (на 4 Гбайт). В комплект процессора (processor package) входит плата центрального процессора, унифицированный кэш уровня L2 и соединяющая их шина кэш-памяти. Сама плата центрального процессора содержит четыре различных вида кэш: TLB-буфер команд, TLB-буфер данных, i-кэш уровня L1 и d-кэш уровня L1. Буферы TLB допускают виртуальную адресацию. Кэш уровня L1 и L2 допускает физическую адресацию. Все кэш в Pentium (включая и буфер TLB) представляют собой четырехканальные связные наборы.



Комплект процессора

Рис. 10.24. Система организации памяти Pentium

Буфер TLB кэшируют 32-разрядные элементы таблицы страниц. В буфере TLB команд кэшируются PTE для виртуальных адресов, сгенерированных блоком выборки команд. В TLB данных кэшируются элементы PTE виртуальных адресов данных. Буфер TLB команд содержит 32 элемента. Буфер TLB данных имеют 64 элемента. Размер страницы может быть задан во время запуска, он составляет 4 Кбайт, либо 4 Мбайт. Операционная система Linux, исполняющаяся на платформе Pentium, использует страницы размером 4 Кбайт.

Кэш уровня L1 и L2 имеет 32-байтовые блоки. Кэш уровня L1 имеет размер 16 Кбайт и 128 наборов, каждый из которых содержит четыре строки. Размер кэша уровня L2 может изменяться от минимального значения 128 Кбайт до максимального значения 2 Мбайт. Обычно используется размер 512 Кбайт.

10.7.1. Трансляция адресов системой Pentium

В этом разделе рассматривается процесс трансляции адресов в системе Pentium. Для справки, на рис. 10.25 представлена итоговая картина полного процесса, начиная с момента, когда центральный процессор генерирует виртуальный адрес, и до момента, когда слово данных будет получено из памяти.

Оптимизация трансляции адресов

При обсуждении устройства механизма трансляции адресов мы описали последовательный двухэтапный процесс, где блок MMU транслирует виртуальный адрес в физический адрес и затем передаст физический адрес в кэш уровня L1. Однако реальные аппаратные реализации используют искусственный прием, который позволяет частично перекрывать по времени эти этапы, ускоряя, таким образом, доступ к кэшу уровня L1.

- 32-битное адресное пространство**
- 4 Кбайт размер страниц**
- L1, L2 и TLB**
 - 4-канальный связанный набор
- Inst TLB**
 - 32 входов, 8 наборов
- Данные TLB**
 - 64 входов, 16 наборов
- L1 I-кэш и d-кэш**
 - 16 Кбайт, 128 наборов
 - 32 байт размер блока
- L2 кэш**
 - унифицированный
 - 128 Кбайт — 2 Мбайт
 - 32 байт размер блока

Например, виртуальный адрес на платформе Pentium при использовании страницы размером 4 Кбайт имеет 12-битовое смещение VPO, и эти биты идентичны 12 битам смещения PPO в соответствующем физическом адресе. Поскольку четырехканальные блок-ассоциативные физически адресуемые кэши уровня L1 имеют 128 наборов и 32-байтовые блоки, каждый физический адрес имеет пять ($\log_2 32$) разрядов смещения в кэш и семь ($\log_2 128$) разрядов индекса. Эти 12 разрядов точно соответствуют разрядам смещения VPO виртуального адреса, и это не случайное совпадение! Когда центральному процессору потребуется оттранслированный виртуальный адрес, он посыпает номер VPN в блок MMU, и смещение VPO — в кэш уровня L1. В то время, как блок MMU запрашивает из соответствующего буфера TLB элемент таблицы страниц, кэш L1 занят тем, что, используя разряды VPO, ищет соответствующий набор и считывает из него четыре тега и соответствующие слова данных. Когда блок MMU снова получает номер PPN из буфера TLB, кэш уже готов выполнить попытку сравнения номера PPN с одним из этих четырех указанных выше тегов.

Предлагаем вам поразмыслить над следующим вопросом: какие варианты могут предложить специалисты компании Intel, если они захотят увеличить размер кэша уровня L1 в будущих системах и по-прежнему использовать этот прием?

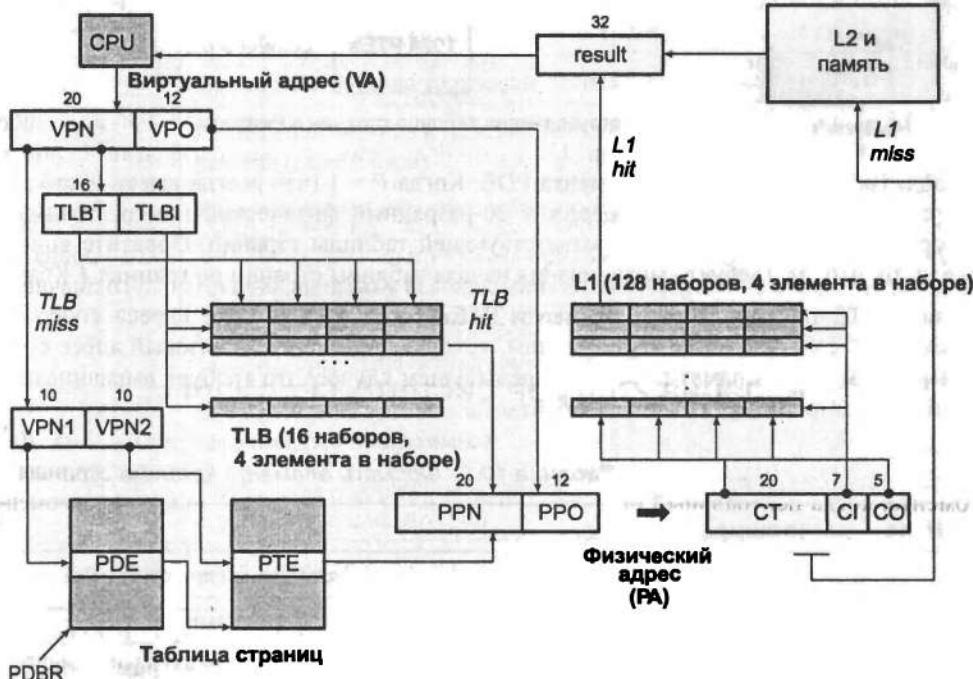


Рис. 10.25. Итоговая картина трансляции адресов в системе Pentium

Таблицы страниц в Pentium

Каждая система Pentium использует двухуровневую таблицу страниц, показанную на рис. 10.26. Таблица уровня 1, известная как *каталог страниц* (page directory), содержит 1024 32-разрядных элемента каталога страниц (PDE, Page Directory Entry), каждый из которых указывает на одну из 1024 таблиц страниц уровня 2. Каждая таблица страниц содержит 1024 32-разрядных элемента *таблицы страниц* (PTE, Page Table Entry), каждый из которых указывает на страницу в физической памяти или на диске.

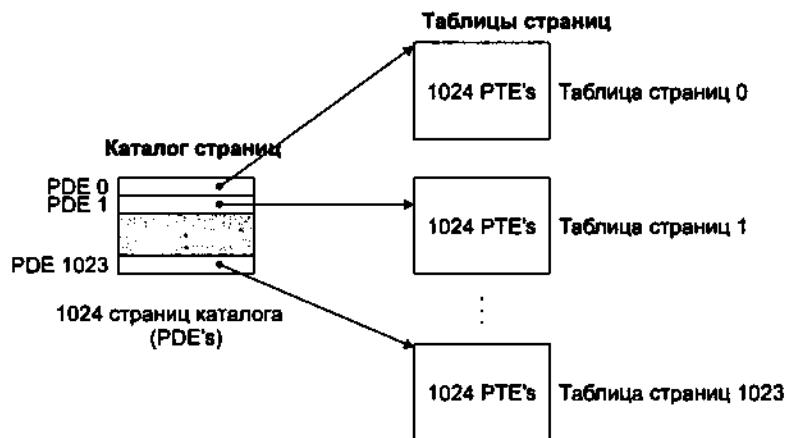


Рис. 10.26. Многоуровневая таблица страниц в Pentium

В табл. 10.2 показан формат элемента PDE. Когда $P = 1$ (что всегда имеет место для системы Linux), поле адреса содержит 20-разрядный физический номер страницы, который указывает на начало соответствующей таблицы страниц. Обратите внимание на то, что это требует выравнивания начала таблицы страниц по границе 4 Кбайт.

В табл. 10.3 показан формат элемента PTE. Когда $P = 1$, поле адреса содержит 20-битовый физический номер страницы, который указывает на базовый адрес страницы в физической памяти. Как и в предыдущем случае, это требует выравнивания физических страниц по границе 4 Кбайт.

Таблица 10.2. Форматы элемента каталога страницы

Поле	Описание
P	Таблица страниц находится (1) или отсутствует (0) в физической памяти
R/W	Разрешение доступа только для чтения или для чтения/записи
U/S	Разрешение доступа в пользовательском или привилегированном режиме (в режиме ядра)
WT	Режим сквозной или отложенной записи в кэш для данной таблицы страниц

Таблица 10.2 (окончание)

Поле	Описание
CD	Кэш заблокирован (1) или задействован (0)
A	Было ли обращение к странице? (Устанавливается блоком MMU при чтении и записи, сбрасывается программно)
PS	Размер страницы 4 Кбайт (0), или 4 Мбайт (1)
G	Глобальная страница (не удаляется из буфера TLB при переключении с одной задачи на другую)
Адрес базы РТ	20 старших разрядов физического адреса таблицы страниц

Таблица 10.3. Форматы элемента таблицы входов

Поле	Описание
P	Таблица страниц находится (1) или отсутствует (0) в физической памяти
R/W	Разрешение доступа только для чтения или для чтения и записи
U/S	Разрешение доступа в пользовательском или привилегированном режиме (режим ядра)
WT	Режим сквозной или отложенной записи в кэш для данной таблицы страниц
CD	Кэш заблокирован (1) или задействован (0)
A	Признак обращения (устанавливается блоком MMU при чтении и записи, сбрасывается программно)
D	Бит изменения (устанавливается блоком MMU при чтении, сбрасывается программно)
G	Глобальная страница (не удаляется из буфера TLB при переключении с одной задачи на другую)
Адрес базы страницы	20 старших разрядов физического адреса страницы

Элемент PTE имеет два разряда разрешения, которые управляют доступом к странице. Разряд R/W определяет, предназначено ли содержимое страницы для чтения и записи или только для чтения. Бит U/S, который определяет, можно ли к странице обратиться в непrivилегированном режиме, защищает программные коды и данные в ядре операционной системы от пользовательских программ.

Поскольку блок MMU преобразует каждый виртуальный адрес, он также модифицирует два других разряда, которые могут использоваться обработчиком ошибки обращения к отсутствующей странице, включенным в состав ядра. Каждый раз, когда происходит обращение к странице, блок MMU устанавливает разряд А, который известен как разряд обращения к странице памяти (*reference bit*). Ядро может использовать разряд обращения при реализации своего алгоритма замены страниц. Блок MMU устанавливает разряд D или разряд изменения (*dirty bit*) каждый раз, когда в страницу производится запись. Иногда измененную страницу называют недействительной страницей (*dirty page*). Разряд изменения сообщает ядру, должно ли оно записать страницу, подлежащую удалению, на диске, прежде чем оно скопирует на ее место страницу замены. Чтобы удалить ссылки или разряды изменения, ядро может вызвать специальную команду привилегированного режима.

Разрешение исполнения и атаки посредством переполнения буфера

Обратите внимание на то обстоятельство, что элемент таблицы страниц в системе Pentium не содержит разряда разрешения, управляющего исполнением содержимого страницы. Атаки с целью переполнения буфера используют это упущение, загружая и выполняя программный код непосредственно на базе пользовательского стека (см. разд. 3.13). Если бы существовал такй разряд разрешения, ядро могло бы устранить угрозу таких атак путем ограничения привилегий при доступе к сегменту программного кода, разрешая доступ к сегментам, предназначенным только для чтения.

Трансляция таблицы страниц в системе Pentium

На рис. 10.27 показано, как в системе Pentium блок MMU использует двухуровневую таблицу страниц для преобразования виртуального адреса в физический адрес. 20-разрядный номер VPN делится на два участка по 10 битов каждый. Номер VPN 1 индексирует элемент PDE в каталоге страниц, на который указывает регистр PDBR. Адрес в PDE указывает на базу некоторой таблицы страниц, которую индексирует номер VPN 2.

Номер PPN в элементе PTE, индексированный номером VPN 2, посредством конкатенации со смещением VPO образует физический адрес.

Преобразование TLB в системе Pentium

На рис. 10.28 схематически представлен процесс преобразования буфера TLB в системе Pentium. Если элемент PTE кэширован в наборе, индексируемом TLBI (TLB непосредственно доступен), то чтобы сформировать физический адрес, из этого кэшированного PTE извлекается PPN, и образуется его конкатенация со смещением VPO. Если PTE не кэширован, а PDE — кэширован (так называемая частичная доступность TLB), то блок MMU должен выбрать соответствующий элемент PTE из памяти, прежде чем он сможет сформировать физический адрес. Наконец, если ни PDE, ни PTE не кэшированы (буфер TLB не доступен), чтобы сформировать физический адрес, блок MMU должен выбрать из памяти и PDE, и PTE.

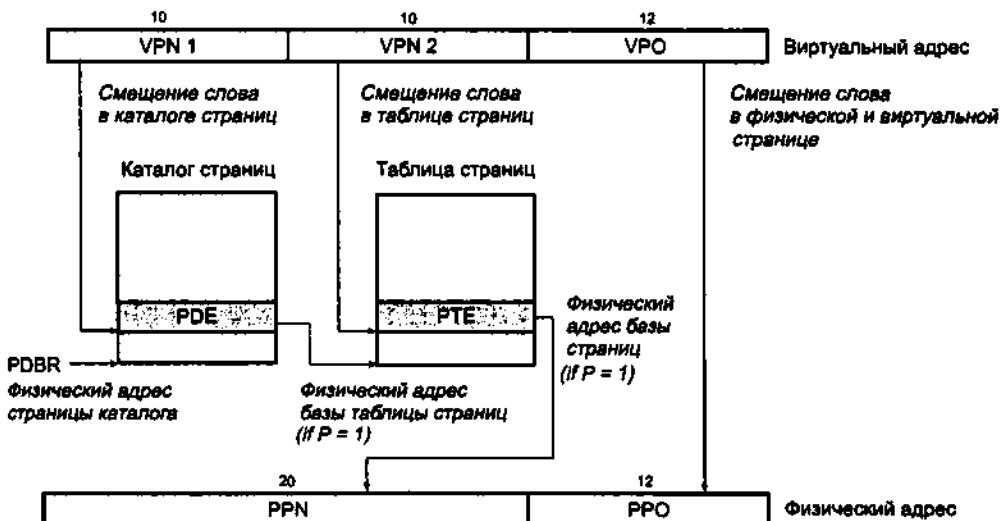


Рис. 10.27. Трансляция таблицы страниц в Pentium

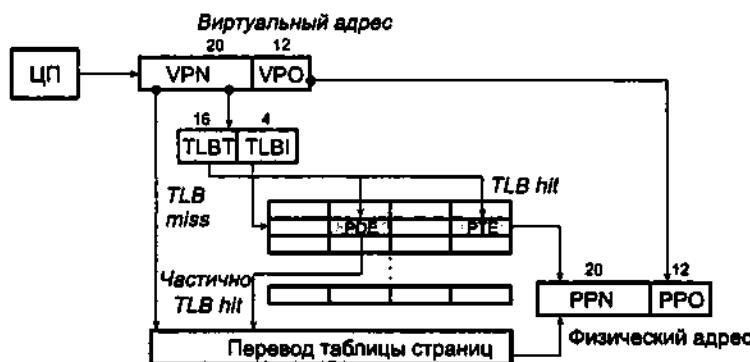


Рис. 10.28. Преобразование буфера TLB в системе Pentium

10.7.2. Система виртуальной памяти Linux

Система виртуальной памяти требует тесного взаимодействия между аппаратной частью и программными средствами ядра системы. Детальное описание этого взаимодействия не является предметом нашего рассмотрения. В этом разделе наша задача состоит в том, чтобы дать достаточное представление о системе виртуальной памяти операционной системы Linux и на примере реально показать, как организована виртуальная память и как в ней обрабатываются обращения к отсутствующей странице.

Операционная система Linux поддерживает раздельные виртуальные адресные пространства для каждого процесса с помощью схемы распределения, представленной

на рис. 10.29. Нам уже неоднократно приходилось сталкиваться с этой схемой, с ее известными сегментами программных кодов, данных, динамической памяти, совместно используемых библиотек и стеков. Теперь, когда мы приступаем к изучению преобразования адресов, мы должны ознакомиться еще с некоторыми особенностями виртуальной памяти ядра, расположенной выше адреса 0xc0000000.

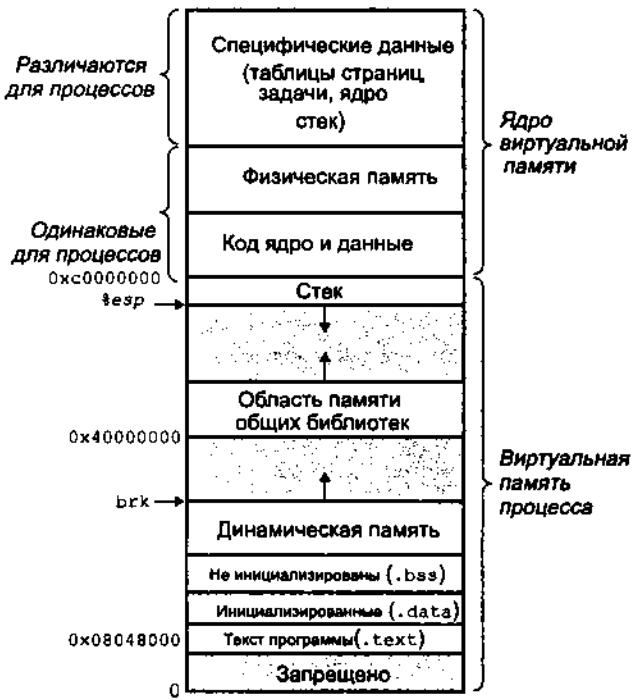


Рис. 10.29. Виртуальная память процесса в Linux

Виртуальная память ядра содержит программные коды и структуры данных ядра. Некоторые области виртуальной памяти ядра отображаются на физические страницы, которые совместно используются всеми процессами. Например, каждый процесс использует программные коды ядра и глобальные структуры данных. Интересно отметить, что система Linux также отображает множество последовательно расположенных виртуальных страниц (суммарный объем памяти которых равен размеру DRAM системы) на соответствующее множество последовательно расположенных физических страниц. Это обеспечивает ядру удобный способ доступа к любому заданному месту в физической памяти, например, когда оно должно выполнить отображаемые в память операции ввода-вывода на устройствах, которые отображаются на конкретные ячейки физической памяти.

Другие области виртуальной памяти ядра содержат данные, специфичные для каждого процесса. В качестве примера можно привести таблицы страниц, стек, используемый ядром, когда оно исполняет программный код в контексте процесса, а также

различные структуры данных, которые отслеживают текущую организацию виртуального адресного пространства.

Области виртуальной памяти в Linux

В системе Linux виртуальная память организована как совокупность областей (называемых также сегментами) памяти. Область — это последовательно расположенный участок существующей (выделенной) виртуальной памяти, страницы которой связаны между собой тем или иным способом. Например, сегмент программного кода, сегмент данных, динамическая память, сегмент совместно используемой библиотеки и пользовательский стек — все это различные виды областей. Каждая существующая виртуальная страница содержится в некоторой области, любая виртуальная страница, которая не содержится в той или иной области памяти, вообще не существует, а процессы не могут на нее ссылаться. Понятие области памяти важно уже потому, что оно допускает наличие промежутков в виртуальном адресном пространстве. Ядро не отслеживает виртуальные страницы, которые не существуют, и такие страницы требуют дополнительных затрат ресурсов в памяти, на диске или в самом ядре.

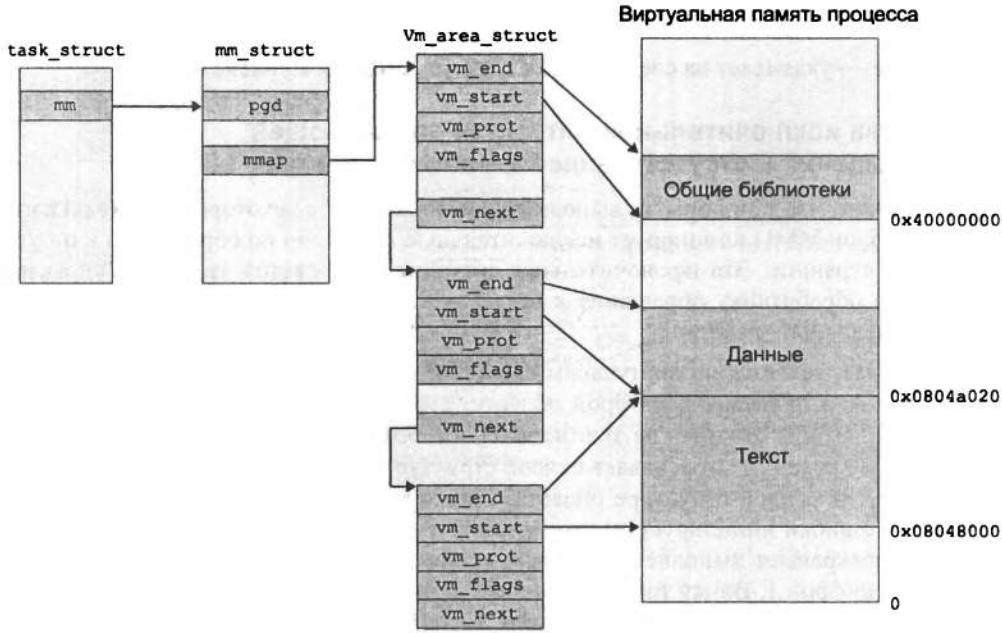


Рис. 10.30. Как организована виртуальная память в операционной системе Linux

На рис. 10.30 выделены структуры данных ядра, которые отслеживают области виртуальной памяти конкретного процесса. Ядро поддерживает специальную структуру задач (`task_struct` в исходном программном коде) для каждого процесса, выполняющегося в системе. Элементы этой структуры задач либо содержат всю информацию, которая необходима ему для управления исполнением конкретного процесса

(например, идентификатор процесса, указатель на пользовательский стек, имя используемого файла и счетчика команд), либо указатель на нее.

Один из входов в структуре задачи (`task_struct`) указывает на структуру `mm_struct`, которая характеризует текущее состояние виртуальной памяти. Для нас интерес представляют два поля: `pgd`, указывающее базовый адрес каталога страниц, и `ptmpar`, указывающее на список `vm_area_structs` структур областей, каждая из которых характеризует область текущего виртуального адресного пространства. Когда ядро исполняет этот процесс, оно хранит `pgd` в регистре команд `PDBR`.

Для наших целей достаточно предположить, что структура для каждой отдельной области содержит следующие поля:

- `vm_start` — указывает на начало области;
- `vm_end` — указывает на конец области;
- `vm_prot` — разрешает или запрещает чтение/запись для всех страниц, содержащихся в данной области;
- `vm_flags` — определяет (среди прочего), используются страницы данной области совместно с другими процессами или предназначены только для данного конкретного процесса;
- `vm_next` — указывает на следующую структуру области в списке.

Обработка исключительной ситуации, возникающей при обращении к отсутствующей странице в системе Linux

Предположим, что при попытке выполнить преобразование некоторого виртуального адреса А, блок MMU инициирует исключительную ситуацию по обращению к отсутствующей странице. Эта исключительная ситуация завершается передачей управления в ядро обработчику обращения к отсутствующей странице, он после этого выполняет следующие действия:

- Проверяет, является ли виртуальный адрес А допустимым. Другими словами, лежит ли А в пределах некоторой области, определяемой какой-либо структурой области? Чтобы ответить на этот вопрос, обработчик ошибки обращения к отсутствующей странице отыскивает список структур области, сравнивая А с `vm_start` и `vm_end` в каждой структуре области. Если данная команда недопустима, обработчик ошибки инициирует исключительную ситуацию ошибки сегментации, которая прекращает выполнение данного процесса. На рис. 10.31 эта ситуация помечена цифрой 1. Ввиду того, что процесс может генерировать произвольное число новых областей виртуальной памяти (используя с этой целью функцию `ptmpar`, которая будет описана в следующем разделе), последовательный перебор списка структур области может оказаться весьма дорогостоящим. Так на практике система Linux осуществляет наложение дерева на список, используя с этой целью некоторые поля, которые мы здесь не показали, и выполняет поиск на этом дереве.
- Проверяет, допустима ли осуществляемая попытка доступа к памяти. Другими словами, имеет ли право данный процесс выполнять операции чтения или записи в страницы этой области? Например, действительно ли ошибка обращения к от-

существующей странице возникла в результате выполнения команды запоминания, попытавшейся осуществить запись в страницу только-для-чтения в сегменте программных кодов? Является ли на самом деле ошибка обращения к отсутствующей странице результатом выполнения пользователем процесса в непrivилегированном режиме, который предпринял попытку читать слово в виртуальной памяти ядра? Если попытка доступа не является допустимой, то обработчик ошибки инициирует исключительную ситуацию, которая с целью защиты прекращает выполнение процесса. На рис. 10.31 эта ситуация отмечена цифрой 2.

- На этой стадии ядро знает, что обращение к отсутствующей странице последовало в результате выполнения допустимой операции над допустимым виртуальным адресом. Оно обрабатывает сбой, выбирая страницу, подлежащую удалению, выгружает ее, если она подвергалась изменениям, и загружает новую страницу, обновляя по ходу дела таблицу страниц. Когда обработчик обращения к отсутствующей странице возвращает управление, центральный процессор повторно начинает выполнение команды, вызвавшей сбой, которая, как и раньше, повторно отсылает А в блок MMU. На сей раз MMU нормально выполняет преобразование адреса А без возбуждения ошибки обращения к отсутствующей странице.

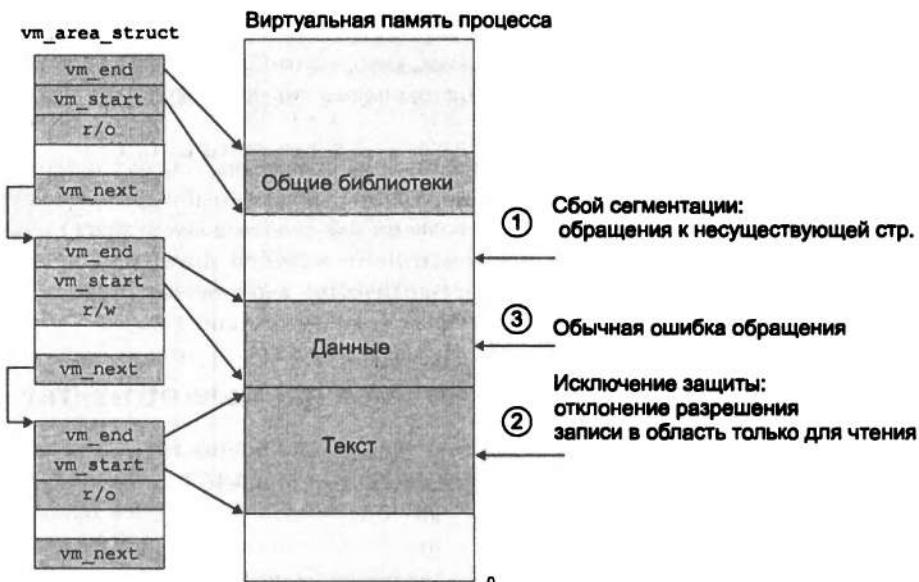


Рис. 10.31. Как решается проблема обращения к отсутствующей странице в системе Linux

10.8. Отображение в памяти

Система Linux (подобно другим формам операционной системы Unix) инициализирует содержимое области виртуальной памяти, связывая его с объектом (object) на диске. Этот процесс называется отображением в памяти (memory mapping).

Области могут быть отображены на один из двух типов объектов:

1. Обычный файл из файловой системы Unix — область может быть отображена на непрерывный участок обычного файла на диске, такого, например, как исполняемый объектный файл. Раздел файла разбит на участки размером в одну страницу, где каждый такой участок содержит начальное содержимое виртуальной страницы. В режиме замещения страниц по требованию ни одна из этих виртуальных страниц фактически не перекачивается в физическую память до тех пор, пока центральный процессор сначала не затронет (*touch*) эту страницу (т. е., пока не выдаст виртуальный адрес, который попадет в пределы области адресного пространства, занимаемого этой страницей). Если область больше, чем участок файла, то неиспользованная ее часть заполняется нулями.
2. Анонимный файл (*anonymous file*) — область может также быть отображена на анонимный файл, образованный ядром, который содержит только двоичные нули. Когда центральный процессор первый раз затрагивает виртуальную страницу в такой области, ядро находит в физической памяти подходящую страницу для удаления, выгружает страницу, подлежащую удалению, если она подвергалась изменениям, заполняет эту страницу двоичными нулями и вносит в таблицу страниц корректирующие, относящие эту страницу к числу резидентных. Следует отметить, что на самом деле никакого обмена данными между диском и памятью не происходит. По этой причине страницы в областях, которые отображаются на анонимные файлы, иногда называются нулевыми страницами по требованию (*demand-zero pages*).

В любом случае, если виртуальная страница инициализирована, она будет перекачиваться в специальный файл подкачки (*swap file*), поддерживаемый ядром, и обратно. Файл подкачки иначе называется *пространством для свопинга* (*swap space*) или *областью подкачки* (*swap area*). Важно понимать, что в любой момент времени пространство для свопинга ограничивает общее количество виртуальных страниц, которые могут распределяться процессами, исполняемыми на текущий момент.

10.8.1. Разделяемые повторно посещаемые объекты

Сама идея отображения в памяти возникла из четкого понимания того, что если система виртуальной памяти может быть интегрирована в стандартную файловую систему, то она сможет обеспечить простой и эффективный способ загрузки программ и данных в память.

Как мы уже видели, понятие процесса предполагает предоставление каждому процессу своего собственного закрытого виртуального адресного пространства, которое защищено от произвольных записей или чтений другими процессами. В то же время некоторые процессы могут иметь одни и те же текстовые области памяти с доступом только для чтения. Например, каждый процесс, который исполняет программу `tcslib` командного процессора операционной системы Unix, имеет одну и ту же текстовую область памяти. Более того, многие программы требуют доступа к одним и тем же копиям библиотечных программ в режиме только для чтения. Например, каждая программа на языке C требует стандартных функций С-библиотеки, например, `printf`.

Было бы чрезвычайно расточительно держать для каждого процесса дубликаты этих широко используемых программ в физической памяти. К счастью, отображение в памяти предоставляет нам тщательно продуманный механизм для управления объектами сразу несколькими процессами.

Любой объект может быть отображен на область виртуальной памяти либо как *совместно используемый* (разделяемый) (*shared*), либо как *закрытый* (*private*) объект. Если процесс отображает совместно используемый объект на некоторую область своего виртуального адресного пространства, то любая запись, производимая процессом в эту область, видна всем другим процессам, которые также отображают данный совместно используемый объект в свою виртуальную память. Более того, все изменения отражаются также в исходном объекте на диске.

С другой стороны, изменения, произведенные в области, отображаемой на закрытый объект, не видны другим процессам, и все записи, произведенные процессом в данную область, не отображаются в объект на диске. Область виртуальной памяти, в которую отображается совместно используемый объект, часто называют совместно используемой (разделяемой) областью (*shared area*). То же самое можно сказать и о закрытой области (*private area*).

Предположим, что процесс 1 отображает совместно используемый объект в область своей виртуальной памяти, как показано на рис. 10.32. Предположим также, что процесс 2 отображает тот же самый совместно используемый объект в свое адресное пространство (не обязательно с тем же самым виртуальным адресом, что у процесса 1).

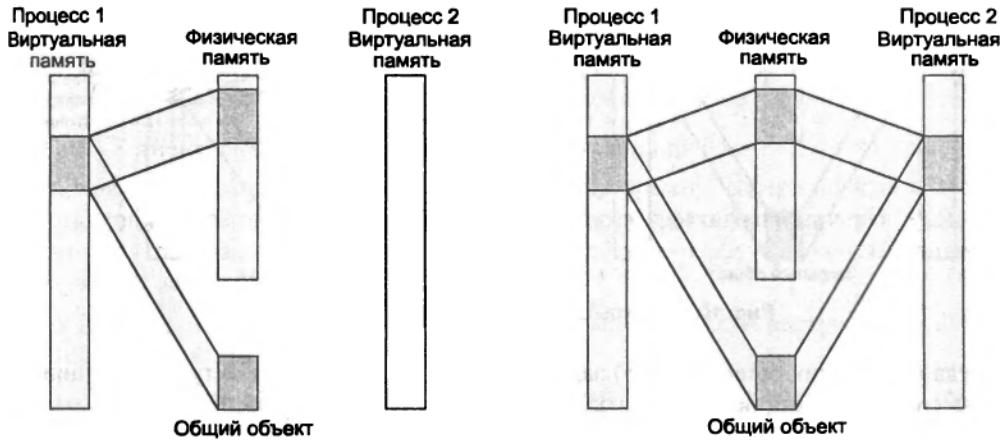


Рис. 10.32. Совместно используемый объект

Поскольку каждый объект имеет уникальное имя файла, ядро может быстро определить, что процесс 1 уже отобразил этот объект и может установить указатели в элементах таблиц страниц в процессе 2 на соответствующие физические страницы. Важно здесь то, что в физической памяти должна быть сохранена только одна-единственная копия совместно используемого объекта, даже если этот объект отображен

в несколько различных совместно используемых областей. Для удобства на рисунке физические страницы изображены смежными, но в общем случае это совсем не обязательно.

Закрытые объекты отображаются в виртуальную память с использованием тщательно продуманной методики, которая называется *копированием при записи* (copy-on-write). Закрытый объект начинает свой жизненный путь точно таким же образом, как и совместно используемый объект, имея только одну-единственную копию, хранящуюся в физической памяти. Например, на рис. 10.33 показана ситуация, где два процесса отобразили закрытый объект в различные области своей виртуальной памяти, но в то же время совместно используют одну и ту же физическую копию этого объекта. Для каждого процесса, который отображает закрытый объект, элементы таблицы страниц для соответствующей закрытой области помечены как обеспечивающие доступ только для чтения, а структура этой области помечена как допускающая закрытое копирование при записи (private copy-on-write). До тех пор, пока никакой процесс не пытается сделать запись в свою закрытую область, все эти процессы продолжают совместно использовать единственную копию данного объекта в физической памяти. Однако, как только тот или иной процесс попытается сделать запись в какую-либо страницу в закрытой области, операция записи возбуждает состояние ошибки нарушения защиты.

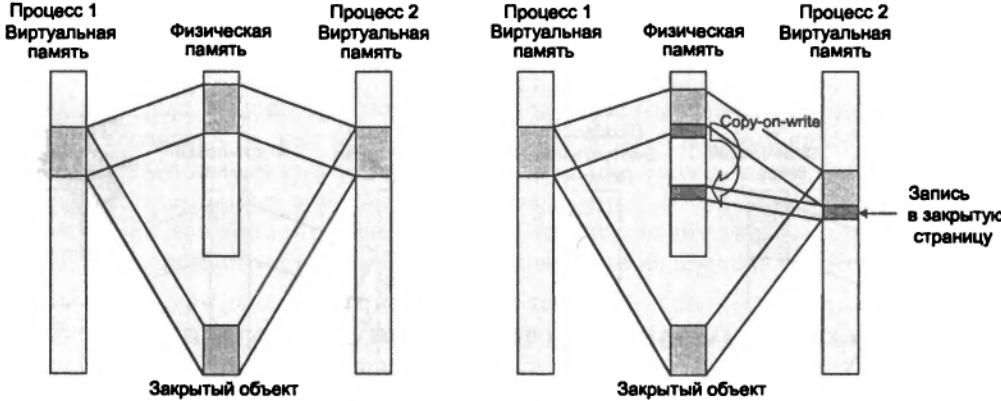


Рис. 10.33. Закрытый объект, копируемый при записи

Когда обработчик ошибки (сбоя) замечает, что исключительная ситуация инициирована системой защиты при попытке записи в страницу закрытой области, он создает новую копию страницы в физической памяти, корректирует соответствующий элемент таблицы страниц с тем, чтобы тот указывал на новую копию, и затем устанавливает разрешение записи в эту страницу, как показано на рис. 10.33. Когда обработчик ошибки возвратит управление, центральный процессор повторно производит запись, которая выполняется без сбоев на вновь созданной странице. Откладывая копирование страниц в закрытых объектах до самого последнего момента, копирование при записи существенно повышает эффективность использования дефицитной физической памяти.

10.8.2. Еще раз о функции fork

Теперь, когда мы получили представление о виртуальной памяти и об отображении в памяти, мы можем ознакомиться с тем, как функция `fork` образует новый процесс, обладающий собственным независимым виртуальным адресным пространством.

Когда функция `fork` вызывается текущим процессом (current process), ядро образует различные структуры данных для нового процесса (new process) и назначает ему уникальный PID (Process Identifier, идентификатор процесса). Чтобы образовывать виртуальную память для нового процесса, ядро создает точные копии структуры `mm_struct` текущего процесса, структур области и таблиц страниц. Оно также помечает каждую страницу обоих процессов как получившие разрешение выполнять доступы только для чтения, и помечает каждую структуру области обоих процессов как закрытую, с копированием при записи.

Когда функция `fork` возвращает управление новому процессу, последний уже имеет точную копию виртуальной памяти, какой она была на момент обращения к функции `fork`. Когда какой-либо из процессов выполняет ту или иную последовательность операций записи, механизм копирования при записи создает новую страницу, сохраняя, таким образом, актуальность идеи закрытого адресного пространства для каждого процесса.

10.8.3. Еще раз о функции execve

Механизмы виртуальной памяти и отображения в памяти играют ключевую роль в процессе загрузки программ в память. Теперь, когда мы подробно ознакомились с этими понятиями, мы вполне можем разобраться с тем, как функция `execve` на самом деле загружает и исполняет программы. Предположим, что программа, исполняемая в текущем процессе, производит следующее обращение к функции `execve`:

```
execve("a.out", argv, environ);
```

Функция `execve` загружает и запускает на счет программы, содержащуюся в исполняемом объектном файле `a.out` текущего процесса, фактически заменяя текущую программу программой `a.out`. Загрузка и исполнение `a.out` требуют выполнения следующих действий:

- Удалить существующие области в пользовательской части виртуального адреса текущего процесса.
- Отобразить закрытые области. Создайте новые структуры областей для текста, данных, неинициализированных данных и стековые области новой программы. Все эти новые области суть закрытые области, копируемые при записи. Области текста и данных отображаются на разделы текста и данных файла `a.out`. Область неинициализированных данных заполняется нулями и отображается на анонимный файл, размер которого содержится в файле `a.out`. Стек и область динамической памяти также суть области, заполняемые нулями, и первоначально имеют нулевую длину. Рис. 10.34 служит иллюстрацией различных отображений закрытых областей.

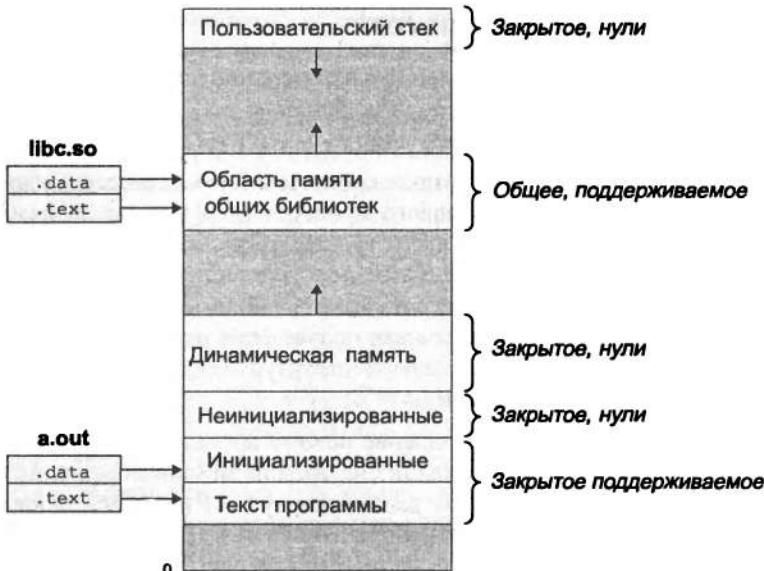


Рис. 10.34. Как загрузчик отображает области пользовательского адресного пространства

- Отобразить совместно используемые области. Если программа `a.out` была скомпонована с совместно используемыми объектами, такими как стандартная библиотека `libc.so` на языке C, то эти объекты динамически связываются в программе и затем отображаются на совместно используемую область виртуального адресного пространства пользователя.
- Установить счетчик команд (PC, Program counter). Последнее, что должна сделать `execve`, — это установить счетчик команд в контексте текущего процесса на точку входа в текстовой области.

В следующий раз, в момент, запланированный для этого процесса, он начнет исполняться с точки входа. По мере необходимости, операционная система Linux будет загружать на диск страницы программного кода и данных.

10.8.4. Отображение в памяти на уровне пользователя с помощью функции `mmap`

Процессы, протекающие в операционной системе Unix, могут использовать функцию `mmap` для образования новых областей виртуальной памяти и отображения объектов в эти области.

```
#include <unistd.h>
#include <sys/mman.h>

void *mmap(void *start, size_t length, int prot,
int flags, int fd, off_t offset);
```

Функция `mmap` обращается с запросом к ядру с требованием создания новой области виртуальной памяти, предпочтительно той, которая начинается с адреса `start`, и отображения непрерывного участка объекта, указанного дескриптором (описателем) файла `fd`, на новую область. Этот непрерывный участок объекта имеет размер `length` байтов и смещение `offset` байтов относительно начала файла. Адрес `start` представляет собой условную точку отсчета, обычно его значение `NULL`. Для наших целей мы всегда будем полагать, что начальный адрес `NULL`. Рис. 10.35 иллюстрирует назначение этих аргументов.

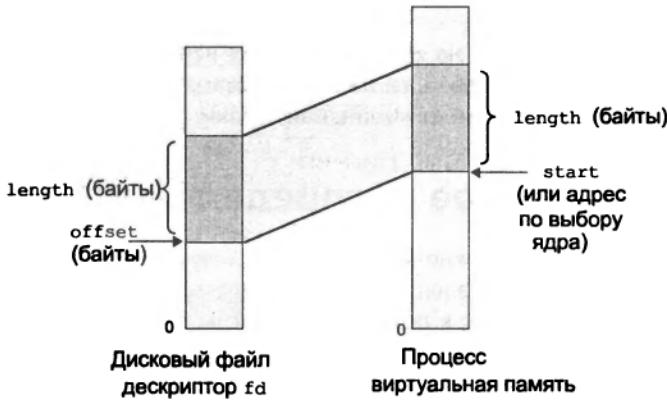


Рис. 10.35. Графическая интерпретация аргументов функции `mmap`

Аргумент `prot` содержит биты, которые описывают разрешение на доступ к вновь отображенной области виртуальной памяти (т. е. разряды `vm_prot` в соответствующей структуре области).

- `PROT_EXEC` — страницы области, состоящие из команд, которые могут быть исполнены центральным процессором;
- `PROT_READ` — страницы области, доступные для чтения;
- `PROT_WRITE` — страницы области, доступные для записи;
- `PROT_NONE` — страницы области, не доступные для обращения.

Аргумент `flags` состоит из разрядов, которые описывают тип отображаемого объекта. Если флаговый разряд `MAP_ANON` установлен и `fd` `NULL`, то страницная память есть анонимный объект, и соответствующие виртуальные страницы заполнены нулями. Разряд `MAP_PRIVATE` означает закрытый объект, копируемый при записи, а разряд `MAP_SHARED` — совместно используемый объект. Например, команда

```
bufp = Mmap(NULL, size, PROT_READ, MAP_PRIVATE|MAP_ANON, 0, 0);
```

есть обращение с запросом к ядру создать новую закрытую область виртуальной памяти с доступом только для чтения, заполненную нулями, содержащую байты `size`. Если запрос успешен, то `bufp` содержит адрес новой области.

Функция `munmap` удаляет заданные области виртуальной памяти:

```
#include <unistd.h>
#include <sys/mman.h>
int munmap(void *start, size_t length);
```

Функция `munmap` удаляет область, начинающуюся с виртуального адреса `start` и включающую следующие `length` байтов. Последующие ссылки на удаленную область завершатся ошибками сегментации.

УПРАЖНЕНИЕ 10.5

Напишите программу `мпарсору` на языке С, которая использовала бы `мпар` для копирования файла произвольного размера из диска в стандартный вывод. Имя входного файла нужно передать как аргумент командной строки.

10.9. Динамическое распределение памяти

Несмотря на то, что вполне можно воспользоваться функциями низкого уровня `мпар` и `тилмпар` для образования и удаления областей виртуальной памяти, большая часть программ на языке С прибегает к помощи программы *динамического распределения памяти* (*dynamic memory allocation*), когда необходимо получить дополнительную виртуальную память во время исполнения.

Программа динамического распределения памяти обслуживает область виртуальной памяти процесса, называемую *динамической памятью* (*heap*) (рис. 10.36). В большинстве Unix-подобных систем это область памяти, заполненная нулями, которая начинается сразу же после неинициализированной области `bss` и возрастает вверх, в направлении возрастания адресов. Для каждого процесса ядро поддерживает переменную `brk` (произносится как "брэйк"), которая указывает на вершину.

Программа распределения памяти поддерживает динамическую память как совокупность блоков различных размеров. Каждый блок представляет собой непрерывный участок виртуальной памяти, который либо распределен (*allocated*), либо свободен (*free*). Распределенный блок явно резервируется для использования приложением. Свободный блок доступен для распределения. Свободный блок остается таковым до тех пор, пока он явно не будет распределен приложением. Распределенный блок остается таковым, пока он не будет освобожден либо явно приложением, либо неявно самой программой распределения памяти.

Программы распределения памяти могут поставляться в двух разновидностях. Обе они требуют, чтобы само приложение явно выделяло блоки памяти. Они отличаются тем, какой логический объект отвечает за освобождение выделенных блоков.

Программы явного распределения памяти (*explicit allocator*) требуют, чтобы приложение явным образом освобождало любой распределенный блок. Например, стандартная библиотека программ на языке С предоставляет программу явного распределения памяти, получившей название `malloc`. Программы на языке С размещают блоки памяти с помощью функции `malloc` и освобождают блоки памяти с помощью

функции `free`. Вызовы функций `new` и `free` в программах на языке C++ дают один и тот же результат.

С другой стороны, программы неявного выделения памяти (*implicit allocator*) требуют, чтобы программа распределения памяти обнаруживала, когда выделенный блок памяти больше не используется приложением, после чего освобождает этот блок. Программы неявного распределения памяти называются также *сборщиками мусора* (*garbage collector*), а сам процесс автоматического освобождения неиспользуемых распределенных блоков памяти называется *сборкой мусора* (*garbage collection*). Например, языки высокого уровня Lisp, ML и Java при освобождении выделенных блоков используют программу сборки мусора.

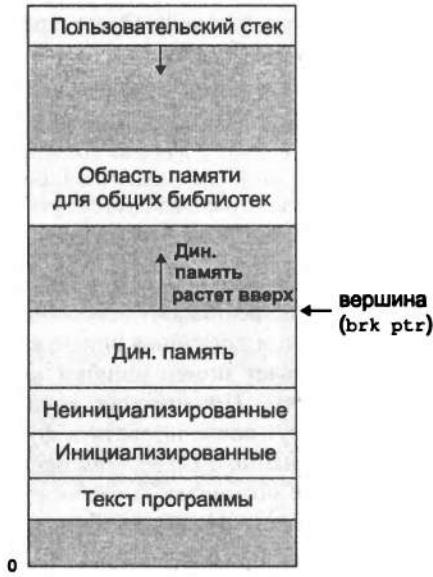


Рис. 10.36. Динамическая память

Оставшаяся часть этого раздела посвящена вопросам разработки и реализации программ явного распределения памяти. В разд. 10.10 мы рассмотрим программы неявного распределения памяти. Для конкретности, мы сосредоточимся на изучении программ распределения памяти, которые манипулируют динамической памятью. При этом следует знать, что распределение памяти — это всего лишь основная идея, которая в различных контекстах интерпретируется по-разному. Например, приложения, которые связаны с интенсивной обработкой таких структур, как графы, часто используют стандартную программу распределения памяти с целью получения крупного блока виртуальной памяти, а затем применяют специально приспособленную для данного приложения программу распределения памяти, управляя с ее помощью памятью в пределах выделенного блока памяти, по мере того, как образуются и уничтожаются те или иные узлы графа.

10.9.1. Функции *malloc* и *free*

Стандартная библиотека программ на языке С предоставляет программу явного распределения памяти, представляющую собой пакет обслуживающих программ *malloc*. Конкретные программы выделяют блоки из динамической памяти, вызывая функцию *malloc*.

```
#include <stdlib.h>
void *malloc(size_t size);
```

Функция *malloc* возвращает указатель на блок памяти размером, по меньшей мере, *size* байтов, соответствующим образом выровненный для любого вида объектов данных, которые могут содержаться в этом блоке. В системах Unix, с которыми мы уже знакомы, *malloc* возвращает блок, выровненный по границе 8 байтов (двойное слово). Тип *size_t* определен как целое без знака.

Каков размер слова?

При обсуждении машины IA32 в главе 3 мы уже говорили о том, что Intel рассматривает 4-байтовые объекты как двойные слова (double-word). Однако далее в этом разделе мы везде будем полагать, что слова (word) суть 4-байтовые объекты, и что двойные слова суть 8-байтовые объекты, что согласуется с общепринятой терминологией.

Если при исполнении функции *malloc* возникают осложнения (например, программа запрашивает блок памяти, больший чем доступная виртуальная память), то эта функция возвращает NULL и устанавливает номер ошибки *errno*. Функция *malloc* не инициализирует возвращаемую память. Приложения, которые намерены инициализировать динамическую память, могут воспользоваться функцией *calloc*, представляющей собой тощую оболочку функции *malloc*, инициализирующую выделенную память нулями. В приложениях, где возникает потребность изменить размер ранее выделенного блока, следует использовать функцию *realloc*.

Такие программы распределения динамической памяти, как функция *malloc*, могут выделить или освобождать память явным образом, воспользовавшись функциями *mmap* и *munmap*, либо они с этой целью могут воспользоваться функцией *sbrk*:

```
#include <unistd.h>
void *sbrk(int incr);
```

Функция *sbrk* увеличивает или уменьшает динамическую память путем добавления приращения *incr* к указателю *brk* ядра. При успешном завершении она возвращает старое значение *brk*, в противном случае она возвращает -1 и устанавливает в *errno* значение ENOMEM. Если *incr* принимает значение нуль, то *sbrk* возвращает текущее значение *brk*. Вызов *sbrk* с отрицательным значением приращения *incr* является допустимым, но ненадежным ввиду того, что возвращаемое значение (старое значение *brk*) указывает на новую вершину динамической памяти со смещением *abs(incr)* байтов.

Программы освобождают распределенные блоки динамической памяти, вызывая функцию *free*:

```
#include <stdlib.h>
void free(void *ptr);
```

Аргумент `ptr` должен указывать на начало выделенного блока, который был получен от функции `malloc`. В противном случае поведение `free` не определено. Более того, после того, как будет возвращено пустое значение, функция `free` никаким способом не уведомляет приложение и о том, что не все в порядке. Как мы увидим в разд. 10.11, это обстоятельство может вызвать определенные затруднения и появление ошибок времени исполнения.

На рис. 10.37 показано, как с помощью функций `malloc` и `free` можно управлять небольшой динамической памятью размером 16 слов в программе на языке С. Каждая клеточка представляет собой 4-байтовое слово. Выстроенные в ряд клеточки соответствуют распределенным (заштрихованы) и свободным (не заштрихованы) блокам. В исходном состоянии динамическая память состоит из единственного выровненного по границе двойного слова свободного блока, состоящего из 16 слов.

- В верхней части рис. 10.37 программа запрашивает блок размером в 4 слова. Функция `malloc` отвечает тем, что вырезает блок с 4 словами из начальной части свободного блока и возвращает указатель на первое слово выделенного блока;

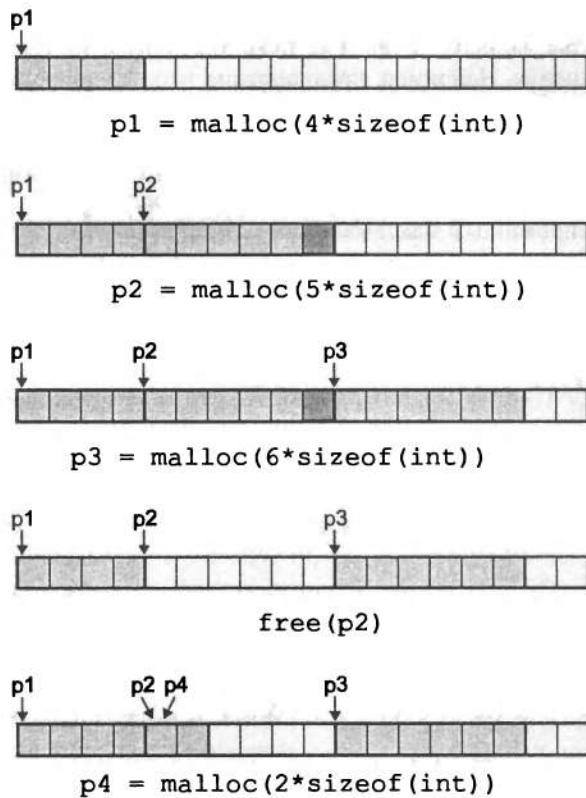


Рис. 10.37. Выделение и освобождение блоков с помощью `malloc`

- далее программа запрашивает блок размером 5 слов, malloc отвечает, выделяя блок с 6 словами с начальной части свободного блока. В этом примере malloc присоединяет к блоку дополнительное слово, чтобы обеспечить выравнивание свободного блока по границе двойного слова;
- на следующем изображении программа запрашивает блок размером в 6 слов, и malloc отвечает тем, что вырезает блок с 6 словами из свободного блока;
- программа освобождает блок из 6 слов, который был выделен. Заметьте, что когда после вызова функция free возвращает управление, указатель p2 все еще указывает на освобожденный блок. Как и раньше, само приложение несет ответственность за использование p2. Последующее использование p2 допустимо только после его инициализации новым обращением к функции malloc;
- в нижней части рис. 10.37 программа запрашивает блок размером 2 слова. В этой ситуации malloc выделяет часть блока, который был освобожден на предыдущем шаге и возвращает указатель на этот новый блок.

10.9.2. Что дает динамическое распределение памяти

Наиболее важной причиной того, что в программе используют динамическое распределение памяти, является тот факт, что часто не знают, какие размеры принимают некоторые структуры данных до тех пор, пока эта программа фактически не будет запущена на исполнение. Например, предположим, что нас попросили написать программу на языке C, которая читает из stdin в массив С список *n* целых чисел ASCII, по одному числу в строке. Ввод представляет собой целое число *n*, за которым следуют *n* целых чисел, которые должны быть считаны и сохранены в массиве. Самый простой подход заключается в том, что массив объявляется статически с некоторым жестко запрограммированным максимальным размером (листинг 10.1).

Листинг 10.1. Чтение массива целых чисел

```

1 #include "csapp.h"
2 #define MAXN 15213
3
4 int array[MAXN];
5
6 int main()
7 {
8     int i, n;
9
10    scanf("%d", &n);
11    if (n > MAXN)
12        app_error("Input file too big");
13    for (i = 0; i < n; i++)
14        scanf("%d", &array[i]);
15    exit(0);
16 }
```

Размещение массивов с жестко запрограммированными размерами, как в данном случае, часто оказывается не слишком удачным решением. Значение MAXN выбирается произвольно и не имеет никакого отношения к фактическому объему доступной виртуальной памяти заданной машины. Далее, если бы пользователь этой программы захотел бы прочитать файл, который по размеру был бы больше, чем MAXN, единственная возможность, которая была бы полезна в этом случае, — это перекомпиляция программы с использованием значения, превосходящего прежнее значение MAXN. В рамках рассматриваемого простого примера сделать это совсем не сложно, тем не менее, наличие жестко запрограммированных границ массива может стать кошмаром при обслуживании больших программных продуктов с миллионами строк программного кода и многочисленными пользователями.

Более приемлемый подход состоит в том, чтобы размещать массив динамически во время исполнения программы, после того как значение *n* становится известным. При таком подходе максимальный размер массива ограничен только объемом доступной виртуальной памяти (листинг 10.2).

Листинг 10.2 Ограничение объема

```
1 #include "csapp.h"
2
3 int main()
4 {
5     int *array, i, n;
6
7     scanf("%d", &n);
8     array = (int *)Malloc(n * sizeof(int));
9     for (i = 0; i < n; i++)
10        scanf("%d", &array[i]);
11     exit(0);
12 }
```

Динамическое распределение памяти — это полезный и важный технический прием, используемый в программировании. Однако чтобы правильно и эффективно использовать программы выделения памяти, программисты должны обладать пониманием того, как они работают. В разд. 10.11 мы рассмотрим некоторые из "ужасных" ошибок, которые могут возникать вследствие неподходящего использования программ выделения памяти.

10.9.3. Назначение программы распределения памяти и требования к ней

Программы явного распределения памяти должны быть работоспособны в условиях некоторых довольно строгих ограничений:

- Обработка произвольных последовательностей запросов. Приложение может выдавать произвольную последовательность запросов на распределение и освобож-

дение памяти при условии, что каждый запрос на освобождение памяти должен соответствовать распределенному на текущий момент блоку памяти в результате предыдущего запроса на выделение памяти. Таким образом, программа распределения памяти не может делать каких-либо прогнозов относительно упорядоченности запросов на распределение и освобождение памяти. Например, программа распределения памяти не может предположить, что каждый запрос на размещение сопровождается соответствующим запросом на освобождение, или что соответствующие запросы на распределение и освобождение попарно упорядочены.

- Немедленное реагирование на запросы. Программа распределения памяти должна реагировать немедленно при обслуживании запросов на распределение памяти. Таким образом недопустимо, чтобы программа выделения памяти меняла порядок запросов или отправляла запросы в буфер с целью повышения эффективности их обслуживания.
- Использование только динамической памяти. Чтобы программа распределения памяти была масштабируемой, любые нескалярные структуры данных, используемые этой программой, должны быть размещены в динамической памяти.
- Выравнивание блоков (требование выравнивания). Программа распределения памяти должна выровнять блоки таким образом, чтобы они могли содержать объекты данных любого типа. Для большинства систем это означает, что блок, возвращенный программой распределения памяти, выровнен по границе восьми байтов (двойное слово).
- Неизменность содержимого выделенных блоков. Программы распределения памяти могут только манипулировать свободными блоками или менять число свободных блоков. В частности, им не разрешается модифицировать или перемещать блоки, после того как они распределены. Такие методы, как уплотнение выделенных блоков, не разрешаются.

Работая в условиях этих ограничений, авторы программ распределения памяти часто пытаются удовлетворять противоречивым требованиям достижения максимальной производительности программ и максимально эффективного использования памяти.

Первое требование — достижение максимальной производительности. Пусть задана некоторая последовательность из n запросов на распределение и освобождение памяти

$$R_0, R_1, \dots, R_k, \dots, R_{n-1}.$$

Требуется обеспечить максимальную производительность (throughput) программы распределения памяти, которая определяется как число запросов, выполненных в единицу времени. Например, если программа выделения памяти выполняет 500 запросов на выделение и 500 запросов на освобождение памяти за 1 секунду, то ее производительность равна 1000 операций в секунду. В общем случае, мы можем достичь максимальной производительности, минимизируя средний промежуток времени, необходимый для удовлетворения запроса на распределение или освобождения памяти. Как мы убедимся далее, не так уж трудно создать программы распределения памяти с достаточно высокой производительностью в условиях, когда в худшем случае продолжительность обслуживания запроса линейно зависит от числа свободных блоков, а время обслуживания запроса на освобождение постоянно.

Второе требование — достижение максимальной эффективности использования памяти. Некоторые начинающие программисты часто неправильно полагают, что виртуальная память — это неограниченный ресурс. Фактически, общая сумма виртуальной памяти, отведенной всем процессам в системе, ограничена объемом пространства на диске, отведенном для свопинга. Опытные программисты понимают, что виртуальная память представляет собой ограниченный ресурс, который следует использовать эффективно. Это особенно справедливо для программ выделения динамической памяти, работающих в режиме распределения и освобождения больших блоков памяти.

Имеется несколько способов оценки того, насколько эффективно программа распределения памяти использует динамическую память. В нашем случае наиболее подходящим показателем является коэффициент пиковой загруженности (peak utilization). Как и прежде, пусть задана некоторая последовательность запросов на и операций выделения и освобождения памяти

$$R_0, R_1, \dots, R_k, \dots, R_{n-1}.$$

Если приложение запрашивает блок, состоящий из p байтов, то выделяемый в результате обслуживания этого запроса блок имеет полезный размер (payload) p байтов. После того, как запрос R_k будет обслужен, положим, что P_k , представляющий сумму полезных размеров выделенных на текущий момент блоков, есть полный размер (aggregate payload), а через H_k обозначим текущее (монотонно неубывающее) значение размера динамической памяти. Тогда пиковый коэффициент загруженности (peak utilization) по первым k запросам, обозначаемый как U_k , задается следующим выражением:

$$U_k = \frac{\max_{i \leq k} P_i}{H_k}.$$

Теперь можно сформулировать цель программы выделения памяти, она заключается в том, чтобы достичь максимума пикового коэффициента загруженности U_n на всей последовательности. Как мы увидим далее, есть противоречие между максимизацией производительности и коэффициентом загруженности. В частности, не трудно написать программу распределения памяти, которая обеспечивает максимальную производительность благодаря использованию динамической памяти. Одной из интересных проблем в любой разработке программы распределения памяти является поиск подходящего баланса между этими двумя показателями.

Смягчение допущения о монотонности

В нашем определении U_k мы могли бы ослабить условие монотонного неубывания и позволить динамической памяти расти вверх и вниз, принимая H_k в качестве предельного максимума по первым k запросам.

10.9.4. Фрагментация

Основная причина неэффективного использования динамической памяти есть явление, получившее название *фрагментация* (fragmentation), которое имеет место, когда память, не использованная для других целей, недоступна для выделения по запросу.

Всего имеется две формы фрагментации: внутренняя фрагментация (internal fragmentation) и внешняя фрагментация (external fragmentation).

Внутренняя фрагментация имеет место, когда выделенный блок по размеру больше, чем полезный размер. Это может происходить по нескольким причинам. Например, конкретная реализация программы распределения памяти может ограничивать минимальный размер выделяемого блока, который может оказаться больше, чем тот или иной запрошенный полезный размер. Или, как показано на рис. 10.37, программа распределения памяти может увеличить размер блока, чтобы удовлетворить ограничениям выравнивания.

Внутренняя фрагментация может быть задана количественно. Это не что иное, как просто сумма разностей между размерами выделенных блоков и их полезных размеров. Таким образом, в любой момент времени общий объем внутренней фрагментации зависит только от последовательности предыдущих запросов и от реализации программы распределения памяти.

Внешняя фрагментация имеет место, когда имеется достаточное пространство свободной памяти, необходимой для удовлетворения запроса на распределение, но никакой отдельный свободный блок не имеет достаточного размера, чтобы обработать данный запрос. Например, если бы запрос на рис. 10.37 требовал выделения шести слов, а не двух, то этот запрос не мог бы быть удовлетворен без запроса на подкачку ядром дополнительной виртуальной памяти, несмотря на то, что в динамической памяти имеется всего шесть свободных слов. Осложнения возникают в связи с тем, что эти шесть слов содержатся в двух свободных блоках.

Измерить внешнюю фрагментацию намного сложнее, чем внутреннюю, поскольку она зависит не только от последовательности предыдущих запросов и реализации программы распределения памяти, но также и от последовательности будущих запросов. Например, предположим, что после выполнения k запросов все свободные блоки имеют размер в четыре слова. Страдает ли такая динамическая память от внешней фрагментации? Ответ зависит от последовательности будущих запросов. Если все будущие запросы на выделение ориентированы на блоки, меньшие по размеру, чем четыре слова, то нет никакой внешней фрагментации. С другой стороны, если один или несколько запросов потребуют блок размером больше четырех слов, то динамическая память действительно пострадает от внешней фрагментации.

Поскольку внешняя фрагментация трудно поддается количественной оценке, и ее, как правило, невозможно предсказать, то программы распределения памяти обычно используют эвристику, которая держит в резерве небольшое число крупных свободных блоков памяти, но не большое число свободных блоков памяти меньших размеров.

10.9.5. Вопросы реализации

Простейшая воображаемая программа распределения памяти организует динамическую память как большой массив байтов и указатель p , который в исходном состоянии указывает на первый байт массива. Чтобы распределить $size$ байтов, функция `malloc` должна сохранить текущее значение p в стеке, увеличить p на величину $size$

и возвратить вызывающей программе старое значение `p`. Функция `free` просто возвращает управление в вызывающую программу, не производя при этом никаких действий.

Такая простейшая программа распределения памяти представляет собой крайнюю точку в пространстве решений. Поскольку функции `malloc` и `free` исполняют только очень небольшое число команд, производительность такой конструкции была бы исключительно высокой. Однако, поскольку программа распределения памяти никогда не использует повторно никаких блоков, использование памяти было бы исключительно неэффективным. Практическая программа распределения памяти, которая достигает приемлемого баланса между производительностью и использованием памяти, должна быть способной ответить на следующие вопросы:

- Организация свободных блоков: как мы ведем учет свободных блоков?
- Размещение: как мы выбираем подходящий свободный блок памяти, чтобы в нем разместить только что распределенный блок?
- Разбиение: после того как мы поместим вновь выделенный блок в некоторый свободный блок памяти, что мы делаем с оставшейся частью этого свободного блока?
- Объединение: что мы делаем с только что освобожденным блоком?

Далее эти вопросы рассматриваются более подробно. Поскольку основные методы размещения, разбиения и объединения имеют много общего в различных методах организации свободных блоков, мы представим их в контексте простой организации свободных блоков, которая называется неявным списком свободных блоков.

10.9.6. Неявные списки свободных блоков

Любая работающая программа распределения памяти требует наличия тех или иных структур данных, которые позволяют распознавать границы блоков и отличать выделенные блоки от свободных. Многие программы распределения памяти встраивают соответствующую информацию непосредственно в блоки. Один простой подход показан на рис. 10.38.



Рис. 10.38. Формат простого блока динамической памяти

В рассматриваемом случае блок состоит из заголовка сверху (header) размером в одно слово, используемой части и, возможно, дополнительной неиспользуемой части (padding). Заголовок сверху содержит сведения о размере блока (включая заголовок и неиспользуемую часть), а также информацию о том, выделен ли данный блок, или он свободен. Если мы установим выравнивание по границе двойного слова, то размер блока всегда кратен восьми, и три младших бита размера блока суть всегда нули. Таким образом, мы должны сохранить только 29 старших разрядов величины размера блока, а остающиеся три разряда позволяют кодировать другую информацию. В рассматриваемом случае мы используем самый младший из этих разрядов (разряд распределения, allocated bit), чтобы показать, выделен ли данный блок или он свободен. Предположим, например, что мы имеем выделенный блок размером 24 (0x18) байтов. Тогда его заголовок сверху должен быть таким:

`0x000000018 | 0x1 = 0x000000019`

Аналогично заголовок вверху свободного блока размером 40 (0x28) байтов будет иметь такой вид:

`0x000000028 | 0x0 = 0x000000028`

За заголовком сверху следует полезное используемое пространство, которое было затребовано приложением при вызове функции `malloc`. За полезным пространством следует неиспользуемый участок памяти, который в общем случае может иметь любой размер. Имеется несколько причин для выделения неиспользуемой части. Например, неиспользуемая часть может играть особую роль в реализации стратегии программы распределения памяти, как средство противодействия внешней фрагментации. Неиспользуемая часть может также понадобиться в целях выравнивания адресов.

Для формата блока, заданного на рис. 10.38, мы можем организовать динамическую память в виде некоторой последовательности выделенных и свободных блоков, как показано на рис. 10.39.

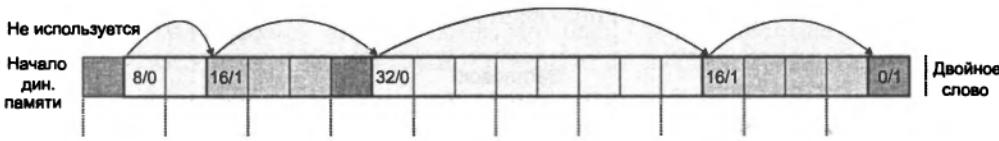


Рис. 10.39. Организация динамической памяти с использованием неявного списка свободных блоков

Такую организацию памяти мы будем называть неявным списком свободных блоков (*implicit free list*), поскольку свободные блоки неявно связаны между собой полями размера в заголовках сверху. Программа распределения памяти может косвенным образом просмотреть все множество свободных блоков, последовательно обходя все блоки динамической памяти. Обратите внимание на то, что в этой структуре необходим некоторый специально помеченный концевой блок. В нашем примере таковым является заголовок снизу, в котором установлен разряд распределения, а его размер

равен нулю. Как мы увидим в разд. 10.9.12, установка разряда распределения упрощает объединение свободных блоков в связную цепочку.

Преимущество неявного списка свободных блоков — это его простота. Существенным недостатком является то, что затраты на каждую операцию, такую как размещение выделяемого блока, которая требует поиска в списке свободных блоков, линейно зависят от общего числа выделенных и освобожденных блоков в динамической памяти.

Важно понять, что системные требования к выравниванию и выбор формата блока для программы распределения памяти определяют минимальный размер блока (*minimum block size*). Никакой распределенный или свободный блок не может быть меньше этого минимума. Например, если мы производим выравнивание по двойному слову, то размер каждого блока должен быть кратен длине двух слов (8 байтов). Таким образом, из формата блока на рис. 10.38 следует, что минимальный размер блока составляют два слова: одно слово для заголовка, а другое обеспечивает удовлетворение требования выравнивания. Даже если бы приложение запросило всего лишь один байт, программа распределения памяти и в этом случае штамповала блоки размером в два слова.

УПРАЖНЕНИЕ 10.6

Вычислите размер блока и код в заголовке для приводимой ниже последовательности запросов к функции `malloc`. Должны быть выполнены следующие условия:

- программа распределения памяти поддерживает выравнивание по границе двойного слова и использует неявный список свободных блоков с форматом, представленным на рис. 10.38;
- размеры блока округлены до ближайшего большего целого числа, кратного восьми байтам.

Запрос	Размер блока (в десятичном формате)	Заголовок сверху блока (в шестнадцатеричном формате)
<code>malloc(1)</code>		
<code>malloc(5)</code>		
<code>malloc(12)</code>		
<code>malloc(13)</code>		

10.9.7. Размещение распределенных блоков

Когда приложение запрашивает блок размером k байтов, программа распределения памяти просматривает список свободных блоков, отыскивая в нем незанятый блок, размер которого достаточен для того, чтобы вместить затребованный блок. Способ,

которым программа распределения памяти осуществляет этот поиск, определяется стратегией размещения (placement policy). Наиболее распространенные стратегии суть метод первого подходящего (first fit), метод следующего подходящего (next fit) и метод наиболее согласованного (best fit) блоков.

Метод первого подходящего блока просматривает список свободных блоков с самого начала и выбирается первый подходящий свободный блок, соответствующий запросу. Метод следующего подходящего блока подобен методу первого подходящего блока, но вместо того, чтобы каждый раз начинать поиск с самого начала списка, он продолжает поиск с того места, где остановился предыдущий поиск. Метод наиболее согласованного блока исследует каждый свободный блок и выбирает тот из блоков, соответствующих запросу, который имеет наименьший размер.

Преимущество метода первого подходящего блока состоит в том, что он имеет тенденцию оставлять большие свободные блоки в конце списка. Недостаток его в том, что он имеет тенденцию оставлять в начале списка маленькие "осколки" свободных блоков, что увеличивает время поиска больших блоков. Метод следующего подходящего блока был впервые предложен Дональдом Кнутом (Donald Knuth) в качестве альтернативы методу первого подходящего блока. Основная идея этого метода заключается в том, что если мы в прошлый раз обнаружили некоторый пригодный свободный блок, то велика вероятность того, что в следующий раз мы обнаружим пригодный блок в оставшейся части списка. Метод следующего подходящего блока может дать результат значительно быстрее, чем метод первого подходящего блока, особенно если начальная часть списка становится засоренной большим количеством маленьких осколков. Однако некоторые исследователи высказывают предположения о том, что метод следующего подходящего блока по эффективности использования памяти несколько хуже, чем метод первого подходящего блока. Исследователи пришли к выводу, что метод наиболее согласованного блока, как правило, отличается более оптимальным использованием памяти, чем метод первого подходящего блока или чем метод следующего подходящего блока. Однако недостаток использования метода наиболее согласованного блока при организации такого простого списка свободных блоков, как, например, неявный список свободных блоков, состоит в том, что требуется полный перебор динамической памяти. Позже мы рассмотрим некоторые более сложные организации списка свободных блоков, которые реализуют наиболее согласованную политику без полного перебора динамической памяти.

10.9.8. Разбиение свободных блоков

После того как программа распределения памяти определила местонахождение подходящего по размеру свободного блока, она, в соответствии с выбранной стратегией, должна принять следующее решение: определить, сколько свободных блоков памяти необходимо выделить под запрос. Один из возможных вариантов предусматривает использование всего свободного блока. Несмотря на то, что он прост в реализации и обладает хорошим быстродействием, он не лишен недостатков, основной его недостаток состоит в том, что он ведет к появлению внутренней фрагментации. Если такая стратегия позволяет получить приемлемые результаты, то некоторая дополнительная внутренняя фрагментация может оказаться допустимой.

Однако если выбранный блок памяти далек от оптимального (по размеру намного больше запрашиваемого), то программа распределения памяти обычно решает разбить (*split*) этот свободный блок на две части. Первая часть становится выделенным блоком, а оставшаяся часть — новым свободным блоком. На рис. 10.40 показано, как программа распределения памяти разбивает свободный блок памяти размером восемь слов, изображенный на рис. 10.39, чтобы удовлетворить запрос приложения на выделение трех слов динамической памяти.

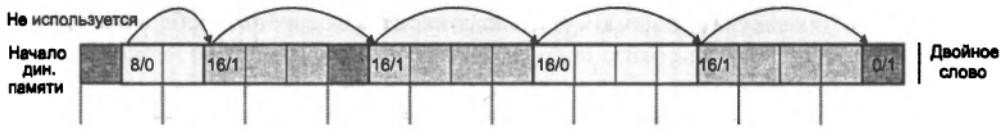


Рис. 10.40. Разбиение свободного блока с целью обслуживания запроса на выделение трех слов

10.9.9. Получение дополнительной динамической памяти

Что произойдет в случае, если программа распределения памяти не сможет найти подходящий участок для запрошенного блока? Один возможный вариант — попытаться построить какой-нибудь больший свободный блок путем объединения свободных блоков, последовательно расположенных в физической памяти (см. следующий раздел). Однако если это не приводит к образованию достаточно большого блока или если свободные блоки уже максимально объединены, то программа распределения памяти запрашивает у ядра дополнительную динамическую память, вызывая функцию `sbrk` или `mmap`. В любом случае программа распределения памяти преобразует эту дополнительную память в один большой свободный блок, вставляет этот блок в список свободных блоков памяти, а затем размещает запрашиваемый блок в этом новом свободном блоке памяти.

10.9.10. Объединение свободных блоков

Когда программа распределения памяти освобождает выделенный блок, может оказаться, что смежными с ним могут оказаться другие свободные блоки. Наличие таких смежных свободных блоков может вызвать эффект, называемый ложной фрагментацией (*false fragmentation*), когда имеется много доступной свободной памяти, нарезанной на маленькие, непригодные для размещения свободные блоки. Например, на рис. 10.41 показан результат освобождения блока, который был выделен на рис. 10.40. В результате получаем два смежных свободных блока с полезным пространством размером три слова каждый. В силу этого обстоятельства последующий запрос полезного пространства размером четыре слова останется невыполненным, хотя общий размер этих двух свободных блоков достаточен для того, чтобы удовлетворить запрос.

Чтобы бороться с ложной фрагментацией, любая работающая программа распределения памяти должна объединять смежные свободные блоки в процессе, называю-

щемся объединением (coalescing). Вопрос о том, когда именно выполнять объединение, имеет немаловажное значение для стратегии. Программа распределения памяти может принять решение осуществлять немедленное объединение (immediate coalescing), в соответствии с которым смежные блоки объединяются каждый раз при освобождении того или иного блока. Объединение может быть отложенным (deferred coalescing), в условиях которого свободные блоки объединяются несколько позднее. Например, программа распределения памяти могла бы отложить объединение до тех пор, пока обслуживание какого-нибудь запроса не завершится неудачей, а затем просмотреть всю динамическую память с целью слияния всех свободных блоков.

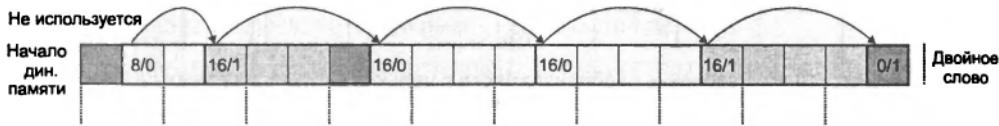


Рис. 10.41. Пример ложной фрагментации

Немедленное объединение осуществляется достаточно просто и может быть выполнено за постоянное время, но при некоторых форматах запросов оно может войти в одну из форм продолжительных циклов, когда блоки многократно объединяются и сразу после этого разбиваются на части. Например, на рис. 10.40 повторяющиеся выделения и освобождения блоков из трех слов приводят к ненужным разбиениям и объединениям. При анализе программы распределения памяти мы принимаем решение об использовании модели немедленного слияния, но вы должны иметь в виду, что быстродействующие программы распределения памяти часто отдают предпочтение той или иной форме отложенного слияния.

10.9.11. Объединение с использованием граничных тегов

Каким образом программа распределения памяти осуществляет слияние? Блок, который необходимо освободить, будем называть *текущим блоком* (current block). Влияние со следующим свободным блоком (в памяти) выполняется просто и эффективно. Заголовок сверху текущего блока указывает на заголовок сверху следующего блока, что может быть использовано с целью определить, свободен ли следующий блок. Если это так, то его размер просто добавляется к размеру текущего блока, указанному в заголовке сверху текущего блока, благодаря чему операции объединения блоков будут выполняться в течение постоянного времени.

Но как быть, если нужно объединить блок с предыдущим? При наличии неявного списка свободных блоков с заголовками сверху, единственным возможным вариантом остается просмотр полного списка с запоминанием местоположения предыдущего блока до тех пор, пока не достигнем текущий блок. При использовании неявного списка свободных блоков это означает, что каждый вызов функции `free` требует времени, линейно зависящего от размера динамической памяти. Даже при использо-

вании более сложной организации списка свободных блоков, время поиска не будет постоянным.

Кнут разработал стройную и универсальную методику, использующую *граничные теги* (boundary tag), которая позволяет производить объединение с предыдущим блоком за постоянное время. Идея метода, представленная на рис. 10.42, основана на добавлении заголовка снизу (footer) (граничный тег) в конец каждого блока, являющегося точной копией заголовка сверху. Если каждый блок включает такой заголовок снизу, то программа распределения памяти может определить начальное местоположение и состояние предыдущего блока путем просмотра своего заголовка снизу, который всегда расположен со смещением на одно слово от начала текущего блока.

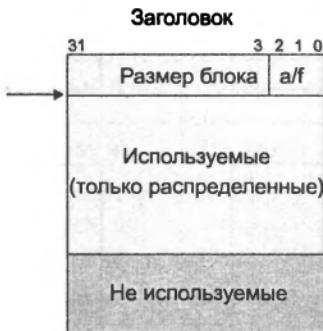


Рис. 10.42. Формат блока динамической памяти, использующего граничный тег

Рассмотрим все возможные случаи, когда программа распределения памяти освобождает текущий блок:

- Предыдущий и следующий блоки заняты.
- Предыдущий блок распределен, а следующий блок свободен.
- Предыдущий блок свободен, а следующий блок распределен.
- Предыдущий и следующий блоки свободны.

На рис. 10.43 показано, как можно осуществить объединение блоков в каждом из этих четырех случаев.

- В случае 1 оба смежных блока распределены, следовательно, никакое объединение невозможно. Таким образом, состояние текущего блока просто изменяется с распределенного на свободное.
- В случае 2 текущий блок сливается со следующим блоком. Заголовок сверху текущего блока и заголовок снизу следующего блока подвергаются модификации, в рамках которой производится суммирование размеров текущего и следующего блоков.
- В случае 3 предыдущий блок сливается с текущим блоком. Заголовок сверху предыдущего блока и заголовок снизу текущего блока модифицируются путем суммирования размеров этих двух блоков.

- В случае 4 все три блока объединяются, формируя один свободный блок, с заголовком сверху предыдущего блока и заголовком снизу следующего блока, модифицированных путем суммирования размеров этих трех блоков. В каждом случае объединение выполняется за постоянное время.

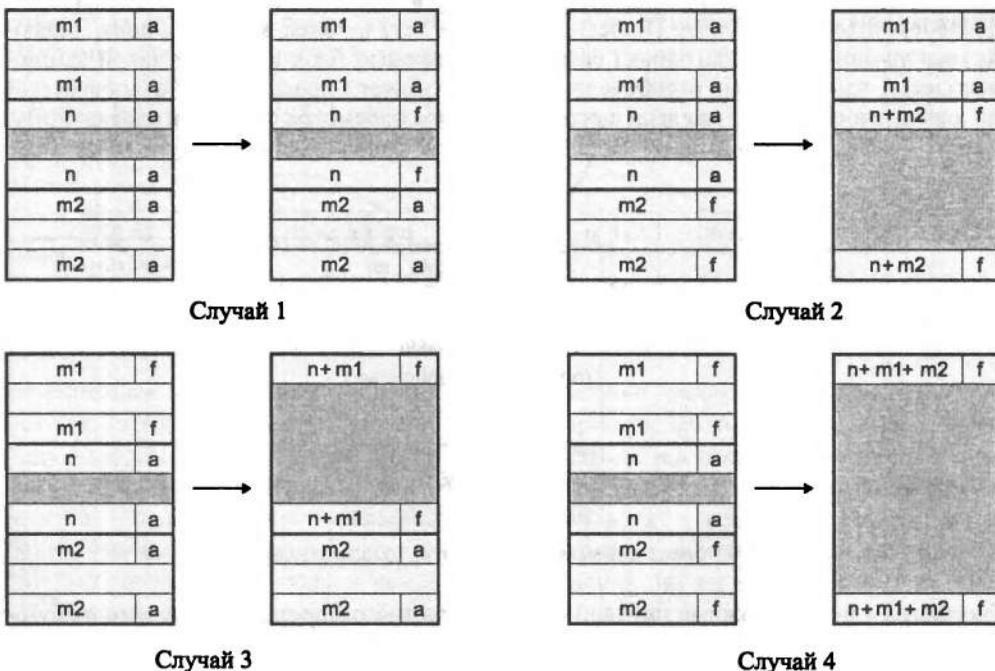


Рис. 10.43. Объединение с использованием граничных тегов

Идея граничных тегов проста и элегантна, она обобщает несколько различных видов распределения памяти и списков свободных блоков. В то же время, она имеет потенциальный недостаток. Требование, чтобы каждый блок содержал как заголовок, так и поле признаков, может вызвать существенные непроизводительные затраты памяти, если приложение управляет большим количеством маленьких блоков. Например, если приложение связано с обработкой графов и динамически порождает и уничтожает узлы графа, делая повторные запросы к функциям `malloc` и `free`, и каждый узел графа требует всего лишь несколько слов памяти, то заголовок сверху и заголовок снизу будут потреблять половину памяти, отведенной для каждого распределенного блока.

Положение облегчается тем, что существует вариант оптимизации граничных тегов, в условиях которого нет необходимости использовать заголовки снизу выделенных блоков. Напоминаем, что в тех случаях, когда мы пытаемся объединять текущий блок с предыдущим и следующим блоками в памяти, поле размера заголовка снизу предыдущего блока необходимо, только если предыдущий блок свободен. Если бы нам нужно было сохранить разряд "распределен/свободен" предыдущего блока в од-

ном из избыточных младших разрядов текущего блока, то распределенные блоки не требовали бы наличия заголовков, и мы могли бы использовать это освободившееся пространство для полезной нагрузки блока. Тем не менее, обратите внимание на то, что свободные блоки по-прежнему требуют наличия заголовков снизу.

УПРАЖНЕНИЕ 10.7

Определите минимальный размер блока для каждой из следующих комбинаций требований по выравниванию и форматов блоков. При этом предполагается выполнение следующих условий: неявный список свободных блоков, полезная нагрузка нулевого размера не допускается, а заголовки сверху и снизу хранятся в четырехбайтовых словах.

Выравнивание	Распределенный блок	Свободный блок	Минимальный размер блока
Одинарное слово	Заголовки сверху и снизу	Заголовки сверху и снизу	
Одинарное слово	Заголовок сверху, но без заголовка снизу	Заголовки сверху и снизу	
Двойное слово	Заголовки сверху и снизу	Заголовки сверху и снизу	
Двойное слово	Заголовок сверху, но без заголовка снизу	Заголовки сверху и снизу	

10.9.12. Реализация простой программы распределения памяти

Разработка программы распределения памяти представляет собой достаточно трудную задачу. Пространство решений обширно, с многочисленными альтернативами формата блока, формата списка свободных блоков, размещения, разбиения и слияния блоков. Другая трудность заключается в том, что вы часто будете вынуждены программировать, выходя за пределы безопасной, привычной системы типов, полагаясь на подверженное ошибкам приведение указателей и на арифметические операции над указателями, которые обычно применяются для программирования низкоуровневых систем. Несмотря на то, что для программ распределения памяти не характерны огромные объемы программного кода, они требуют большого искусства и не прощают ошибок. Читатели, знакомые с языками программирования высокого уровня, такими как C++ или Java, часто упираются в концептуальную стену, когда впервые сталкиваются с этим стилем программирования. Чтобы помочь вам преодолеть это препятствие, далее мы займемся разработкой простой программы распределения памяти с неявным списком свободных блоков и с немедленным объединением с использованием граничных тегов.

Разработка программы выделения памяти

Разрабатываемая нами программа распределения памяти использует модель системы памяти, предоставляемую модулем memlib.c и изображенную в листинге 10.3. Назначение модели состоит в том, чтобы дать возможность выполнять программу распределения памяти без обращения к существующей на системном уровне функции malloc.

Листинг 10.3. Модель системы памяти

```

1 #include "csapp.h"
2
3 /* закрытые глобальные переменные */
4 static char mem_start_brk; /* указывает на первый байт динамической памяти */
5 static char mem_brk;        /* указывает на последний байт динамической
                               памяти */
6 static char mem_max_addr;   /* максимальный виртуальный адрес динамической
                               памяти */
7
8 /*
9  * mem_init - инициализирует модель системы памяти
10 */
11 void mem_init(int size)
12 {
13     mem_start_brk = (char)Malloc(size); /* моделирует доступную виртуальную
                                         память */
14     mem_brk = mem_start_brk;           /* динамическая память в исходном
                                         состоянии пуста */
15     mem_max_addr = mem_start_brk + size; /* максимальный адрес виртуальной
                                         памяти для динамической памяти */
16 }
17
18 /*
19 * mem_sbrk - простая модель функции sbrk. Расширяет динамическую память
20 * на incr байтов и возвращает начальный адрес новой области;
21 * в этой модели динамическая память не может быть сокращена
22 */
23 void *mem_sbrk(int incr)
24 {
25     char *old_brk = mem_brk;
26
27     if ( (incr < 0) || ((mem_brk + incr) > mem_max_addr) ) {
28         errno = ENOMEM;
29         return (void *)-1;
30     }

```

```

31     mem_brk += incr;
32     return old_brk;
33 }

```

Функция `mem_init` моделирует виртуальную память, доступную из динамической памяти (динамической памяти) как крупный, выровненный по границе двойного слова массив байтов. Байты, заключенные между `mem_start_brk` и `mem_brk`, представляют собой распределенную виртуальную память. Байты, следующие за `mem_brk`, представляют собой незанятую виртуальную память. Программа распределения памяти запрашивает дополнительную память из динамической памяти, обращаясь с этой целью к функции `mem_sbrk`, которая имеет тот же самый интерфейс, что и системная функция `sbrk`, а также ту же семантику, за исключением того, что она не принимает запросы на сокращение динамической памяти.

Сама программа распределения памяти содержится в исходном файле (`malloc.c`), который пользователи могут компилировать и компоновать в свои приложения. Программа распределения памяти экспортирует в прикладные программы три функции:

```

1 int mm_init(void);
2 void *mm_malloc(size_t size);
3 void mm_free(void *bp);

```

Функция `mm_init` инициализирует программу распределения памяти, возвращая 0 при успешном завершении инициализации и -1 в противном случае. Функции `mm_malloc` и `mm_free` имеют такие же интерфейсы и семантику, как и их системные аналоги. Программа распределения памяти использует формат блока, показанный на рис. 10.42. Минимальный размер блока 16 байтов. Список свободных блоков организован как неявный список свободных блоков в инвариантной форме, показанной на рис. 10.44.

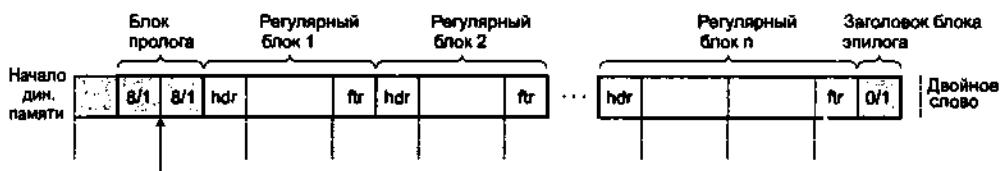


Рис. 10.44. Инвариантная форма неявного списка свободных блоков

Первое слово заполнено незначащей информацией и используется для выравнивания по границе двойного слова. Далее следует специальный выделенный блок пролога (prologue block), представляющий собой восемьбайтовый распределенный блок и состоящий только из заголовка сверху и заголовка снизу. Блок пролога образуется в процессе инициализации и никогда не освобождается. За блоком пролога следует нуль или большее число стандартных блоков, которые образуются в результате обращений к функциям `malloc` и `free`. В конце динамической памяти всегда располагается специальный блок эпилога — распределенный блок нулевого размера, который состоит только из заголовка. Блоки пролога и эпилога суть элементы, служащие для

устранения граничных эффектов при слиянии блоков. Программа распределения памяти использует отдельную закрытую (статическую) глобальную переменную (`heap_listp`), которая всегда указывает на блок пролога. (С целью небольшой оптимизации мы установили ее указатель на следующий блок вместо блока пролога.)

Основные константы и макроопределения для управления списком свободных блоков

В листинге 10.4 показаны некоторые основные константы, которые мы будем использовать в программе распределения памяти.

Листинг 10.4. Основные константы и макроопределения для управления списком свободных блоков

```

1  /* основные константы и макроопределения */
2 #define WSIZE      4      /* размер слова (в байтах) */
3 #define DSIZE      8      /* размер двойного слова (в байтах) */
4 #define CHUNKSIZE (1<<12) /* начальный размер динамической памяти
                           (в байтах) */
5 #define OVERHEAD   8      /* размер заголовка и поля признаков (в байтах) */
6
7 #define MAX(x, y) ((x) > (y)? (x) : (y))
8
9 /* упаковывает в слово размер и разряд распределения */
10#define PACK(size, alloc) ((size) | (alloc))
11
12/* читает и записывает слово по адресу p */
13#define GET(p) (*((size_t *) (p)))
14#define PUT(p, val) (*((size_t *) (p)) = (val))
15
16/* читает размер и распределенные поля по адресу p */
17#define GET_SIZE(p) (GET(p) & ~0x7)
18#define GET_ALLOC(p) (GET(p) & 0x1)
19
20/* для заданного указателя блока bp вычисляет адрес его заголовка сверху
   и заголовка снизу */
21#define HDRP(bp) ((char *) (bp) - WSIZE)
22#define FTRP(bp) ((char *) (bp) + GET_SIZE(HDRP(bp)) - DSIZE)
23
24/* для заданного указателя блока bp вычисляет адрес следующего
   и предыдущего блоков */
25#define NEXT_BLKP(bp) ((char *) (bp) + GET_SIZE(((char *) (bp) - WSIZE)))
26#define PREV_BLKP(bp) ((char *) (bp) - GET_SIZE(((char *) (bp) - DSIZE)))

```

В строках 2—5 заданы значения некоторых основных констант, определяющих размеры: размер слов (`WSIZE`) и двойных слов (`DSIZE`), размер начального свободного блока и размер по умолчанию блока расширения динамической памяти (`CHUNKSIZE`), а

также количество дополнительных байтов, используемых заголовком сверху и заголовком снизу (*overhead*) и относящихся к непроизводительным затратам памяти.

Манипулирование заголовками сверху и снизу в списке свободных блоков может вызвать некоторые неудобства, поскольку оно связано с интенсивным использованием операций приведения и адресной арифметики. В силу этого обстоятельства, мы находим полезным дать небольшой набор макроопределений для осуществления доступа и просмотра списка свободных блоков (строки 10—26).

- Макрокоманда `PACK` (строка 10) объединяет размер и разряд распределения и возвращает значение, которое может быть сохранено в заголовке сверху или в заголовке снизу.
- Макрокоманда `GET` (строка 13) читает и возвращает слово, на которое ссылается параметр `p`. Преобразование типов здесь имеет важное значение. Параметр `p` — это, как правило, указатель `void*`, непосредственное разыменование которого невозможно.
- Аналогично, макрокоманда `PUT` (строка 14) сохраняет значение `val` в слове, адрес которого задан параметром `p`.
- Макроопределения `GET_SIZE` и `GET_ALLOC` (строки 17—18) возвращают размер и разряд распределения соответственно из заголовка сверху или заголовка снизу по адресу `p`. Остальные макроопределения оперируют *указателями блоков* (*block pointer*), обозначенными через `bp`, которые адресуют первый байт полезного пространства.
- Для заданного указателя блока `bp` макроопределения `HDRP` и `FTRP` (строки 21—22) возвращают указатели соответственно на заголовок сверху и снизу блока.
- Макроопределения `NEXT_BLKP` и `PREV_BLKP` (строки 25—26) возвращают соответственно указатели на следующий и предыдущий блоки.

Макроопределения для манипулирования списком свободных блоков могут быть составлены различными способами. Например, для определения размера следующего блока в памяти при заданном указателе `bp` на текущий блок можно воспользоваться следующей строкой программных кодов:

```
size_t size = GET_SIZE(HDRP(NEXT_BLKP(bp)));
```

Создание начального списка свободных блоков

Прежде чем вызывать функции `mm_malloc` или `mm_free`, приложение должно инициализировать динамическую память, обратившись с этой целью к функции `mm_init` (листинг 10.5).

Листинг 10.5. Инициализация динамической памяти

```
1 int mm_init(void)
2 {
3     /* образуем исходную пустую динамическую память */
4     if ((heap_listp = mem_sbrk(4*WSIZE)) == NULL)
5         return -1;
```

```

6     PUT(heap_listp, 0);                      /* дополнение для выравнивания */
7     PUT(heap_listp+WSIZE, PACK(OVERHEAD, 1)); /* заголовок сверху пролога
*/
8     PUT(heap_listp+DSIZE, PACK(OVERHEAD, 1)); /* заголовок снизу пролога */
9     PUT(heap_listp+WSIZE+DSIZE, PACK(0, 1)); /* заголовок сверху
                                                 эпилога */
10    heap_listp += DSIZE;
11
12    /* расширяем пустую динамическую память со свободным блоком размером
       CHUNKSIZE байтов */
13    if (extend_heap(CHUNKSIZE/WSIZE) == NULL)
14        return -1;
15    return 0;
16 }

```

Функция `mm_init` получает четыре слова из системы памяти и инициализирует их, образуя пустой список свободных блоков (строки 4—10). После этого она вызывает функцию `extend_heap` (листинг 10.6), которая расширяет динамическую память динамической памяти на `CHUNKSIZE` байтов и образует начальный свободный блок. В этом месте инициализируется программа распределения памяти, и она готова принимать от приложений запросы на выделение и освобождение памяти.

Листинг 10.6. Расширение динамической памяти

```

1 static void *extend_heap(size_t words)
2 {
3     char *bp;
4     size_t size;
5
6     /* выделяем четное число слов с целью соответствия требованиям
       выравнивания */
7     size = (words % 2) ? (words+1) * WSIZE : words * WSIZE;
8     if ((int)(bp = mem_sbrk(size)) < 0)
9         return NULL;
10
11    /* инициализируем заголовок сверху/снизу свободного блока и заголовок
       сверху эпилога */
12    PUT(HDRP(bp), PACK(size, 0));           /* заголовок сверху свободного
                                               блока */
13    PUT(FTRP(bp), PACK(size, 0));           /* заголовок снизу свободного
                                               блока */
14    PUT(HDRP(NEXT_BLKP(bp)), PACK(0, 1)); /* заголовок снизу нового
                                               эпилога */
15
16    /* Слияние, если предыдущий блок был свободен */
17    return coalesce(bp);
18 }

```

Функция `extend_heap` вызывается в двух различных случаях:

- при инициализации динамической памяти;
- когда функция `mm_malloc` не может обнаружить подходящий блок.

С целью удовлетворения требований выравнивания функция `extend_heap` округляет запрошенный размер до ближайшего большего целого числа, кратного 2 словам (8 байтов), а затем запрашивает у системы памяти дополнительную область динамической памяти (строки 7—9).

Оставшаяся часть функции `extend_heap` (строки 12—17) несколько сложнее. Динамическая память начинается на границе двойного слова, и каждый запрос к функции `extend_heap` возвращает блок, размер которого есть целое число двойных слов. Таким образом, каждый запрос к функции `mem_sbrk` возвращает выровненный по границе двойного слова участок памяти, следующий сразу же после заголовка блока эпилога. Этот заголовок становится заголовком нового свободного блока (строка 12), и последнее слово этого участка памяти становится новым заголовком блока эпилога (строка 14). Наконец, в том возможном случае, когда предыдущая динамическая память заканчивалась свободным блоком, мы вызываем функцию `coalesce`, которая объединяет эти два свободных блока и возвращает указатель на объединенный блок (строка 17).

Освобождение и объединение блоков

Приложение освобождает выделенный ранее блок, обращаясь к функции `mm_free` (листинг 10.7), которая освобождает запрошенный блок (`bp`) и затем объединяет смежные свободные блоки, используя метод граничных тегов, описанный в разд. 10.9.11.

Листинг 10.7. Метод граничных тегов

```

1 void mm_free(void *bp)
2 {
3     size_t size = GET_SIZE(HDRP(bp));
4
5     PUT(HDRP(bp), PACK(size, 0));
6     PUT(FTRP(bp), PACK(size, 0));
7     coalesce(bp);
8 }
9
10 static void *coalesce(void *bp)
11 {
12     size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLKP(bp)));
13     size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
14     size_t size = GET_SIZE(HDRP(bp));
15

```

```

16     if (prev_alloc && next_alloc) {           /* случай 1 */
17         return bp;
18     }
19
20     else if (prev_alloc && !next_alloc) {        /* случай 2 */
21         size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
22         PUT(HDRP(bp), PACK(size, 0));
23         PUT(FTRP(bp), PACK(size, 0));
24         return(bp);
25     }
26
27     else if (!prev_alloc && next_alloc) {        /* случай 3 */
28         size += GET_SIZE(HDRP(PREV_BLKP(bp)));
29         PUT(FTRP(bp), PACK(size, 0));
30         PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
31         return(PREV_BLKP(bp));
32     }
33
34     else {                                         /* случай 4 */
35         size += GET_SIZE(HDRP(PREV_BLKP(bp))) +
36         GET_SIZE(FTRP(NEXT_BLKP(bp)));
37         PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
38         PUT(FTRP(NEXT_BLKP(bp)), PACK(size, 0));
39         return(PREV_BLKP(bp));
40     }
41 }

```

Программный код вспомогательной функции `coalesce` представляет собой непосредственную реализацию четырех случаев, отмеченных на рис. 10.43. Здесь есть одна особенность, на которую следует обратить внимание. Выбранный нами формат списка свободных блоков, формат с блоками пролога и эпилога, которые всегда отмечаются как распределенные (т. е. занятые), позволяет игнорировать потенциально чреватые неприятностями краевые условия, когда запрашиваемый блок `bp` расположен в начале или в конце динамической памяти. Без этих специальных блоков программный код был бы менее прозрачен, больше подвержен ошибкам и работал бы медленнее, ввиду того, что мы должны были бы проверять эти редко встречающиеся краевые условия при каждом запросе на освобождение блока памяти.

Выделение блоков

Приложение запрашивает блок размером `size` байтов памяти, вызывая функцию `mm_malloc` (листинг 10.8). После проверки допустимости запроса (строки 8—9) программа распределения памяти должна откорректировать размер запрошенного блока, чтобы зарезервировать в нем участки для заголовков сверху и снизу и удовлетворить требованиям выравнивания по границе двойного слова. Строки 12—13 задают минимальный размер блока 16 байтов: восемь (`DSIZE`) байтов, чтобы удовлетворить требованиям выравнивания, и еще восемь (`OVERHEAD`) — для заголовков сверху и снизу. Для

запросов, превышающих восемь байтов (строка 15), общее правило состоит в том, чтобы сначала добавить дополнительные байты, а затем округлить до ближайшего большего числа, кратного восьми (DSIZE).

Листинг 10.8 Распределение памяти

```

1 void *mm_malloc(size_t size)
2 {
3     size_t asize;      /* установленный размер блока */
4     size_t extendsize; /* размер приращения динамической памяти,
                           если выделенный блок не подходит */
5     char *bp;
6
7     /* неправильные запросы игнорируются */
8     if (size <= 0)
9         return NULL;
10
11    /* корректируем размер блока, включая байты дополнения и выравнивания */
12    if (size <= DSIZE)
13        asize = DSIZE + OVERHEAD;
14    else
15        asize = DSIZE * ((size + (OVERHEAD) + (DSIZE-1)) / DSIZE);
16
17    /* поиск подходящего блока в списке свободных блоков */
18    if ((bp = find_fit(asize)) != NULL) {
19        place(bp, asize);
20        return bp;
21    }
22
23    /* подходящих блоков не найдено; получить дополнительную память
       и разместить блок */
24    extendsize = MAX(asize,CHUNKSIZE);
25    if ((bp = extend_heap(extendsize/WSIZE)) == NULL)
26        return NULL;
27    place(bp, asize);
28    return bp;
29 }
```

Как только программа распределения памяти откорректирует запрашиваемый размер, она просматривает список свободных блоков в поиске подходящего свободного блока (строка 18). Если подходящий блок будет найден, то программа распределения памяти разместит запрошенный блок и, возможно, отделит избыточную память (строка 19), а затем возвратит адрес вновь выделенного блока (строка 20).

Если программа распределения памяти не сможет найти подходящий блок, она расширяет пространство динамической памяти, добавляя новый свободный блок (строки 24—26), размещает запрошенный блок в новом свободном блоке, возможно,

разбивает блок (строка 27) и затем возвращает указатель на вновь распределенный блок (строка 28).

УПРАЖНЕНИЕ 10.8

Реализуйте функцию `find_fit` для простой программы распределения памяти, описанной в разд. 10.9.12.

```
static void *find_fit (size_t asize)
```

Решением должна быть программа, осуществляющая выбор первого подходящего блока в неявном списке свободных блоков.

УПРАЖНЕНИЕ 10.9

Реализуйте функцию `place` для демонстрационной программы распределения памяти.

```
static void place(void *bp, size_t asize)
```

Решением должна быть программа, размещающая запрошенный блок в начале свободного блока и разбивающая его только в случае, если размер избыточной памяти равняется или превышает минимальный размер блока.

10.9.13. Явные списки свободных блоков

Неявный список свободных блоков предоставляет в наше распоряжение простой способ определения некоторых основных понятий для программ распределения памяти. Однако поскольку время распределения линейно зависит от общего количества блоков динамической памяти, неявный список свободных блоков не целесообразно использовать в универсальной программе распределения памяти (хотя он мог бы быть вполне подходящим для специальных программ распределения памяти, если заранее известно, что количество блоков динамической памяти невелико).

Более приемлемый подход состоит в том, чтобы организовать свободные блоки в форме некоторой явной структуры данных. Поскольку по определению тело свободного блока не востребуется программой, то указатели, которые реализуют структуру данных, можно хранить в телах свободных блоков. Например, динамическая память может быть организована как двусвязный список свободных блоков, содержащий указатели `pred` (предшественник) и `succ` (преемник) в каждом свободном блоке, как показано на рис. 10.45.

Использование двусвязного списка вместо неявного свободного списка уменьшает время распределения памяти по методу первого подходящего блока: от линейно зависящего от общего количества блоков до линейно зависящего от количестве свободных блоков. Наряду с этим, время освобождения блока может быть либо линейным, либо постоянным, в зависимости от стратегии, которую мы выбираем при упорядочивании блоков в свободном списке.

При одном из подходов список обслуживается в соответствии с дисциплиной "последним пришел — первым обслужен" (LIFO, Last-In First-Out), при которой последний освобожденный блок становится в начало списка. При порядке обслуживания

LIFO и стратегии размещения по методу первого подходящего блока программа распределения памяти просматривает сначала самые последние использованные блоки. В этом случае освобождение блока может быть выполнено за постоянное время. Если используются граничные теги, то слияние блоков также может быть выполнено за постоянное время.

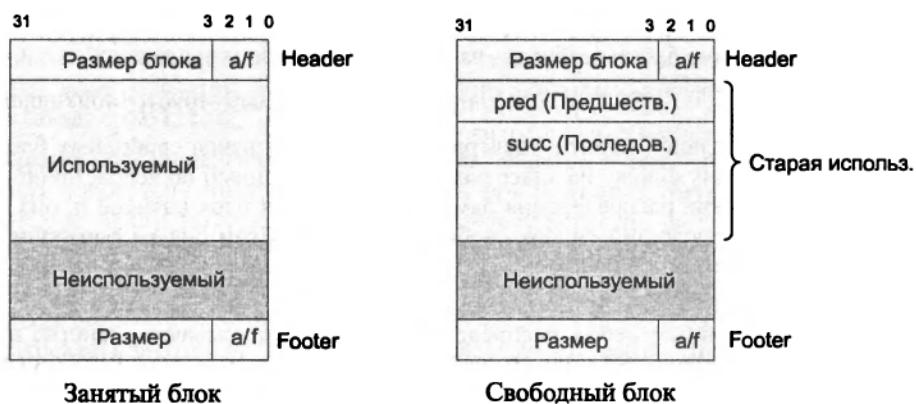


Рис. 10.45. Формат блоков динамической памяти при использовании двусвязных списков свободных блоков

Другой подход состоит в том, чтобы обслуживать список в порядке следования адресов (address order), когда адрес каждого блока в списке меньше чем адрес его преемника. В этом случае для освобождения блока требуется выполнить поиск соответствующего предшественника. Время поиска его местонахождения определяется линейной зависимостью от числа блоков. Компромиссный вариант, когда список обслуживается в порядке следования адресов по методу первого подходящего блока, отличается лучшим использованием памяти, чем обслуживание по правилу LIFO с использованием метода первого подходящего блока, и приближается по этому показателю к методу наиболее согласованного блока.

В общем случае недостаток явных списков заключается в том, что свободные блоки должны быть достаточно большими, чтобы содержать все необходимые указатели, а также заголовок и, возможно, заголовок снизу. Это приводит к увеличению минимального размера блока и, потенциально, к более высокой степени внутренней фрагментации.

10.9.14. Раздельные свободные списки

Как мы видели, программа распределения памяти, которая использует единственный связанный список свободных блоков, затрачивает для размещения блока время, линейно зависящее от числа свободных блоков. Распространенный подход, применяемый для сокращения времени размещения, известный как метод раздельной памяти (*segregated storage*), состоит в том, чтобы поддерживать несколько свободных списков, где каждый список содержит блоки примерно одного и того же размера (попрядка).

Общая идея основана на разбиении множества всех возможных размеров блоков на эквивалентные классы, называемые классами размеров (size class). Могут быть различные способы определения классов размеров. Например, можно было бы разбивать размеры блоков по степени числа два:

{1}, {2}, {3, 4}, {5–8}, …, {1025–2048}, {2049–4096}, {4097–∞}

Или можно было бы назначить каждому небольшому блоку класс размера, равный его размеру, а большие блоки разбивать на классы по степеням числа два:

{1}, {2}, {3}, …, {1023}, {1024}, …, {1025–2048}, {2049–4096}, {4097–∞}

Программа распределения памяти поддерживает массив списков свободных блоков, по одному свободному списку на класс размера, упорядоченный по возрастанию размера. Когда программе распределения памяти понадобится блок размера n , она просматривает соответствующий список свободных блоков. Если она не сможет найти блок, соответствующий предъявленному требованию, в одном списке, она просматривает следующий список и т. д.

В литературе по динамическому распределению памяти описываются многие варианты организации раздельных списков свободных блоков памяти, которые отличаются друг от друга тем, как организованы классы размеров, как производятся слияния блоков, как производятся запросы к операционной системе на выделение дополнительной динамической памяти, позволяют ли они разбивать блоки и т. п. Чтобы дать вам представление о том, что в подобных случаях можно, а что нельзя, мы опишем два из наиболее часто используемых подходов: *простое разделение памяти* (simple segregated storage) и *разделение с учетом размера* (segregated fits).

Простое разделение памяти

При простом разделении памяти список свободных блоков для каждого класса размера содержит одинаковые блоки, размер каждого из них равен размеру наибольшего элемента в классе. Например, если некоторый класс размера определен как {17–32}, то список свободных блоков для этого класса состоит только из блоков размера 32.

Чтобы распределить блок некоторого заданного размера, мы просматриваем соответствующий список свободных блоков. Если список не пуст, мы просто выделяем ему первый блок полностью. Свободные блоки никогда не подвергаются разбиению с тем, чтобы удовлетворить запрос на выделение памяти. Если список пуст, программа распределения памяти запрашивает от операционной системы порцию дополнительной памяти фиксированного размера (обычно кратный размеру страницы), делит эту порцию на блоки равного размера, и связывает блоки вместе, формируя новый список свободных блоков. Чтобы освободить блок, программа распределения памяти просто переставляет блок на первое место соответствующего списка свободных блоков.

Эта простая схема обладает множеством преимуществ. Распределение и освобождение блоков суть операции, обладающие высоким быстродействием и выполняющиеся за постоянное время. Далее, использование блоков одинаковых размеров, отсутствие необходимости разбивать и объединять блоки означает, что на манипулирование блоком потребуется незначительные непроизводительные затраты ресурсов. По-

скольку все блоки занимают участки памяти одинакового размера, размер занятого блока может быть легко определен по его адресу. Поскольку объединения не применяются, в заголовке блока не нужен флагок "распределен/свободен". Распределяемые таким образом блоки не требуют никаких заголовков, а в силу того, что они не подвергаются слиянию, отпадает необходимость в заголовках снизу. Поскольку операции выделения и освобождения памяти вставляют и удаляют блоки из начала списка свободных блоков, этот список должен быть всего лишь односвязным, а не двусвязным. В итоге в каждом блоке необходимым является единственное поле — указатель `succ` размером в одно слово, таким образом, минимальный размер блока составляет одно слово.

Существенный недостаток этой схемы заключается в том, что простая разделенная память подвержена внутренней и внешней фрагментации. Внутренняя фрагментация возможна ввиду того, что свободные блоки никогда не подвергаются разбиению. Хуже того, некоторые специальные методы могут вызвать недопустимую внешнюю фрагментацию, в силу того, что свободные блоки никогда не объединяются.

В целях борьбы с внешней фрагментацией исследователи предложили примитивную форму слияния блоков. Программа распределения памяти отслеживает число свободных блоков в каждом участке памяти, возвращенном операционной системой. Всякий раз, когда участок памяти состоит исключительно из свободных блоков, программа распределения памяти удаляет этот участок памяти из своего текущего класса размеров и делает его доступным для других классов размеров.

УПРАЖНЕНИЕ 10.10

Опишите один специальный метод распределения памяти, который приводит к серьезной внешней фрагментации при использовании программы выделения памяти, основанной на простом разделении памяти.

Разделение с учетом размера

Этот подход характерен тем, что программа выделения памяти поддерживает массив списков свободных блоков. Каждый список свободных блоков связан с классом размера и организован как некоторая разновидность явного или неявного списка. Каждый такой список потенциально может содержать блоки различного размера, и их размеры являются членами класса размера. Существует много вариантов программ распределения памяти в зависимости от методов разделения с учетом размера. Здесь мы опишем одну из простых версий такой программы.

Чтобы распределить блок памяти, мы определяем класс размера для запроса и производим поиск нужного блока по методу первого подходящего блока в списке свободных блоков соответствующего размера. Если мы находим такой блок, то мы производим его разбиение (при необходимости) и вставляем полученный фрагмент в соответствующий список свободных блоков. Если мы не сможем найти подходящий блок заданного размера, то мы просматриваем список свободных блоков следующего, большего класса размера. Мы повторяем эту процедуру до тех пор, пока не найдем подходящий блок. Если просмотр всех имеющихся списков свободных блоков не дает результата, то мы запрашиваем дополнительную динамическую память у операционной системы, размещаем блок в этом новом участке динамической памяти, и

помещаем остаток в класс наибольшего размера. Чтобы освободить блок, мы выполняем слияние и помещаем полученную порцию динамической памяти в соответствующий список свободных блоков.

Метод раздельного распределения с учетом размера блока используется во многих промышленных программах распределения памяти типа пакета программ malloc проекта GNU, поставляемого в составе стандартной библиотеки. Этот метод характеризуется и быстродействием, и эффективностью. Время поиска уменьшено за счет того, что поиск сосредоточен только на некоторой части, а не всей динамической памяти. Эффективность использования памяти может быть повышена благодаря тому интересному факту, что поиск по упрощенному методу первого подходящего блока в раздельном списке свободных блоков близок к поиску по методу наиболее согласованного блока в полной динамической памяти.

Метод близнецов

Метод близнецов (buddy system) представляет собой специальный случай раздельного поиска подходящего блока с учетом размера, где каждый класс размера есть степень числа два. Основная идея метода заключается в том, что для заданной динамической памяти размером 2^m слов мы поддерживаем отдельный список свободных блоков для каждого размера блока, равного 2^k , где $0 \leq k \leq m$. Запрашиваемые размеры блока округляются до наименьшей степени числа два, превосходящей размер блока. Первоначально имеется один свободный блок размером 2^m слов.

Чтобы выделить память для блока размером 2^k , мы находим первый доступный блок размера 2^j , такой, что $k \leq j \leq m$. Если $j = k$, то задача успешно решена. В противном случае мы рекурсивно разбиваем блок пополам до достижения $j = k$. Всякий раз, когда мы выполняем такое разбиение, каждая получающаяся половина, известная как близнец (buddy), помещается в соответствующий список свободных блоков. Чтобы освободить блок размером 2^k , мы последовательно объединяем свободные блоки-близнецы. Когда мы сталкиваемся с распределенным близнецом, мы прекращаем объединение.

Ключевым моментом в системах с близнецами является то, что для заданного адреса и размера блока легко вычислить адрес его близнеца. Например, блок размером 32 байта с адресом

xxx...x00000

имеет адрес близнеца

xxx...x10000

Другими словами, адрес блока и его близнеца различается в позиции всего лишь одного разряда.

Главное преимущество программы выделения памяти по методу близнецов заключается в быстром поиске и объединении. Главный недостаток состоит в том, что требование к размеру блока быть степенью числа два может вызвать существенную внутреннюю фрагментацию. По этой причине программы распределения памяти по методу близнецов не подходят для всех выполняемых задач. Однако для некоторых специфических приложений, где заранее известно, что размеры блоков являются сте-

пенями числа два, использование программ распределения памяти по методу близнецов может оказаться целесообразным.

10.10. Сборка мусора

С помощью программы явного распределения памяти, такой как пакет программ `malloc` на языке C, приложение выделяет и освобождает блоки динамической памяти, обращаясь с этой целью к функциям `malloc` и `free`. Это позволяет приложению высвобождать любые ранее распределенные блоки, которые ему в дальнейшем не потребуются.

Неудачное освобождение распределенных блоков — обычная ошибка в программировании. Например, рассмотрим функцию на языке C, которая распределяет блок временной памяти каждый раз, когда она вызывается (листинг 10.9).

Листинг 10.9. Распределение блока временной памяти

```
1 void garbage()
2 {
3     int *p = (int *)malloc(15213);
4
5     return; /* в этой точке массив p становится мусором */
6 }
```

Поскольку `p` больше не нужно программе, эта память должна быть освобождена перед тем, как управление будет возвращено из функции `garbage`. К сожалению, случилось так, что программист забыл освободить этот блок. Он остается занятым в течение всего жизненного цикла этой программы, напрасно занимая пространство динамической памяти, которое могло бы быть использовано для удовлетворения последующих запросов на выделение памяти.

Сборщик мусора (*garbage collector*) — это программа, связанная с распределением динамической памяти, которая автоматически освобождает те из занятых блоков, которые больше не будут востребованы программой. Такие блоки известны как мусор (*garbage*), откуда и происходит термин "сборщик мусора". Процесс автоматического восстановления динамической памяти известен как сборка мусора (*garbage collection*). В системах, которые поддерживают сборку мусора, приложения явным образом распределяют блоки динамической памяти, но никогда явно их не освобождают. В контексте программ на языке C это означает, что приложение вызывает функцию `malloc`, но никогда не вызывает функцию `free`. Вместо этого сборщик мусора периодически опознает блоки мусора и делает соответствующие вызовы функции `free`, возвращая такие блоки в список свободных блоков.

Сборка мусора восходит к системам *Lisp*, разработанным Джоном МакКарти (John McCarthy) в Массачусетском технологическом институте еще в начале шестидесятых годов прошлого столетия. Это важная часть современных языков программирования типа Java, ML, Perl и Mathematica, и тема эта до сих пор остается актуальной и важной областью исследований. В литературе можно найти описания удивительно

большого числа подходов к сборке мусора. В своих рассуждениях мы ограничимся рассмотрением оригинального алгоритма *Mark&Sweep*, предложенного МакКарти, который интересен прежде всего тем, что он может быть построен на базе существующего пакета программ *malloc*. Этот алгоритм дает возможность организовать сборку мусора в программах на языках C++ и C.

10.10.1. Основные принципы функционирования программ сборки мусора

Программа сборки мусора (или сборщик мусора) рассматривает память как направленный граф достижимости (reachability graph) в форме, представленной на рис. 10.46. Узлы графа делятся на множество корневых узлов (root node) и множество узлов динамической памяти (heap nodes). Каждый узел динамической памяти соответствует распределенному блоку динамической памяти. Ориентированное ребро $p \rightarrow q$ означает, что некоторая ячейка в блоке p содержит указатель на некоторую ячейку в блоке q . Корневой узел соответствует ячейке за пределами динамической памяти, которая содержит указатель на ячейку, расположенную в динамической памяти. Такие ячейки могут быть регистрами, переменными в стеке или глобальными переменными в области данных чтения и записи виртуальной памяти.

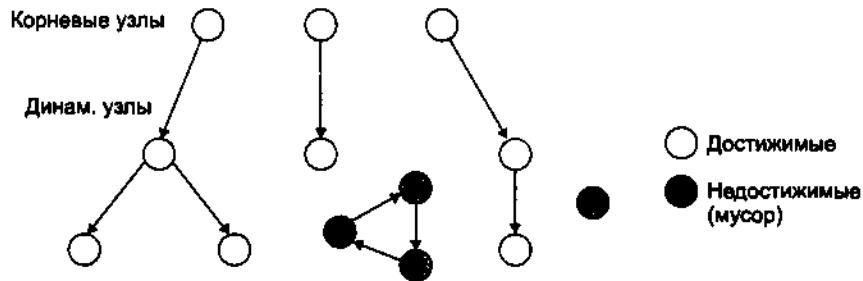


Рис. 10.46. Программа сборки мусора представляет память как ориентированный граф

Мы говорим, что узел p достижим (reachable), если существует направленный путь от любого корневого узла к узлу p . В любой момент времени недостижимые узлы соответствуют мусору, который никогда не может быть снова использован приложением. Роль сборщика мусора заключается в том, чтобы поддерживать некоторое представление графа достижимости и периодически отыскивать недостижимые узлы, освобождая их и возвращая в список свободных блоков.

Сборщики мусора для языков, подобных ML и Java, которые осуществляют строгий контроль над тем, как приложения создают и используют указатели, могут поддерживать точное представление графа достижимости и таким образом могут возвратить в динамическую память весь мусор. Однако сборщики для языков программирования, подобных С и C++, в общем не могут поддерживать точные представления графа достижимости. Такие сборщики называются *консервативными сборщиками мусора* (conservative garbage collectors). Они консервативны в том смысле, что каждый

достижимый блок опознается правильно как достижимый, тогда как некоторые недостижимые узлы могут быть неправильно идентифицированы как достижимые.

Сборщики могут предоставлять свои услуги по требованию, или же они могут исполняться как отдельные потоки параллельно с приложением, непрерывно модифицируя граф достижимости и возвращая памяти мусор. Например, рассмотрим, как можно было бы включить консервативный сборщик для программ на языке C в существующий пакет malloc, по схеме, показанной на рис. 10.47.

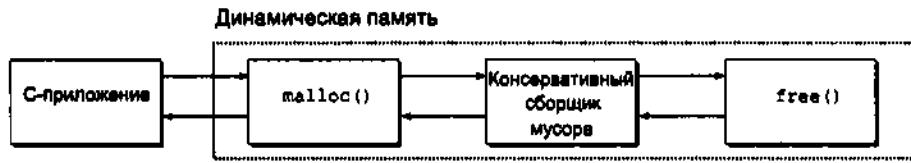


Рис. 10.47. Встраивание консервативного сборщика мусора в пакет malloc языка C

Приложение вызывает функцию `malloc` обычным способом всякий раз, когда требуется получить пространство памяти из динамической памяти. Если `malloc` не сможет найти свободный блок, который соответствует запрашиваемому размеру, то она вызывает программу сборки мусора в надежде, что та возвратит некоторое пространство, занимаемое мусором, в список свободных блоков. Программа сборки идентифицирует блоки мусора и возвращает их в динамическую память, вызывая функцию `free`. Характерная особенность этого метода заключается в том, что вместо приложения функцию `free` вызывает сборщик мусора. Когда сборщик возвращает управление, `malloc` снова предпринимает попытку найти свободный блок соответствующего размера. Если эта попытка неудачна, то сборщик может запросить у операционной системы дополнительную память. В конечном счете, функция `malloc` возвращает указатель на запрашиваемый блок (если попытка была успешной) или указатель `NULL` (в случае неудачи).

10.10.2. Программы сборки мусора, реализующие алгоритм Mark&Sweep

Работа сборщика мусора `Mark&Sweep` происходит в два этапа: *этап разметки* (*mark phase*), на котором отмечаются все достижимые и распределенные потомки корневых узлов, и *этап очистки* (*sweep phase*), на котором освобождается каждый неотмеченный распределенный блок. Как правило, один из резервных младших разрядов в заголовке блока используется для указания, отмечен ли блок или нет.

В нашем описании алгоритма `Mark&Sweep` мы будем использовать следующие функции, в которых переменная `ptr` определена как `typedef char *ptr`:

- `ptr isPtr(ptr p)` — возвращает указатель `b` на начало блока, если `p` указывает на некоторое слово в распределенном блоке, в противном случае возвращает `NULL`;
- `int blockMarked(ptr b)` — возвращает значение `true`, если блок `b` уже отмечен;
- `int blockAllocated(ptr b)` — возвращает значение `true`, если блок `b` занят;
- `void markBlock(ptr b)` — отмечает блок `b`;

- int length(b) — возвращает длину блока b в словах (исключая заголовок);
- void unmarkBlock(ptr b) — изменяет состояние блока b с отмеченного на неотмеченное;
- ptr nextBlock(ptr b) — возвращает преемника блока b в динамической памяти.

На этапе разметки функция mark, показанная в листинге 10.10, вызывается один раз для каждого корневого узла. Если p не указывает на распределенный и немаркированный блок динамической памяти, то функция mark немедленно возвращает управление. В противном случае, она помечает блок и рекурсивно вызывает сама себя для каждого слова в блоке. При каждом вызове функция mark помечает все неотмеченные и достижимые потомки некоторого корневого узла. В конце этапа разметки каждый распределенный блок, который не был отмечен, гарантированно недостижим и, следовательно, является мусором, который может быть оприходован на этапе очистки.

Этап очистки представляет собой единственный вызов функции sweep, показанной в листинге 10.10. Функция sweep просматривает каждый блок динамической памяти, освобождая все неотмеченные распределенные блоки (т. е. мусор), которые ей встречаются.

Листинг 10.10. Разметка и очистка

```
void mark(ptr p) {
    if ((b = isPtr(p)) == NULL)
        return;
    if (blockMarked(b))
        return;
    markBlock(b);
    len = length(b);
    for (i=0; i < len; i++)
        mark(b[i]);
    return;
}

void sweep(ptr b, ptr end) {
    while (b < end) {
        if (blockMarked(b))
            unmarkBlock(b);
        else if (blockAllocated(b))
            free(b);
        b = nextBlock(b);
    }
    return;
}
```

На рис. 10.48 в графическом виде представлена последовательность действий при исполнении алгоритма Mark&Sweep для динамической памяти небольшого размера.

Границы блоков обозначены утолщенными линиями. Каждый квадратик соответствует слову памяти. Каждый блок имеет заголовок размером в одно слово, который может быть либо отмечен, либо не отмечен.

В исходном положении динамическая память на рис. 10.48 состоит из шести распределенных блоков, ни один из которых не отмечен. Блок 3 содержит указатель на блок 1. Блок 4 содержит указатели на блоки 3 и 6. Корневой блок указывает на блок 4. После этапа разметки блоки 1, 3, 4 и 6 становятся отмеченными, поскольку они достижимы из корневого блока. Блоки 2 и 5 не отмечены, поскольку они недостижимы. После этапа очистки два недостижимых блока возвращены в список свободных блоков.

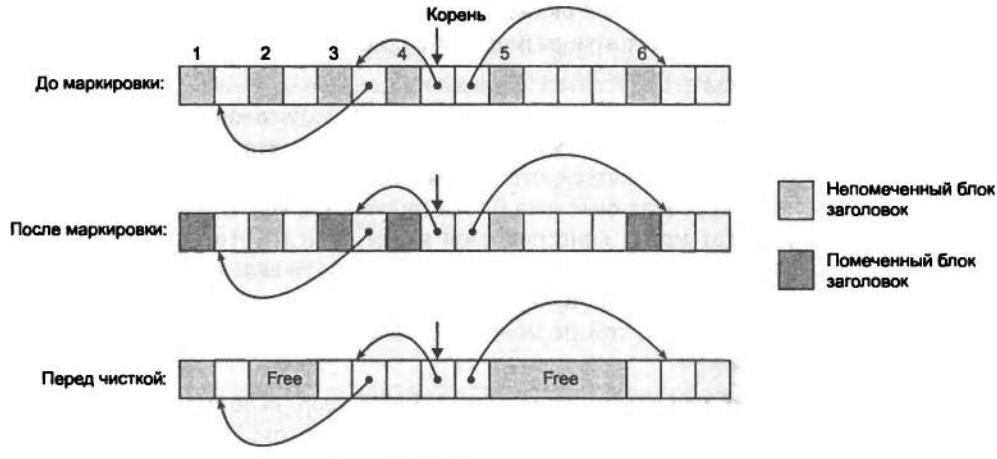


Рис. 10.48. Динамическая память

10.10.3. Консервативный алгоритм Mark&Sweep для программ на С

Алгоритм Mark&Sweep целесообразно использовать в собирающих мусор программах на языке C, поскольку он работает на одном месте, не перемещая никаких блоков. В то же время язык C предлагает несколько интересных подходов к реализации функции `isPtr`.

Прежде всего, С не связывает ячейки памяти с какой-либо информацией о типе. Таким образом, нет никакого очевидного способа, позволяющего функции `isPtr` определять, является ли ее входной параметр `p` указателем или нет. Во-вторых, даже если бы мы знали, что `p` есть указатель, не существует никакого явного способа, позволяющего функции `isPtr` определять, указывает ли `p` на некоторую ячейку в полезной части размещенного блока.

Одно из решений последней проблемы состоит в том, чтобы представить множество распределенных блоков в виде сбалансированного двоичного дерева, чтобы все блоки в левом поддереве были расположены в меньших адресах, а все блоки в правом

поддереве были расположены в больших адресах. Как показано на рис. 10.49, это потребует двух дополнительных полей *left* (левое) и *right* (правое) в заголовке каждого размещенного блока. Каждое поле указывает на заголовок некоторого размещенного блока.

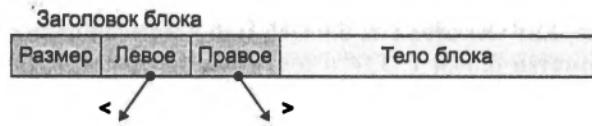


Рис. 10.49. Левые и правые указатели в сбалансированном дереве размещенных блоков

Функция *isPtr* (*ptr p*) выполняет двоичный поиск на дереве распределенных блоков. На каждом шаге она, используя размер поля, содержащегося в заголовке блока, определяет, попадает ли *p* в пределы блока.

Метод сбалансированного дерева правилен в том смысле, что он гарантирует, что будут отмечены все узлы, достижимые из начального блока. Это необходимая гарантия, поскольку можно не сомневаться, что самим пользователям приложений отнюдь не понравится, если распределенные ими блоки будут возвращены в список свободных блоков. Однако этот метод консервативен в том смысле, что он может неправильно отметить блоки, которые фактически недостижимы, и вследствие этого может произойти сбой при освобождении мусора. При этом не затрагивается правильность работы прикладных программ, тем не менее это может привести к излишней внешней фрагментации.

Основная причина того, что сборщики мусора *Mark&Sweep*, используемые программами на С, должны быть консервативными, заключается в том, что язык С не ассоциирует ячейки памяти с информацией о типе данных. Таким образом, скалярные типы, подобные *int* или *float*, могут имитировать указатели. Например, предположим, что некоторый достижимый распределенный блок содержит величину типа *int* в своей полезной части, которая случайно соответствует адресу в полезной части некоторого другого распределенного блока *b*. Не существует никакого способа, посредством которого сборщик мусора может определить, что это данные типа *int*, а вовсе не указатель. Таким образом, программа распределения памяти должна обязательно отметить блок *b* как достижимый, в то время как на самом деле он таковым не является.

10.11. Часто встречающиеся ошибки

Управление виртуальной памятью и ее использование могут оказаться трудной задачей для программистов на языке С, чреватой многочисленными ошибками. Ошибки, вызванные неправильным использованием памяти, относятся к числу наиболее неприятных, поскольку они часто проявляются не сразу, а на значительном удалении во времени и в пространстве от источника их возникновения. Стоит записать неправильные данные по неправильному адресу, и ваша программа может прокручиваться часами, прежде чем она, наконец, прекратит выполняться в какой-то другой отдален-

ной части. Мы завершаем наше изучение виртуальной памяти обсуждением нескольких общих ошибок, связанных с неправильным использованием памяти.

10.11.1. Разыменование плохих указателей

Как следует из разд. 10.7.2, в виртуальном адресном пространстве процесса имеются большие дыры, которые не отображаются ни на какие имеющие смысл данные. Если мы пытаемся разыменовывать указатель на место одной из этих дыр, операционная система прекратит выполнение нашей программы, инициируя исключительную ситуацию по нарушению сегментации. Кроме того, некоторые области виртуальной памяти доступны только для чтения. Попытка осуществить запись в одну из этих областей приводит к исключительной ситуации по нарушению защиты.

Известный пример разыменования плохого указателя представляет собой пример классической ошибки при выполнении функции `scanf`. Предположим, что мы хотим использовать `scanf`, чтобы прочитать целое число из `stdin` в некоторую переменную. Это делается правильно, когда в функцию `scanf` передается строка формата и адрес переменной:

```
scanf("%d", &val)
```

Однако программисты, не имеющие достаточного опыта работы с языком С (и опытные также!), просто передают содержимое `val` вместо ее адреса:

```
scanf("%d", val)
```

В этом случае функция `scanf` интерпретирует содержимое `val` как адрес и будет пытаться записывать слово по этому адресу. В лучшем случае, программа заканчивается немедленно, возбуждая исключительную ситуацию. В худшем случае, содержимое `val` соответствует некоторой допустимой для чтения/записи области виртуальной памяти, и мы ошибочно переписываем нужную запись в памяти, что обычно намного позже приводит к непредсказуемым и, зачастую, пагубным последствиям.

10.11.2. Чтение неинициализированной области памяти

В то время как ячейки памяти `.bss` (такие, как неинициализированные глобальные переменные языка С) всегда инициализируются загрузчиком нулями, это не относится к динамической памяти. Распространенная ошибка состоит в том, что заранее предполагается, что динамическая память инициализируется нулями (листинг 10.11).

Листинг 10.11. Распространенная ошибка

```
1 /* возвращает у = Ax */
2 int *matvec(int **A, int *x, int n)
3 {
4     int i, j;
```

```

6     int *y = (int *)Malloc(n * sizeof(int));
7
8     for (i = 0; i < n; i++)
9         for (j = 0; j < n; j++)
10            y[i] += A[i][j] * x[j];
11
12 }

```

В этом примере программист неправильно предположил, что вектор *y* проинициализирован нулевыми значениями. В правильной реализации следует обнулить *y[i]* кодами в строках 8 и 9 или использовать функцию *calloc*.

10.11.3. Переполнение буфера стека

Как следует из разд. 3.13, программа возбуждает ошибку переполнения буфера (buffer overflow bug), если она делает запись в целевой буфер в стеке без проверки размера входной строки. Например, приводимая функция (листинг 10.12) вызывает ошибку переполнения буфера, поскольку функция *gets* копирует в буфер строку произвольной длины. Чтобы не допустить этого, мы должны воспользоваться функцией *fgets*, которая ограничивает размер входной строки.

Листинг 10.12. Переполнение буфера стека

```

1 void bufoverflow()
2 {
3     char buf[64];
4
5     gets(buf); /* здесь ошибка переполнения буфера стека */
6     return;
7 }

```

10.11.4. Предположение о размере указателей и объектов

Одна из распространенных ошибок вызывается предположением о том, что указатели на объекты имеют тот же размер, что и объекты, на которые они указывают (листинг 10.13).

Листинг 10.13. Ошибка оценки размера

```

1 /* образуем массив pxm */
2 int **makeArray1(int n, int m)
3 {
4     int i;

```

```

5     int **A = (int **) Malloc (n * sizeof(int));
6
7     for (i = 0; i < n; i++)
8         A[i] = (int *) Malloc (m * sizeof(int));
9     return A;
10 }

```

В рассматриваемом случае цель заключается в том, чтобы построить массив из *n* указателей, каждый из которых указывает на массив из *m* целых чисел. Однако поскольку программист в строке 5 написал `sizeof (int)` вместо `sizeof (int *)`, программа фактически создает некоторый массив целых чисел. Эта программа будет прекрасно работать на машинах, где целые числа и указатели на целые числа одного и того же размера.

Но если мы запустим эту программу на такой машине как Alpha, где указатель больше чем `int`, то цикл в строках 7 и 8 попытается сделать запись за пределами границ массива. Поскольку одно из этих слов, по-видимому, будет заголовком снизу граничного тега распределенного блока, мы не сможем обнаружить ошибку, пока не освободим блок в этой программе гораздо позже, в той точке, где программа объединения в программе распределения памяти прекратит выполняться по непонятной причине. Это пример коварного "действия на расстоянии", который так характерен для ошибок программирования, связанных с неправильным использованием памяти.

10.11.5. Ошибки занижения или завышения на единицу

Ошибки занижения или завышения на единицу числа подсчитываемых объектов представляют собой еще один распространенный пример ошибок наложения записей (листинг 10.14).

Листинг 10.14. Ошибка наложения записей

```

1 /* образуем массив pxm */
2 int **makeArray2(int n, int m)
3 {
4     int i;
5     int **A = (int **) Malloc(n * sizeof(int));
6
7     for (i = 0; i <= n; i++)
8         A[i] = (int *) Malloc(m * sizeof(int));
9     return A;
10 }

```

Это еще одна версия программы из предыдущего раздела. Здесь в строке 5 мы создали массив из *n* указателей, но затем в строках 7 и 8 попытались инициализировать

$n + 1$ его элементов в процессе наложения записей в памяти, которая следует за массивом A.

10.11.6. Ссылка на указатель вместо объекта

Если не отнеслись с должным вниманием к правилам старшинства и ассоциативности операций в C, то это может привести к некорректному использованию указателя вместо объекта, на который он указывает. Например, рассмотрим следующую функцию, назначение которой состоит в том, чтобы удалять первый элемент из бинарной динамической памяти, содержащей $*size$ элементов, и затем реорганизовать оставшиеся $*size - 1$ элементов (листинг 10.15).

Листинг 10.15. Ссылка на указатель

```

1 int *binheapDelete(int **binheap, int *size)
2 {
3     int *packet = binheap[0];
4
5     binheap[0] = binheap[*size - 1];
6     size--; /* здесь должно быть (*size)-- */
7     heapify(binheap, *size, 0);
8     return(packet);
9 }
```

Строка 6 предназначается для того, чтобы уменьшить целочисленное значение, на которое указывает $(*size) --$. Однако, ввиду того, что одноместные операции $--$ и $*$ имеют одно и то же старшинство и ассоциированы справа налево, программа в строке 6 фактически уменьшает сам указатель вместо целочисленного значения, на которое он указывает. Если нам повезет, то программа прекратит выполняться немедленно, но вероятнее всего нам придется искать ошибку, когда эта программа на более поздней стадии исполнения выдаст неправильный ответ. Отсюда можно сделать вывод, что следует использовать круглые скобки всякий раз, когда имеется сомнение относительно старшинства и ассоциативности. Например, в строке 6 мы могли бы четко сформулировать наше намерение, используя выражение $(*size) --$.

10.11.7. Неправильное понимание арифметических операций над указателями

Еще одна распространенная ошибка заключается в том, что иногда забывают, что арифметические операции над указателями выполняются в единицах, которые имеют размер тех объектов, на которые они указывают, а это не обязательно байты. Например, назначение следующей функции состоит в том, чтобы просмотреть массив целочисленных значений и возвратить указатель на первое вхождение величины val (листинг 10.16).

Листинг 10.16. Ошибка в арифметических операциях

```

1 int *search(int *p, int val)
2 {
3     while (*p && *p != val)
4         p += sizeof(int); /* здесь должно быть p++ */
5     return p;
6 }
```

Однако, ввиду того, что на каждом проходе цикла в строке 4 указатель увеличивается на четыре (количество байтов в целом числе), функция ошибочно просматривает только каждое четвертое целое число в массиве.

10.11.8. Ссылки на несуществующие переменные

Неопытные программисты на языке С, которые не понимают устройства стека, иногда будут ссылаться на локальные переменные, которые больше уже не существуют, как в следующем примере (листинг 10.17).

Листинг 10.17. Ссылка на локальную переменную

```

1 int *stackref ()
2 {
3     int val;
4
5     return &val;
6 }
```

Эта функция возвращает указатель (пусть это будет *p*) на локальную переменную в стеке и затем сдвигает указатель на вершину стека. Хотя указатель *p* указывает на допустимый адрес памяти, однако он уже не указывает на правильную переменную. Когда позже эта программа будет обращаться к другим функциям, память будет многократно использована для размещения стековых фреймов этих функций. Если затем в данной программе **p* получит некоторое новое значение, то это может фактически изменять вход стекового фрейма другой функции, с потенциально катастрофическими и непредсказуемыми последствиями.

10.11.9. Ссылка на данные в свободных блоках динамической памяти

Подобная ошибка возникает при ссылке на данные в блоках динамической памяти, которые уже были освобождены. Например, рассмотрим следующий случай, где в строке 6 размещается целочисленный массив *x*, затем в строке 10 освобождается блок *x*, а позже на него ссылаются в строке 14 (листинг 10.18).

Листинг 10.18. Ошибка ссылки

```

1 int *heapref(int n, int m)
2 {
3     int i;
4     int *x, *y;
5
6     x = (int *)Malloc(n * sizeof(int));
7
8     /* ... */ / здесь производятся другие вызовы функций malloc и free */
9
10    free(x);
11
12    y = (int *)Malloc(m * sizeof(int));
13    for (i = 0; i < m; i++)
14        y[i] = x[i]++; /* оказывается, что x[i] - слово в свободном блоке */
15
16    return y;
17 }
```

В зависимости от того, какова последовательность вызовов функций `malloc` и `free`, осуществляемых в промежутке между строками 6 и 10, при ссылке программы на `x[i]` в строке 14 массив `x` мог бы быть некоторой частью другого блока, размещенного в динамической памяти, и мог быть перезаписан. Как и в случае многих ошибок, вызванных неправильным использованием памяти, данная ошибка проявится в программе на более поздней стадии, когда мы заметим, что значения в `у` запорчены.

10.11.10. Представление об утечках в памяти

Утечки в памяти можно сравнить безжалостными молчаливыми убийцами, которые появляются, когда программисты неизбежно создают мусор в динамической памяти, забывая освобождать распределенные блоки. Например, следующая далее функция выделяет блок `x` в динамической памяти и затем возвращает управление, не освобождая его (листинг 10.19).

Листинг 10.19. Утечки в памяти

```

1 void leak(int n)
2 {
3     int *x = (int *)Malloc(n * sizeof(int));
4
5     return; /* в этом месте x становится мусором */
6 }
```

Если функцию `leak` вызывать часто, то динамическая память постепенно заполнится мусором, который заполнит полностью все виртуальное адресное пространство.

Утечки памяти имеют особенно тяжелые последствия для таких программ, например, как демоны и серверы, исполнение которых никогда не должно прерываться.

10.12. Сводка некоторых ключевых понятий, связанных с виртуальной памятью

В этой главе мы ознакомились с тем, как работает виртуальная память, как она используется системой для исполнения таких функций, как загрузка программ, отображение совместно используемых библиотек и предоставление процессам закрытых защищенных адресных пространств. Кроме того, мы рассмотрели множество способов как правильного, так и неправильного использования виртуальной памяти прикладными программами.

Главным уроком является то, что при всем том, что виртуальная память предоставляется системой автоматически, она является конечным ресурсом. Как нам удалось выяснить при изучении программ распределения динамической памяти, управление виртуальными ресурсами памяти требует достижения искусственных компромиссов в отношении расхода пространства и времени. Другой ключевой урок состоит в том, что в программах на языке C очень просто допустить ошибки, обусловленные неправильным обращением с памятью. Неправильные значения указателя, освобождение и без того свободных блоков, неправильное приведение типов и некорректные арифметические операции над указателями, а также перезапись структур динамической памяти — это только некоторые из множества путей, которые приводят к ошибкам в программах. По сути дела, серьезные последствия ошибок, связанных с неправильным использованием памяти, стали важным стимулом в разработке языка Java, который существенно ограничивает управление доступа к виртуальной памяти, не позволяя получать адресов переменных и сохраняя за собой полный контроль над программой динамического распределения памяти.

10.13. Резюме

Виртуальная память представляет собой абстракцию оперативной памяти. Процессоры, которые поддерживают виртуальную память, ссылаются на оперативную память, используя форму косвенной адресации, известную как виртуальная адресация. Процессор генерирует виртуальные адреса, которые транслируются в физические адреса перед тем, как начнется работа с оперативной памятью. Преобразование адресов виртуального адресного пространства в адреса физического адресного пространства требует тесного взаимодействия между аппаратными и программными средствами. Специально разработанные аппаратные средства преобразуют виртуальные адреса, используя таблицы страниц, содержимое которых поддерживается операционной системой.

Виртуальная память служит базой для трех важных системных средств. Прежде всего, это автоматическое кэширование только что использованного содержимого виртуального адресного пространства, хранимого на диске и в оперативной памяти. Блок

в виртуальном кэше памяти называется страницей. Ссылка на страницу, хранящуюся на диске, вызывает ошибку обращения к отсутствующей странице, которая передается обработчику ошибок в операционной системе. Обработчик ошибок копирует страницу из диска в кэш в оперативной памяти и в случае необходимости записывает на диск подлежащую удалению страницу. Во-вторых, при наличии виртуальной памяти упрощается управление памятью, которое, в свою очередь, упрощает связывание, совместное использование данных различными процессами, распределение памяти для процессов, а также загрузку программы. Наконец, наличие виртуальной памяти упрощает защиту памяти путем включения разряда защиты в каждый элемент таблицы страниц.

Процесс преобразования адреса должен быть интегрирован с работой всех аппаратных кэш в системе. Большинство элементов таблицы страниц расположено в кэше L1, но увеличение стоимости доступа к элементам таблицы страниц из L1 обычно нивелируется кэшированием элементов таблицы страниц, расположенной на плате и называемой буфером TLB.

Современные системы инициализируют участки виртуальной памяти, связывая их с участками файлов на диске — процесс, известный как отображение в памяти. Отображение в памяти представляет собой эффективный механизм для совместного использования данных, создания новых процессов, и для загрузки программ. Приложения могут самостоятельно создавать и удалять области виртуального адресного пространства, используя функцию `mmap`. Однако большинство программ полагается на такую программу распределения динамической памяти памяти, как `malloc`, которая управляет памятью в области виртуального адресного пространства, называемого динамической памятью. Программы динамического распределения памяти суть приложения на уровне с выходом на системный уровень, они непосредственно управляют памятью независимо от системы типов. Программы распределения памяти бывают двух видов. Явные программы распределения памяти требуют, чтобы приложение явно освобождало свои блоки памяти. Неявные программы распределения памяти (сборщики мусора) освобождают все неиспользуемые и недостижимые блоки автоматически.

Управление виртуальной памятью и ее использование для программистов, работающих на языке C, является трудной задачей, чреватой всевозможными ошибками. К распространенным ошибкам относятся неправильное разыменование указателей, чтение неинициализированной памяти, доведение буфера стека до переполнения, предположение о том, что указатели и объекты, на которые они указывают, имеют один и тот же размер, ссылки на указатель вместо ссылок на объект, на который они указывают, непонимание сути арифметических операций над указателями, ссылки на несуществующие переменные и создание условий для утечки памяти.

Библиографические замечания

Килбурн (Kilburn) и его коллеги опубликовали первое описание виртуальной памяти [42]. Тексты описания архитектуры содержат дополнительные детали о роли аппаратных средств в организации виртуальной памяти [33]. Тексты операционных систем содержат дополнительную информацию о роли операционной системы [70, 83, 75].

В 1968 Кнут (Knuth) написал классическую работу о выделении памяти [43]. С тех пор в этой области появилось огромное количество работ. Вилсон (Wilson), Джонстон (Johnstone), Нилай (Neely) и Болз (Boles) написали прекрасный обзор, в котором дали оценку производительности программ явного распределения памяти [88]. Основным содержанием этих публикаций является производительность программ распределения памяти и выбор различных стратегий, реализованных в этих программах. Эта книга стала естественным продолжением указанного ранее обзора. Джонс (Jones) и Линс (Lins) в [37] предложили всесторонний обзор проблемы сборки мусора. Керниган (Kernighan) и Ричи (Ritchie) [40] представили законченный программный код для простой программы распределения памяти, основанный на использовании явного списка свободных блоков с фиксацией размера блока и указателя на преемника в каждом свободном блоке. Этот программный код интересен тем, что он использует слияния для устранения большого количества сложных арифметических операций над указателями, но это достигается за счет отказа от постоянного времени исполнения в операциях освобождения в пользу линейной зависимости времени исполнения.

По адресу www.cs.colorado.edu/~zorn/DSA.html можно найти удобный царновский ресурс Dynamic Storage Allocation Repository (Сборник материалов по динамическому выделению памяти). Он включает разделы по средствам отладки и реализации программ сборки мусора и функций malloc/free.

Задачи для домашнего решения

УПРАЖНЕНИЕ 10.11 ◆

В предлагаемой ниже последовательности упражнений вы должны показать, как система памяти из примера в разд. 10.6.4 преобразует виртуальный адрес в физический адрес и получает доступ к кэш. Для заданного виртуального адреса укажите доступный элемент буфера TLB, физический адрес и возвращаемое значение байта кэша. Покажите, нужен ли буфер TLB, имеет ли место обращение к отсутствующей странице и происходит ли потеря кэш. Если кэш отсутствует, сделайте отметку "—" в графе возвращаемого байта кэш). В случае ошибки обращения к отсутствующей странице сделайте в графе PPN отметку "—" и оставьте последние пункты незаполненными.

Виртуальный адрес 0x027c:

- формат виртуального адреса;
- преобразование адреса;

Параметр	Значение
VPN	
Индекс TLB	
Ter TLB	
TLB имеется? (Да/Нет)	

(окончание)

Параметр	Значение
Ошибка страницы? (Да/Нет)	
PPN	

- формат физического адреса;
- ссылка на физическую память.

Параметр	Значение
Байт смещения	
Индекс в кэш	
Тег кэша	
Кэш имеется? (Да/Нет)	
Байт, возвращаемый кэш	

УПРАЖНЕНИЕ 10.12 ◆

Повторите упр. 10.11 для следующего адреса.

Виртуальный адрес 0х03a9:

- формат виртуального адреса;
- преобразование адреса;

Параметр	Значение
VPN	
Индекс TLB	
Тег TLB	
TLB имеется? (Да/Нет)	
Ошибка страницы? (Да/Нет)	
PPN	

- формат физического адреса;
- ссылка на физическую память.

Параметр	Значение
Байт смещения	
Индекс в кэш	
Тег кэша	
Кэш имеется? (Да/Нет)	
Байт, возвращаемый кэш	

УПРАЖНЕНИЕ 10.13 ◆

Повторите задание 10.11 для следующего адреса.

Виртуальный адрес 0x0040:

- формат виртуального адреса;
- преобразование адреса;

Параметр	Значение
VPN	
Индекс TLB	
Тег TLB	
TLB имеется? (Да/Нет)	
Ошибка страницы? (Да/Нет)	
PPN	

- формат физического адреса;
- ссылка на физическую память.

Параметр	Значение
Байт смещения	
Индекс в кэш	
Тег кэша	
Кэш имеется? (Да/Нет)	
Байт, возвращаемый кэш	

УПРАЖНЕНИЕ 10.14 ♦♦

Для заданного входного файла `hello.txt`, который содержит только строку "Hello, world! \n", напишите программу на С, использующую функцию `ptrap` для изменения содержимого `hello.txt` на "Jello, world! \n".

УПРАЖНЕНИЕ 10.15 ♦

Определите размер блока и код в заголовке, исходя из следующей ниже последовательности запросов функции `malloc`. Предварительные условия:

1. Программа распределения памяти поддерживает выравнивание по границе двойного слова и использует неявный список свободных блоков с форматом блока, данном на рис. 10.38.
2. Размеры блока округлены до ближайшего большего числа, кратного восьми байтам.

Запрос	Размер блока (байт, десятичное)	Заголовок блока (шестнадцатеричное)
<code>malloc(3)</code>		
<code>malloc(11)</code>		
<code>malloc(20)</code>		
<code>malloc(21)</code>		

УПРАЖНЕНИЕ 10.16 ♦

Определите минимальный размер блока для каждой из следующих комбинаций требований к выравниванию и формату блока. Предварительные условия: явный список свободных блоков, четырехбайтовые указатели `pred` и `succ` в каждом свободном блоке, полезное пространство нулевого размера не допускается, и заголовки и поля признаков хранятся в четырехбайтовых словах.

Выравнивание	Распределенный блок	Свободный блок	Минимальный размер блока (байт)
Одинарное слово	Заголовки сверху и снизу	Заголовки сверху и снизу	
Одинарное слово	Заголовок сверху, но без заголовка снизу	Заголовки сверху и снизу	
Двойное слово	Заголовки сверху и снизу	Заголовки сверху и снизу	
Двойное слово	Заголовок сверху, но без заголовка снизу	Заголовки сверху и снизу	

УПРАЖНЕНИЕ 10.17 ♦♦♦

Внесите изменения в версию программы распределения памяти из разд. 10.9.12, которая осуществляла бы поиск по методу следующего подходящего блока вместо поиска по методу первого подходящего блока.

УПРАЖНЕНИЕ 10.18 ♦♦♦

Программа распределения памяти из разд. 10.9.12 требует и заголовка сверху, и заголовка снизу для каждого блока, чтобы выполнить объединение блоков за постоянное время. Измените программу распределения памяти так, чтобы свободные блоки требовали наличия заголовка сверху и заголовка снизу, но выделенные блоки требуют только заголовка сверху.

УПРАЖНЕНИЕ 10.19 ♦

Ниже приведены три группы высказываний, касающихся управления памятью и сборкой мусора. В каждой группе только одно высказывание является истинным. Ваша задача состоит в том, чтобы указать, какое именно высказывание истинно.

1. • В методе близнецов до 50% объема памяти может быть потрачено впустую из-за внутренней фрагментации.
 - Алгоритм распределения памяти по методу первого подходящего блока обладает меньшим быстродействием, чем алгоритм по методу наиболее согласованного блока (в среднем).
 - Освобождение с использованием граничных тегов обладает достаточным быстродействием только тогда, когда список свободных блоков упорядочен по возрастанию адреса памяти.
 - Для метода близнецов характерна внутренняя, но не внешняя фрагментация.
2. • Использование алгоритма по методу первого подходящего блока применительно к списку свободных блоков, упорядоченному по уменьшению размера блока, приводит к уменьшению производительности при распределении памяти, зато предотвращает внешнюю фрагментацию.
 - Для алгоритма, реализующего метод наиболее согласованного блока, список свободных блоков нужно упорядочивать по возрастанию адресов памяти.
 - По методу наиболее согласованного блока выбирается наибольший свободный блок, в котором может поместиться запрошенный сегмент.
 - Использование алгоритма, реализующего метод первого подходящего блока на списке свободных блоков, который упорядочен по возрастанию размера блока, эквивалентно использованию алгоритма по методу наиболее согласованного блока.
3. Программы сборки мусора, работающие по методу "отметить и очистить", называются консервативными:
 - если они объединяют освобожденную память только тогда, когда запрос на выделение памяти не может быть обслужен;

- если они рассматривают как указатель все, что напоминает указатель;
- если они выполняют сборку мусора, только память будет израсходована;
- если они не освобождают блоки памяти, образующие циклический список.

УПРАЖНЕНИЕ 10.20 ◆◆◆

Напишите свои собственные версии функций `malloc` и `free` и сравните время их выполнения и использование пространства памяти с версией `malloc`, предоставляемой стандартной библиотекой языка C.

Решение упражнений

РЕШЕНИЕ УПРАЖНЕНИЯ 10.1

Это упражнение дает некоторую оценку размеров различных адресных пространств. Было время, когда 32-разрядное адресное пространство казалось невероятно большим. Но теперь, когда имеются базы данных и научные приложения, требующие еще больших ресурсов, можно ожидать, что эта тенденция сохранится. Можно также ожидать, что наступит такой момент в вашей деятельности, когда вы будете жаловаться на недостаток адресного 64-разрядного пространства на вашем персональном компьютере!

Число разрядов виртуального адреса	Число виртуальных адресов	Максимально возможный виртуальный адрес
8	$2^8 = 256$	$2^8 - 1 = 255$
16	$2^{16} = 64\text{K}$	$2^{16} - 1 = 64\text{K} - 1$
32	$2^{32} = 4\text{G}$	$2^{32} - 1 = 4\text{G} - 1$
48	$2^{48} = 256\text{T}$	$2^{48} - 1 = 256\text{T} - 1$
64	$2^{64} = 16\text{ T}84\text{P}$	$2^{64} - 1 = 16\text{ T}84\text{P} - 1$

РЕШЕНИЕ УПРАЖНЕНИЯ 10.2

Поскольку каждая виртуальная страница имеет размер $P = 2^P$ байтов, то всего в системе может быть $2^n/2^P = 2^{n-P}$ страниц, каждая из которых требует своего элемента в таблице страниц (PTE).

n	$P = 2^P$	Количество PTE
16	4K	16
16	8K	8
32	4K	1M
32	8K	512K

РЕШЕНИЕ УПРАЖНЕНИЯ 10.3

Вы должны хорошо разбираться в такого рода задачах, чтобы полностью понимать принципы преобразования адреса. Вот как решается первая подзадача: нам заданы виртуальные адреса длиной $n = 32$ разряда и физические адреса длиной $m = 24$ разряда. Размер страницы $P = 1$ Кбайт означает, что нам нужно $\log_2(1 \text{ Кбайт}) = 10$ разрядов и для VPO, и для PPO. (Вспомните, что смещения VPO и PPO идентичны.) Остающиеся в адресе разряды суть, соответственно, VPN и PPN.

P	Число разрядов в VPN	Число разрядов в VPO	Число разрядов в PPN	Число разрядов в PPO
1 Кбайт	22	10	14	10
2 Кбайт	21	11	13	11
4 Кбайт	20	12	12	12
8 Кбайт	19	13	11	13

РЕШЕНИЕ УПРАЖНЕНИЯ 10.4

Построение этих нескольких ручных моделей представляет собой хороший способ закрепить понимание принципов преобразования адреса. Полезно было бы выписать все разряды адреса, такие как VPN, TLBI и т. д., и затем заключить их в клеточки. В этой конкретной задаче ничего не пропущено: буфер TLB содержит копию элемента PTE, а кэш содержит копию запрошенных слов данных. См. упр. 10.11—10.13, в которых рассматриваются различные комбинации наличия и отсутствия различных элементов.

1. 00 0011 1101 0111

2. VPN: 0xf
 TLBI: 0x3
 TLBT: 0x3
 TLB hit? Y
 page fault? N
 PPN: 0xd
 4. 0011 0101 0111
 5. CO: 0x3
 CI: 0x5
 CT: 0xd
 cache hit? Y
 cache byte? 0x1d

РЕШЕНИЕ УПРАЖНЕНИЯ 10.6

Решение этой задачи даст вам уверенность в том, что вы хорошо понимаете, что происходит при отображении в памяти. Пробуйте выполнить его самостоятельно. Мы не

рассматривали функций open, fstat и write, так что вам придется прочитать соответствующие страницы документации, чтобы разобраться, как они работают.

```

1 #include "csapp.h"
2
3 /*
4 * mmapcopy — использует функцию mmap для копирования файла fd в stdout
5 */
6 void mmapcopy(int fd, int size)
7 {
8     char *bufp; /* указывает на область виртуальной памяти, отображенной
9                  памяти */
10    bufp = Mmap(NULL, size, PROT_READ, MAP_PRIVATE, fd, 0);
11    Write(1, bufp, size);
12    return;
13 }
14
15 /* драйвер mmapcopy */
16 int main(int argc, char **argv)
17 {
18     struct stat stat;
19     int fd;
20
21     /* проверка наличия необходимого параметра командной строки */
22     if (argc != 2) {
23         printf("usage: %s <filename>\n", argv[0]);
24         exit(0);
25     }
26
27     /* копирование входного параметра в stdout */
28     fd = Open(argv[1], O_RDONLY, 0);
29     fstat(fd, &stat);
30     mmapcopy(fd, stat.st_size);
31     exit(0);
32 }
```

РЕШЕНИЕ УПРАЖНЕНИЯ 10.6

Решение этой задачи потребует использования некоторых основных понятий, таких как требования выравнивания, минимальные размеры блока, и кодирование заголовка. Общий подход к определению размера блока — это округление суммы запрашиваемого полезного пространства и заголовка, с последующей установкой размера равным ближайшему кратному требованиям выравнивания (в нашем случае восемь байтов). Например, размер блока для malloc (1) вычисляется так: запрос $4 + 1 = 5$, округляем до восьми. Размер блока для malloc (13) вычисляется так: запрос $13 + 4 = 17$, округляем до 24.

Запрос	Размер блока (в байтах, десятичное значение)	Заголовок блока (шестнадцатеричное значение)
malloc(1)	8	0x9
malloc(5)	16	0x11
malloc(12)	16	0x11
malloc(13)	24	0x19

РЕШЕНИЕ УПРАЖНЕНИЯ 10.7

Значение минимального размера блока может оказать существенное влияние на внутреннюю фрагментацию. Таким образом, важно понимать, что минимальные размеры блока связаны с различиями в реализациях программы распределения памяти и в требованиях на выравнивание. Сложность состоит в том, чтобы понять, что в различные моменты времени один и тот же блок может находиться в выделенном или свободном состоянии. Минимальный размер блока — это максимальное значение из минимального размера распределенных блоков и минимального размера свободных блоков. Например, в последнем упражнении минимальный размер выделенного блока есть четырехбайтовый заголовок и однобайтовое полезное пространство, округленные до восьми байтов. Минимальный размер свободного блока есть четырехбайтовый верхний заголовок и четырехбайтовый нижний заголовок, которые в сумме кратны восьми и не требуют округления. Следовательно, минимальный размер блока для этой программы выделения памяти составляет восемь байтов.

Выравнивание	Занятый блок	Свободный блок	Минимальный размер блока
Одинарное слово	Заголовки сверху и снизу	Заголовок сверху и снизу	12
Одинарное слово	Заголовок сверху, но без заголовка снизу	Заголовки сверху и снизу	8
Двойное слово	Заголовки сверху и снизу	Заголовки сверху и снизу	16
Двойное слово	Заголовок сверху, но без заголовка снизу	Заголовки сверху и снизу	8

РЕШЕНИЕ УПРАЖНЕНИЯ 10.8

Здесь нет ничего сложного. Но решение этой задачи требует от вас четкого понимания того, как работает наша простая программа распределения памяти с использованием неявного списка и как вызывать и проводить обход ее блоков.

```

1 static void *find_fit(size_t asize)
2 {
3     void *bp;
4
5     /* поиск по методу первого подходящего */
6     for (bp = heap_listp; GET_SIZE(HDRP(bp)) > 0; bp = NEXT_BLKP(bp)) {
7         if (!GET_ALLOC(HDRP(bp)) && (asize <= GET_SIZE(HDRP(bp)))) {
8             return bp;
9         }
10    }
11    return NULL; /* подходящих нет */
12 }

```

РЕШЕНИЕ УПРАЖНЕНИЯ 10.9

Это еще одно упражнение для разминки, помогающее понять, как правильно использовать программу распределения памяти. Обратите внимание на то обстоятельство, что минимальный размер блока для рассматриваемой программы распределения памяти равен 16 байтам. Если бы остаток от блока после его разбиения больше или равен минимальному размеру блока, то мы продвигаемся дальше и разбиваем блок (строки 6—10). Единственная сложность здесь состоит в том, чтобы понять, что новый выделенный блок следует сначала разместить (строки 6 и 7) и только потом переходить к следующему блоку (строка 8).

```

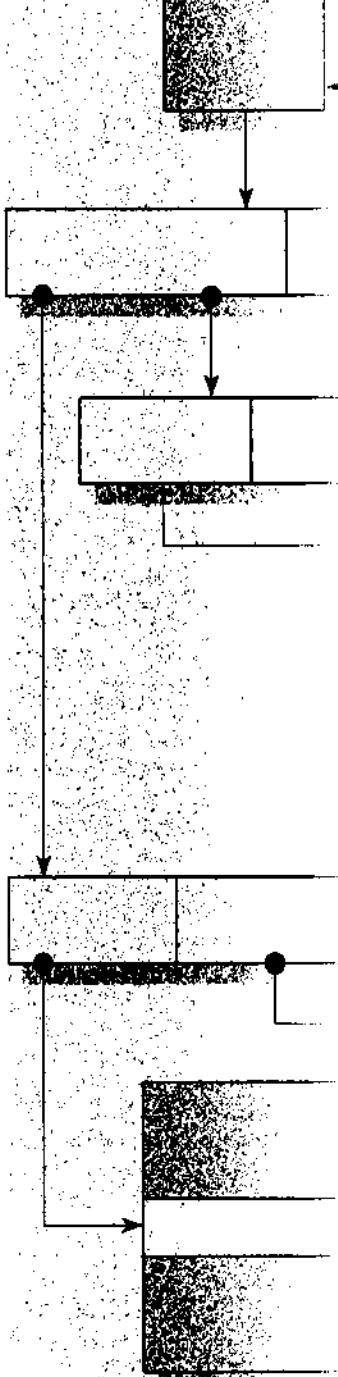
1 static void place(void *bp, size_t asize)
2 {
3     size_t csize = GET_SIZE(HDRP(bp));
4
5     if ((csize - asize) >= (DSIZE + OVERHEAD)) {
6         PUT(HDRP(bp), PACK(asize, 1));
7         PUT(FTRP(bp), PACK(asize, 1));
8         bp = NEXT_BLKP(bp);
9         PUT(HDRP(bp), PACK(csize-asize, 0));
10        PUT(FTRP(bp), PACK(csize-asize, 0));
11    }
12    else {
13        PUT(HDRP(bp), PACK(csize, 1));
14        PUT(FTRP(bp), PACK(csize, 1));
15    }
16 }

```

РЕШЕНИЕ УПРАЖНЕНИЯ 10.10

Здесь мы имеем некоторую модель реализации, которая вызывает внешнюю фрагментацию: приложение делает многочисленные запросы на выделение и на освобож-

дение памяти первого класса размера, за которыми следуют многочисленные запросы на выделение и освобождение памяти второго класса размера, за которыми следуют многочисленные запросы на выделение и освобождение памяти третьего класса размера и т. д. На каждый класс размера программа распределения памяти отводит большую порцию памяти, которая никогда не будет востребована, поскольку программа распределения памяти не производит слияний, а приложение никогда не осуществляет повторных запросов блоков этого класса размера.



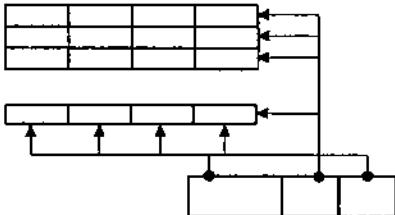
ЧАСТЬ III

Взаимодействие и взаимосвязи программ

До сих пор при изучении компьютерных систем авторами делалось допущение, что программы выполняются изолированно, с минимумом информации на входе и на выходе. Однако в реальной практике программные приложения используют услуги операционных систем, направленные на взаимодействие с устройствами ввода-вывода, а также с другими программами.

Данная часть книги дает возможность понимания основных услуг ввода-вывода, предоставляемых операционными системами Unix, а также принципы использования этих услуг для создания таких приложений, как Web-клиенты и серверы, взаимодействующие друг с другом по сети Internet. Студентам предоставляется возможность изучения методик написания параллельных программ (например, Web-серверов), способных к одновременному обслуживанию большого количества клиентов.

По окончании изучения данной части предполагается, что читатель по-настоящему приблизится к цели досконального понимания компьютерных систем и их влияния на вновь создаваемые программы.



ГЛАВА 11

Системный уровень ввода-вывода

- Ввод-вывод Unix.
 - Открытие и закрытие файлов.
 - Считывание и запись файлов.
 - Устойчивое считывание и запись с помощью пакета Rio.
 - Считывание метаданных файла.
 - Совместное использование файлов.
 - Переадресация данных ввода-вывода.
 - Стандартный ввод-вывод.
 - Окончательная сборка: какие функции ввода-вывода стоит использовать.
 - Резюме.
-

Ввод-вывод (I/O) — это процесс копирования данных между основной памятью и внешними устройствами: дисковыми накопителями, терминалами и сетями. При операции ввода данные копируются из устройства ввода-вывода в основную память, а при операции вывода данные копируются из памяти на соответствующее устройство.

Все языковые системы поддержки выполнения программ обеспечивают средства высокого уровня для выполнения ввода-вывода. Например, ANSI C предоставляет стандартную библиотеку ввода-вывода с такими функциями, как `printf` и `scanf`, выполняющими буферизованный ввод-вывод. Язык C++ обеспечивает похожую функциональность с перегруженными операторами `<<` (ввести) и `>>` (получить). В системах Unix эти функции ввода-вывода высокого уровня реализуются с использованием функций Unix ввода-вывода системного уровня, предоставляемых ядром. Большую часть времени функции ввода-вывода высокого уровня работают хорошо, поэтому непосредственного использования ввода-вывода Unix не требуется. Тогда возникает вопрос: зачем вообще изучать ввод-вывод Unix?

- Понимание ввода-вывода Unix позволяет понять принципы работы других систем. Ввод-вывод не отделим от работы системы, и по этой причине пользователи часто

столкиваются с круговыми зависимостями между понятиями ввода-вывода системы и другими концепциями. Например, ввод-вывод играет ключевую роль в создании и выполнении процесса, и наоборот, создание процесса играет ключевую роль в совместном использовании файлов различными процессами. Таким образом, для реального понимания ввода-вывода необходимо понимание процессов, и наоборот. При рассмотрении иерархии памяти, связывания и загрузки, процессов и виртуальной памяти авторы уже затрагивали аспекты ввода-вывода. Теперь, когда эти концепции стали понятными, можно завершить круг и рассмотреть процесс ввода-вывода более подробно.

- Иногда нет выбора, кроме как использовать ввод-вывод Unix. Иногда возникают принципиальные ситуации, когда использовать функции ввода-вывода высокого уровня либо невозможно, либо неуместно. Например, стандартная библиотека ввода-вывода не дает доступа к метаданным файла, таким как его размер или время создания. Более того, со стандартной библиотекой ввода-вывода возникают проблемы, создающие риски при ее использовании в сетевом программировании.

В данной главе представлены общие концепции ввода-вывода систем Unix, стандартного ввода-вывода, рассказывается об их надежном использовании при создании программ на языке С. Помимо того, что глава служит общим введением в предмет, она закладывает твердую основу последующего изучения сетевого программирования и параллельности.

11.1. Ввод-вывод Unix

Файл Unix представляет собой последовательность из m байтов

$$B_0, B_1, \dots, B_k, \dots, B_{m-1}$$

Все устройства ввода-вывода: сети, дисковые накопители и терминалы — смоделированы в виде файлов. Как ввод, так и вывод выполняются путем считывания и записи соответствующих файлов. Такое тонкое отображение устройств в файлы позволяет ядру Unix экспорттировать простой интерфейс приложения низкого уровня, называемый Unix I/O, обеспечивающий равномерное и последовательное выполнение операций ввода и вывода:

- Открытие файлов. Программное приложение объявляет о своем намерении получить доступ к устройству ввода-вывода путем запроса ядра на открытие соответствующего файла. Ядро возвращает небольшое неотрицательное целое число, называемое *дескриптором*, идентифицирующее файл во всех последующих операциях с ним. Ядро отслеживает всю информацию об открытом файле. Приложение отслеживает только работу дескриптора. Каждый процесс, создаваемый оболочкой Unix, начинает свое существование с трех открытых файлов:

- стандартного ввода (дескриптор 0);
- стандартного вывода (дескриптор 1);
- стандартной ошибки (дескриптор 2).

- Файл заголовка <unistd.h> определяет константы STDIN_FILENO, STDOUT_FILENO и STDERR_FILENO, которые можно использовать вместо явных значений дескриптора.
- Изменение текущей позиции файла. Ядро поддерживает позицию файла k , изначально равную 0, для каждого открытого файла. Позиция файла — это байтовый сдвиг с начала файла. Программное приложение может явно установить текущую позицию файла k путем выполнения операции поиска (seek).
- Считывание и запись файлов. Операция считывания копирует $n > 0$ байтов из файла в память, начиная с текущей позиции файла k , после чего k увеличивается на n . При условии, что файл имеет размер в m байтов, выполнение операции считывания, когда $k \geq m$, запускает условие, известное как *окончание файла* (EOF), которое выявляется приложением. В конце файла нет явного символа EOF.
- Аналогично, операция записи копирует $n > 0$ байтов из памяти в файл, начиная с текущей позиции файла k , после чего обновляет k .
- После того как приложение заканчивает доступ к файлу, оно информирует ядро запросом закрытия файла. Ядро отвечает освобождением созданных при открытии файла структур данных и восстановлением дескриптора в пуле имеющихся дескрипторов. Когда по какой-либо причине процесс прерывается, ядро закрывает все открытые файлы и освобождает их ресурсы памяти.

11.2. Открытие и закрытие файлов

Процесс открывает существующий файл или создает новый файл вызовом функции open:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open (char *filename, int flags, mode_t mode);
```

Функция open преобразует имя файла filename в файловый дескриптор и возвращает номер дескриптора. Возвращаемый дескриптор — всегда наименьший дескриптор, не открытый в данный момент в процессе. Аргумент flags указывает на то, как процесс планирует доступ к файлу:

- O_RDONLY — только считывание;
- O_WRONLY — только запись;
- O_RDWR — считывание и запись.

Вот как, например, открывается для считывания существующий файл:

```
fd = open ("foo.txt", O_RDONLY, 0);
```

Аргумент flags также может быть только считан с одной или несколькими битовыми масками, обеспечивающими дополнительные команды для записи:

- O_CREAT** — если файла не существует, тогда следует создать его усеченную (пустую) версию;
- O_TRUNC** — если файл существует, тогда его следует усечь;
- O_APPEND** — до выполнения каждой операции записи установите позицию файла в конец файла.

Например, вот как можно открыть существующий файл с намерением добавления определенных данных:

```
fd = Open ("foo.txt", O_WRONLY | O_APPEND, 0);
```

Аргумент *mode* задает биты разрешения доступа новых файлов. Символические имена этих битов показаны в табл. 11.1.

Таблица 11.1. Биты разрешения доступа

Маска	Описание
S_IRUSR	Пользователь (владелец) может считать этот файл
S_IWUSR	Пользователь (владелец) может записывать этот файл
S_IXUSR	Пользователь (владелец) может выполнять этот файл
S_IRGRP	Компоненты группы владельца могут считать этот файл
S_IWGRP	Компоненты группы владельца могут записывать этот файл
S_IXGRP	Компоненты группы владельца могут выполнять этот файл
S_IROTH	Другие пользователи (любые) могут считать этот файл
S_IWOTH	Другие пользователи (любые) могут записывать этот файл
S_IXOTH	Другие пользователи (любые) могут выполнять этот файл

В части своего контекста каждый процесс имеет маску, задаваемую функцией *umask*. Когда процесс создает новый файл вызовом функции *open* с некоторым аргументом *mode*, тогда биты разрешения доступа к файлу устанавливаются на *mode & ~umask*. Предположим, например, что даны следующие значения по умолчанию для *mode* и *umask*:

```
#define DEF_MODE S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH
#define DEF_UMASK S_IWGRP | S_IWOTH
```

Тогда следующий фрагмент кода создает новый файл, в котором владелец файла считал и записывал разрешение на запись, а все другие пользователи считали это разрешение:

```
umask (DEF_UMASK);
fd = Open ("foo.txt", O_CREAT | O_TRUNC | O_WRONLY, DEF_MODE);
```

Наконец, процесс закрывает открытый файл вызовом функции `close`.

```
#include <unistd.h>
int close (int fd);
```

Закрытие уже закрытого дескриптора является ошибкой.

УПРАЖНЕНИЕ 11.1

Каковы выходные данные следующей программы?

```
1 #include "csapp.h"
2
3 int main ()
4 {
5     int fd1, fd2;
6
7     fd1 = Open ("foo.txt", O_RDONLY, 0);
8     Close (fd1);
9     fd2 = Open ("baz.txt", O_RDONLY, 0);
10    printf ("fd2 = %d\n", fd2);
11    exit (0);
12}
```

11.3. Считывание и запись файлов

Программные приложения выполняют ввод и вывод вызовом функций `read` и `write` соответственно.

```
#include <unistd.h>

ssize_t read (int fd, void *buf, size_t n);
    Возврат: число считанных байтов, если OK, 0 при EOF, -1 при ошибке

ssize_t write (int fd, const void *buf, size_t n);
```

Функция `read` копирует максимум n байтов из текущей позиции файла дескриптора fd в ячейку памяти buf . Возвращаемое значение -1 указывает на ошибку, а возвращаемое значение 0 — на конец файла. В противном случае возвращаемое значение указывает на число фактически перенесенных байтов.

Функция `write` копирует максимум n байтов из ячейки памяти buf в текущую позицию файла дескриптора fd . В листинге 11.1 показана программа, использующая вызовы `read` и `write` для копирования стандартного ввода в стандартный вывод, по одному байту за каждый раз.

Листинг 11.1: Копирование стандартного ввода в стандартный вывод

```

1 #include "csapp.h"
2
3 int main (void)
4 {
5     char c;
6
7     while (Read (STDIN_FILENO, &c, 1) != 0)
8         Write (STDOUT_FILENO, &c, 1);
9     exit (0);
10 }
```

Программные приложения могут явно модифицировать текущую позицию файла вызовом функции `lseek`; в книге не описывается.

Какова разница между `ssize_t` и `size_t`?

Возможно, читатель уже заметил, что функция `read` имеет аргумент ввода `size_t` и возвращаемое значение `ssize_t`. Какова же разница между этими двумя типами? `size_t` определяется `unsigned int`, а `ssize` (размер со знаком) определяется как `int`. Функция `read` возвращает размер со знаком, а `ssize` — без знака, потому что при ошибке должно возвращаться значение `-1`. Интересно, что возможность возвращения одного значения `-1` снижает максимальный размер `read` на коэффициент 2 — с 4 до 2 Гбайт.

В некоторых ситуациях `read` и `write` переносят меньшее количество байтов, чем приложение запрашивает. Такие короткие единицы счета не указывают на ошибку. Они происходят по ряду причин:

- Столкновение с EOF при считывании. Предположим, что все готово к считыванию из файла, содержащего только на 20 байтов больше, из текущей позиции файла, и что данный файл считывается кусками по 50 байтов. Тогда следующая операция `read` возвратит короткую единицу счета 20, и после этого функция `read` просигнализирует об окончании файла возвратом короткой единицы счета 0.
- Считывание строк текста с терминала. Если открытый файл совмещен с терминалом (т. е. с клавиатурой и монитором), тогда каждая функция `read` будет переносить за один раз одну строку текста, возвращая короткую единицу счета с размером текстовой строки.
- Считывание и запись сетевых сокетов. Если открытый файл соответствует сетевому сокету (см. разд. 12.3.3), тогда ограничения внутренней буферизации и продолжительные задержки срабатывания сети могут вызвать возврат коротких единиц счета со стороны `read` и `write`. Короткие импульсы также могут иметь место при вызове `read` и `write` на канале Unix — механизме межпроцессорного взаимодействия, которое здесь не рассматривается.

На практике столкнуться с короткими единицами счета при считывании с дисковых файлов невозможно, за исключением случаев EOF, а короткие единицы не будут

иметь место при записи в дисковые файлы. Впрочем, если преследуется цель создания устойчивых (надежных) сетевых приложений, таких как Web-серверы, тогда придется иметь дело с короткими единицами счета путем многократного вызова функций `read` и `write` до тех пор, пока не будут перенесены все запрошенные байты.

11.4. Устойчивое считывание и запись с помощью пакета RIO

В данном разделе разрабатывается пакет ввода-вывода, называемый пакетом RIO (Robust I/O), автоматически управляющий короткими единицами счета. Пакет RIO обеспечивает удобный, надежный и эффективный ввод-вывод в таких приложениях, как сетевые программы, подверженные коротким единицам (импульсам) счета. RIO обеспечивает два разных типа функций:

- Небуферизованные функции ввода-вывода. Эти функции передают данные непосредственно между памятью и файлом без буферизации на уровне приложения. Они особенно полезны для считывания и записи двоичных данных в сеть и из нее.
- Буферные функции ввода. Данные функции позволяют эффективно считывать текстовые строки и двоичные данные из файла, содержимое которого кэшировано в буфер на уровне приложения, аналогично стандартным функциям ввода-вывода, например, `printf`. В отличие от процедур буферного ввода-вывода, представленных в [81], буферные функции ввода RIO являются защищенными от потока (см. разд. 13.7.1) и могут произвольно чередоваться на одном и том же дескрипторе. Например, с дескриптора можно считывать некоторые текстовые строки, затем некоторые двоичные данные, после чего снова текстовые строки.

Рутинные процедуры RIO представлены по двум причинам. Во-первых, они будут использоваться при разработке сетевых приложений. Во-вторых, при изучении кодов для этих процедур читатели глубже поймут общие принципы функционирования ввода-вывода Unix.

11.4.1. Небуферизованные функции ввода и вывода RIO

Приложения могут переносить данные напрямую между памятью и файлом путем вызова функций `rio_readn` и `rio_writen`.

```
#include <csapp.h>
ssize_t rio_readn (int fd, void *usrbuf, size_t n);
ssize_t rio_writen (int fd, void *usrbuf, size_t n);

Возрат: число переданных байтов, если OK, 0 при EOF (только rio_readn),
-1 при ошибке
```

Функция `rio_readn` передает до `n` байтов из текущей позиции файла дескриптора `fd` в ячейку памяти `usrbuf`. Аналогичным образом функция `rio_writen` передает `n` байтов из ячейки `usrbuf` в дескриптор `fd`. Функция `rio_readn` может только возвращать короткую единицу счета, если она сталкивается с окончанием файла. Функция

rio_written никогда не возвращает короткую единицу счета. Вызовы rio_readn и rio_written можно произвольно чередовать на одном дескрипторе.

В листингах 11.2—11.3 показан код для rio_readn и rio_written. Обратите внимание на то, что каждая функция перезапускает функцию read или write, если она прерывается возвратом из программного приложения обработчика сигналов. Для максимальной мобильности авторами предусмотрены прерывистые системные вызовы и возможность их перезагрузки (при необходимости) (см. обсуждение прерывистых системных вызовов в разд. 8.5.4).

Листинг 11.2. Стандартный readn

```

1 ssize_t rio_readn (int fd, void *usrbuf, size_t n)
2 {
3     size_t nleft = n;
4     ssize_t nread;
5     char *bufp = usrbuf;
6
7     while (nleft > 0) {
8         if ((nread = read (fd, bufp, nleft)) < 0) {
9             if (errno == EINTR) /* прерывается возвратом обработчика сигналов */
10                nread = 0; /* и очередной раз вызов read () */
11            else
12                return -1/* установка errno функцией read */;
13        }
14        else if (nread == 0)
15            break; /* EOF */
16        nleft -= nread;
17        bufp += nread;
18    }
19    return (n - nleft); /* возврат >= 0 */
20 }
```

Листинг 11.3. Стандартный written

```

1 ssize_t rio_written (int fd, void *usrbuf, size_t n)
2 {
3     size_t nleft = n;
4     ssize_t nwritten;
5     char *bufp = usrbuf;
6
7     while (nleft > 0) {
8         if ((nwritten = write (fd, bufp, nleft)) <= 0) {
9             if (errno == EINTR) /* прерывается возвратом обработчика сигналов */
10                nwritten = 0; /* и очередной раз вызов write () */
11            else
12                return -1/* установка errno функцией write */;
13        }
14        nleft -= nwritten;
15        bufp += nwritten;
16    }
17    return (n - nleft); /* возврат >= 0 */
18 }
```

```

11 else
12 return -1 /* установка errno функцией write */
13 }
14 nleft -= nwritten;
15 bufp += nwritten;
16 }
17 return n;
18 }

```

11.4.2. Буферные функции ввода RIO

Текстовая строка — это последовательность символов ASCII, прерываемая символом новой строки. В системах Unix символ новой строки \n — тот же, что и символ перевода строки (LF) ASCII, и имеет числовое значение 0x0a. Предположим, что необходимо написать программу, считающую количество текстовых строк в текстовом файле. Как это сделать? Одним из способов будет использование функции `read` для переноса за каждый раз одного байта из файла в память пользователя, с проверкой каждого байта для символа новой строки. Недостатком такого подхода является его непроизводительность, требующая ловушки для того, чтобы ядро считывало каждый байт в файле.

Более совершенным подходом является вызов интерфейсной функции `rio_readlineb`, копирующий текстовую строку из внутреннего буфера считывания, с автоматическим созданием вызова `read` для заполнения буфера всякий раз, когда он оказывается пустым. Для файлов, в которых содержатся как текстовые строки, так и двоичные данные (например, для ответов HTTP, описанных в разд. 12.5.3) также предусмотрена буферная версия `rio_readnb`, называемая `rio_readnb`, переносящая необработанные байты из того же буфера считывания, что и `rio_readlineb`.

```

#include <csapp.h>

void rio_readinitb (rio_t *rp, int fd);
                                         Ничего не возвращает
ssize_t rio_readlineb (rio_t *rp, void *usrbuf, size_t maxlen);
ssize_t rio_readnb (rio_t *rp, void *usrbuf, size_t n);

```

Функция `rio_readinitb` вызывается один раз на открытый дескриптор. Она связывает дескриптор `fd` с буфером считывания типа `rio_t` в адресе `rp`.

Функция `rio_readlineb` считывает следующую текстовую строку из файла `rp` (включая прерывающий символ новой строки), копирует ее в ячейку памяти `usrbuf` и прерывает текстовую строку нулевым символом. Функция `rio_readlineb` считывает максимум `maxlen-1` байтов, оставляя место для прерывающего нулевого символа. Текстовые строки, превышающие `maxlen-1` байтов, усекаются и прерываются нулевым символом.

Функция `rio_readnb` считывает до `n` байтов из файла `rp` в ячейку памяти `usrbuf`. Вызовы `rio_readlineb` и `rio_readnb` можно произвольно чередовать на том же дескрип-

торе. Однако вызовы этих буферных функций не должны чередоваться с вызовами небуферизуемой функции `rio_readn`. В оставшейся части данного текста функции RIO будут встречаться достаточно часто. В листинге 11.4 показано использование функций RIO для копирования текстового файла из стандартного ввода в стандартный вывод по одной строке за каждый раз.

Листинг 11.4 Копирование текстового файла из стандартного ввода в стандартный вывод

```
1 #include "csapp.h"
2
3 int main (int argc, char **argv)
4 {
5     int n;
6     rio_t rio;
7     char buf [MAXLINE];
8
9     Rio_readinitb (&rio, STDIN_FILENO);
10    while ((n = Rio_readlineb (&rio, buf, MAXLINE)) !=0)
11        Rio_writen (STDOUT_FILENO, buf, n);
12 }
```

В листинге 11.5—11.6 показан формат буфера считывания вместе с кодом для инициализирующей его функции `rio_readinitb`. Функция `rio_readinitb` задает пустой буфер считывания и связывает дескриптор открытого файла с этим буфером.

Листинг 11.5 Формат буфера считывания

```
1 #define RIO_BUFSIZE 8192
2 typedef struct {
3     int rio_fd; /* дескриптор для данного внутреннего буфера */
4     int rio_cnt; /* несчитанные байты во внутреннем буфере */
5     char *rio_bufptr; /* следующий несчитанный байт во внутреннем буфере */
6     char rio_buf [RIO_BUFSIZE]; /* внутренний буфер */
7 } rio_t;
```

Листинг 11.6 Инициализирующая его функция (то же самое)

```
1 void rio_readinitb (rio_t *rp, int fd)
2 {
3     rp->rio_fd = fd;
4     rp->rio_cnt = 0;
5     rp->rio_bufptr = rp->rio_buf;
6 }
```

"Сердцем" рутинных процедур считывания RIO является функция `rio_read`, показанная в листинге 11.7. Функция `rio_read` является буферной версией функции считывания Unix.

Листинг 11.7. Буферная версия функции считывания

```

1 static ssize_t rio_read (rio_t *rp, char *usrbuf, size_t n)
2 {
3     int cnt;
4
5     while (rp->rio_cnt <= 0) /* повторное заполнение, если буфер пуст */
6         rp->rio_cnt = read (rp->rio_fd, rp->rio_buf,
7         sizeof (rp->rio_buf));
8     if (rp->rio_cnt < 0) {
9         if (errno != EINTR) /* прервано возвратом обработчика сигналов */
10            return -1;
11    }
12    else if (rp->rio_cnt == 0) /* EOF */
13        return 0;
14    else
15        rp->rio_bufptr = rp->rio_buf; /*перезагрузка буфера ptr */
16    }
17
18    /* Копирование минимум (n, rp->rio_cnt) байтов из внутреннего буфера
   в буфер пользователя */
19    cnt = n;
20    if (rp->rio_cnt < n)
21        cnt = rp->rio_cnt;
22    memcpy (usrbuf, rp->rio_bufptr, cnt);
23    rp->rio_bufptr += cnt;
24    rp->rio_cnt -= cnt;
25    return cnt;
26 }
```

Когда `rio_read` вызывается с запросом считать `n` байтов, тогда в буфере считывания имеется `rp->rio_cnt` несчитанных байтов. Если буфер пуст, тогда он пополняется вызовом `read`. Получение короткой единицы считывания от такой активизации `read` не является ошибкой и просто оказывает эффект частичного заполнения буфера считывания. Как только буфер не пуст, `rio_read` копирует минимум `n` и `rp->rio_cnt` байтов из буфера считывания в пользовательский буфер и возвращает число скопированных байтов.

Для прикладной программы функция `rio_read` имеет ту же семантику, что и функция `read` Unix. При ошибках он возвращает `-1` и соответствующим образом устанавливает `errno`. По окончании файла (EOF) функция возвращает 0. Она возвращает короткую единицу счета, если количество запрошенных байтов превышает количество несчитанных байтов в буфере считывания. Сходство этих двух функций упрощает

построение различного рода буферных функций считывания путем замены `rio_read` на `read`. Например, функция `rio_readnb`, показанная в листинге 11.8, имеет ту же структуру, что и `rio_readn`, где `rio_read` замещена на `read`.

Листинг 11.8. Функция `rio_readnb`

```

1 ssize_t rio_readnb (rio_t, *rp, void *usrbuf, size_t maxlen)
2 {
3     int n, rc;
4     char c, *bufp = usrbuf;
5
6     for (n = 1; n < maxlen; n++) {
7         if ((rc = rio_read (rp, &c, 1)) == -1) {
8             *bufp++ = c;
9             if (c == '\n')
10                break;
11        } else if (rc == 0) {
12            if (n == 1)
13                return 0; /* EOF, отсутствие считывания данных */
14        } else
15            break; /* EOF, некоторые данные считаны */
16    } else
17        return -1; /* ошибка */
18 }
19 * bufp = 0;
20 return n;
21 }
```

Аналогичным же образом, рутинная процедура `rio_readlineb`, показанная в листинге 11.9, вызывает `rio_read` максимум `maxlen-1` раз. Каждый вызов возвращает один байт из буфера считывания, который впоследствии проверяется на принадлежность к входящей новой строке.

Листинг 11.9. Процедура `rio_readlineb`

```

1 ssize_t rio_readlineb (rio_t, *rp, void *usrbuf, size_t n)
2 {
3     size_t nleft = n;
4     ssize_t nread;
5     char *bufp = usrbuf;
6
7     while (nleft > 0) {
8         if ((nread = rio_read (rp, bufp, nleft)) < 0) {
9             if (errno == EINTR) /* прерывается возвратом обработчика сигналов */
10                nread = 0; /* очередной вызов read () */
11         }
12         if (bufp[nread] == '\n') /* обработка новой строки */
13             break;
14         bufp += nread;
15         nleft -= nread;
16     }
17 }
```

```

11 else
12 return -1; /* errno устанавливается read () */
13 }
14 else if (nread == 0)
15 break; /* EOF */
16 nleft -= nread;
17 bufp += nread;
18 }
19 return (n - nleft); /* возврат >= 0 */
20 )

```

Происхождение пакета RIO

Функции RIO произошли от функций readline, readn и written, описанные В. Ричардом Стивенсом (W. Richard Stevens) в его классической работе по сетевому программированию [81]. Функции rio_readn и rio_written идентичны функциям readn и written Стивенса. Однако функция readline Стивенса имеет определенные ограничения, скорректированные в RIO. Во-первых, по причине того, что readline — функция буферная, а readn — нет, их нельзя использовать вместе на одном и том же дескрипторе. Во-вторых, из-за использования буфера static функция readline Стивенса не является потокозащищенной, что потребовало от Стивенса представления другой потокозащищенной версии, называемой readline_r. Авторы устранили эти два недостатка с помощью функций rio_readlineb и rio_readnb, которые являются взаимозаменяемыми и потокозащищенными.

11.5. Считывание метаданных файла

Программное приложение может извлекать информацию о файле (иногда называемую *метаданными файла*) путем вызова функций stat и fstat.

```

#include <unistd.h>
#include <sys/stat.h>

int stat (const char *filename, struct stat *buf);
int fstat (int fd, struct stat *buf);

```

Функция stat принимает в качестве входных данных имя файла и заполняет компоненты структуры stat, показанные в листинге 11.10. Функция fstat аналогична, но вместо имени файла использует дескриптор файла. При рассмотрении Web-серверов в разд. 12.5 потребуются компоненты st_mode и st_size структуры stat. Другие компоненты авторами не рассматриваются.

Листинг 11.10. Компоненты структуры stat

```

/* Метаданные возвращаются функциями stat и fstat */
struct stat {
    dev_t st_dev; /* устройство */

```

```

ino_t          st_ino; /* inode */
mode_t         st_mode; /* защита и тип файла */
nlink_t        st_nlink; /* количество жестких связок */
uid_t          st_uid; /* идентификатор пользователя владельца */
gid_t          st_gid; /* группа идентификатора владельца */
dev_t          st_rdev; /* тип устройства (если устройство inode) */
off_t          st_size; /* общий размер в байтах */
unsigned long st_blksize; /* размер блока для ввода-вывода системы файлов */
unsigned long st_blocks; /* количество размещенных блоков */
time_t          st_atime; /* время последнего доступа */
time_t          st_mtime; /* время последней модификации */
time_t          st_ctime; /* время последнего изменения */

```

Компонент `st_size` содержит размер файла в байтах. Компонент `st_mode` кодирует биты файла доступа (рис. 11.1) и тип файла. Unix распознает несколько различных типов файлов. Обычный файл содержит некоторый тип двоичных или текстовых данных. Ядро не различает текстовые и двоичные файлы. Каталог содержит информацию о других файлах. *Сокет* — это файл, используемый для взаимодействия с другим процессом в рамках сети (см. разд. 12.4).

Unix предоставляет предикаты макроса для определения типа файла из компонента `st_mode`. В табл. 11.2 показано подмножество этих макросов.

Таблица 11.2. Описание структуры

Макрос	Описание
<code>S_ISREG ()</code>	Это обычный файл?
<code>S_ISDIR ()</code>	Это каталог?
<code>S_ISSOCK ()</code>	Это сетевой сокет?

В листинге 11.11 показано то, как можно использовать эти макросы и функцию `stat` для считывания и интерпретации битов `st_mode` файла.

Листинг 11.11. Запросы и манипуляция битами

```

1 #include "csapp.h"
2
3 int main (int argc, char **argv)
4 {
5 struct stat stat;
6 char *type, *readok;
7
8 Stat (argv[1], &stat);
9 if (S_ISREG (stat.st_mode)) /* Определение типа файла */
10 type = "regular";

```

```
11 else if (S_ISDIR (stat.st_mode))
12 type = "directory";
13 else
14 type = "other";
15 if ((stat.st_mode & S_IRUSR)) /* Проверка доступа к считыванию */
16 readok = "yes";
17 else
18 readok = "no"
19
20 printf ("type: %s, read: %s\n", type, readok);
21 exit (0);
22 }
```

11.6. Совместное использование файлов

Файлы Unix можно использовать совместно по-разному. В отсутствие четкого представления того, как ядро представляет открытые файлы, концепция совместного использования может показаться довольно запутанной. Ядро представляет открытые файлы с использованием трех соотносящихся структур данных:

1. Таблица дескрипторов. Каждый процесс обладает своей собственной таблицей дескрипторов, элементы описания в которой индексируются дескрипторами открытых файлов процесса. Каждый элемент описания открытого дескриптора указывает на элемент описания в таблице файлов.
2. Таблица файлов. Набор открытых файлов представлен таблицей файлов, используемой совместно всеми процессами. Каждый элемент записи таблицы файлов состоит (для описываемых здесь целей) из текущей позиции файла, количества элементов записи дескриптора, в данный момент указывающих на нее, и указателя на элемент записи в таблице узла *v*. Закрытие дескриптора уменьшает исходную единицу счета в соответствующем элементе записи таблицы файлов. Ядро не будет удалять элемент записи таблицы файлов до тех пор, пока исходная единица счета не будет равной нулю.
3. Таблица узла *v*. Как и таблица файлов, таблица узла *v* совместно используется всеми процессами. В каждом элементе записи содержится большая часть информации в структуре *stat*, включая компоненты *st_mode* и *st_size*.

На рис. 11.1 показан пример, где дескрипторы 1 и 4 делают ссылку на два разных файла через записи таблицы различных открытых файлов. Это типичная ситуация, когда файлы не используются совместно, и каждый дескриптор соответствует одному явно различимому файлу.

Множественные дескрипторы также могут ссылаться на тот же файл через различные элементы записи таблицы, как показано на рис. 11.2. Например, это может произойти, если дважды вызвать функцию *open* с одним и тем же именем файла *filename*. Ключевая идея заключается в том, что каждый дескриптор имеет свою собственную

явно выраженную позицию файла, поэтому разные операции считывания на разных дескрипторах могут выбирать данные из разных ячеек в файле.

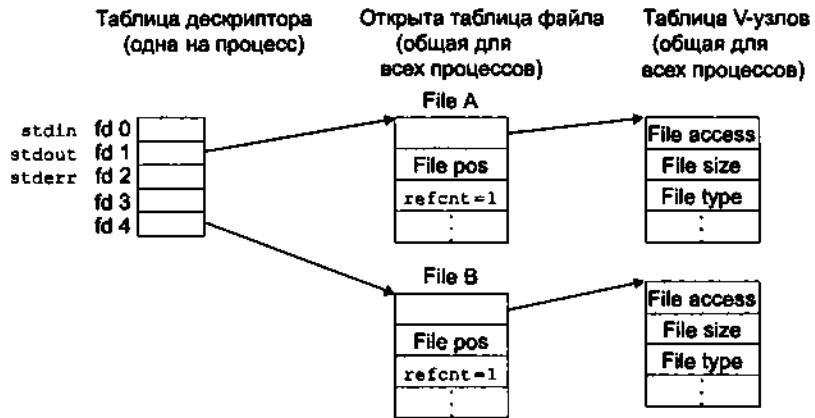


Рис. 11.1. Типичные структуры данных ядра для открытых файлов

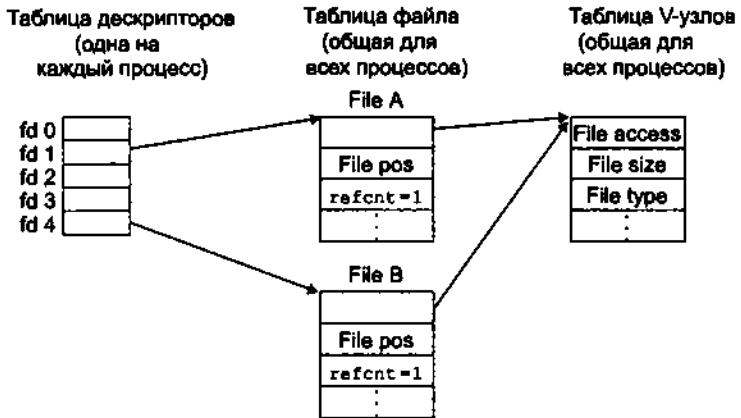


Рис. 11.2. Совместное использование файлов

Теперь можно понять, как порождающий и порожденный процессы совместно используют файлы. Предположим, что перед вызовом `fork` порождающий процесс имеет открытые файлы, показанные на рис. 11.1. На рис. 11.3 показана ситуация после вызова `fork`.

Порожденный процесс получает свой дубликат таблицы дескриптора порождающего процесса. Как один, так и другой процесс совместно используют один и тот же набор таблиц открытых файлов и, таким образом, совместно используют одну и ту же позицию файла. Важным последствием этого является то, что порождающий и порожденный процессы должны закрыть свои дескрипторы до того, как ядро удалит соответствующий элемент записи таблицы файлов.

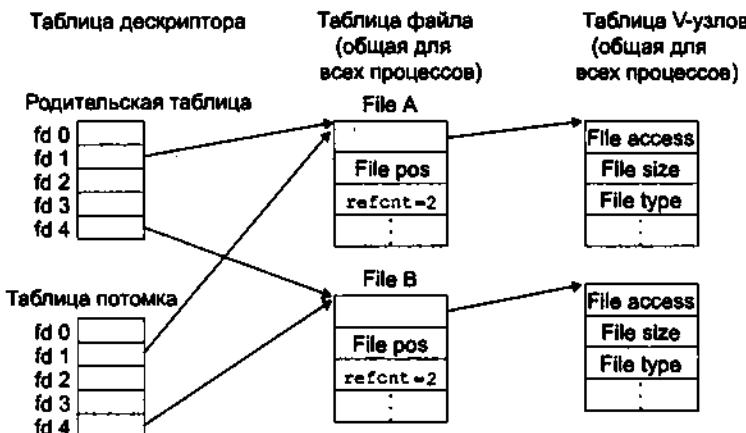


Рис. 11.3. Как порожденный процесс наследует открытые файлы порождающего процесса

УПРАЖНЕНИЕ 11.2

Предположим, что дисковый файл `foobar.txt` состоит из 6 символов `foobar` ASCII. Каковы выходные данные следующей программы?

```

1 #include "csapp.h"
2
3 int main ()
4 {
5     int fd1, fd2;
6     char c;
7
8     fd1 = Open ("foobar.txt", O_RDONLY, 0);
9     fd2 = Open ("foobar.txt", O_RDONLY, 0);
10    Read (fd1, &c, 1);
11    Read (fd2, &c, 1);
12    printf ("c = %c\n", c);
13    exit (0);
14 }
```

УПРАЖНЕНИЕ 11.3

Как и ранее, предположим, что дисковый файл `foobar.txt` состоит из 6 символов `foobar` ASCII. Каковы выходные данные следующей программы?

```

1 #include "csapp.h"
2
3 int main ()
4 {
5     int fd;
```

```

6 char c;
7
8 fd1 = Open ("foobar.txt", O_RDONLY, 0);
9 if (Fork () == 0) {
10 Read (fd, &c, 1);
11 exit (0);
12 }
13 Wait (NULL);
14 Read (fd, &c, 1);
15 printf ("c = %c\n", c);
16 exit (0)
17 }

```

11.7. Переадресация данных ввода-вывода

В оболочках Unix предоставлены операторы *переадресации данных ввода-вывода*, позволяющие пользователям ассоциировать (связывать) стандартный ввод и вывод с дисковыми файлами. Например, ввод строки

```
unix> ls > foo. txt
```

заставляет оболочку загружать и выполнять программу ls с переадресацией стандартного вывода в дисковый файл foo.txt. Как будет видно в разд. 12.5, Web-сервер выполняет аналогичный тип переадресации при выполнении программы CGI от имени клиента. Так как же работает переадресация данных ввода-вывода? Одним из способов является использование функции dup2.

```
#include <unistd.h>

int dup2 (int oldfd, int newfd);
```

Функция dup2 копирует элемент записи oldfd таблицы дескриптора в элемент записи newfd таблицы дескриптора, перезаписывая предыдущее содержимое записи newfd таблицы дескриптора. Если элемент записи newfd уже открыт, тогда dup2 закрывает newfd до копирования oldfd.

Предположим, что перед вызовом dup2 (4, 1) получаем ситуацию, показанную на рис. 11.1, где дескриптор 1 (стандартный вывод) соответствует файлу А (скажем, терминалу), а дескриптор 4 — файлу В (скажем, дисковому файлу). Количество ссылок для А и В равно 1. На рис. 11.4 показана ситуация после вызова dup2 (4, 1). Теперь оба дескриптора указывают на файл В; файл А был закрыт, и его элементы записи таблицы файлов и таблицы узла *v* были удалены; ссылка единицы счета для В была увеличена на единицу. Начиная с этой точки, все данные, записанные в стандартный вывод, переадресованы в файл В.

Правые и левые стрелочные указатели

Во избежание путаницы с другими операторами скобочного типа, например] и [, оператор оболочки > авторы всегда называют "правым стрелочным указателем", а оператор < — "левым стрелочным указателем".

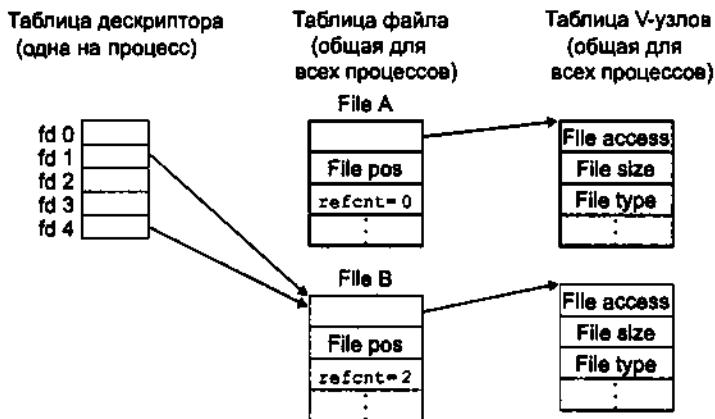


Рис. 11.4. Структуры данных ядра после переадресации стандартного вывода

УПРАЖНЕНИЕ 11.4

Как использовать dup2 для переадресации стандартного ввода в дескриптор 5?

УПРАЖНЕНИЕ 11.5

Каков будет выход следующей программы, если предположить, что дисковый файл foobar.txt состоит из 6 символов foobar ASCII?

```

1 #include "csapp.h"
2
3 int main ()
4 {
5     int fd1, fd2;
6     char c;
7
8     fd1 = Open ("foobar.txt", O_RDONLY, 0);
9     fd2 = Open ("foobar.txt", O_RDONLY, 0);
10    Read (fd2, &c, 1);
11    Dup2 (fd2, &c, 1);
12    Read (fd1, &c, 1);
13    printf ("c = %c\n", c);
14    exit (0);
15 }
```

11.8. Стандартный ввод-вывод

ANSI C определяет набор функций ввода и вывода высокого уровня, называемый **стандартной библиотекой ввода-вывода**, который обеспечивает программистов альтернативой высокого уровня вводу-выводу Unix. Данная библиотека libc обеспечи-

вает функции открытия и закрытия файлов `fopen` и `fclose`, считывания и записи байтов `fread` и `fwrite`, считывания и записи строк `fgets` и `fputs`, а также функции сложного форматированного ввода-вывода `scanf` и `printf`.

Стандартная библиотека ввода-вывода моделирует открытый файл в виде *потока*. Для программиста поток представляет собой указатель на структуру типа `FILE`. Каждая программа ANSI C начинается с трех открытых потоков `stdin`, `stdout` и `stderr`, соответствующих стандартному вводу, стандартному выводу и стандартной ошибке соответственно:

```
#include <stdio.h>
extern FILE *stdin; /* стандартный ввод (дескриптор 0) */
extern FILE *stdout; /* стандартный вывод (дескриптор 1) */
extern FILE *stderr; /* стандартная ошибка (дескриптор 2) */
```

Поток типа `FILE` — абстракция для дескриптора файлов и буфера потока. Назначение буфера потока — то же, что и буфера считывания RIO: минимизация количества дорогостоящих системных вызовов ввода-вывода Unix. Например, предположим, что имеется программа, выполняющая множественные вызовы стандартной функции ввода-вывода `getc`, где каждый вызов возвращает из файла следующий символ. При первом вызове `getc` библиотека заполняет буфер потока однократным вызовом функции `read`, после чего возвращает в приложение первый байт в буфере. До тех пор, пока в буфере остаются несчитанные байты, последующие вызовы `getc` могут обрабатываться непосредственно из буфера потока.

11.9. Окончательная сборка: какие функции ввода-вывода стоит использовать?

На рис. 11.5 показана общая схема различных пакетов ввода-вывода, рассмотренных в данной главе.

Ввод-вывод Unix реализован в ядре операционной системы. Он доступен для приложений через такие функции, как `open`, `close`, `lseek`, `read` и `write`. Функции высокого уровня RIO и функции стандартного ввода-вывода реализованы "поверх" функций ввода-вывода Unix (используя их). Функции RIO — устойчивые упаковщики для `read` и `write`, разработанные специально для данной книги. Они автоматически обрабатывают короткие единицы счета (импульсы) и обеспечивают эффективный буферный подход для считывания строк текста. Функции стандартного ввода-вывода обеспечивают более полную буферную альтернативу функциям ввода-вывода Unix, включая рутинные процедуры форматированного ввода-вывода.

Так какие из этих функций следует использовать при написании программ? Функции стандартного ввода-вывода являются методом выбора ввода-вывода на диске и периферийных устройствах. Большинство программистов, работающих на С, используют исключительно стандартный ввод-вывод на протяжении всей карьеры, никогда не обращаясь к функциям ввода-вывода Unix низкого уровня. Там, где это возможно, авторы рекомендуют придерживаться того же принципа.



Рис. 11.5. Взаимосвязь между вводом-выводом Unix, стандартным вводом-выводом и RIO

К сожалению, стандартный ввод-вывод становится источником очень неприятных проблем при попытке использования в сетях. В разд. 12.4 описывается файл, называемый сокетом и служащий абстракцией Unix для сети. Как и любой файл Unix, файловые дескрипторы делают ссылки на сокеты и в этом случае называются *дескрипторами сокета*. Процессы программного приложения взаимодействуют с процессами, выполняющимися на других компьютерах, путем считывания и записи дескрипторов сокета.

Потоки стандартного ввода-вывода являются *полнодуплексными* в том смысле, что программы могут осуществлять ввод и вывод в одном и том же потоке. Однако существуют ограничения на потоки, плохо взаимодействующие с ограничениями на сокетах:

1. Функции ввода, следующие за функциями вывода. Функция ввода не может следовать за функцией вывода без промежуточного вызова `fflush`, `fseek`, `fsetpos` или `rewind`. Функция `fflush` освобождает буфер, ассоциированный с потоком. Три последние функции используют функцию `lseek` ввода-вывода Unix для сброса текущей позиции файла.
2. Функции вывода, следующие за функциями ввода. Функция вывода не может следовать за функцией ввода без промежуточного вызова `fseek`, `fsetpos` или `rewind`, за исключением случаев, когда функция ввода сталкивается с окончанием файла.

Эти ограничения вызывают проблемы для сетевого приложения, потому что использовать функцию `lseek` на сокете запрещено. Первое ограничение на потоковый ввод-вывод можно "обойти" принятием на вооружение практики очистки буфера перед каждым выполнением операции ввода. Однако единственным способом обхода второго ограничения является открытие двух потоков на том же самом открытом дескрипторе сокета: одного для считывания, а другого — для записи:

```
FILE *fpin, *fpout;
```

```
fpin = fdopen (sockfd, "r");
fpout = fdopen (sockfd, "w");
```

Однако этот подход также сталкивается с проблемами, потому что он требует от приложения вызова `fclose` на обоих потоках для освобождения ресурсов памяти, связанными с каждым потоком, во избежание "утечки памяти":

```
fclose (fpin);
fclose (fpout);
```

Каждая из этих операций делает попытку закрыть один и тот же базовый дескриптор сокета, поэтому вторая операция `close` не срабатывает. Для последовательных программ это не проблема, однако закрытие уже закрытого дескриптора в поточной программе является ее решением (см. разд. 13.7.4).

Таким образом, авторы рекомендуют не использовать стандартные вводы-выводы для этих операций на сетевых сокетах. Вместо этого следует использовать функции RIO. Если необходим форматированный вывод, используйте функцию `sprint` для форматирования строки в памяти, после чего отправьте ее на сокет, используя `rio_writen`. Если необходим форматированный ввод, используйте `rio_readlineb` для считывания текстовой строки целиком, после чего используйте `sscanf` для извлечения различных полей из текстовой строки.

11.10. Резюме

Unix предоставляет небольшое количество функций системного уровня, позволяющих открывать, закрывать, считывать и записывать файлы, выбирать метаданные файлов и выполнять переадресацию ввода-вывода. Операции считывания и записи Unix подвержены коротким единицам счета (импульсам), которые прикладные программы должны прогнозировать и корректно обрабатывать. Вместо непосредственного вызова функций ввода-вывода Unix в приложениях следует использовать пакет RIO, автоматически обрабатывающий короткие единицы счета путем многократного выполнения операций считывания и записи до полного переноса всех запрошенных данных.

Ядро Unix использует три взаимосвязанных структуры данных для представления открытых файлов. Элементы записи в таблице дескрипторов указывают на элементы записи в таблице открытых файлов, которые, в свою очередь, указывают на элементы записи в таблице узла *v*. Каждый процесс имеет свою собственную особую таблицу дескриптора, тогда как все процессы совместно используют одну таблицу открытых файлов и таблицу узла *v*. Общая организация этих структур способствует пониманию как принципов совместного использования файлов, так и переадресации операций ввода-вывода.

Стандартная библиотека ввода-вывода реализуется "поверх" ввода-вывода Unix и представляет мощный набор рутинных процедур ввода-вывода высокого уровня. Для большинства прикладных программ стандартный ввод-вывод является более простой и более предпочтительной альтернативой функциям ввода-вывода Unix. Впрочем, из-за определенных взаимно несовместимых ограничений между стандартным вводом-выводом и сетевыми файлами для сетевых приложений следует использовать ввод-вывод Unix, а не стандартные функции ввода-вывода.

Библиографические примечания

Стивенс написал стандартный эталонный текст для функций ввода-вывода Unix [76]. Керниган (Kernighan) и Риччи (Ritchie) представляют четкое и исчерпывающее обсуждение функций стандартного ввода-вывода [40].

Задачи для домашнего решения

УПРАЖНЕНИЕ 11.6 ◆

Каковы выходные данные следующей программы:

```
1 #include "csapp.h"
2
3 int main ()
4 {
5     int fd1, fd2;
6
7     fd1 = Open ("foobar.txt", O_RDONLY, 0);
8     fd2 = Open ("foobar.txt", O_RDONLY, 0);
9     Close (fd2);
10    fd2 = Open ("baz.txt", O_RDONLY, 0);
11    printf ("fd2 = %d\n", fd2);
12    exit (0);
13 }
```

УПРАЖНЕНИЕ 11.7 ◆

Модифицируйте программу cpfile, показанную в листинге 11.4 так, чтобы в ней использовались функции RIO для копирования стандартного ввода в стандартный вывод, по MAXBUF байтов за один раз.

УПРАЖНЕНИЕ 11.8 ◆◆

Напишите версию программы statcheck, показанную в листинге 11.11, под названием fstatcheck, которая в командной строке принимает номер дескриптора, а не номер файла.

УПРАЖНЕНИЕ 11.9 ◆◆

Рассмотрите следующий вызов программы fstatcheck из упр. 11.8:

```
unix> fstatcheck 3 < foo.txt
```

Можно ожидать, что этот вызов fstatcheck выберет и отобразит метаданные для файла foo.txt. Однако при его выполнении на данной системе он не срабатывает с сообщением "неверный файловый дескриптор". При данном поведении заполните псевдокод, который оболочка может выполнять между вызовами fork и execve:

```

if (Fork () == 0)  /* порожденный процесс */
    /* Какой код является выполняющейся здесь оболочкой? */
    Execve ("fstatcheck", argv, envp);
}

```

УПРАЖНЕНИЕ 11.10 ••

Модифицируйте программу cpfile, показанную в листинге 11.4 так, чтобы она принимала необязательный аргумент infile командной строки. Если дан infile, скопируйте его в стандартный вывод, в противном случае скопируйте, как и ранее, стандартный ввод в стандартный вывод. "Трюк" заключается в том, что в решении для обоих случаев должен использоваться оригинальный цикл копирования (строки 9—11). Допускается только вставка кода, но изменять что-либо в существующем коде нельзя.

Решение упражнений

РЕШЕНИЕ УПРАЖНЕНИЯ 11.1

Процессы Unix начинают свое существование с открытыми дескрипторами, привязанными к stdin (дескриптор 0), stdout (дескриптор 1), stderr (дескриптор 2). Функция open всегда возвращает самый нижний не открытый дескриптор, поэтому первый вызов open возвращает дескриптор 3. Вызов функции close освобождает дескриптор 3. Последний вызов open возвращает дескриптор 3, следовательно, выходные данные программы составляют fd2 = 3.

РЕШЕНИЕ УПРАЖНЕНИЯ 11.2

Дескрипторы fd1 и fd2 имеют свои собственные элементы записи таблицы открытых файлов, поэтому каждый дескриптор имеет свою собственную позицию файла для foobar.txt. Таким образом, при считывании из fd2 сначала считывается первый байт foobar.txt, а выход

c = f,

а не

c = o,

как можно было предположить ранее.

РЕШЕНИЕ УПРАЖНЕНИЯ 11.3

Вспомните о том, что порожденный процесс наследует таблицу дескрипторов родителя, и что все процессы совместно использовали одну и ту же таблицу открытых файлов. Таким образом, дескриптор fd порождающего и порожденного процесса указывает на один элемент таблицы открытых файлов. Когда порожденный процесс считывает первый байт файла, позиция файла увеличивается на единицу. Следовательно, порождающий процесс считывает второй байт и выходные данные составляют c = o.

РЕШЕНИЕ УПРАЖНЕНИЯ 11.4

Для переадресации стандартного вывода (дескриптор 0) на дескриптор 5 необходимо вызвать

`dup2 (5, 0)`

или, эквивалентно,

`dup2 (5, STDIN_FILENO).`

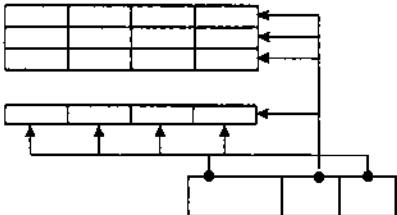
РЕШЕНИЕ УПРАЖНЕНИЯ 11.5

На первый взгляд можно подумать, что выходные данные составят

`c = f;`

но из-за того, что `fd1` переадресовано на `fd2`, реальный выход

`c = o.`



ГЛАВА 12

Сетевое программирование

- Программная модель клиент-сервер.
 - Компьютерные сети.
 - Глобальная сеть Internet.
 - Интерфейс сокетов.
 - Web-серверы.
 - Разработка небольшого Web-сервера TINY.
 - Резюме.
-

Сетевые приложения можно встретить везде. Каждый раз, когда вы просматриваете Web-страницы, посылаете сообщения по электронной почте или открываете на экране X-окно, вы используете соответствующее сетевое приложение. Интересно отметить, что основой всех сетевых приложений служит одна и та же базовая модель программирования. Такие приложения имеют общую логическую структуру и используют один и тот же программный интерфейс.

В основу сетевых приложений положены многие из тех идей и понятий, которые мы изучали в предыдущих главах. Например, процессы, сигналы, упорядочение байтов, отображение в памяти и динамическое распределение памяти. Теперь нам придется осваивать новые понятия. Нам необходимо получить представление о базовом методе программирования приложений типа клиент-сервер, которые пользуются услугами служб, предоставляемых Internet. В завершение, мы воспользуемся всеми этими идеями при разработке небольшого, но вполне пригодного для практического использования Web-сервера, который может служить как статическим (неизменяемым), так и динамическим (изменяемым) содержимым, в смысле текстовой и графической частей для реальных Web-браузеров.

12.1. Программная модель клиент-сервер

В основу каждого сетевого приложения положена модель клиент-сервер. Приложение, построенное на базе этой модели, состоит из процесса сервера и процесса клиен-

та. Сервер управляет некоторым ресурсом, он предоставляет определенную услугу своим клиентам, манипулируя этим ресурсом. Например, специальный Web-сервер управляет множеством файлов на дисках, он осуществляет их поиск и исполнение по запросам клиентов. Сервер FTP (File Transfer Protocol, протокол передачи файлов) управляет набором файлов на дисках, которые он сохраняет и отыскивает для клиентов. Аналогично, сервер электронной почты управляет файлом списка писем, который он читает и обновляет по запросам клиентов.

Основной операцией модели клиент-сервер является *транзакция* (*transaction*) (рис. 12.1). Транзакция в модели клиент-сервер предусматривает выполнение четырех действий:

1. Когда клиент нуждается в какой-либо услуге, он инициирует транзакцию, послав запрос. Например, когда Web-браузеру нужен тот или иной файл, он посыпает запрос на Web-сервер.
2. Сервер получает запрос, выполняет его интерпретацию и манипулирует своим ресурсом соответствующим образом. Например, когда Web-сервер получает запрос от браузера, он считывает соответствующий дисковый файл.
3. Сервер отсылает ответ (*response*) клиенту, после чего ждет следующий запрос. Например, Web-сервер отсылает файл назад клиенту.
4. Клиент получает ответ и использует его по своему усмотрению. Например, после того, как Web-браузер получает страницу от сервера, он отображает ее на экране.

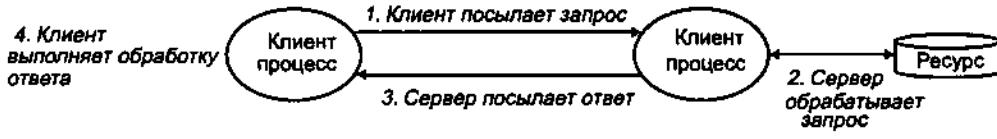


Рис. 12.1. Транзакция клиент-сервер

Важно понимать, что клиенты и серверы суть процессы, а не машины или хосты (*hosts*), как их еще часто называют в этом контексте. На одном и том же хосте одновременно может выполняться множество различных клиентов и серверов, а клиентские и серверные транзакции могут быть как на одном и том же, так и различных хостах. Модель клиент-сервер одна и та же, независимо от того, как размещаются клиенты и серверы на хостах.

Сходство и различие транзакций

Транзакции клиент-сервер не являются транзакциями базы данных и не обладают свойствами, характерными для транзакций баз данных, такими как атомарность. В нашем контексте транзакции — это просто последовательность действий, выполняемых клиентом и сервером.

12.2. Компьютерные сети

Клиенты и серверы часто исполняются на отдельных хостах и обмениваются данными, используя для этой цели программные и аппаратные ресурсы компьютерных се-

тей. Сети представляют собой сложные системы, и мы надеемся, что нам здесь удастся немножко приоткрыть завесу над тем, как это работает. Наша цель заключается в том, чтобы дать вам работоспособную мысленную модель сети, с точки зрения программиста.

Для хоста сеть представляет собой еще одно устройство ввода-вывода, которое выступает в роли источника и приемника данных, как показано на рис. 12.2. Адаптер, подключенный через расширительное гнездо в шине ввода-вывода, обеспечивает физический интерфейс с сетью. Данные, полученные из сети через шины ввода-вывода и шины памяти, копируются в память, обычно посредством метода DMA (Direct Memory Access, прямой доступ к памяти). Аналогичным образом данные могут быть скопированы из памяти в сеть.



Рис. 12.2. Аппаратная организация сетевого хоста

В физическом плане сеть представляет собой иерархическую систему, организованную по географическому принципу. На нижнем уровне это локальная сеть (LAN, Local Area Network), охватывающая здание или студенческий лагерь. В настоящее время наиболее популярной из локальных сетей является сеть Ethernet, которая была разработана в середине семидесятых годов прошлого столетия компанией Xerox PARC. Сеть Ethernet зарекомендовала себя как исключительно гибкая и живучая локальная сеть, скорость передачи данных которой возросла с 3 Мбайт/с до 1 Гбайт/с.

Сегмент сети Ethernet (Ethernet segment) состоит из нескольких перемычек (обычно это витые пары) и небольшой коробки, которая называется концентратором (hub), как показано на рис. 12.3. Сегменты Ethernet обычно охватывают небольшие области, например, комнаты или этаж здания. Все витые пары обладают одной и той же полосой пропускания, обычно 100 Мбайт/с или 1 Гбайт/с. Один ее конец присоединен к

адаптеру хоста, другой — к порту концентратора. Концентратор послушно копирует каждый бит, который он получает, во все остальные порты. Следовательно, каждый хост видит каждый бит.

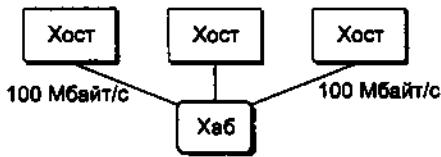


Рис. 12.3. Сегмент локальной сети Ethernet

Каждый адаптер Ethernet имеет уникальный глобальный 48-разрядный адрес, который хранится в энергонезависимой памяти адаптера. Хост может послать порцию битов, называемую *фреймом* (frame), другому хосту сегмента. Каждый фрейм содержит несколько чисел заголовка (header), которые указывают на источник и место назначения фрейма, а также длину фрейма, после чего следует полезная нагрузка (payload) в виде битов данных. Каждый хост видит фрейм, но читает его только тот хост, для которого этот фрейм предназначен.

Многочисленные сегменты сети Ethernet могут быть объединены в локальные сети больших размеров, так называемые мостовые сети Ethernet (bridged Ethernets), как показано на рис. 12.4. В мостовой сети Ethernet одни кабели соединяют шлюз со шлюзом, в то время как другие соединяют шлюзы с концентраторами. Пропускные способности кабелей могут быть различными. В рассматриваемом примере кабельное соединение шлюз-шлюз имеет пропускную способность 1 Гбайт/с, в то время как четыре соединения концентратор-шлюз имеют пропускную способность 100 Мбайт/с.

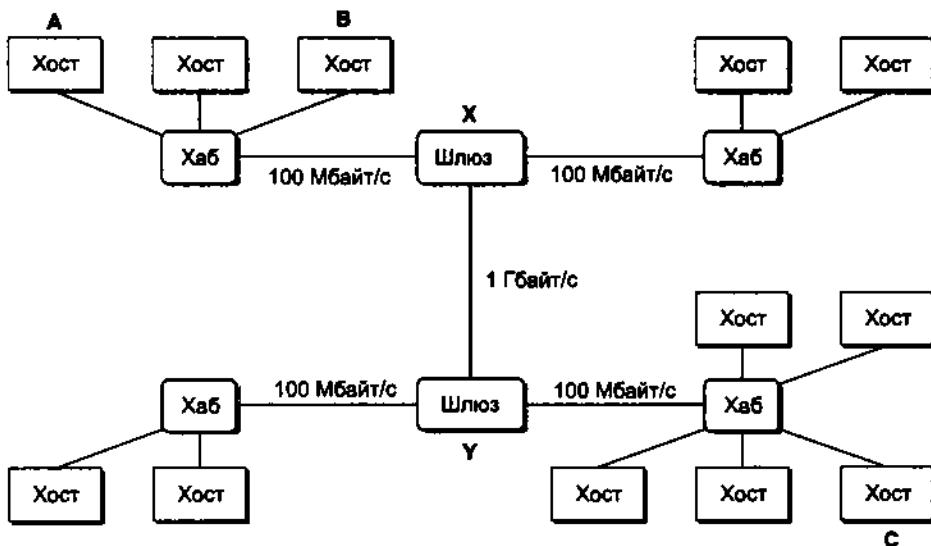


Рис. 12.4. Сегменты Ethernet, объединенные в единую сеть посредством шлюзов

Шлюзы с большей эффективностью используют существующие кабельные линии, чем концентраторы. Используя искусственный распределенный алгоритм, они автоматически фиксируют время, необходимое для установления соединения между различными хостами и портами, а затем производят выборочное копирование фреймов из одного порта в другой только при необходимости. Например, если хост А посыпает фрейм на хост В, который находится в этом же сегменте, то шлюз X отбрасывает этот фрейм, когда тот появляется в его порте ввода, благодаря чему не загружаются линии передачи данных других сегментов. В то же время, если хост А посыпает фрейм на хост С другого сегмента, то шлюз X скопирует этот фрейм только в порт, подключенный к сегменту шлюза С.

Чтобы упростить наше представление о локальной сети, нарисуем концентраторы и линии связи, их соединяющие, в виде горизонтальной линии, как показано на рис. 12.5.

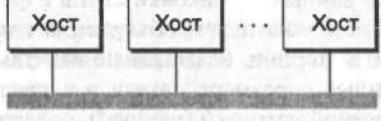


Рис. 12.5. Концептуальное представление локальной сети

На более высоком уровне иерархии многочисленные несовместимые локальные сети могут быть подключены друг к другу посредством специализированных компьютеров, получивших название *маршрутизаторов* (routers), образуя при этом *интерсеть* (сеть с внутренней коммутацией).

У каждого маршрутизатора имеется *адаптер* (порт) для каждой сети, к которой он подключен. Маршрутизаторы могут устанавливать высокоскоростные двухточечные коммутируемые соединения, которые представляют собой примеры *сетей*, известных как сети WAN (Wide Area Networks, глобальные сети). Их так назвали в силу того факта, что они охватывают намного **большие географические** территории, чем могут охватить **локальные** сети. В общем случае, маршрутизаторы могут быть использованы для построения интерсетей из произвольной совокупности локальных и глобальных сетей. Например, на рис. 12.6 показана интерсеть с парой локальных и парой глобальных сетей, соединенных между собой тремя маршрутизаторами.

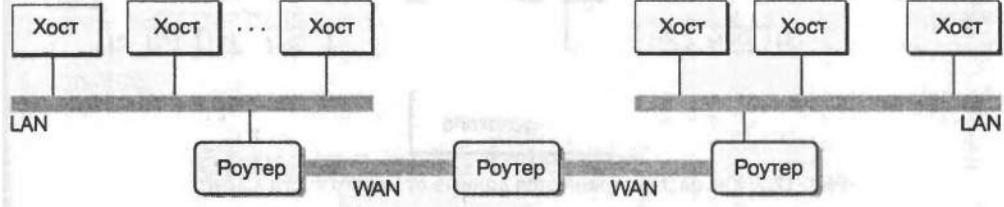


Рис. 12.6. Интерсеть небольших размеров

Главным свойством интерсети является тот факт, что она может состоять из совершенно различных локальных и глобальных сетей с несовместимыми технологиями.

Каждый хост имеет физическую связь с остальными хостами, но как сделать, чтобы некоторый хост-источник (source host) посыпал биты данных хосту назначения (destination host) через все эти несовместимые сети?

Решением этой задачи является многоуровневое программное обеспечение протоколов, функционирующих на хосте и маршрутизаторе, сглаживая различие между разными сетями. Этот протокол должен обеспечить две базовые возможности:

- Схема именования (Naming scheme). Технологии передачи данных по различным локальным сетям реализуют различные несовместимые способы присвоения адресов хостам. Межсетевой протокол сглаживает эти различия путем объявления единобразного формата для адресов хоста. Каждому хосту назначается, по меньшей мере, один из таких межсетевых адресов (internet addresses), которые однозначно его идентифицируют.
- Механизм доставки (Delivery mechanism). Различные технологии передачи данных по сети предусматривают различные и несовместимые способы кодирования битов в линиях передачи данных и упаковки битов в фреймы. Межсетевой протокол сглаживает все эти различия путем объявления единого способа связывания битов данных в отдельные порции, называемые пакетами. Пакет состоит из заголовка, содержащего данные о размере пакета и адреса хоста-источника и хоста назначения, а также полезной нагрузки (payload), содержащей биты данных, передаваемых с хоста-источника.

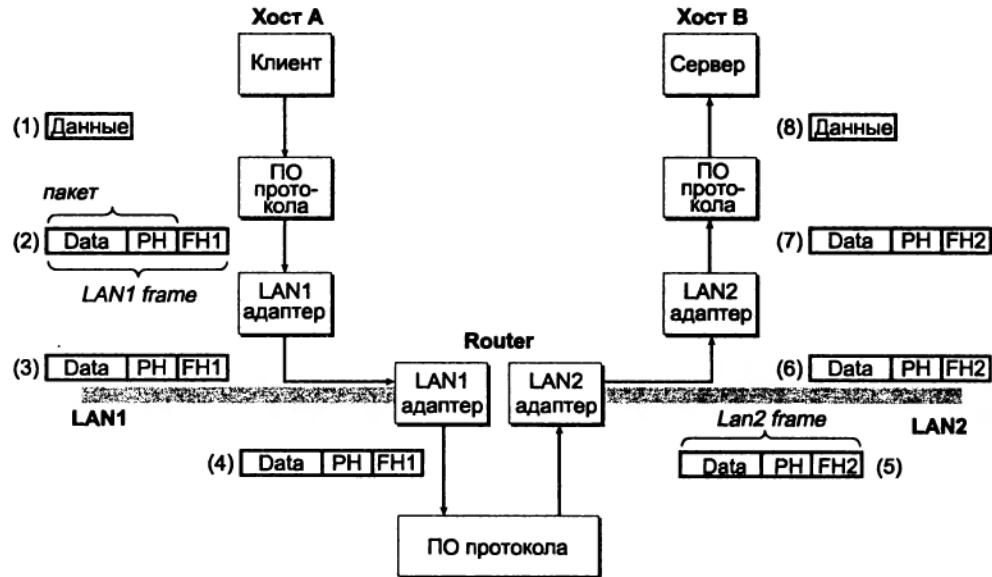


Рис. 12.7. Как распространяются данные от одного хоста к другому

На рис. 12.7 показан пример того, как хосты и маршрутизаторы используют межсетевой протокол передачи данных через несовместимые локальные сети. Рассматриваемая в этом примере сеть состоит из двух локальных сетей, соединенных друг с другом.

гом через маршрутизатор. Клиент, исполняемый на хосте А, который подключен к локальной сети LAN1, посыпает некоторую последовательность байтов данных в сервер, исполняемый на хосте В, который подключен к локальной сети LAN2. При этом выполняются следующие восемь основных действий:

1. Клиент, исполняемый на хосте А, обращается к системе с запросом скопировать данные из виртуального адресного пространства в буфер ядра.
2. Программное обеспечение протокола на хосте А создает фрейм локальной сети LAN1, добавляя к данным межсетевой заголовок и заголовок фрейма LAN1. Межсетевой заголовок адресован межсетевому хосту В. Заголовок фрейма LAN1 адресован маршрутизатору. Затем он передает этот фрейм адаптеру. Обратите внимание на тот факт, что фрейм LAN1 является межсетевым пакетом, полезное содержимое которого составляют фактические данные пользователя. Такой вид инкапсуляции (encapsulation) является одним из фундаментальных принципов работы сети.
3. Адаптер локальной сети LAN1 копирует фрейм в сеть.
4. Когда фрейм попадает на маршрутизатор, адаптер локальной сети LAN1 маршрутизатора считывает его с линии передачи данных и передает его программному обеспечению протокола.
5. Маршрутизатор извлекает межсетевой адрес из заголовка межсетевого пакета и использует его как указатель таблицы маршрутизации с целью определить, куда направить этот пакет в сети LAN2. Затем маршрутизатор удаляет старый заголовок LAN1, присоединяет впереди заголовок фрейма LAN2, адресованный хосту В, после чего передает построенный таким образом фрейм на адаптер.
6. Адаптер LAN2 маршрутизатора копирует этот фрейм в сеть.
7. Когда фрейм поступает на хост В, его адаптер считывает фрейм из линии передачи данных и передает его программному обеспечению протокола.
8. И в завершение, программное обеспечение протокола хоста В удаляет заголовок пакета и заголовок фрейма, копирует полученные данные в виртуальное адресное пространство сервера, когда сервер выполняет системный вызов, который читает эти данные.

Разумеется, здесь мы не заостряем ваше внимание на многих возникающих при этом сложных проблемах. Как быть в тех случаях, когда разные сети работают с различными максимальными размерами фреймов? Как маршрутизаторы получают информацию о том, что топология сети изменилась? Тем не менее наш пример отражает суть метода межсетевой передачи данных, а инкапсуляция является основным его принципом.

12.3. Глобальная сеть Internet

Глобальная сеть Internet, функционирующая на базе протокола IP (Internet Protocol), — наиболее известная и успешная реализация интерсети. В той или иной форме она существует с 1969 года. Несмотря на то, что внутренняя архитектура сети Internet

сложна и постоянно меняется, организация приложений типа клиент-сервер остается удивительно устойчивой, начиная с восьмидесятых годов прошлого столетия. На рис. 12.8 показана базовая организация Internet-приложения клиент-сервер.

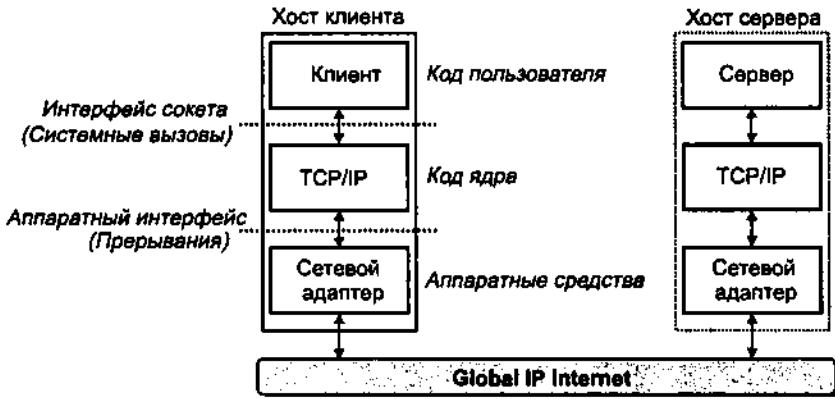


Рис. 12.8. Организация аппаратных и программных средств Internet-приложения

На каждом хосте Internet осуществляется прогон программного обеспечения, реализующего протокол TCP/IP (Transmission Control Protocol/Internet Protocol, протокол управления передачей/протокол Internet), который поддерживается практически каждой современной компьютерной системой. Клиенты и серверы Internet обмениваются данными, используя с этой целью комбинацию функций интерфейса сокетов и функций ввода-вывода системы Unix. Интерфейс сокетов мы опишем в разд. 12.4. Функции сокетов обычно выполняются как системные вызовы, которые попадают в ядро и вызывают различные функции, исполняемые в режиме ядра, в протокол TCP/IP.

Протокол TCP/IP — это фактически семейство протоколов, каждый из которых выполняет свою задачу. Например, протокол IP предоставляет базовую схему именования и механизм доставки, который посылает пакеты, известные как дейтаграммы (datagrams), с одного хоста сети Internet на другой хост. Механизм IP ненадежен в том смысле, что он не предпринимает никаких усилий для восстановления дейтаграмм в тех случаях, когда они утеряны или продублированы в сети. Протокол UDP (Unreliable Datagram Protocol, протокол ненадежных дейтаграмм) слегка расширяет протокол IP, благодаря чему пакеты можно передавать из процесса в процесс, а не с хоста на хост. TCP представляет собой сложный протокол, действующий на базе IP при установлении надежных полнодуплексных (двунаправленных) соединений между процессами. Чтобы упростить наш анализ, будем считать сочетание TCP/IP единым и монолитным протоколом. Мы не будем обсуждать его внутреннее устройство, мы будем рассматривать только некоторые основные свойства, которые протоколы TCP и IP предоставляют прикладным программам. Мы не будем рассматривать протокол UDP.

С точки зрения программиста, Internet можно рассматривать как совокупность всех хостов мира, которые обладают свойствами, приведенными ниже.

- Множество хостов отображается на множество 32-разрядных IP-адресов (IP addresses).
- Множество IP-адресов отображается на множество идентификаторов, так называемых доменных имен Internet (Internet domain names).
- Процесс, выполняемый на одном хосте Internet, может обмениваться данными с процессом, выполняемым на другом хосте Internet по установленному соединению (connection).

В трех следующих разделах мы будем изучать эти фундаментальные идеи, положенные в основу Internet, более подробно.

12.3.1. IP-адреса

IP-адрес есть 32-разрядное целое число без знака. Сетевые программы сохраняют IP-адреса в структуре IP-адресов.

```
struct in_addr {
    unsigned int s_addr; /* порядок следования байтов в сети */
};
```

Зачем хранить скалярный IP-адрес в структуре

Хранение скалярного адреса в структуре представляет собой неудачное наследие более ранних реализаций интерфейса сокетов. Целесообразнее объявить для IP-адресов специальный скалярный тип, но слишком поздно сейчас вносить какие-либо изменения, в силу того, что уже накопилась огромная база инсталлированных приложений.

Поскольку хосты Internet могут иметь различные порядки следования байтов, протокол TCP/IP объявляет единый порядок следования байтов в сети (порядок следования тупоконечника) для любых целочисленных элементов данных, таких как IP-адреса, которые переносятся через сеть в заголовках пакетов. Адреса в структурах IP-адресов всегда в прямом порядке следования байтов в сети, даже если на хосте используется обратный порядок следования байтов. Система Unix предоставляет следующие функции для изменения порядка следования байтов между хостом и сетью:

```
#include <netinet/in.h>;
unsigned long int htonl(unsigned long int hostlong);
unsigned short int htons(unsigned short int hostshort);

unsigned long int ntohl(unsigned long int netlong);
unsigned short int ntohs(unsigned short int netshort);
```

Функция `htonl` меняет порядок следования байтов 32-разрядного целого числа, принятый на хосте в порядок следования байтов, принятый на хосте. Функция `ntohl` меняет порядок следования байтов 32-разрядного целого числа, принятый в хосте, на порядок следования байтов, принятый в сети. Функции `htons` и `ntohs` выполняют соответствующие преобразования для 16-разрядных целых чисел.

Обычно человек работает с IP-адресами, представленными в формате, известном как десятичное представление с разделительными точками (dotted-decimal notation), в котором каждый байт представлен своим десятичным значением и отделен от других байтов точкой. Например, 128.2.194.242 есть десятичное представление адреса 0x8002c2f2. В операционной системе Linux вы можете воспользоваться командой `hostname`, чтобы узнать адрес вашего собственного компьютера в десятичном представлении с разделительными точками:

```
linux> hostname -i
128.2.194.242
```

Программы для Internet производят преобразования IP-адреса в строки десятичных значений, разделенных точками, и обратные преобразования, используя для этой цели функции `inet_aton` и `inet_ntoa`:

```
#include <arpa/inet.h>
int inet_aton(const char *cp, struct in_addr *inp);

char *inet_ntoa(struct in_addr in);
```

Функция `inet_aton` преобразует строку десятичных значений, разделенных точками, в IP-адрес с порядком следования байтов, принятым в сети. Аналогично, функция `inet_ntoa` преобразует IP-адреса с порядком следования байтов, принятых в сети, в соответствующую ей строку десятичных значений, разделенных точками. Обратите внимание на тот факт, что вызов функции `inet_aton` передает указатель на структуру, в то время как вызов функции `inet_ntoa` передает саму структуру.

Что означают `ntoa` и `aton`

Буква "n" означает сетевое (network) представление, буква "a" — представление, принятое в приложении (application), "to" означает передачу.

УПРАЖНЕНИЕ 12.1

Заполните следующую таблицу:

Шестнадцатеричный адрес	Адрес в десятичном представлении с разделительными точками
0x0	
0xffffffff	
0xef000001	
	205.188.160.121
	64.12.149.13
	205.188.146.23

УПРАЖНЕНИЕ 12.2

Напишите программу hex2dd.c, которая преобразует свой шестнадцатеричный аргумент в строку десятичных значений, разделенных точками, и распечатайте результат преобразования. Например,

```
unix> ./hex2dd 0x8002c2f2  
128.2.94.242
```

УПРАЖНЕНИЕ 12.3

Напишите программу dd2hex.c, которая преобразует свою строку десятичных значений, разделенных точками, в шестнадцатеричный аргумент, и распечатайте результат преобразования. Например,

```
unix> ./dd2hex 128.2.94.242  
0x8002c2f2
```

12.3.2. Имена доменов в Internet

Клиенты и серверы Internet используют IP-адреса, когда они обмениваются между собой данными. В то же время, люди плохо запоминают длинные числа, поэтому Internet кроме IP-адресов объявляет специальный набор дружественных по отношению к пользователю имен доменов, а также механизм, который отображает множество имен доменов на множество IP-адресов. Имя домена есть последовательность слов (букв, чисел и тире), отделенных друг от друга точками, например:

kittyhawk.cmcl.cs.cmu.edu

Множество имен доменов образует иерархию, и каждое имя домена кодирует свою позицию в этой иерархии. Проще всего понять это на примере. На рис. 12.9 показана некоторая часть иерархии имен доменов. Эта иерархия представлена в виде дерева. Узлы дерева представляют имена доменов, которые образованы посредством прохода по дереву в направлении его корня. Поддеревья интерпретируются как субдомены. Первый уровень иерархии образует неименованный корневой узел. Следующий уровень образует совокупность имен доменов первого уровня, определенных некоммерческой организацией под названием ICANN (Internet Corporation for Assigned Names and Numbers, новая некоммерческая организация по назначению адресов и имен в Internet, параметров протоколов, управлению системами доменных имен). Обычно домены первого уровня включают домены com, edu, gov, org и net.

На следующем уровне находятся доменные имена второго уровня, такие как, например, cmu.edu, которые назначаются по принципу "первым пришел — первым обслужен" различными уполномоченными агентами организации ICANN. Как только какая-либо организация получила доменное имя второго уровня, она получает возможность создавать другие новые доменные имена в пределах своего субдомена.

Internet определяет соответствие между множеством доменных имен и множеством IP-адресов. До 1988 года это соответствие устанавливалось вручную в специальном

текстовом файле с именем hosts.txt. С тех пор это соответствие поддерживалось в распределенной по всему миру базе данных, известной как DNS (Domain Naming System, служба имен доменов). По идее, база данных DNS состоит из многих миллионов входных структур доменов, изображенных в листинге 12.1, каждая из которых объявляет соответствие между множеством доменных имен (официальное имя и список альтернативных имен) и множеством IP-адресов. В математическом смысле вы можете представлять себе ввод хоста как класс эквивалентности доменных имен и IP-адресов.

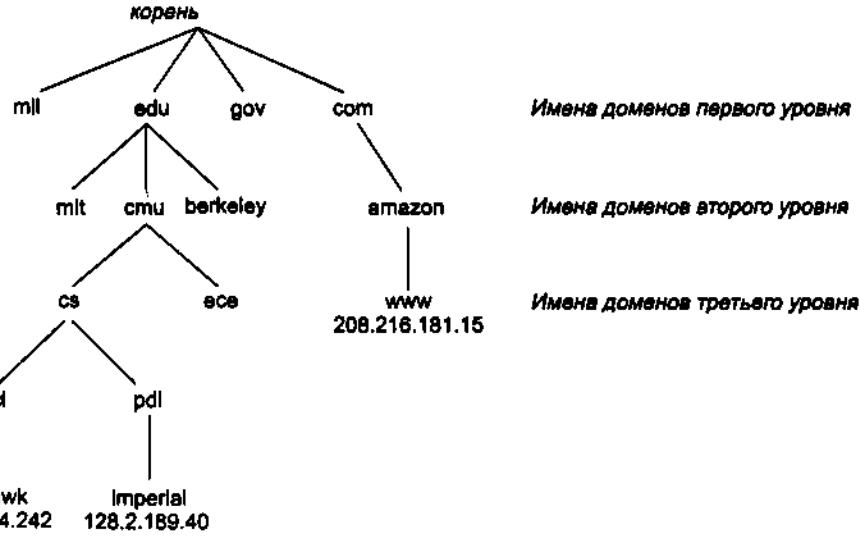


Рис. 12.9. Подмножество иерархии доменных имен в Internet

Листинг 12.1. Входная структура хостов DNS

```

struct hostent {
char *h_name; /* официальное доменное имя хоста */
char **h_aliases; /* массив доменных имен с завершающим нулем */
int h_addrtype; /* тип адреса хоста (AF_INET) */
int h_length; /* длина адреса в байтах */
char **h_addr_list; /* массив с завершающим нулем в структурах in_addr */
};
  
```

Internet-приложения получают произвольные вводы хоста из базы данных DNS путем обращения к функциям gethostbyname (найти хост по имени) и gethostbyaddr (найти хост по адресу).

```

#include <netdb.h>
struct hostent *gethostbyname(const char *name);
struct hostent *gethostbyaddr(const char *addr, int len, 0);
  
```

Функция `gethostbyname` возвращает ввод хоста, связанный с именем домена name. Функция `gethostbyaddr` возвращает ввод хоста, связанный с IP-адресом addr. Второй аргумент представляет длину IP-адреса в байтах, которая в текущей версии Internet всегда равна четырем байтам. Для наших целей третий аргумент всегда есть ноль.

Мы можем исследовать некоторые свойства отображения DNS с помощью программы `HOSTINFO`, представленной листингом 12.2, которая считывает имя домена или десятичный адрес с разделяющими точками из командной строки и отображает соответствующий ввод хоста.

Листинг 12.2 Программа осуществляет поиск и печатает вхождение хоста в DNS

```

1 #include "csapp.h"
2
3 int main(int argc, char **argv)
4 {
5     char **pp;
6     struct in_addr addr;
7     struct hostent *hostp;
8
9     if (argc != 2) {
10         fprintf(stderr, "usage: %s <domain name or dotted-decimal>\n",
11                 argv[0]);
12         exit(0);
13     }
14
15     if (inet_aton(argv[1], &addr) != 0)
16         hostp = Gethostbyaddr((const char *)&addr, sizeof(addr), AF_INET);
17     else
18         hostp = Gethostbyname(argv[1]);
19
20     printf("official hostname: %s\n", hostp->h_name);
21
22     for (pp = hostp->h_aliases; *pp != NULL; pp++)
23         printf("alias: %s\n", *pp);
24
25     for (pp = hostp->h_addr_list; *pp != NULL; pp++) {
26         addr.s_addr = *((unsigned int *)*pp);
27         printf("address: %s\n", inet_ntoa(addr));
28     }
29     exit(0);
30 }
```

Каждый хост Internet обладает локально определенным доменным именем `localhost`, которое всегда отображается на адрес обратной связи (loopback address) 127.0.0.1:

```
unix> ./hostinfo localhost
official hostname: localhost
alias: localhost.localdomain
address: 127.0.0.1
```

Имя `localhost` позволяет воспользоваться удобным и переносимым способом ссылки клиентов и серверов, исполняемых на одной и той же машине, что может быть очень полезным при отладке. Мы можем воспользоваться программой `HOSTNAME` для определения реального доменного имени нашего локального хоста:

```
unix> ./hostname
kittyhawk.cmcl.cs.cmu.edu
```

В простейшем случае существует взаимно однозначное отображение между доменным именем и IP-адресом:

```
unix> ./hostinfo kittyhawk.cmcl.cs.cmu.edu
official hostname: kittyhawk.cmcl.cs.cmu.edu
address: 128.2.194.242
```

Тем не менее, в некоторых случаях, на один и тот же IP-адрес отображаются сразу несколько доменных имен:

```
unix> ./hostinfo cs.mit.edu
official hostname: EECS/MIT.EDU
alias: cs.mit.edu
address: 18.62.1.6
```

В самом общем случае несколько доменных имен могут отображаться на несколько IP-адресов:

```
unix> ./hostinfo www.aol.com
official hostname: aol.com
alias: www.aol.com
address: 205.188.160.121
address: 64.12.149.13
address: 205.188.146.23
```

И наконец, мы замечаем, что некоторые действительные доменные номера не отображаются ни на какие IP-адреса:

```
unix> ./hostinfo edu
Gethostbyname error: Не существует адрес, связанный с этим именем
unix> ./hostinfo cmcl.cs.cmu.edu
Gethostbyname error: Не существует адрес, связанный с этим именем
```

Примечание

С 1987 года дважды в год консорциум программного обеспечения Internet (Internet Software Consortium) (www.isc.org) проводит обзор доменов Internet. Отчет, который дает оценку числа хостов Internet путем подсчета числа IP-адресов, которые

были назначены конкретному доменному имени, обнаружил любопытную тенденцию. Начиная с 1987 года, когда в Internet было всего лишь примерно 20 тыс. хостов, число хостов ежегодно удваивалось примерно вдвое. К июню 2001 года в Internet было более 120 млн хостов.

УПРАЖНЕНИЕ 12.4

Выполните компиляцию программы `HOSTINFO`, представленной в листинге 12.2. Затем выполните `hostinfo aol.com` три раза подряд в вашей системе.

1. Что вы можете сказать об упорядочении IP-адресов в трех вводах хоста?
2. Как это упорядочение может оказаться полезным?

12.3.3. Internet-соединения

Клиенты и серверы Internet обмениваются между собой данными, посылая и принимая потоки байтов. Соединение называется двухточечным (*point-to-point*) в том смысле, что оно устанавливает канал передачи данных, связывающий два процесса. Это полнодуплексное соединение (*full-duplex*), при котором данные могут одновременно передаваться в обоих направлениях. В то же время, такое соединение считается надежным (*reliable*) в том смысле, что, если отвлечься от некоторых катастрофических событий, таких как повреждение кабеля пресловутым беспечным оператором экскаватора, поток байтов, посыпаемый процессом-источником, в конечном итоге будет получен процессом назначения в том порядке, в каком он был отправлен.

Сокет представляет собой конечную точку соединения. Каждый сокет обладает собственным адресом сокета, который состоит из адреса в Internet и 16-разрядного целочисленного порта и обозначается как `address:port`. Порт в адресе сокета клиента назначается ядром автоматически в тот момент, когда клиент делает запрос на соединение, и называется эфемерным портом (*ephemeral port*). В то же время, порт в адресе хоста сервера — это обычно хорошо известный порт, который ассоциирован с соответствующей службой. Например, Web-серверы обычно используют порт 80, в то время как серверы электронной почты применяют порт 25. На машинах, работающих под управлением операционной системы Unix, файл `/etc/services` содержит подробный список услуг, предоставляемых на такой машине, а также список хорошо известных портов.

Соединение однозначно идентифицируется адресами сокетов обеих его конечных точек. Такая пара адресов сокетов называется парой сокетов и обозначается следующим кортежем (`cliaddr:cliport, servaddr:servport`), где `cliaddr` есть IP-адрес клиента, `cliport` — порт клиента, `servaddr` — IP-адрес сервера и `servport` — порт сервера. Например, на рис. 12.10 показано соединение между Web-клиентом и Web-сервером. В этом примере адрес сокета Web-клиента есть `128.2.194.242:51213`, где `51213` есть обозначение эфемерного порта, назначенное ядром. Адрес сокета Web-сервера есть `208.216.181.15:80`, где порт `80` есть замечательный порт, связанный с соответствующей Web-службой. Когда заданы эти адреса клиента и сервера, соединение между клиентом и сервером единственным образом идентифицируется парой сокетов (`128.2.194.242:51213, 208.216.181.15:80`).

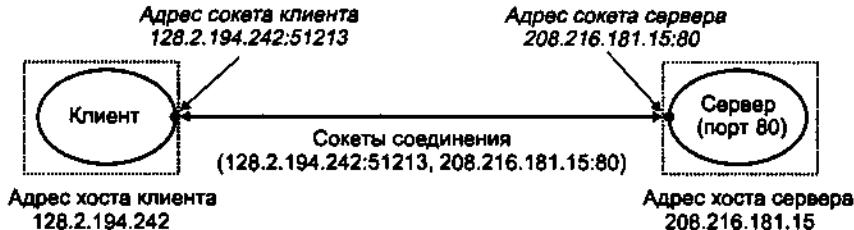


Рис. 12.10. Анатомия Internet-соединения

Происхождение сети Internet

Сеть Internet является примером одного из наиболее успешных взаимодействий правительства, университета и промышленности. Многие факторы внесли свой вклад в ее успех, но, по нашему мнению, особенно важным является тот факт, что правительство Соединенных Штатов оказывало ему финансовую поддержку на протяжении 30-летнего периода, а также та самоотверженность, с которой разработчики трудились над реализацией проекта.

Семена Internet были посеяны в 1957 г., когда в самый разгар холодной войны Советский Союз потряс весь мир, запустив в космос свой спутник, первый искусственный спутник Земли. В ответ правительство Соединенных Штатов создает агентство ARPA (Advanced Research Projects Agency, Управление перспективного планирования научно-исследовательских работ), перед которым была поставлена задача восстановить лидерство США в науке и технологиях. В 1967 г. Лоренс Робертс (Lawrence Roberts) из агентства ARPA опубликовал планы проведения работ по созданию новой сети, получившей название ARPANET (Advanced Research Projects Agency network). Первые узлы ARPANET начали работать в 1969 г. В 1971 г. существовало уже 13 узлов сети ARPANET, а электронная почта появилась как первое важное сетевое приложение.

В 1972 г. Роберт Кан (Robert Kahn) сформулировал общие принципы обеспечения межсетевого обмена: совокупность взаимосвязанных сетей, обмен данными между этими сетями производится независимо, по принципу обязательного приложения максимальных усилий посредством использования черных ящиков, получивших название "маршрутизаторов". В 1974 г. Кан и Винтон Серф (Vinton Cerf) опубликовали первые детали протокола TCP/IP, который к 1982 г. стал стандартным протоколом межсетевого взаимодействия для сети ARPANET. С 1 января 1983 года каждый узел сети ARPANET был переключен на протокол TCP/IP, эта дата стала днем рождения глобальной сети Internet, функционирующей на базе протокола IP.

В 1985 г., когда Пол Мокапетрис (Paul Mockapetris) изобрел систему DNS (Domain Name System, служба имен доменов), существовало порядка 1000 хостов Internet. В следующем году фонд NSF (National Science Foundation, Национальный научный фонд США) построил магистральную сеть NSFNET, соединившую 13 площадок посредством коммутируемых телефонных линий со скоростью передачи данных, равной 56 Кбайт/с. Позже, в 1988 г. были использованы линии связи T1, обеспечивающие скорость передачи данных 1.5 Мбайт/с, а в 1991 г. были использованы линии связи T3, обеспечивающие скорость передачи данных 45 Мбайт/с. В 1988 г. было более 50 тыс. хостов. В 1989 г. исходная сеть ARPANET официально пере-

стала существовать. В 1995 г., когда в сети Internet было порядка 10 млн хостов, фонд NSF отправил в небытие сеть NSFNET, заменив ее современной сетью с архитектурой на базе специализированных коммерческих магистральных сетей, соединенных посредством общедоступных мест сетевого доступа.

12.4. Интерфейс сокетов

Интерфейс сокетов представляет собой некоторое множество функций, которые используются совместно с функциями ввода-вывода операционной системы Unix с целью построения сетевых приложений. Он реализован в большинстве современных систем, включая все варианты операционных систем Unix, Windows и Macintosh. Рис. 12.11 дает представление о интерфейсе сокетов в контексте обычной транзакции клиент-сервер. Вы должны пользоваться этим рисунком, как дорожной картой при обсуждении отдельных функций.

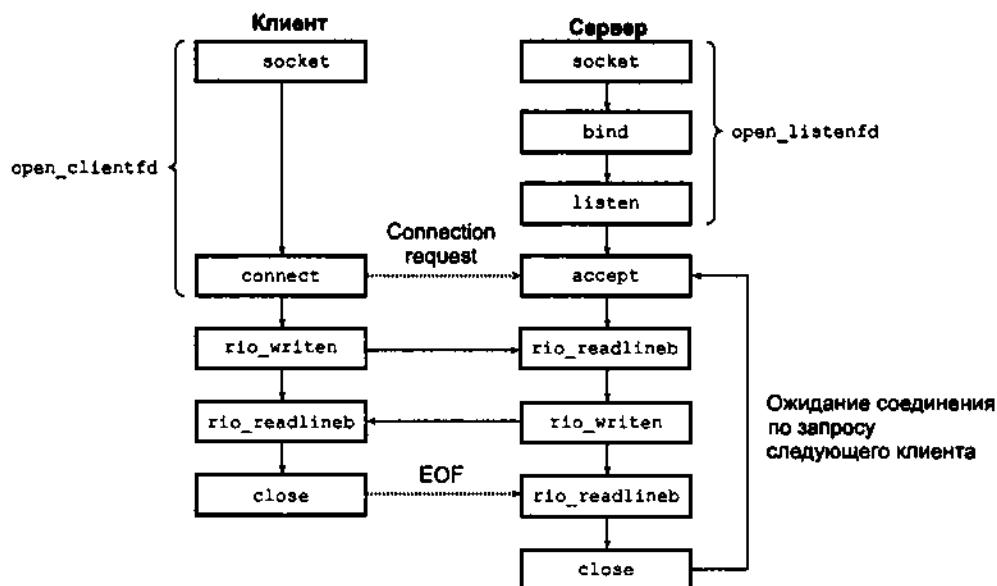


Рис. 12.11. Обзор интерфейса сокетов

Происхождение интерфейса сокетов

Интерфейс сокетов был разработан исследователями Калифорнийского университета в Беркли в начале восьмидесятых годов. По этой причине его часто называют сокетами Беркли (Berkeley sockets). Исследователи из Беркли разработали интерфейс сокетов, позволяющий работать с любым базовым протоколом. Первой реализацией был протокол, который они включили ядро Unix 4.2BSD (Berkeley Software Distribution, программное изделие Калифорнийского университета) и распределили по многочисленным университетам и лабораториям. Это было важное

событие в истории Internet. Практически в одну ночь тысячи пользователей получили доступ к протоколу TCP/IP и его исходным кодам. Это породило чрезвычайное волнение в среде пользователей и стимулировало дальнейшие исследования в области построения сетей и межсетевого обмена.

12.4.1. Адресные структуры сокетов

С точки зрения ядра операционной системы Unix, сокет представляет собой конечную точку соединения. С точки зрения программы системы Unix сокет есть открытый файл с соответствующим дескриптором.

Адреса сокетов в Internet хранятся в виде 16-байтовых структур типа `sock_addr_in`, показанных в листинге 12.3. Для Internet-приложений элемент семейства `sin_family` есть `AF_INET`, элемент `sin_port` есть 16-разрядный номер порта, а элемент `sin_addr` есть 32-разрядный IP-адрес. IP-адрес и номер порта всегда хранятся в порядке следования байтов, принятом в сети (тупоконечник).

Листинг 12.3. Адресные структуры сокетов

```
/* Общий адрес структуры сокета (для соединения, связывания и приема) */
struct sockaddr {
    unsigned short sa_family; /* семейство протоколов */
    char sa_data[14]; /* адресные данные. */
};

/* Адресная структура сокета в стиле Internet */
struct sockaddr_in {
    unsigned short sin_family; /* семейство адресов (always AF_INET) */
    unsigned short sin_port; /* номер порта в порядке следования байтов, принятом
        в сети */
    struct in_addr sin_addr; /* IP-адреса в сетевом порядке следования байтов */
    unsigned char sin_zero[8]; /* место для значения sizeof(struct sockaddr) */
};
```

Функции соединения требуют указателя на адресную структуру сокета, ориентированную на конкретный протокол. Проблема, с которой приходится сталкиваться разработчикам интерфейса сокетов — как определить эти функции с тем, чтобы иметь возможность работать с любой адресной структурой сокета. Теперь мы будем употреблять общий указатель `void*`, который не существовал в то время в языке C. Решение заключалось в том, чтобы определить функции сокетов, ожидающих указателя на общую структуру `sockaddr`, а затем потребовать от приложений, чтобы они привели указатели на эту общую структуру к протокольно-специфическим структурам. Чтобы упростить наши программные коды, последуем указаниям Стивенса и определим следующий тип:

```
typedef struct sockaddr SA;
```

Затем мы будем применять этот тип всякий раз, когда нам понадобится привести структуру `sockaddr_in` к общей структуре `sockaddr`.

12.4.2. Функция `socket`

Клиенты и серверы используют функцию `socket` для построения описателя сокета:

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

В наших программных кодах мы всегда вызываем функцию `socket` с аргументами

```
clientfd = Socket(AF_INET, SOCK_STREAM, 0);
```

где `AF_INET` указывает на то, что мы собираемся воспользоваться сетью Internet, а `SOCK_STREAM` указывает на то, что сокет будет конечной точкой Internet-соединения. Описатель `clientfd`, возвращаемый функцией `socket`, открыт только частично и не может быть использован в операциях считывания и записи. Как мы завершим процедуру открывания сокета, зависит от того, чем мы являемся, клиентом или сервером. В следующем подразделе описано, как мы завершаем процедуру открывания сокета, будучи клиентом.

12.4.3. Функция `connect`

Клиент устанавливает соединение с сервером путем вызова функции `connect`:

```
#include <sys/socket.h>
int connect(int sockfd, struct sockaddr *serv_addr, int addrlen);
```

Функция `connect` предпринимает попытку установить соединение Internet с сервером по адресу сокета `serv_addr`, где `addrlen` есть значение `sizeof(sockaddr_in)`. Функция `connect` блокирует выполнение до тех пор, пока либо соединение будет успешно установлено, либо возникнет ошибка. Если соединение будет успешно установлено, описатель `sockfd` в этом случае готов выполнять чтение и запись, а установленное соединение описывается парой сокета

```
(x:y, serv_addr.sin_addr:serv_addr.sin_port)
```

где `x` есть IP-адрес клиента, а `y` есть эфемерный порт, который единственным образом идентифицирует процесс клиента, исполняемый на хосте клиента.

12.4.4. Функция `open_clientfd`

Мы пришли к заключению, что удобно представить функции `socket` и `connect` в виде вспомогательной функции под именем `open_clientfd`, которой может воспользоваться клиент, с целью установить соединение:

```
#include "csapp.h"
int open_clientfd(char *hostname, int port);
```

Функция `open_clientfd` устанавливает соединение с сервером, функционирующим на хосте `hostname`, и прослушивает запросы на установление соединения на `port`. Она возвращает описатель открытого сокета, который готов осуществлять ввод и вывод посредством функций ввода-вывода системы Unix. В листинге 12.4 приводится программный код функции `open_clientfd`.

Листинг 12.4. Вспомогательная функция, которая устанавливает соединение

```

1 int open_clientfd(char *hostname, int port)
2 {
3     int clientfd;
4     struct hostent *hp;
5     struct sockaddr_in serveraddr;
6
7     if ((clientfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
8         return -1; /* проверьте номер ошибки с целью выяснить причину ошибки */
9
10    /* Введите IP-адрес и порт сервера */
11    if ((hp = gethostbyname(hostname)) == NULL)
12        return -2; /* проверьте h_errno с целью выяснить причину ошибки */
13    bzero((char *) &serveraddr, sizeof(serveraddr));
14    serveraddr.sin_family = AF_INET;
15    bcopy((char *)hp->h_addr,
16          (char *)&serveraddr.sin_addr.s_addr, hp->h_length);
17    serveraddr.sin_port = htons(port);
18
19    /* Установить соединение с сервером */
20    if (connect(clientfd, (SA *)&serveraddr, sizeof(serveraddr)) < 0)
21        return -1;
22    return clientfd;
23 }
```

После создания описателя сокета (строка 7) мы находим запись DNS, содержащую данные о хосте, на котором находится сервер, и копируем первый IP-адрес записи хоста (который уже представлен в порядке следования байтов, установленном в сети) в структуру адреса сокета сервера (строки 11—16). После инициализации адресной структуры сокета с номером порта в сетевом порядке следования байтов (строка 17) мы инициируем запрос на соединение с сервером (строка 20). Когда функция `connect` возвращает результат, мы возвращаем описатель сокета клиенту, который теперь немедленно может использовать функции ввода-вывода системы Unix для обмена данными с сервером.

12.4.5. Функция `bind`

Остальные функции сокетов — `bind`, `listen` и `accept` — используются серверами для установления соединения с клиентами.

```
#include <sys/socket.h>
int bind(int sockfd, struct sockaddr *my_addr, int addrlen);
```

Функция `bind` требует от ядра связать адрес сокета сервера в `my_addr` с дескриптором сокета `sockfd`. Аргумент функции `addrlen` есть значение `sizeof(sockaddr_in)`.

12.4.6. Функция *listen*

Клиенты активны и инициируют запросы на установление соединения. Серверы пассивны и ожидают запросов на установление соединений от клиентов. По умолчанию ядро полагает, что описатель, построенный функцией `socket`, соответствует активному сокету, который существует на другом конце соединения. Сервер вызывает функцию `listen`, чтобы сообщить ядру, что описатель будет использован сервером, а не клиентом.

```
#include <sys/socket.h>
int listen(int sockfd, int backlog);
```

Функция `listen` преобразует активный сокет `sockfd` в прослушивающий сокет, который может принимать запросы на установление соединения от клиентов. Аргумент `backlog` есть указание на число запросов на установление соединений, которые ядро должно поставить в очередь, прежде чем начнет отклонять запросы. Точное понимание аргумента `backlog` требует подробного изучения протокола TCP/IP, что выходит за пределы материала, рассматриваемого в данной книге. В дальнейшем мы будем назначать ему большое значение, например, 1024.

12.4.7. Функция *listenfd*

Включим функции `socket`, `bind` и `listen` в одну вспомогательную функцию под названием `open_listenfd`, которой сервер может воспользоваться для создания описателя прослушивания.

```
#include "csapp.h"
int open_listenfd(int port);
```

Функция `open_listenfd` открывает и возвращает описатель прослушивания, который готов принимать запросы на установление соединения через порт. В листинге 12.5 приводится код функции `open_listenfd`. После того как мы создадим описатель сокета `listenfd`, мы используем функцию `setsockopt` (ее описание здесь не приводится) с целью конфигурирования сервера, чтобы его можно было остановить, а затем немедленно запустить повторно. По умолчанию повторно запускаемый сервер будет отвергать запросы клиентов на установление соединений в течение примерно 30 с, что серьезно затрудняет отладку.

Листинг 12.5. Вспомогательная функция

```
1  int open_listenfd(int port)
2  {
3      int listenfd, optval=1;
```

```

4   struct sockaddr_in serveraddr;
5
6   /* Создать дескриптор сокета */
7   if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
8       return -1;
9
10  /* Удаляет ошибку "Адрес уже используется" из bind. */
11  if (setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR,
12      (const void *)&optval, sizeof(int)) < 0)
13      return -1;
14
15  /* Listenfd будет конечной точкой для всех запросов к порту
16  по любому IP-адресу для данного хоста */
17 bzero((char *)&serveraddr, sizeof(serveraddr));
18 serveraddr.sin_family = AF_INET;
19 serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
20 serveraddr.sin_port = htons((unsigned short)port);
21 if (bind(listenfd, (SA *)&serveraddr, sizeof(serveraddr)) < 0)
22     return -1;
23
24  /* Сделать его прослушивающим сокетом, готовым к приему запросов
25  на установление соединений */
26 if (listen(listenfd, LISTENQ) < 0)
27     return -1;
28 }

```

Далее инициализируем адресную структуру сокета сервера с тем, чтобы подготовиться к вызову функции `bind`. В этом случае мы использовали `INADDR_ANY`, чтобы сообщить ядру, что данный сервер принимает запросы по любым IP-адресам этого хоста (строка 19) через порт (строка 20). Обратите внимание на то обстоятельство, что мы пользуемся функциями `htonl` и `htons` для преобразования IP-адресов и номеров портов, заменяя порядок следования байтов, принятый для хоста, на порядок следования, принятый в сети. В завершение мы преобразуем функцию `listenfd` в прослушивающий дескриптор (строка 25) и возвращаем его вызывающему объекту.

12.4.8. Функция `accept`

Серверы ожидают запросы на установление соединений от клиентов, вызывая функцию `accept`:

```
#include <sys/socket.h>
int accept(int listenfd, struct sockaddr *addr, int *addrlen);
```

Функция `accept` ожидает от клиентов запросов на установление соединений, которые поступают в прослушивающем дескрипторе функции `listenfd`, затем записывает

адрес сокета клиента в `addr` и возвращает связный дескриптор, который используется для обмена данными с клиентом посредством функций ввода-вывода операционной системы Unix.

Различия между прослушивающим дескриптором и связным дескриптором приводят в замешательство многих студентов. Прослушивающий дескриптор служит конечной точкой запросов клиента на установление соединения. Обычно он создается один раз и существует на протяжении продолжительности жизни соответствующего сервера. Связный дескриптор есть конечная точка соединения, установленного между клиентом и сервером. Он создается всякий раз, когда сервер принимает запрос на установление соединения, и существует только на протяжении периода обслуживания клиента.

На рис. 12.12 показано, какие роли возложены на прослушивающий и связный дескрипторы. На этапе 1 сервер вызывает функцию `accept`, которая ждет, когда запрос на установление соединения появится на прослушивающем дескрипторе, который мы, для конкретности, назовем дескриптором 3. Напомним, что дескрипторы 0—2 зарезервированы для стандартных файлов.

На этапе 2 клиент вызывает функцию `connect`, которая посылает запрос на установление соединения функции `listenfd`. На этапе 3 функция `accept` открывает новый связный дескриптор `connfd` (который мы будем рассматривать как дескриптор 4), устанавливает соединение между `clientfd` и `connfd`, а затем возвращает `connfd` приложению. Клиент возвращается из функции `connect`, и начиная с этого момента, клиент и сервер могут передавать данные туда и назад, считывая и записывая, соответственно, посредством `clientfd` и `connfd`.

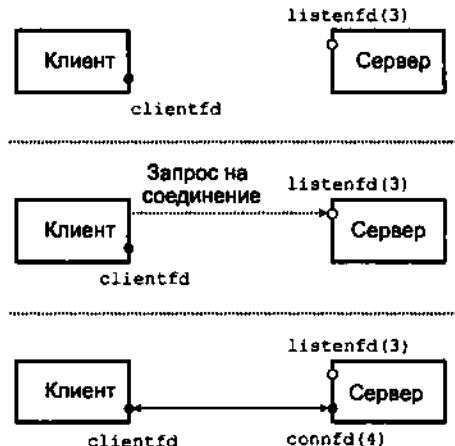


Рис. 12.12. Роли прослушивающих и связных дескрипторов

Почему возникает различие между прослушивающими и связными дескрипторами

Вас, возможно, удивляет, почему интерфейс сокетов проводит различие между прослушивающими и связными дескрипторами. На первый взгляд это выглядит

ненужным осложнением. Тем не менее фиксация различия между этими двумя дескрипторами приносит определенную пользу, поскольку оно позволяет нам строить параллельно работающие серверы, которые способны осуществлять обработку многих клиентских соединений одновременно. Например, всякий раз, когда на прослушивающем дескрипторе появляется запрос на установление соединения, мы можем отложковать новый процесс, который обменивается данными с клиентом через связанный дескриптор. Более подробно параллельные серверы будут описаны в главе 13.

12.4.9. Примеры эхо-клиента и эхо-сервера

Наилучший способ изучения интерфейса сокетов заключается в изучении примеров программных кодов. В листинге 12.6 представлен программный код эхо-клиента. После установления соединения с сервером клиент входит в цикл, в котором много-кратно считывает строки текстов со стандартного ввода, посыпает строку текста на сервер, считывает эхо-строку с сервера и передает результат на стандартный вывод. Этот цикл прерывается в тот момент, когда функция `fgets` получает признак EOF со стандартного ввода, либо в силу того, что пользователь ввел с клавиатуры комбинацию `<Ctrl>+<D>` в командной строке, либо в силу того, что в переадресованном входном файле текстовые строки исчерпаны.

После того как цикл закончится, клиент закрывает дескриптор. Сообщение об этом событии в виде признака EOF пересыпается на сервер, который он обнаруживает, когда получает код возврата, равный нулю, от `rio_readlineb`. После того как клиент закроет дескриптор, он завершает работу. По завершении процесса ядро клиента автоматически закрывает все открытые дескрипторы, и команда `close` в строке 24 больше не нужна. Тем не менее хороший тон в программировании требует, чтобы мы явным образом закрывали все дескрипторы, которые мы когда-то открыли.

Листинг 12.6. Главная программа эхо-клиента

```

1  #include "csapp.h"
2
3  int main(int argc, char **argv)
4  {
5      int clientfd, port;
6      char *host, buf[MAXLINE];
7      rio_t rio;
8
9      if (argc != 3) {
10          fprintf(stderr, "usage: %s <host> <port>\n", argv[0]);
11          exit(0);
12     }
13     host = argv[1];
14     port = atoi(argv[2]);
15 }
```

```

16 clientfd = Open_clientfd(host, port);
17 Rio_readinitb(&rio, clientfd);
18
19 while (Fgets(buf, MAXLINE, stdin) != NULL) {
20     Rio_writen(clientfd, buf, strlen(buf));
21     Rio_readlineb(&rio, buf, MAXLINE);
22     Fputs(buf, stdout);
23 }
24 Close(clientfd);
25 exit(0);
26 }

```

В листинге 12.7 представлена основная программа эхо-сервера. После того как будет открыт прослушивающий дескриптор, он оказывается в бесконечном цикле. Каждая итерация этого цикла работает в режиме ожидания поступления от клиентов запроса на установление соединения, она распечатывает доменное имя и IP-адрес подключенного клиента и вызывает эхо-функцию, которая обслуживает клиента. После того как эхо-программа возвратит управление, основная программа закрывает связный дескриптор. Как только клиент и сервер закроют соответствующие дескрипторы, соединение разрывается.

Листинг 12.7. Итеративная программа эхо-сервера

```

1 #include "csapp.h"
2
3 void echo(int connfd);
4
5 int main(int argc, char **argv)
6 {
7     int listenfd, connfd, port, clientlen;
8     struct sockaddr_in clientaddr;
9     struct hostent *hp;
10    char *haddrp;
11    if (argc != 2) {
12        fprintf(stderr, "usage: %s <port>\n", argv[0]);
13        exit(0);
14    }
15    port = atoi(argv[1]);
16
17    listenfd = Open_listenfd(port);
18    while (1) {
19        clientlen = sizeof(clientaddr);
20        connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
21
22        /* определить доменное имя и IP-адрес клиента */
23        hp = Gethostbyaddr((const char *)&clientaddr.sin_addr.s_addr,
24                           sizeof(clientaddr.sin_addr.s_addr), AF_INET);

```

```

25     haddrp = inet_ntoa(clientaddr.sin_addr);
26     printf("server connected to %s (%s)\n", hp->h_name, haddrp);
27
28     echo(connfd);
29     Close(connfd);
30 }
31 exit(0);
32 }
```

Что означает признак EOF при установлении соединения

Идея использования признака EOF часто воспринимается с трудом, особенно в контексте Internet-соединений. Во-первых, мы должны осознавать, что не существует таких вещей, как символ EOF. Скорее наоборот, EOF есть условие, которое обнаруживается ядром. Приложение обнаруживает условие EOF, когда оно в качестве кода возврата получает от функции read нуль. Для дисковых файлов условие EOF возникает, когда позиция в текущем файле превышает длину файла. Для Internet-соединений EOF возникает в тех случаях, когда какой-либо процесс закрывает свой конец соединения. Процесс на другом конце соединения обнаруживает EOF, когда он делает попытку прочитать последний байт из потока байтов.

Обратите внимание на тот факт, что простой эхо-сервер может обслуживать одновременно только одного клиента. Сервер этого типа, просматривающий клиентов по одному за раз, называется итеративным сервером. В главе 13 мы узнаем, как можно построить более сложные параллельные серверы (concurrent servers), которые могут обслуживать одновременно несколько клиентов.

И наконец, в листинге 12.8 представлен код эхо-программы, которая периодически читает и пишет строки текста до тех пор, пока функция rio_readlineb признак конца файла в строке 10.

Листинг 12.8. Эха-функция

```

1 #include "csapp.h"
2
3 void echo(int connfd)
4 {
5     size_t n;
6     char buf[MAXLINE];
7     rio_t rio;
8
9     Rio_readinitb(&rio, connfd);
10    while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
11        printf("server received %d bytes\n", n);
12        Rio_writen(connfd, buf, n);
13    }
14 }
```

12.5. Web-серверы

До сих пор мы обсуждали сетевое программирование в контексте простого эхосервера. В данном разделе мы покажем, как следует использовать базовые идеи сетевого программирования, чтобы построить ваш собственный небольшой, но в то же время полностью работоспособный Web-сервер.

12.5.1. Основные сведения о Web

Web-клиенты и Web-серверы взаимодействуют, используя текстовый протокол уровня приложений, известный как HTTP (Hypertext Transfer Protocol, протокол передачи гипертекстовых данных). Протокол HTTP достаточно прост. Web-клиент (браузер) открывает Internet-соединение с сервером и запрашивает некоторое содержимое. Сервер отвечает тем, что представляет затребованное содержимое, после чего разрывает соединение. Браузер считывает содержимое и отображает его на экране.

Что отличает Web-сервер от обычной службы поиска файлов, таких как FTP (File Transfer Protocol, протокол передачи файлов)? Основное их отличие заключается в том, что Web-содержимое может быть записано в языке HTML (Hypertext Markup Language, гипертекстовый язык описания документов). Программа на языке HTML (страница) содержит инструкции (теги), которые указывают браузеру, как отображать различные тексты и графические объекты на странице. Например, код

```
<b> Make me bold! </b>
```

представляет собой команду распечатки текста между тегами `` и `` в полужирном шрифте. Однако основное достоинство языка HTML заключается в том, что страница может содержать указатели (гипертекстовые связи) с содержимым, хранящимся на одном из хостов сети. Например, строка HTML вида

```
<a href="http://www.cmu.edu/index.html">Carnegie Mellon</a>
```

представляет собой команду выделения текстового объекта "Carnegie Mellon" и создания гипертекстовой связи с файлом HTML с именем `index.html`, который хранится на Web-сервере CMU (Carnegie-Melone University, университет Карнеги-Меллона). Если пользователь щелкнет на выделенном текстовом объекте, то браузер затребует соответствующий файл HTML с сервера CMU, после чего отобразит его на экране.

Происхождение World Wide Web

World Wide Web (Всемирная паутина) была изобретена Тимом Бернерсом-Ли (Tim Berners-Lee), специалистом по программному обеспечению в швейцарской физической лаборатории CERN (European Organisation for Nuclear Research, Европейская организация ядерных исследований). В 1989 г. Бернерс-Ли опубликовал в международных научных изданиях заметку с предложением распределенной гипертекстовой системы, которая соединяла бы сеть узлов линиями передачи данных. Назначение предложенной системы заключалось в том, чтобы помочь ученым CERN совместно использовать и управлять информацией. Через два с небольшим года после того, как Бернерс-Ли реализовал первый Web-сервер и Web-браузер, Web

разработала небольшую сеть, аналогичную сети CERN, и несколько других сайтов. Фундаментальное событие произошло в 1993 г., когда Марк Андерсен (Marc Andreessen) (он впоследствии основал компанию *Netscape*) и его коллеги из NCSA (National Center for Supercomputing Applications, Национальный центр по использованию суперкомпьютеров) разработали и внедрили графический браузер MOSAIC для всех трех основных платформ: Unix, Windows и Macintosh. С появлением браузера MOSAIC интерес к Web приобрел характер взрыва, при этом число Web-сайтов ежегодно увеличивалось в 10 и более раз. К 2002 г. в мире существовало более 36 млн сайтов (см. отчет Netcraft Web Survey на сайте www.netcraft.com).

12.5.2. Содержимое Web

Для Web-клиентов и Web-серверов содержимое (content) представляет собой последовательность байтов со связанным с ними типом MIME (Multipurpose Internet Mail Extensions, многоцелевые расширения электронной почты в сети Internet). В табл. 12.1 показаны некоторые общеупотребительные типы MIME.

Таблица 12.1. Примеры типов MIME

Тип MIME	Описание
text/html	Страница HTML
text/plain	Неформатированный текст
application/postscript	Документ в языке Postscript
image/gif	Бинарное отображение, представленное в формате GIF
image/gif jpeg	Бинарное отображение, представленное в формате JPEG

Web-серверы предоставляют клиентам содержимое двумя различными способами:

- Вызовите дисковый файл и возвратите его содержимое клиенту. Этот дисковый файл представляет собой неизменяемое содержимое, процесс доставки файла клиенту называется обслуживанием статического содержимого.
- Выполните исполняемый файл и возвратите его выходные данные клиенту. Выходные данные, полученные при прогоне исполняемого файла, называются динамическим содержимым, а сам процесс исполнения программы и доставки ее выходных результатов клиенту называется обслуживанием динамического содержимого.

Каждая порция содержимого, возвращенного Web-сервером, ассоциируется с некоторым файлом, которым она управляет. Каждый из этих файлов имеет уникальное имя, называемое указателем URL (Universal Resource Locator, унифицированный указатель информационного ресурса). Например, URL <http://www.aol.com:80/index.html> идентифицирует файл HTML с именем /index.html на хосте www.aol.com сети Internet, который управляет Web-сервером, прослушивающим порт 80. Номер пор-

та не обязательно указывать, по умолчанию таким портом является порт HTTP с номером 80. Указатели URL исполняемых файлов могут включать аргументы программ после имени файла. Символ ? отделяет имя файла от аргументов, а каждый аргумент отделен символом &. Например, указатель URL `http://kittyhawk.cmcl.cs.cmu.edu:8000/cgi-bin/adder?15000&213` идентифицирует исполняемый файл под именем `/cgi-bin/adder`, который вызывается с двумя строками аргументов: 15000 и 213. Клиенты и серверы используют различные части указателя URL во время обработки транзакций. Например, клиент использует префикс `http://www.aol.com:80` с целью определить, с каким видом сервера устанавливать контакт, где находится сервер и какой порт он прослушивает. Сервер использует суффикс `/index.html` для поиска файла в своей файловой системе для того, чтобы определить, какое содержимое запрашивается, статическое или динамическое.

Выделим несколько моментов, которые необходимо понимать, чтобы правильно интерпретировать суффиксы указателей URL.

- Не существует стандартных правил для определения, ссылается ли указатель URL на статическое или динамическое содержимое. У каждого сервера имеются собственные правила для файлов, которыми он манипулирует. Обычный подход заключается в идентификации некоторого множества каталогов, таких как `cgi-bin`, в котором должны храниться все исполняемые файлы.
- Начальная черта / в суффиксе не означает корневого каталога Unix. Напротив, он обозначает начальный каталог для любого вида запрашиваемого содержимого. Например, сервер может быть сконфигурирован таким образом, чтобы все файлы статического содержимого хранились в каталоге `/usr/httpd/html`, а все файлы динамического содержимого хранились в каталоге `/usr/httpd/cgi-bin`.
- Суффикс минимальной длины указателя URL есть символ /, который все серверы расширяют до некоторой стандартной начальной страницы, такой как `/index.html`. Это объясняет, почему становится возможным извлечь начальную страницу простым вводом доменного имени в браузер. Этот браузер добавляет отсутствующий символ / к указателю URL и передает его в сервер, который расширяет / до некоторого стандартного имени файла.

12.5.3. Транзакции HTTP

Поскольку протокол HTTP основан на строках текста, передаваемого по соединениям Internet, мы можем пользоваться программой TELNET операционной системы Unix для проведения транзакций с любым Web-сервером по сети Internet. Программа TELNET очень полезна при отладке серверов, которые общаются с клиентами, передавая текстовые строки через сетевые соединения. Например, листинг 12.9 использует TELNET для запроса домашней страницы Web-сервера AOL.

Листинг 12.9. Транзакция HTTP, которая обслуживает статическое содержимое

```
1 unix> telnet www.aol.com 80      Клиент: открывает соединение
2 Trying 205.188.146.23...          Telnet печатает 3 строки на терминал
3 Connected to aol.com.
```

```

4 Escape character is '€'.
5 GET / HTTP/1.1
6 host: www.aol.com
7
8 HTTP/1.0 200 OK
9 MIME-Version: 1.0

10 Date: Mon, 08 Jan 2001 04:59:42 GMT
11 Server: NaviServer/2.0 AOLserver/2.3.3
12 Content-Type: text/html
13 Content-Length: 42092
14
15 <html>
16 ...
17 </html>
18 Connection closed by foreign host.
19 unix>

```

Клиент: запрос строки
 Клиент: требуется заголовок HTTP/1.1
 Клиент: пустая строка завершает заголовки
 Сервер: строка ответа
 Сервер: далее следуют пять заголовков ответа
 Сервер: ожидает HTML в теле ответа
 Сервер: ожидает 42092 байт в теле ответа
 Сервер: пустая строка завершает заголовки ответа
 Сервер: первая строка HTML в теле ответа
 Сервер: 766 строк HTML не показаны
 Сервер: последняя строка HTML в теле ответа
 Сервер: разрывает соединение
 Клиент: разрывает соединение и прекращает работу

В строке 1 мы запускаем в работу программу TELNET из командного интерпретатора Unix и выставляем требование установить соединение с Web-сервером AOL. Программа TELNET распечатывает три строки выходных данных, направленных на терминал, открывает соединение, а затем ждет, когда мы введем текст (строка 5). Каждый раз, когда мы вводим текстовую строку и нажимаем клавишу ввода, TELNET считывает эту строку, добавляет в конец символы перевода строки (\r\n в нотации языка С) и отправляет строку на сервер. Все это не противоречит стандарту HTTP, который требует, чтобы каждая строка оканчивалась парой символов возврата каретки и перевода строки. Чтобы инициировать транзакцию, мы вводим запрос HTTP (строки 5—7). Сервер выдает ответ HTTP (строки 8—17), после чего разрывает соединение (строка 18).

Запросы HTTP

Запрос HTTP состоит из строки запроса (строка 5), за которой может следовать несколько заголовков запросов, а может и не быть ни одного такого заголовка (строка 6), после которых следует пустая текстовая строка, которая завершает список заголовков (строка 7). Запрос имеет вид

<method> <uri> <version>

HTTP поддерживает некоторое число различных методов, в том числе GET, POST, OPTIONS, HEAD, PUT, DELETE и TRACE. Мы рассмотрим только наиболее часто употребляемый метод GET, который, в соответствии с результатами исследований, служит основой не менее 99% запросов HTTP [79]. Метод GET требует от сервера генерировать и возвратить содержимое, обозначенное идентификатором URI (Uniform Resource Identifier, универсальный идентификатор ресурса). Идентификатор

URI есть суффикс, соответствующий указателю URL, который включает имя файла и необязательные аргументы.

Поле `<version>` в строке запроса указывает на версию HTTP, которой соответствует запрос. Самая последняя версия HTTP — это версия HTTP/1.1 [27]. HTTP/1.0 есть предыдущая версия, которая появилась в 1996 г. и используется до сих пор [3]. Версия HTTP/1.1 определяет дополнительные заголовки, которые обеспечивают поддержку более совершенных свойств, таких как безопасность, равно как и механизм, который позволяет клиенту и серверу выполнять многочисленные транзакции по одному и тому же установленному соединению. На практике две указанные выше версии совместимы, поскольку клиенты и серверы HTTP/1.0 просто игнорируют неизвестные заголовки HTTP/1.1.

Запрос в строке 5 требует от сервера отыскать и возвратить HTML-файл `/index.html`. Он также информирует сервер о том, что остальная часть запроса будет представлена в формате HTTP/1.1.

Заголовки запросов предоставляют дополнительную информацию серверу, такую как, например, торговая марка браузера или тип MIME, который воспринимает браузер. Заголовки запросов имеют вид

```
<header name>: <header data>
```

Для нашей цели единственный заголовок, который мы должны учитывать, — это заголовок `Host` (строка 6), который возвращается в запросах HTTP/1.1, но не в запросах HTTP/1.0. Заголовок `Host` использует кэш модулей доступа (*proxy caches*), который иногда служит посредником между браузером и исходным сервером, который манипулирует затребованным файлом. Между клиентом и исходным сервером может находиться множество посредников в так называемой цепи посредников (*proxy chain*). Данные в заголовке `Host`, который идентифицирует доменное имя исходного сервера, позволяет посреднику из середины цепи определить локально кэшированную копию запрашиваемого содержимого.

Возвращаясь к нашему примеру, представленному в листинге 12.9, пустая текстовая строка, показанная в строке 7 программы (которая образуется после нажатия клавиши `<Enter>` на клавиатуре), завершает заголовки и дает серверу команду послать запрашиваемый файл HTML.

Ответы HTTP

Ответы HTTP подобны запросам HTTP. Ответ HTTP состоит из строки ответа (строка 8 листинга 12.9), за которой могут следовать несколько, а может не быть ни одного заголовка ответа (строки 9—13), за которым следует пустая строка, завершающая раздел заголовков (строка 14), после которой идет тело ответа (строки 15—17). Стока ответа имеет вид

```
<version> <status code> <status message>
```

Поле версии показывает версию HTTP, которой соответствует ответ. Код состояния есть положительное целое число из трех цифр, которое указывает диспозицию запроса. Сообщение о состоянии отображает эквивалент кода ошибки. В табл. 12.2 показан

список некоторых наиболее часто употребляемых кодов состояния и соответствующих им сообщений.

Таблица 12.2. Коды состояния среды HTTP

Код состояния	Сообщение о состоянии	Описание сообщения
200	OK	Запрос был обработан без ошибок
301	Moved permanently (Передается постоянно)	Содержимое было передано на хост, имя которого указано в заголовке Location (местоположение)
400	Bad request (Запрос сформулирован неправильно)	Запрос может быть не понят сервером
403	Forbidden (Запрещено)	Сервер не получил разрешения на доступ к затребованному файлу
404	Not found (Не найден)	Сервер не смог найти затребованный файл
501	Not implemented (Не реализован)	Сервер не поддерживает метод запроса
505	HTTP version not supported (Данная версия HTTP не поддерживается)	Сервер не поддерживает версию, указанную в запросе

Заголовки ответов в строках 9—13 листинга 12.9 предоставляют дополнительную информацию об ответе. Для наших целей самыми важными заголовками являются Content-Type (тип содержимого) (строка 12), который сообщает клиенту тип MIME содержимого в теле ответа, и Content-Length (длина содержимого) (строка 13), которая доставляет его размер в байтах.

За пустой текстовой строкой, помещенной в строке 14 программы, завершающей раздел заголовков, следует тело ответа, которое и содержит затребованное содержимое.

12.5.4. Обслуживание динамического содержимого

Если мы прекратим на момент думать о том, как сервер может предоставить динамическое содержимое клиенту, возникают определенные вопросы. Например, как клиент передает серверу программные аргументы? Как сервер передает эти аргументы порожденному процессу, который он создает? Как сервер передает другую информацию порожденному процессу, которая может ему понадобиться для генерации затребованного содержимого? Куда дочерний процесс пересыпает свои выходные данные? На все эти вопросы дает ответы фактический стандарт, получивший название интерфейса CGI (Common Gateway Interface, общий шлюзовой интерфейс).

Как клиент передает аргументы программы серверу

Аргументы запроса GET передаются в URI. Как мы уже знаем, символ ? отделяет имя файла от его аргументов, а каждый аргумент отделяется от остальных аргументов символом &. Пробелы между аргументами недопустимы и должны быть представлены в виде %20. Подобного рода коды существуют и для других специальных символов.

Передача аргументов в запросах HTTP POST

Аргументы для запросов HTTP POST передаются в теле запроса, но не в идентификаторе URI.

Как сервер передает аргументы порожденному процессу

После того как сервер получит запрос типа

GET /cgi-bin/adder?15000&213 HTTP/1.1

он вызывает функцию fork с целью создания порожденного процесса и вызывает функцию execve с целью выполнения программы /cgi-bin/adder в контексте порожденного процесса. Программы, подобные программе adder, часто называют программами CGI в силу того, что они подчиняются правилам стандарта CGI. И поскольку многие программы CGI написаны как сценарии языка Perl, программы CGI часто называют сценариями CGI. Прежде чем обратиться к функции execve, порожденный процесс присваивает переменной QUERY_STRING среды CGI значение 15000&213, на которое программа adder может ссылаться во время исполнения, используя для этой цели функцию getenv системы Unix.

Как сервер передает порожденному процессу другую информацию

Стандарт CGI определяет число переменных среды, которые программа CGI может устанавливать во время исполнения. В табл. 12.3 показано одно из подмножеств таких переменных.

Таблица 12.3. Примеры переменных среды CGI

Переменная среды	Описание
QUERY_STRING	Аргументы программы
SERVER_PORT	Порт, который прослушивает родительский процесс
REQUEST_METHOD	Запросы GET или POST
REMOTE_HOST	Доменное имя клиента
REMOTE_ADDR	IP-адрес клиента в десятичном представлении с разделительными точками
CONTENT_TYPE	Только POST: тело запроса имеет MIME
CONTENT_LENGTH	Только POST: размер тела запроса в байтах

Куда порожденный процесс отправляет свои выходные данные

Программа CGI отсылает свое динамическое содержимое на стандартный вывод. Прежде чем порожденный процесс загрузит и исполнит программу CGI, он применяет функцию dup2 системы Unix с тем, чтобы переадресовать стандартный выход на связанный дескриптор, ассоциированный с соответствующим клиентом. Следовательно, все, что программа CGI записывает на стандартный вывод, передается непосредственно клиенту.

Обратите внимание на тот факт, что поскольку родительский процесс не знает ни типа, ни размеров содержимого, которое генерирует порожденный процесс, на последний возлагается задача построения заголовков ответа Content-type и Content-length, а также пустой строки, которая завершает раздел заголовков.

В листинге 12.10 показана простая программа CGI, которая складывает два аргумента и возвращает клиенту файл HTML, содержащий их сумму.

Листинг 12.10. Программа CGI суммирования двух целых чисел

```
1 #include "csapp.h"
2
3 int main(void) {
4     char *buf, *p;
5     char arg1[MAXLINE], arg2[MAXLINE], content[MAXLINE];
6     int n1=0, n2=0;
7
8     /* Извлечь два аргумента */
9     if ((buf = getenv("QUERY_STRING")) != NULL) {
10         p = strchr(buf, '&');
11         *p = '\0';
12         strcpy(arg1, buf);
13         strcpy(arg2, p+1);
14         n1 = atoi(arg1);
15         n2 = atoi(arg2);
16     }
17
18     /* Построить тело ответа */
19     sprintf(content, "Welcome to add.com: ");
20     sprintf(content, "%sTHE Internet addition portal.\r\n<p>", content);
21     sprintf(content, "%sThe answer is: %d + %d = %d\r\n<p>",
22             content, n1, n2, n1 + n2);
23     sprintf(content, "%sThanks for visiting!\r\n", content);
24
25     /* Построить ответ HTTP */
26     printf("Content-length: %d\r\n", strlen(content));
27     printf("Content-type: text/html\r\n\r\n");
28     printf("%s", content);
```

```

29     fflush(stdout);
30     exit(0);
31 }

```

В листинге 12.11 показана HTTP-транзакция, которая представляет динамическое содержимое, возвращаемое программой adder.

Листинг 12.11. Транзакция HTTP, которая обслуживает динамическое содержимое

```

1 unix> telnet kittyhawk.cmcl.cs.cmu.edu 8000 Клиент: установить соединение
2 Trying 128.2.194.242...
3 Connected to kittyhawk.cmcl.cs.cmu.edu.
4 Escape character is ']'.
5 GET /cgi-bin/adder?15000&213 HTTP/1.0      Клиент: строка запроса
6                                         Клиент: пустая строка завершает раздел заголовков
7 HTTP/1.0 200 OK                          Сервер: строка ответа
8 Server: Tiny Web Server                  Сервер: идентификация сервера
9 Content-length: 115                      Сумматор: ожидает 115 байтов в теле ответа
10 Content-type: text/html                 Сумматор: ожидает HTML в теле ответов
11                                         Сумматор: пустая строка завершает раздел заголовков
12 Welcome to add.com: THE Internet addition portal.   Сумматор: первая
13 <p>The answer is: 15000 + 213 = 15213      Сумматор: вторая строка HTML
14                                         в теле ответа
15 <p>Thanks for visiting!      Сумматор: третья строка HTML в теле ответа
16 Connection closed by foreign host.      Сервер: разрыв соединения
17 unix>                                    Клиент: разрывает соединение и завершает работу

```

Передача аргументов в запросах HTTP POST, предназначенных программам CGI

При выполнении запросов POST у порожденного процесса возникает также необходимость переадресации стандартного ввода на связный дескриптор. Программа CGI в этом случае считывает аргументы в тело запроса со стандартного ввода.

УПРАЖНЕНИЕ 12.5

В разд. 11.9 мы предупреждали вас об опасностях использования стандартных функций ввода-вывода языка C в сетевых приложениях. В то же время, программа CGI, представленная в листинге 12.10, может без каких-либо проблем использовать стандартный ввод-вывод. Почему?

12.6. Разработка небольшого Web-сервера TINY

Мы завершаем обсуждение разработкой сетевой программы, представляющей собой небольшой и в то же время исправно работающий Web-сервер, которому мы при-

своили имя TINY (Крохотный). Сервер TINY — во многих отношениях интересная программа. В нем реализованы многие из тех идей, которые мы изучали ранее, такие как управление процессами, применение ввода-вывода системы Unix, интерфейс сокетов, а также протокол HTTP, и при этом программный код размещается всего лишь в 250 строках. И хотя в нем содержится минимум функциональных средств, ему явно не хватает надежности и безопасности реального сервера, тем не менее, он обладает достаточной мощностью. Мы настоятельно рекомендуем вам изучить его и реализовать его собственными силами. Исключительно интересно направить реальный браузер на собственный сервер и наблюдать за тем, как он отображает на экране сложную Web-страницу с текстом и графикой.

Программа *main* сервера TINY

В листинге 12.12 представлена главная программа сервера TINY. TINY является итеративным сервером, который прослушивает в порту требования на установление соединений, которые передаются в командную строку. После того как сервер TINY посредством вызова функции `open_listenfd` откроет прослушивающий сокет, он выполняет бесконечный цикл, характерный для серверов, многократно принимая запросы на установление соединений (строка 31), выполняя транзакции (строка 32) и закрывая свой конец соединения (строка 33).

Листинг 12.12. Web-сервер TINY

```
1      /*
2   * tiny.c - Простой итеративный Web-сервер с протоколом HTTP/1.0, который
3   * использует метод GET для обслуживания статического и динамического
4   * содержания
5   */
6
7 void doit(int fd);
8 void read_requesthdrs(rio_t *rp);
9 int parse_uri(char *uri, char *filename, char *cgiargs);
10 void serve_static(int fd, char *filename, int filesize);
11 void get_filetype(char *filename, char *filetype);
12 void serve_dynamic(int fd, char *filename, char *cgiargs);
13 void clienterror(int fd, char *cause, char *errnum,
14                  char *shortmsg, char *longmsg);
15
16 int main(int argc, char **argv)
17 {
18     int listenfd, connfd, port, clientlen;
19     struct sockaddr_in clientaddr;
20
21     /* Проверка аргументов командной строки */
22     if (argc != 2) {
23         fprintf(stderr, "usage: %s <port>\n", argv[0]);
```

```

24         exit(1);
25     }
26     port = atoi(argv[1]);
27
28     listenfd = Open_listenfd(port);
29     while (1) {
30         clientlen = sizeof(clientaddr);
31         connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
32         doit(connfd);
33         Close(connfd);
34     }
35 }
```

Функция *doit*

Функция *doit*, представленная в листинге 12.13, выполняет обработку одной HTTP-транзакции. Прежде всего, мы считываем и проводим грамматический анализ строки запроса (строки 11—12). Обратите внимание на то, как мы используем функцию *rio_readlineb*, представленную в листинге 11.8 для того, чтобы прочитать строку запроса. Сервер TINY поддерживает только метод GET. Если клиент запрашивает другой метод (например, метод POST), мы отправляем ему сообщение об ошибке и возвращаемся в главную программу (строки 13—17), которая затем разрывает соединение и переходит в режим ожидания следующего запроса на установление соединения. В противном случае мы читаем и (как мы сможем убедиться далее) игнорируем любые заголовки запросов (строка 18).

Листинг 12.13. Функция выполняет обработку одной транзакции

```

1 void doit(int fd)
2 {
3     int is_static;
4     struct stat sbuf;
5     char buf[MAXLINE], method[MAXLINE], uri[MAXLINE], version[MAXLINE];
6     char filename[MAXLINE], cgiargs[MAXLINE];
7     rio_t rio;
8
9     /* Чтение строки запроса и заголовков */
10    Rio_readinitb(&rio, fd);
11    Rio_readlineb(&rio, buf, MAXLINE);
12    sscanf(buf, "%s %s %s", method, uri, version);
13    if (strcasecmp(method, "GET")) {
14        clienterror(fd, method, "501", "Not Implemented",
15                    "Сервер Tiny не реализует этот метод ");
16        return;
17    }
```

```

18     read_requesthdrs(&rio);
19
20     /* Синтаксический разбор идентификатора URI из запроса GET */
21     is_static = parse_uri(uri, filename, cgiargs);
22     if (stat(filename, &sbuf) < 0) {
23         clienterror(fd, filename, "404", "Not found",
24                     "Сервер Tiny не смог найти этот файл");
25         return;
26     }
27
28     if (is_static) { /* Обслуживание статического содержимого */
29         if (!(S_ISREG(sbuf.st_mode)) || !(S_IRUSR & sbuf.st_mode)) {
30             clienterror(fd, filename, "403", "Forbidden",
31                         "Сервер Tiny не смог прочесть этот файл");
32             return;
33         }
34         serve_static(fd, filename, sbuf.st_size);
35     }
36     else { /* Обслуживание динамического содержимого */
37         if (!(S_ISREG(sbuf.st_mode)) || !(S_IXUSR & sbuf.st_mode)) {
38             clienterror(fd, filename, "403", "Forbidden",
39                         "Tiny couldn't run the CGI program");
40             return;
41         }
42         serve_dynamic(fd, filename, cgiargs);
43     }
44 }

```

Далее мы выполняем синтаксический анализ идентификатора URI, разбивая его на имя файла и, возможно, на пустую строку аргумента CGI, и при этом мы устанавливаем флаг, который указывает, к какому содержимому относится запрос, к статическому или динамическому (строка 21). Если файл на диске отсутствует, мы немедленно отправляем клиенту сообщение об ошибке и возвращаемся в главную программу (строки 22—26).

И наконец, если запрос относится к статическому содержимому, мы проверяем, является ли файл регулярным, и есть ли у нас полномочия на чтение (строка 29). Если есть, то мы обслуживаем статическое содержимое (строка 34) для клиента. Аналогично, если поступил запрос динамического содержимого, мы проверяем, является ли файл исполняемым (строка 37), и если это так, мы продвигаемся дальше и обслуживаем динамическое содержимое (строка 42).

Функция `clienterror`

В сервере TINY отсутствуют многие функции обработки ошибок, которые имеются в реальных серверах. Тем не менее он выполняет проверку на наличие некото-

рых очевидных ошибок и уведомляет о них соответствующего клиента. Функция `clienterror` (ошибка клиента), представленная в листинге 12.14, отсылает клиенту ответ HTTP с соответствующим кодом состояния и сообщением о состоянии в строке ответа, а также файл HTML в теле ответа, который объясняет пользователю браузера, в чем состоит ошибка.

Листинг 12.14. Функция посыпает клиенту сообщение об ошибке

```
1 void clienterror(int fd, char *cause, char *errnum,
2                   char *shortmsg, char *longmsg)
3 {
4     char buf[MAXLINE], body[MAXBUF];
5
6     /* Построение тела ответа HTTP */
7     sprintf(body, "<html><title>Tiny Error</title>");
8     sprintf(body, "%s<body bgcolor=\"ffffff\">\r\n", body);
9     sprintf(body, "%s%s: %s\r\n", body, errnum, shortmsg);
10    sprintf(body, "%s<p>%s: %s\r\n", body, longmsg, cause);
11    sprintf(body, "%s<hr><em>The Tiny Web server</em>\r\n", body);
12
13    /* Печать ответа HTTP */
14    sprintf(buf, "HTTP/1.0 %s %s\r\n", errnum, shortmsg);
15    Rio_writen(fd, buf, strlen(buf));
16    sprintf(buf, "Content-type: text/html\r\n");
17    Rio_writen(fd, buf, strlen(buf));
18    sprintf(buf, "Content-length: %d\r\n\r\n", strlen(body));
19    Rio_writen(fd, buf, strlen(buf));
20    Rio_writen(fd, body, strlen(body));
21 }
```

Напомним вам, что в ответе HTML должен быть указан размер и тип содержимого тела ответа. Таким образом, мы предпочли построить содержимое HTML в виде единой строки (в программе строки 7—11), благодаря чему мы легко можем определить его размер (строка 18). Обратите также внимание на тот факт, что мы используем устойчивую функцию `Rio_writen`, представленную в листинге 11.3 для всех выводов.

Функция `read_requesthdrs`

Сервер TINY не использует никакую информацию, содержащуюся в заголовках запросов. Он просто читает и игнорирует ее посредством вызова функции `read_requesthdrs` (читать заголовки запросов), представленной в листинге 12.15. Обратите внимание на тот факт, что пустая текстовая строка, которая завершает раздел заголовков запросов, состоит из пары символов возврата каретки и символа перевода строки, наличие которой мы проверяем в строке 6.

Листинг 12.15. Функция читает и игнорирует заголовки запросов

```

1 void read_requesthdrs(rio_t *rp)
2 {
3     char buf[MAXLINE];
4
5     Rio_readlineb(rp, buf, MAXLINE);
6     while(strcmp(buf, "\r\n"))
7         Rio_readlineb(rp, buf, MAXLINE);
8     return;
9 }
```

Функция parse_uri

Сервер TINY действует в предположении, что домашний каталог для статического содержимого есть его текущий каталог, и что домашним каталогом для исполняемых файлов является каталог ./cgi-bin. Предполагается, что любой идентификатор URI, который содержит строку cgi-bin, обозначает запрос динамического содержимого. Файл по умолчанию есть файл ./home.html.

Функция `parse_uri` (синтаксический анализ идентификатора `uri`), представленная в листинге 12.16, реализует эти стратегии. Она выполняет синтаксический анализ идентификатора `URI`, выделяя из него имя файла и необязательную строку аргументов `CGI`. Если запрашивается статическое содержимое (строка 5), мы очищаем строку аргументов `CGI` (строка 6), после чего преобразуем идентификатор `URI` в имя относительного пути доступа в системе Unix, такое как, например, `./index.html` (строки 7—8). Если идентификатор `URI` заканчивается символом `/` (строка 9), то мы добавляем в конец имя файла по умолчанию (строка 10). С другой стороны, если запрос требует динамическое содержимое (строка 13), мы извлекаем все возможные аргументы `CGI` (строки 14—20) и преобразуем остальную часть идентификатора `URI` в относительное имя файла в Unix (строки 21—22).

Листинг 12.16. Функция выполняет синтаксический анализ идентификатора URI протокола HTTP

```

1 int parse_uri(char *uri, char *filename, char *cgiargs)
2 {
3     char *ptr;
4
5     if (!strstr(uri, "cgi-bin")) { /* Статическое содержимое */
6         strcpy(cgiargs, "");
7         strcpy(filename, ".");
8         strcat(filename, uri);
9         if (uri[strlen(uri)-1] == '/')
10             strcat(filename, "home.html");
11         return 1;
12     }
```

```

13     else { /* Динамическое содержимое */
14         ptr = index(uri, '?');
15         if (ptr) {
16             strcpy(cgiargs, ptr+1);
17             *ptr = '\0';
18         }
19     else
20         strcpy(cgiargs, "");
21     strcpy(filename, ".");
22     strcat(filename, uri);
23     return 0;
24 }
25 }
```

Функция `serve_static`

Сервер TINY обслуживает четыре различных типа статического содержимого: файлы HTML, неформатированные текстовые файлы и изображения, закодированные в форматах GIF и JPG. Эти типы файлов составляют большую часть статических данных, обслуживаемых через Web.

Функция `serve_static`, представленная в листинге 12.17, пересыпает ответ HTTP, тело которого содержит содержимое локального файла. Сначала мы определяем тип путем просмотра суффикса имени файла (строка 7), а затем отсылаем строку ответа и заголовки ответа клиенту (строки 8—12). Обратите внимание на то, что раздел заголовков завершает пустая строка.

Листинг 12.17. Функция обслуживает статическое содержимое по запросу клиента

```

1 void serve_static(int fd, char *filename, int filesize)
2 {
3     int srcfd;
4     char *srcp, filetype[MAXLINE], buf[MAXBUF];
5
6     /* Отослать заголовки ответа клиенту */
7     get_filetype(filename, filetype);
8     sprintf(buf, "HTTP/1.0 200 OK\r\n");
9     sprintf(buf, "%sServer: Tiny Web Server\r\n", buf);
10    sprintf(buf, "%sContent-length: %d\r\n", buf, filesize);
11    sprintf(buf, "%sContent-type: %s\r\n\r\n", buf, filetype);
12    Rio_writen(fd, buf, strlen(buf));
13
14    /* Отослать тело ответа клиенту */
15    srcfd = Open(filename, O_RDONLY, 0);
16    srcp = Mmap(0, filesize, PROT_READ, MAP_PRIVATE, srcfd, 0);
17    Close(srcfd);
```

```

18     Rio_written(fd, srcp, filesize);
19     Munmap(srcp, filesize);
20 }
21
22 /*
23 * Функция get_filetype извлекает тип файла из имени файла
24 */
25 void get_filetype(char *filename, char *filetype)
26 {
27     if (strstr(filename, ".html"))
28         strcpy(filetype, "text/html");
29     else if (strstr(filename, ".gif"))
30         strcpy(filetype, "image/gif");
31     else if (strstr(filename, ".jpg"))
32         strcpy(filetype, "image/jpeg");
33     else
34         strcpy(filetype, "text/plain");
35 }

```

Далее мы отсылаем тело ответа путем копирования содержимого запрашиваемого файла в подключенный дескриптор `fd` (строки 15—19). Соответствующий программный код содержит ряд тонкостей, поэтому его следует тщательно изучить. Стока 15 открывает `filename` для чтения и получает его дескриптор. В строке 16 функция `mmap` системы Unix отображает затребованный файл в область виртуальной памяти. Вспомните из предыдущего обсуждения функции `mmap` в разд. 10.8, что вызов функции `mmap` отображает первые байты `filesize` файла `srcfd` в приватную область только для чтения виртуальной памяти, которая начинается с адреса `srcp`.

Как только мы отобразим файл в памяти, его дескриптор нам больше не нужен, так что мы закрываем файл (строка 17). Если мы этого не сделаем, возникает возможность фатальной утечки памяти. Стока 18 фактически осуществляет передачу файла клиенту. Функция `rio_written` копирует байты размера файла, начиная от ячейки `srcp` (которая, естественно, отображена на затребованный файл), в дескриптор, связанный с клиентом. И, наконец, строка 19 освобождает область виртуальной памяти, отведенной под файл. Это необходимо, чтобы избежать возможной фатальной утечки памяти.

Функция `serve_dynamic`

Сервер TINY обслуживает любой тип динамического содержимого путем ответвления на порожденный процесс с последующим выполнением программы CGI в контексте производного процесса. Функция `serve_dynamic`, представленная в листинге 12.18, начинает пересылки строки ответа, показывающей клиенту, что все идет normally (строки 6—7), вместе с информационным заголовком `Server` (строки 8—9). В обязанности программы CGI входит пересылка остальной части ответа. Обратите внимание на тот факт, что эта операция не так устойчива, как бы нам хотелось.

лось, поскольку она не предусматривает случая возникновения той или иной ошибки при работе программы.

Листинг 12.18. Функция обслугивает динамическое содержимое для клиента

```

1 void serve_dynamic(int fd, char *filename, char *cgiargs)
2 {
3     char buf[MAXLINE], *emptylist[] = { NULL };
4
5     /* Возвратить первую часть ответа HTTP */
6     sprintf(buf, "HTTP/1.0 200 OK\r\n");
7     Rio_writen(fd, buf, strlen(buf));
8     sprintf(buf, "Server: Tiny Web Server\r\n");
9     Rio_writen(fd, buf, strlen(buf));
10
11    if (Fork() == 0) { /* child */
12        /* Реальный сервер устанавливает значения всех переменных CGI в этом
13           месте */
14        setenv("QUERY_STRING", cgiargs, 1);
15        Dup2(fd, STDOUT_FILENO); /* Переадресуйте stdout клиенту */
16        Execve(filename, emptylist, environ); /* Выполнить программу CGI */
17    }
18    Wait(NULL); /* Родительский процесс ждет и запускает порожденный
19                 процесс */

```

После отсылки первой части ответа мы порождаем новый дочерний процесс (строка 11). Порожденный процесс инициализирует переменную среды QUERY_STRING аргументами CGI из запроса идентификатора URI (строка 13). Обратите внимание на то обстоятельство, что реальный сервер устанавливает в этом месте также и значения других переменных среды CGI. Для краткости изложения, мы эти действия здесь опускаем. Кроме того, мы здесь отметим, что системы Solaris используют функцию `putenv` вместо функции `setenv`.

Далее, порожденный процесс переадресует стандартный вывод порожденного процесса дескриптору подключенного файла (строка 14), а затем загружает и исполняет программу CGI (строка 15). Поскольку программа CGI выполняется в контексте порожденного процесса, она имеет доступ к тем же открытым файлам и переменным среды, которые существовали до вызова функции `execve`. Следовательно, все, что программа CGI записывает на стандартный вывод, поступает непосредственно в клиентский процесс, без какого-то ни было вмешательства со стороны родительского процесса.

В то же время, родительский процесс обозначает ожидание посредством обращения к функции `wait`, с целью получить результаты от порожденного процесса, после того как последний завершится (строка 17).

Обработка преждевременно закрытых соединений

Несмотря на то, что базовые функции Web-сервера достаточно просты, мы не хотим вызвать у вас ложное представление о том, что написание настоящего сервера — пустяковое дело. Построение устойчивого Web-сервера, который обеспечивает безаварийную работу в течение достаточно продолжительного периода времени, представляет собой трудную задачу, которая требует глубокого понимания особенностей программирования в Unix-подобных системах. Например, если сервер пересыпает на соединение сообщение о том, что он уже закрыт клиентом, то первое из таких сообщений будет доставлено без осложнений, однако второе такое сообщение повлечет подачу сигнала SIGPIPE, стандартное поведение которого предусматривает останов процесса. Если сигнал SIGPIPE захвачен или проигнорирован, то вторая операция записи возвращает -1 , при этом для переменной `errno` установлено значение EPIPE. Функции `strerr` и `reErrMsg` интерпретируют ошибку EPIPE как "Разрыв канала" — несодержательное сообщение, которое ставило в тупик не одно поколение студентов. В результате устойчивый сервер должен перехватывать эти сигналы SIGPIPE и проверять, не возвратил ли вызов функции `write` ошибки EPIPE.

12.7. Резюме

В основу каждого сетевого приложения положена модель клиент-сервер. В соответствии с этой моделью приложение состоит из сервера и одного или нескольких клиентов. Сервер управляет ресурсами, предоставляя услуги своим клиентам, для чего необходимо тем или иным способом манипулировать ресурсами. Базовой операцией модели клиент-сервер является транзакция клиент-сервер, предусматривающая запрос со стороны клиента, после которого следует ответ со стороны сервера.

Клиент и серверы обмениваются данными через глобальную сеть, известную как сеть Internet. С точки зрения программиста, можно рассматривать Internet как коллекцию хостов со всего мира, обладающих следующими свойствами:

- каждый хост сети Internet имеет уникальное 32-разрядное имя, которое является его IP-адресом;
- множество IP-адресов отображается на множество доменных имен Internet;
- процессы различных хостов Internet могут обмениваться между собой данными по установленным соединениям.

Клиенты и серверы устанавливают соединения с помощью интерфейса сокетов. Сокет есть конечная точка соединения, которая представлена приложению в виде дескриптора файла. Интерфейс сокетов предоставляет функции для открывания и закрывания дескрипторов сокетов. Клиенты и серверы осуществляют обмен данными друг с другом путем записи и считывания содержимого этих дескрипторов.

Web-серверы и их клиенты (такие как, например, браузеры) осуществляют обмен данными между собой посредством использования протокола HTTP. Браузер запрашивает у сервера статическое либо динамическое содержимое. Запрос статического содержимого обслуживается путем извлечения файла с диска сервера в доставки его

клиенту. Запрос динамического содержимого обслугивается путем прогона программы в контексте порожденного процесса сервера и возврата его вывода клиенту. Стандарт CGI формулирует множество правил, которые регламентируют, как клиент передает программные аргументы серверу, как сервер передает эти аргументы и другую информацию производному процессу, и как этот производный процесс пересыпает свой вывод клиенту.

Простой, но вместе с тем нормально функционирующий процесс, который обслуживает как статическое, так и динамическое содержимое, может быть реализован несколькими сотнями программных кодов на языке C.

Библиографические заметки

Официальный источник информации для Internet содержитя во множестве бесплатно распределемых перенумерованных документов, известных как RFC (Requests for Comments, требования к комментариям). Поисковый индекс документов RFC доступен в Web по адресу <http://www.rfc-editor.org/rfc.html>.

Документы RFC, как правило, написаны для разработчиков инфраструктуры Internet и обычно изобилуют многочисленными деталями, не представляющими интереса для случайного читателя. В то же время, не существует более авторитетного источника информации. Протокол HTTP/1.1 задокументирован в RFC 2616. Обязательный для исполнения список типов MIME доступен по адресу

<ftp://ftp.isi.edu/in-notes/iana/assignments/media-types/media-types>

Существует несколько хороших публикаций общего назначения по организации компьютерных сетей [44, 58, 84]. Великий технический писатель У. Ричард Стивенс (W. Richard Stevens) написал серию классических книг по таким вопросам, как современное программирование в системе Unix [76], протоколы Internet [77, 78, 79] и программирование сетевых задач в системе Unix [81, 80]. Студенты, серьезно изучающие программирование в Unix-подобных системах, возможно, захотят изучить вопрос в полном объеме. К нашему величайшему сожалению, Стивенс умер первого сентября 1999 года. Нам очень не хватает его книг.

Задачи для домашнего решения

УПРАЖНЕНИЕ 12.6 ◆◆

1. Внести в сервер TINY такие изменения, чтобы он отображал на выходе каждую функцию запроса и каждый заголовок запроса.
2. Воспользуйтесь своим любимым браузером, чтобы обратиться к серверу TINY с запросом статического содержимого. Зафиксируйте вывод сервера TINY в специальном файле.
3. Исследуйте вывод сервера TINY с тем, чтобы определить, какую версию протокола HTTP использует ваш браузер.
4. Изучите стандарт HTTP/1.1 по документу RFC 2616 с целью определить содержимое каждого заголовка HTTP-запроса, поступившего от вашего браузера. Документ RFC 2616 вы можете получить по адресу www.rfc-editor.org/rfc.html.

УПРАЖНЕНИЕ 12.7 ◆◆

Расширьте сервер TINY таким образом, чтобы он мог обслуживать видеофайлы MPG. Проверьте правильность вашего решения, воспользовавшись реальным браузером.

УПРАЖНЕНИЕ 12.8 ◆◆

Внесите изменения в сервер TINY с тем, чтобы порожденные им процессы помещали результаты CGI в обработчик SIGCHLD вместо того, чтобы ждать, пока они не завершатся сами.

УПРАЖНЕНИЕ 12.9 ◆◆

Внесите в сервер TINY такие изменения, чтобы при обслуживании статического содержимого он копировал затребованный файл в подключенный дескриптор, используя для этой цели функции malloc, rio_readn и rio_writen вместо mmap и rio_written.

УПРАЖНЕНИЕ 12.10 ◆◆

1. Напишите форму HTML для CGI-функции суммирования, представленной листингом 12.10. Ваша форма должна включать два текстовых окна, которые пользователи заполняют двумя числами, которые нужно сложить. Ваша форма должна затребовать содержимое, используя для этой цели метод GET.
2. Проверьте, правильно ли работает ваша программа, воспользовавшись реальным браузером для запроса формы от сервера TINY, предоставьте заполненную форму серверу TINY, а затем отобразите на экране динамическое содержимое, вычисленное программой суммирования.

УПРАЖНЕНИЕ 12.11 ◆◆

Расширьте сервер TINY таким образом, чтобы он мог поддерживать метод HTTP HEAD. Проверьте, правильно ли работает ваш сервер, воспользовавшись для этой цели протоколом TELNET в качестве Web-клиента.

УПРАЖНЕНИЕ 12.12 ◆◆◆

Расширьте сервер TINY таким образом, чтобы он мог обслуживать запрос динамического содержимого, поступившего от метода HTTP POST. Проверьте, как работает ваш сервер, воспользовавшись для этой цели вашим любимым Web-браузером.

УПРАЖНЕНИЕ 12.13 ◆◆◆

Внесите в сервер TINY такие изменения, чтобы он работал плавно (без отключений) с сигналами SIGPIPE и ошибками EPIPE, которые возникают, когда функция записи предпринимает попытку осуществить запись в преждевременно закрытое соединение.

Решение упражнений

РЕШЕНИЕ УПРАЖНЕНИЯ 12.1

Шестнадцатеричный адрес	Десятичный адрес с разделительными точками
0x0	0.0.0.0
0xffffffff	255.255.255.255
0x7f000001	127.0.0.1
0xcdbca079	205.188.160.121
0x400c950d	64.12.149.13
0xcdcbc9217	205.188.146.23

РЕШЕНИЕ УПРАЖНЕНИЯ 12.2

```

1 #include "csapp.h"
2
3 int main(int argc, char **argv)
4 {
5     struct in_addr inaddr; /* адрес в порядке следования байтов, принятом
6                             в сети */
7     unsigned int addr; /* адрес в порядке следования байтов, принятом
8                          на хосте */
9
10    if (argc != 2) {
11        fprintf(stderr, "usage: %s <hex number>\n", argv[0]);
12        exit(0);
13    }
14    sscanf(argv[1], "%x", &addr);
15    inaddr.s_addr = htonl(addr);
16    printf("%s\n", inet_ntoa(inaddr));
17 }
```

РЕШЕНИЕ УПРАЖНЕНИЯ 12.3

```

1 #include "csapp.h"
2
3 int main(int argc, char **argv)
4 {
5     struct in_addr inaddr; /* адрес в порядке следования байтов, принятом
6                             в сети */
```

```

6     unsigned int addr; /* адрес в порядке следования байтов, принятом
7           на хосте */
8
9     if (argc != 2) {
10        fprintf(stderr, "usage: %s <dotted-decimal>\n", argv[0]);
11        exit(0);
12    }
13
14    if (inet_aton(argv[1], &inaddr) == 0)
15        app_error("inet_aton error");
16    addr = ntohl(inaddr.s_addr);
17    printf("0x%x\n", addr);
18
19 }

```

РЕШЕНИЕ УПРАЖНЕНИЯ 12.4

Каждый раз, когда мы обращаемся с запросом адреса хоста aol.com, список соответствующих Internet-адресов возвращается в другом, циклическом порядке.

```

unix> ./hostinfo aol.com
official hostname: aol.com
address: 205.188.146.23
address: 205.188.160.121
address: 64.12.149.13

unix>> ./hostinfo aol.com
official hostname: aol.com
address: 64.12.149.13
address: 205.188.146.23
address: 205.188.160.121

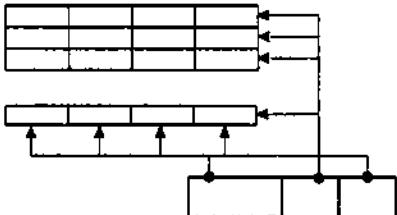
unix>> ./hostinfo aol.com
official hostname: aol.com
address: 205.188.146.23
address: 205.188.160.121
address: 64.12.149.13

```

Различный порядок следования адресов в различных запросах DNS еще называют циклом DNS. Он может быть использован для запросов с целью уравновешивания нагрузки при интенсивно используемых доменных именах.

РЕШЕНИЕ УПРАЖНЕНИЯ 12.5

Причина того, что стандартный ввод-вывод работает в программах CGI, заключается в том, что программам CGI, функционирующими в рамках порожденного процесса, не нужно явным образом закрывать любой из их входных и выходных потоков. Когда порожденный процесс прекращает работу, ядро автоматически закрывает все дескрипторы.



ГЛАВА 13

Параллельное программирование

- Параллельное программирование с процессами.
 - Параллельное программирование с мультиплексированием ввода-вывода.
 - Параллельное программирование с потоками.
 - Совместно используемые переменные в поточных программах.
 - Синхронизация потоков с семафорами.
 - Параллельный сервер на базе предварительной организации поточной обработки.
 - Другие вопросы параллелизма.
 - Резюме.
-

Управляющие логические схемы являются параллельными, если они перекрываются во времени. Этот универсальный феномен, называемый *параллелизмом*, проявляется на многих различных уровнях любой компьютерной системы. В число знакомых примеров входят обработчики исключительных ситуаций аппаратных средств, процессы и обработчики сигналов Unix.

До сих пор параллелизм рассматривался авторами в основном как механизм, используемый ядром для выполнения множественных прикладных программ. Однако параллелизм не ограничивается ядром. Он может играть важную роль в самих прикладных программах. Например, уже рассматривалась ситуация, когда обработчики сигналов Unix позволяли приложениям реагировать на такие асинхронные события, как ввод пользователем `<Ctrl>+<C>` или доступ программы к произвольной области виртуальной памяти. Параллелизм на уровне приложения также полезен в других областях:

- На однопроцессорной вычислительной системе с одним центральным процессором (CPU) параллельные потоки чередуются так, что в каждый отдельно взятый момент времени на CPU фактически выполняется только один поток. Однако существуют машины с многими процессорами, называемыми *многопроцессорами*, на которых одновременно выполняются несколько потоков. На таких машинах параллельные приложения, разделенные на параллельные потоки, иногда выпол-

няются гораздо быстрее. Это особенно важно при работе с обширными базами данных и научными приложениями.

- Доступ к медленнодействующим устройствам ввода-вывода. Когда приложение ожидает поступления данных с медленнодействующего устройства ввода-вывода, например с дискового накопителя, тогда ядро поддерживает работу CPU выполнением других процессов. Подобным же образом отдельные приложения могут использовать параллелизм путем чередования полезных действий с запросами ввода-вывода.
- Пользователям необходима возможность одновременного выполнения задач (многозадачность). Например, может возникнуть потребность изменения размера окна при одновременной распечатке документа. В современных системах оконного типа для этой цели используется параллелизм. Всякий раз при запросе пользователем некой операции (например, при щелчке кнопкой мыши) для ее выполнения создается отдельная параллельная управляющая логическая схема.
- Сокращение времени задержки путем откладывания выполнения. Иногда приложения могут использовать параллелизм для сокращения времени задержки выполнения определенных операций путем задержки выполнения других операций для их параллельного выполнения. Например, распределитель динамического запоминающего устройства может сократить время задержки выполнения отдельных операций free путем их слияния в параллельный поток, выполняемый с более низким приоритетом и поглощающий свободные циклы CPU по мере их появления.
- Обслуживание клиентов сетей. Итеративные сетевые серверы, изученные в главе 12, не реалистичны, потому что за каждый отдельно взятый момент времени они могут обслуживать только одного клиента. Таким образом, один медленнодействующий клиент может отказать в обслуживании других клиентов. Для реального сервера, от которого можно ожидать обслуживания сотен или тысяч клиентов в секунду, неприемлемо иметь одного медленнодействующего клиента, отказывающего в обслуживании других. Более совершенным подходом будет построение *параллельного сервера*, создающего отдельную управляющую логическую схему для каждого клиента. Это позволяет серверу параллельно обслуживать множество клиентов и предотвращать узурпацию сервера медленнодействующими клиентами.

Приложения, в которых используется параллелизм на уровне прикладной программы, называются *параллельными*. В современных операционных системах представлены три основных подхода к построению параллельных программ:

1. Процессы — при данном подходе каждая управляющая логическая схема представляет собой процесс, спланированный и поддерживаемый ядром. Поскольку процессы имеют раздельные виртуальные адресные пространства, потоки, которым необходимо взаимодействовать друг с другом, должны использовать определенный механизм явного межпроцессорного взаимодействия (IPC).
2. Мультиплексирование ввода-вывода представляет собой форму параллельного программирования, когда прикладные программы явно планируют собственную управляющую логическую схему в контексте одного процесса. Логические схемы

моделируются как конечные автоматы, которые основная программа явно переводит из одного состояния в другое в результате поступления данных на файловые дескрипторы. Поскольку программа является единым процессом, все потоки совместно используют одно и то же адресное пространство.

- Потоки — логические схемы, выполняющиеся в контексте одного процесса и планируемые ядром. Потоки можно представить в виде симбиоза двух вышеописанных подходов, спланированных ядром, как потоки процесса, и совместно использующих одно виртуальное адресное пространство, как потоки мультиплексирования ввода-вывода.

В настоящей главе исследуются три разные методики параллельного программирования. Для того чтобы дискуссия была максимально прозрачной, авторы предполагают работать с одним стимулирующим приложением — с параллельной версией итеративного отраженного сервера (эхо-сервера), *рассмотренного в разд. 12.4.9*.

13.1. Параллельное программирование с процессами

Самым простым способом построения параллельной программы является построение с процессами и использованием знакомых функций: `fork`, `exec` и `waitpid`. Например, естественным подходом для построения параллельного сервера является прием запросов клиента на подключение в порождающем процессе с последующим созданием вновь порожденного процесса для обслуживания каждого нового клиента.

Для более наглядной демонстрации предположим, что имеются два клиента и сервер, прослушивающий запросы на подключение через прослушивающий дескриптор (скажем, 3). Теперь предположим, что сервер принимает запрос на подключение от клиента 1 и возвращает подключенный дескриптор (скажем, 4), как показано на рис. 13.1.

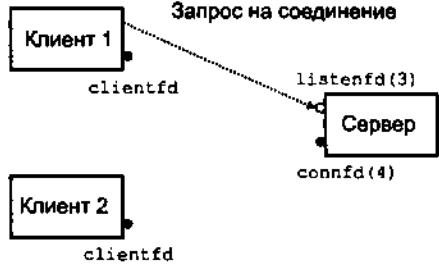


Рис. 13.1. Шаг 1: сервер принимает запрос на подключение от клиента

После приема запроса на подключение сервер разветвляет порожденный процесс, получающий полную копию таблицы дескрипторов сервера. Порожденный процесс закрывает свою копию прослушивающего дескриптора 3, а порождающий процесс — свою копию связного дескриптора 4, поскольку необходимости в них больше нет.

Эта ситуация представлена на рис. 13.2, где порожденный процесс "занят" обслуживанием клиента.

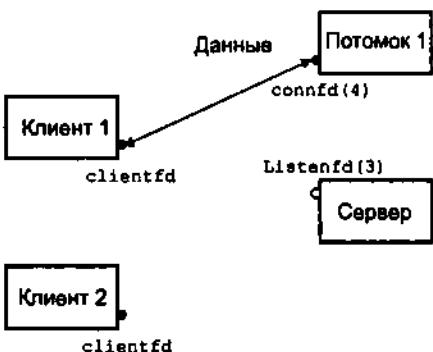


Рис. 13.2. Шаг 2: сервер образует порожденный процесс для обслуживания клиента

Поскольку связные дескрипторы в порождающем и порожденном процессе указывают на один и тот же элемент записи таблицы файлов, для порождающего процесса принципиальным является закрытие копии связного дескриптора. В противном случае, элемент записи таблицы файлов для связного дескриптора 4 никогда не будет отключен, и образующаяся утечка памяти, в конечном итоге, поглотит доступную память и выведет систему из строя.

Теперь предположим, что после того, как порождающий процесс создает процесс для клиента 1, он принимает новый запрос на подключение от клиента 2 и возвращает новый связный дескриптор (например, 5), как показано на рис. 13.3.

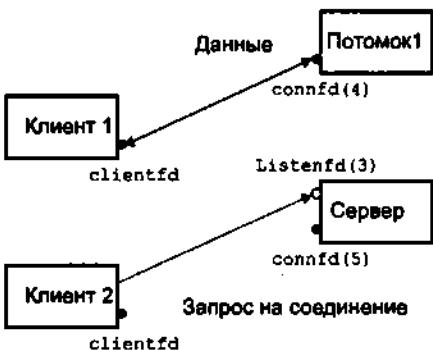


Рис. 13.3. Шаг 3: сервер принимает очередной запрос о соединении

После этого порождающий процесс создает новый процесс, который начинает обслуживать своего клиента, используя связный дескриптор 5, как показано на рис. 13.4. На этом этапе порождающий процесс ожидает следующий запрос о подключении, и два вновь порожденных процесса параллельно обслуживают соответствующих им клиентов.

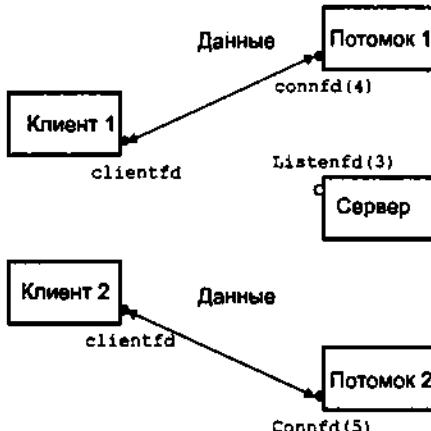


Рис. 13.4. Шаг 4: сервер образует очередной порожденный процесс для обслуживания нового клиента

13.1.1. Параллельный сервер, основанный на процессах

В листинге 13.1 представлен код параллельного эхо-сервера, основанного на процессах.

Листинг 13.1. Параллельный эхо-сервер, основанный на процессах

```

1 #include "csapp.h"
2 void echo (int connfd);
3
4 void sigchld_handler (int sig)
5 {
6     while (waitpid (-1, 0, WNOHANG) > 0)
7     ;
8     return;
9 }
10
11 int main (int argc, char **argv)
12 {
13     int listenfd, connfd, port, clientlen=sizeof (struct sockaddr_in);
14     struct sockaddr_in clientaddr;
15
16     if (argc !=2) {
17         fprintf (stderr, "usage: %s <port>\n", argv [0]);
18         exit (0);
19     }
  
```

```

20 port = atoi (argv [1]);
21
22 Signal (SIGCHLD, sigchld_handler);
23 listenfd = Open_listenfd (port);
24 while (1) {
25 connfd = Accept (listenfd, (SA *) &clientaddr, &clientlen);
26 if (Fork () == 0) {
27 Close (listenfd); /* Порожденный процесс закрывает прослушивающий сокет */
28 echo (connfd); /* Порожденный процесс обслуживает клиента */
29 Close (connfd); /* Порожденный процесс закрывает связь с клиентом */
30 exit (0); /* Порожденный процесс выходит */
31 }
32 Close (connfd); /* Порождающий процесс закрывает подкл. сокет (важно!) */
33 }
34 }
```

В отношении этого сервера существует несколько интересных заслуживающих внимания моментов:

- Как правило, серверы работают довольно продолжительное время, поэтому необходимо включить обработчик SIGCHLD, порождающий так называемые зомби-процессы (строки 4—9). Поскольку сигналы SIGCHLD блокируются во время выполнения обработчика SIGCHLD и поскольку сигналы Unix не организованы в очередь, обработчик SIGCHLD должен быть подготовлен для создания множественных зомби-процессов.
- Порождающий и порожденный процессы должны закрыть соответствующие им копии connfd (строки 32 и 29 соответственно). Как уже отмечалось, это особенно важно для порождающего процесса, который должен закрыть свою копию связного дескриптора, во избежание утечки памяти.
- Из-за количества ссылок в записи таблицы файлов сокета подключение к клиенту не будет прервано до тех пор, пока не будут закрыты копии connfd порождающего и порожденного процессов.

13.1.2. За и против использования процессов

Процессы обладают чистой моделью совместного использования информации о состоянии между порождающими и порожденными процессами: таблицы файлов используются совместно, а пространства адресов пользователей — нет. Наличие раздельных адресных пространств для процессов является как преимуществом, так и недостатком. Одному процессу невозможно случайно осуществить запись поверх виртуальной памяти другого процесса, что устраняет массу очевидных ошибок. Это является преимуществом.

С другой стороны, раздельные адресные пространства затрудняют совместное использование процессами информации о состоянии. Для совместного использования

информации они должны использовать явные механизмы межпроцессорного взаимодействия. Другим недостатком проектов, основанных на процессах, является то, что они работают медленнее из-за высоких непроизводительных издержек на управление процессом и межпроцессорное взаимодействие.

Unix IPC

В данном тексте читатель уже сталкивался с несколькими примерами IPC (межпроцессорное взаимодействие). Функция `waitpid` и сигналы Unix, рассмотренные в главе 8, — примитивные механизмы IPC, позволяющие посыпать крошечные сообщения в процессы, выполняющиеся на одной и той же хост-машине. Интерфейсы сокета, описанные в главе 12, представляют собой важную форму IPC, позволяющую процессам на разных хост-машинах обмениваться произвольными потоками байтов. Однако термин Unix IPC, как правило, "зарезервирован" для великого множества методик, позволяющих одним процессам взаимодействовать с другими, выполняющимися на одной и той же хост-машине. Все подробности можно найти в книге Стивенса (Stevens) [80].

УПРАЖНЕНИЕ 13.1

После того как порождающий процесс закрывает связный дескриптор в строке 32 параллельного сервера, порожденный процесс по-прежнему может взаимодействовать с клиентом, используя свою копию дескриптора. Почему?

УПРАЖНЕНИЕ 13.2

Если бы строка 29 в листинге 13.1, закрывающая связный дескриптор, была удалена, код по-прежнему оставался бы корректным в том смысле, что не было бы утечки памяти. Почему?

13.2. Параллельное программирование с мультиплексированием ввода-вывода

Предположим, что поставлена задача написания программы эхо-сервера интерактивных команд, которые пользователь набирает в стандартном вводе. В этом случае сервер должен отвечать на два независимых события ввода-вывода:

- клиент сети делает запрос на подключение;
- пользователь вводит командную строку с клавиатуры.

Какого события нужно дождаться первым? Ни один выбор не является идеальным. При ожидании запроса о соединении в `accept` реагировать на команды ввода нельзя. Аналогичным же образом при ожидании команды ввода в `wait` нельзя реагировать ни на какие запросы о соединении.

Одним из решений этой дилеммы является методика, называемая *мультиплексированием ввода-вывода*. Основная идея: использование функции `select` для запроса ядра о придержании процесса с возвратом управления приложению только после проявле-

ния одного или нескольких событий ввода-вывода, как показано в следующих примерах:

- возврат, когда любой дескриптор во множестве {0, 4} готов к считыванию;
- возврат, когда любой дескриптор во множестве {1, 2, 7} готов к записи;
- простой, если 152.13 секунд истекли в ожидании события ввода-вывода.

Здесь будет рассмотрен только первый сценарий: ожидание подготовки множества дескрипторов к считыванию. Подробное обсуждение представлено в [76], [81].

```
#include <unistd.h>
#include <sys/types.h>
```

```
int select (int n, fd_set *fdset, NULL, NULL, NULL);
FD_ZERO (fd_set *fdset); /* Очистка всех битов в fdset */
FD_CLR (int fd, fd_set *fdset); /* Очистка бита fd в fdset */
FD_SET (int fd, fd_set *fdset); /* Обращение к биту fd в fdset */
FD_ISSET (int fd, fd_set *fdset); /* Включен ли бит fd в fdset */
```

Функция `select` манипулирует множествами типа `fd_set`, известными как *множества дескрипторов*. Логически множество дескрипторов рассматривается как побитовая маска размера *n*:

$$b_{n-1}, \dots, b_1, b_0.$$

Каждый бит b_k соответствует дескриптору k . Дескриптор k является членом множества дескриптора, если $b_k = 1$. Со множествами дескрипторов пользователю разрешено выполнять только следующие операции:

- размещать их;
- присваивать одну переменную данного типа другой;
- модифицировать и инспектировать их, используя макросы `FD_ZERO`, `FD_SET`, `FD_CLR` и `FD_ISSET`.

Для преследуемых в данной главе целей функция `select` охватывает два ввода: множество дескрипторов `fdset`, называемое множеством ввода, и количество элементов и множества ввода. Функция `select` блокируется до тех пор, пока минимум один дескриптор в множестве считывания не будет готов к считыванию. Дескриптор k готов к считыванию, только если запрос на считывание одного байта из этого дескриптора не будет блокирован. В качестве побочного эффекта функция `select` модифицирует `fd_set` для указания на подмножество множества считывания, называемое готовым множеством. Возвращаемое функцией значение указывает количество элементов (кардинальное значение) множества считывания. Обратите внимание на то, что необходимо обновлять множество считывания всякий раз при вызове `select`.

Наилучшим способом понимания функции `select` является изучение конкретных примеров. В листинге 13.2 показано то, как можно использовать `select` для реализации итеративного эхо-сервера, принимающего команды пользователя на стандартном вводе.

Листинг 13.2. Эхосервер, использующий мультиплексирование ввода-вывода

```

1 #include "csapp.h"
2 void echo (int connfd);
3 void command (void);
4
5 int main (int argc, char **argv)
6 {
7     int listenfd, connfd, port, clientlen = sizeof (struct sockaddr_in);
8     struct sockaddr_in clientaddr;
9     fd_set read_set, ready_set;
10
11    if (argc != 2) {
12        fprintf (stderr, "usage: %s <port>\n", argv [0]);
13        exit (0);
14    }
15    port = atoi (argv [1]);
16    listenfd = Open_listenfd (port);
17
18    FD_ZERO (&read_set);
19    FD_SET (STDIN_FILENO, &read_set);
20    FD_SET (listenfd, &read_set);
21
22    while (1) {
23        ready_set = read_set;
24        Select (listenfd+1, &ready_set, NULL, NULL, NULL);
25        if (FD_ISSET (STDIN_FILENO, &ready_set))
26            command (); /* считывание командной строки из stdin */
27        if (FD_ISSET (listenfd, &ready_set)) {
28            connfd = Accept (listenfd, (SA *) &clientaddr, &clientlen);
29            echo (connfd); /* ввод эхо-клиента до окончания файла (EOF) */
30        }
31    }
32 }
33
34 void command (void) {
35     char buf [MAXLINE];
36     if (!Fgets (buf, MAXLINE, stdin))
37         exit (0); /* Окончание файла */
38     printf ("%s", buf); /* Обработка команды ввода */
39 }
```

Начнем с использования функции `open_listenfd`, показанной в листинге 12.5, для открытия прослушивающего дескриптора (строка 16 листинга 13.2) с последующим применением `FD_ZERO` для создания пустого множества считывания (рис. 13.5).

listenfd	3	2	1	stdin	0
read_set(0):	0	0	0	0	

Рис. 13.5. Создание пустого множества

Далее в строках 19—20 листинга 13.2 определяется множество считывания для состава дескриптора 0 (стандартный ввод) и дескриптора 3 (прослушивающий дескриптор) (рис. 13.6).

listenfd	3	2	1	stdin	0
read_set({0,3}):	1	0	0	1	

Рис. 13.6. Прослушивающий дескриптор

В данной точке начинается цикл типичного сервера. Однако вместо ожидания запроса на подключение вызовом функции accept вызывается функция select, осуществляющая блокирование до тех пор, пока прослушивающий дескриптор или стандартный ввод не будет готов к считыванию (строка 24). Например, имеется значение ready_set, которое будет возвращено функцией select, если пользователь нажмет клавишу ввода и таким образом подготовит дескриптор стандартного ввода к считыванию (рис. 13.7).

listenfd	3	2	1	stdin	0
read_set({0}):	0	0	0	1	

Рис. 13.7. Дескриптор стандартного ввода

После возврата select для определения того, какой из дескрипторов готов к считыванию, используется макрос FD_ISSET. Если стандартный ввод готов (строка 25), то вызывается функция command, считающая, анализирующая и реагирующая на команду до возврата к основной процедуре. Если прослушивающий дескриптор готов (строка 27), тогда вызывается функция accept для получения связного дескриптора, после чего вызывается функция echo, отражающая каждую строку программы клиента до тех пор, пока клиент не закрывает свой конец подключения.

Несмотря на то, что эта программа представляет собой хороший пример использования функции select, она не лишена недостатков. Проблема заключается в том, что после подключения к клиенту она продолжает отражать строки ввода до тех пор, пока клиент не закроет свой конец подключения. Таким образом, если команда вводится в стандартный ввод, ответ не будет получен до тех пор, пока сервер не завершит обслуживание клиента. Лучшим подходом будет мультиплексирование на более мелкие ячейки, отражающие (максимум) одну текстовую строку при каждом прохождении через цикл сервера (см. упр. 13.3).

13.2.1. Параллельный событийно-управляемый сервер на базе мультиплексирования ввода-вывода

Мультиплексирование ввода-вывода можно использовать в качестве основы для параллельных событийно-управляемых программ, где потоки перемещаются в результате определенных событий. Общей идеей является моделирование логических схем в виде конечных автоматов. Говоря неформально, конечным автоматом называется набор состояний, событий ввода и переходов, отображающих состояния и события ввода в состояния. Каждый переход отображает пару (состояние ввода, событие ввода) в состояние вывода. Петля в графе — это переход между одним и тем же состоянием ввода и вывода. Конечные автоматы обычно изображаются в виде ориентированных графов, где узлы обозначают состояния, направленные дуги — переходы, а ярлыки дуг — события ввода. Конечный автомат начинает выполнение в некоем первоначальном состоянии. Каждое событие ввода запускает переход от текущего состояния в следующее состояние.

Для каждого нового клиента k параллельный сервер, основанный на мультиплексировании ввода-вывода, создает новый конечный автомат s_k и ассоциирует его со связным дескриптором d_k . Как показано на рис. 13.8, каждый конечный автомат s_k имеет одно состояние (ожидание подготовки дескриптора d_k к считыванию), одно событие ввода (дескриптор d_k готов к считыванию) и один переход (считывание текстовой строки из дескриптора d_k).

Благодаря функции `select`, сервер использует мультиплексирование ввода-вывода для выявления появлению событий ввода. По мере того, как каждый связный дескриптор оказывается готовым к считыванию, сервер выполняет переход для соответствующего конечного аппарата, считывая в этом случае и отражая текстовую строку из дескриптора.



Рис. 13.8. Конечный автомат для логической схемы в параллельном событийно-управляемом эхо-сервере

В листинге 13.3 показан примерный код для параллельного событийно-управляемого сервера на базе мультиплексирования ввода-вывода. Множество активных клиентов поддерживается в структуре `pool` (строки 3—11). После инициализации накопителя вызовом `init_pool` (строка 28) сервер входит в бесконечный цикл. Во время каждой итерации этого цикла сервер вызывает функцию `select` для выявления двух разных типов событий ввода: запроса на подключение, поступающего от нового клиента, и

связного дескриптора для существующего клиента, готового к считыванию. При поступлении запроса на подключение (строка 35) сервер открывает подключение (строка 36) и вызывает функцию `add_client` для добавления клиента в накопитель (строка 37). Наконец, сервер вызывает функцию `check_client` для отображения одной текстовой строки из каждого готового связного дескриптора (строка 41).

Листинг 13.3. Параллельный эхо-сервер на мультиплексоре ввода-вывода

```

1 #include "csapp.h"
2
3 typedef struct { /* представляет накопитель связных дескрипторов */
4     int maxfd; /*самый крупный дескриптор в read_set */
5     fd_set read_set; /* множество всех активных дескрипторов */
6     fd_set ready_set; /* подмножество дескрипторов, готовых к считыванию */
7     int nready; /* количество готовых дескрипторов из select */
8     int maxi; /* указатель "критического уровня" в массив клиентов */
9     int clientfd [FD_SETSIZE]; /* множество активных дескрипторов */
10    rio_t clientrio [FD_SETSIZE]; /* множество активных буферов считывания */
11    }pool;
12
13    int byte_cnt = 0; /* подсчет общего количества байтов, полученных
14                      сервером */
15
16 }
17 int listenfd, connfd, port, clientlen = sizeof (struct sockaddr_in);
18 struct sockaddr_in clientaddr;
19 static pool pool;
20
21 if (argc !=2) {
22     fprintf (stderr, "usage: %s <port>\n", argv [0]);
23     exit (0);
24 }
25 port = atoi (argv [1]);
26
27 listenfd = Open_listenfd (port);
28 init_pool (listenfd, &pool);
29 while (1) {
30 /* Ожидание прослушивающего/связного дескриптора (ов) */
31     pool.ready_set = pool.read_set;
32     pool.nready = Select (pool.maxfd+1, &pool.ready_set, NULL, NULL, NULL);
33
34 /* Если прослушивавший дескриптор готов, добавление нового клиента
35 в накопитель */
36 if (FD_ISSET (listenfd, &pool.ready_set)) (
37     connfd = Accept (listenfd, (SA *) &clientaddr, &clientlen);

```

```

37 add_client (connfd, &pool);
38 }
39
40 /* Отражение текстовой строки из каждого готового связного дескриптора */
41 check_clients (&pool);
42     }
43 }
```

Функция `init_pool` (листинг 13.4) инициализирует клиентский накопитель. Массив `clientfd` представляет множество связных дескрипторов, где `-1` означает доступный сегмент памяти. Первоначально множество связных дескрипторов пусто, а прослушивающий дескриптор является единственным дескриптором во множестве считывания `select` (строки 10–12).

Листинг 13.4. Инициализация накопителя для активных клиентов

```

1 void init_pool (int listenfd, pool *p)
2 {
3 /* Изначально связных дескрипторов нет */
4 int i;
5 p->maxi = -1;
6 for (i=0; i< FD_SETSIZE; i++)
7 p->clientfd [i] = -1;
8
9 /* Изначально listenfd – единственный член множества считывания select */
10 p->maxfd = listenfd;
11 FD_ZERO (&p ->read_set);
12 FD_SET (listenfd, &p->read_set);
13 }
```

Функция `add_client` (листинг 13.5) добавляет в накопитель активных клиентов нового клиента. После обнаружения пустого сегмента в массиве `clientfd` сервер добавляет связный дескриптор в массив и инициализирует соответствующий буфер считывания `RIO` так, что на дескриптор можно вызвать `rio_readlineb` (строки 8–9). Затем связный дескриптор добавляется во множество считывания `select` (строка 12), и обновляются некоторые глобальные свойства накопителя. Переменная `maxfd` (строки 15–16) отслеживает для `select` самый крупный файловый дескриптор. Переменная `maxi` (строки 17–18) отслеживает самый крупный указатель в массиве `clientfd` так, что функциям `check_clients` нет необходимости просмотра всего массива.

Листинг 13.5. Добавление нового клиентского подключения в накопитель

```

1 void add_client (int connfd, pool *p)
2 {
3 int i;
```

```

4 p->nready--;
5 for (i = 0; i < FD_SETSIZE; i++) /* Поиск доступного сегмента */
6 if (p->clientfd [i] < 0) {
7 /* Добавление в накопитель связного дескриптора */
8 p->clientfd [i] = connfd;
9 Rio_readinitb (&p->clientrio [i], connfd);
10
11 /* Добавление дескриптора во множество дескрипторов */
12 FD_SET (connfd, &p->read_set);
13
14 /* Обновление дескриптора max и отметки ""критического уровня" накопителя */
*/
15 if (connfd > p->maxfd)
16 p->maxfd = connfd;
17 if (i > p->maxi)
18 p->maxi = i;
19 break;
20 }
21 if (i == FD_SETSIZE) /*Невозможно найти пустой сегмент */
22 app_error ("add_client error: Too many clients");
23 }

```

Функция `check_client` (листиг 13.6) отражает текстовую строку из каждого готового связного дескриптора. Если считывание текстовой строки из дескриптора будет успешным, тогда данная строка отражается назад к клиенту (строки 15—18). Обратите внимание, что в строке 15 поддерживается совокупный счетчик всех байтов, полученных от всех клиентов. Если выявляется окончание файла EOF из-за того, что клиент закрыл свой конец подключения, тогда закрываем свой конец подключения (строка 23) и удаляем дескриптор из накопителя (строки 24—25).

Листинг 13.6. Готовые услуги подключения клиентов

```

1 void check_clients (pool *p)
2 {
3 int i, connfd, n;
4 char buf [MAXLINE];
5 rio_t rio;
6
7 for (i = 0; (i <= p->maxi) && (p->nready > 0); i++) {
8 connfd = p->clientfd [i];
9 rio = p->clientrio [i];
10
11 /* Если дескриптор готов, отразить из него текстовую строку */
12 if ((connfd > 0) && (FD_ISSET (connfd, &p->ready_set))) {
13 p->nready--;
14 if ((n = Rio_readlineb (&rio, buf, MAXLINE)) != 0) {
15 byte_cnt += n;

```

```
16 printf ("Server received %d (%d total) bytes on fd %d\n",
17 n, byte_cnt, connfd);
18 Rio_writen (connfd, buf, n);
19 }
20
21 /* Выявлено окончание файла (EOF), удаление дескриптора из накопителя */
22 else {
23 Close (connfd);
24 FD_CLR (connfd, &p->read_set);
25 p->clientfd [i] = -1;
26 }
27 }
28 }
29 }
```

В том, что касается модели конечного состояния, функция `select` выявляет события ввода, а функция `add_client` создает новую логическую схему (конечный автомат). Функция `check_client` выполняет переходы состояний отражением строк ввода, а также удаляет данный конечный автомат, когда клиент прекращает отправку текстовых строк.

13.2.2. За и против мультиплексирования ввода-вывода

Сервер, показанный на рис. 13.8, наглядно представляет прекрасный пример преимуществ и недостатков событийно-управляемого программирования, основанного на мультиплексировании ввода и вывода. Одним из преимуществ является то, что событийно-управляемое проектирование обеспечивает программистам большую степень управления поведением своих программ, чем проектирование, основанное на процессах. Например, можно представить написание программы событийно-управляемого параллельного сервера, обеспечивающего предпочтительное обслуживание определенных клиентов, что было бы сложно для параллельного сервера, основанного на процессах.

Другим преимуществом является то, что событийно-управляемый сервер, основанный на мультиплексировании ввода-вывода, работает в контексте одного процесса, следовательно, каждая логическая схема имеет доступ ко всему адресному пространству процесса. Этим упрощается совместное использование данных между потоками. Связанным с выполнением в виде одного процесса преимуществом является то, что параллельный сервер можно отладить, как любую последовательную программу, используя уже знакомый инструмент отладки, например GDB. И наконец, событийно-управляемые проекты часто значительно более эффективны и производительны, нежели основанные на проектах, потому что они не требуют контекстного переключателя процесса для планирования нового потока.

Значительным недостатком событийно-управляемых проектов является сложность кодирования. К примеру, рассматриваемый параллельный событийно-управляемый

эхо-сервер требует в три раза большего объема кодирования, нежели сервер, основанный на процессах, и к сожалению, эта сложность повышается по мере уменьшения степени распараллеливания процессов. Под распараллеливанием здесь подразумевается количество команд, выполняемое каждой логической схемой в период времени. Например, в рассматриваемом параллельном сервере степень распараллеливания — количество команд, необходимое для считывания целой текстовой строки. Пока определенная логическая схема считывает текстовую строку, никакая другая логическая схема не может "двигаться дальше". Для примера это замечательно, однако делает событийно-управляемый сервер уязвимым для клиентов-злоумышленников, которые отправляют только часть текстовой строки, после чего останавливают процесс. Модификация событийно-управляемого сервера для обработки неполных текстовых строк нетривиальна, однако она выполняется автоматически проектом, основанным на процессах.

УПРАЖНЕНИЕ 13.3

В большинстве систем Unix ввод `<Ctrl>+<d>` указывает на окончание файла в стандартном вводе. Что произойдет, если ввести `<Ctrl>+<d>` в программу, показанную в листинге 13.2, в то время как она заблокирована вызовом `select`?

УПРАЖНЕНИЕ 13.4

В программе сервера, показанной в листинге 13.3, соблюдается большая осторожность при повторной инициализации переменной `pool.ready_set` непосредственно перед каждым вызовом `select`. Почему?

13.3. Параллельное программирование с потоками

До сих пор авторами рассматривались два подхода к созданию параллельных логических схем. При первом подходе используется отдельный процесс для каждой схемы. Ядро планирует каждый процесс автоматически. Каждый процесс имеет свое собственное частное адресное пространство, что затрудняет совместное использование данных логическими схемами. При втором подходе авторы создают свои собственные логические схемы и используют мультиплексирование ввода-вывода для явного планирования потоков. По причине того, что процесс только один, потоки совместно используют все адресное пространство. В данном разделе представлен третий подход, основанный на потоках, являющийся гибридом двух первых.

Поток — это логическая блок-схема, выполняющаяся в контексте процесса. До сих пор описывавшиеся в книге программы содержали один поток на процесс. Однако современные системы позволяют писать программы с несколькими потоками параллельно в одном процессе. Каждый поток имеет свой собственный контекст, включающий уникальный целочисленный идентификатор `TID`, стек, указатель на стек, счетчик команд, регистры общего назначения и коды условия. Все выполняющиеся в процессе потоки совместно используют полное виртуальное адресное пространство этого процесса.

Основанные на потоках логические схемы объединяют в себе качества потоков, основанных на процессах и мультиплексировании операций ввода-вывода. Подобно процессам, ядро автоматически планирует потоки и распознает их по целочисленному идентификатору. Подобно логическим схемам, основанным на мультиплексировании ввода-вывода, множественные потоки выполняются в контексте одного процесса и совместно используют все содержимое виртуального адресного пространства процесса, включая его код, данные, совместно используемые библиотеки и открытые файлы.

13.3.1. Модель выполнения потока

Модель выполнения для множественных потоков в некотором роде сходна с моделью выполнения для множественных процессов. Рассмотрим пример на рис. 13.9.

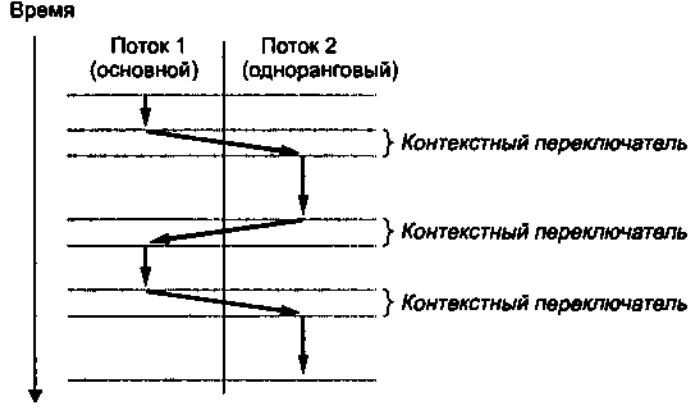


Рис. 13.9. Выполнение параллельного потока

Каждый процесс начинает свое существование в виде одного потока, называемого основным. В определенной точке основной поток создает одноранговый поток, и с этого момента эти два потока выполняются параллельно. Вскоре управление передается одноранговому потоку через контекстный переключатель, потому что основной поток выполняет медленнодействующий системный вызов, такой как `read` или `sleep`, либо он прерывается системным датчиком временных интервалов. Одноранговый поток выполняется немного раньше того момента, когда управление передается назад основному потоку.

Выполнение потоков отличается от процессов некоторыми важными показателями. По причине того, что контекст потока намного меньше контекста процесса, контекстный переключатель потока срабатывает быстрее переключателя контекста процесса. Другим различием является то, что в отличие от процессов, потоки не организованы в жесткую иерархию "родитель-потомок". Потоки, связанные с процессом, формируют накопитель одноранговых процессов. Основной поток отличается от других потоков только тем, что в процессе он всегда выполняется первым. Основное влияние накопителя одноранговых процессов заключается в том, что поток может

уничтожить любой поток либо дождаться завершения любого из одноранговых потоков. Более того, каждый одноранговый процесс может считывать и записывать одни и те же совместно используемые данные.

13.3.2. Потоки интерфейса операционной системы Posix

Потоки Posix — стандартный интерфейс для манипуляций с потоками из программ С. Он был принят на вооружение в 1995 г. и имеется в большинстве систем Unix. Posix определяет порядка 60 функций, позволяющих программам создавать, уничтожать и собирать потоки, безопасно использовать данные совместно с одноранговыми потоками и извещать последние об изменениях состояния системы.

В листинге 13.7 показана простая программа потоков Posix. Основной поток создает одноранговый поток и ожидает окончания его выполнения. Одноранговый поток печатает строку "Hello, world!\n" и завершает выполнение. Когда основной поток обнаруживает окончание выполнения однорангового потока, он завершает процесс вызовом exit.

Листинг 13.7. Потоки Posix

```

1 #include "csapp.h"
2 void *thread (void *vargp);
3
4 int main ()
5 {
6     pthread_t tid;
7     Pthread_create (&tid, NULL, thread, NULL);
8     Pthread_join (tid, NULL);
9     exit (0);
10 }
11
12 void *thread (void *vargp)/* рутинная процедура потока */
13 {
14     printf ("Hello, world!\n");
15     return NULL;
16 }
```

Это первая рассматриваемая потоковая программа, поэтому разберем ее подробнее. Код и локальные данные для потока инкапсулируются в рутинную процедуру потока. Как показано на прототипе в строке 2, каждая рутинная процедура потока берет в качестве данных ввода один родовой указатель и возвращает родовой указатель. При необходимости передачи в рутинную процедуру потока множественных аргументов необходимо поместить аргументы в структуру и передать в структуру указатель. Подобным же образом, при необходимости возврата рутинной процедурой потока множественных аргументов, можно вернуть указатель в структуру.

Строка 4 отмечает начало кода для основного потока. Основной поток объявляет одну локальную переменную `tid`, которая будет использоваться для сохранения идентификатора однорангового потока (строка 6). Основной поток создает новый одноранговый поток вызовом функции `pthread_create` (строка 7). При возврате вызова `pthread_create` основной поток и вновь созданный одноранговый поток выполняются параллельно, а `tid` содержит идентификатор нового потока. Основной поток ожидает завершения выполнения однорангового потока с вызовом `pthread_join` в строке 8. Наконец, основной поток вызывает `exit` (строка 9), завершающий выполнение в данный момент всех потоков процесса (в данном случае только основного).

В строках 12—16 определена рутинная процедура потока для однорангового потока. При этом просто распечатывается строка, после чего выполнение однорангового потока завершается выполнением оператора `return` в строке 15.

13.3.3. Создание потоков

Одни потоки создают другие вызовом функции `pthread_create`:

```
#include <pthread.h>
typedef void * (func) (void *);

int pthread_create (pthread_t *tid, pthread_attr_t *attr,
    func *f, void *arg);
```

Функция `pthread_create` создает новый поток и запускает рутинную процедуру потока в контексте нового потока с аргументом ввода `arg`. Аргумент `attr` можно использовать для изменения атрибутов по умолчанию вновь созданного потока. Изменение этих атрибутов не входит в предмет книги.

При возврате `pthread_create` аргумент `tid` содержит идентификатор вновь созданного потока. Новый поток может определить свой собственный идентификатор вызовом функции `pthread_self`.

```
#include <pthread.h>

pthread_t pthread_self (void);
```

13.3.4. Завершение выполнения потоков

Поток завершает выполнение следующими способами:

- неявно при возврате рутинной процедуры высокого уровня;
- явно путем вызова функции `pthread_exit`, которая вызывает указатель на возвращаемое значение `thread_return`. Если основной поток вызывает `pthread_exit`, он ожидает завершения выполнения других одноранговых потоков, после чего завершает работу основного потока и всего процесса с возвращаемым значением `thread_return`:

```
#include <pthread.h>

int pthread_exit (void *thread_return);
```

- некоторый одноранговый поток вызывает функцию Unix `exit`, которая прекращает выполнение процесса и всех связанных с ним потоков;
- другой одноранговый поток прерывает текущий поток вызовом функции `pthread_cancel` с идентификатором текущего потока:

```
#include <pthread.h>

int pthread_cancel (pthread_t tid);
```

Одни потоки ждут завершения выполнения других вызовом функции `pthread_join`:

```
#include <pthread.h>

int pthread_join (pthread_t tid, void **thread_return);
```

Функция `pthread_join` блокируется до прекращения выполнения потока `tid`, присваивает указатель (`void *`), возвращенный процедурой потока в ячейку, указанную `thread_return`, после чего собирает ("пожинает") ресурсы памяти, удерживаемые прерванным потоком.

Обратите внимание, что, в отличие от функции Unix `wait`, функция `pthread_join` может дождаться прекращения выполнения только особого потока. Нельзя "пронструировать" `pthread_wait` дождаться прекращения выполнения произвольного потока. Это может усложнить код путем принуждения к использованию других, менее интуитивных механизмов для обнаружения прекращения выполнения процесса. И Стивенс (Stevens) вполне убедительно заявляет о том, что в спецификации это является ошибкой [81].

13.3.5. Отделение потоков

В любой момент времени поток является объединяемым или отделенным. *Объединяемый поток* можно собрать и уничтожить другими потоками. Его ресурсы памяти (например, стек) не освобождаются до тех пор, пока он не будет собран другим потоком. Напротив, *отделенный поток* не может быть собран или уничтожен другими потоками. При прекращении выполнения его ресурсы памяти освобождаются системой автоматически.

Потоки по умолчанию создаются объединяемыми. Во избежание утечек памяти каждый объединяемый поток должен быть либо явно собран (сжат) другим потоком, либо отделен обращением к функции `pthread_detach`.

```
#include <pthread.h>

int pthread_detach (pthread_t tid);
```

Функция `pthread_detach` отделяет объединяемый поток `tid`. Потоки могут отделяться самостоятельно вызовом функции `pthread_detach` с аргументом `pthread_self ()`.

Несмотря на то, что в некоторых примерах будут использоваться объединяемые потоки, существуют убедительные причины использования в реальных программах отделенных потоков. Например, высокопроизводительный Web-сервер может создавать новый одноранговый поток каждый раз при получении запроса о подключении с Web-браузера. Поскольку каждое подключение управляет независимо отдельным потоком, то для сервера нет необходимости — и даже нежелательно — явно ожидать, пока каждый одноранговый поток прекратит свое выполнение. В этом случае каждый одноранговый поток должен самостоятельно отделяться до того, как он начнет обрабатывать запрос, с тем, чтобы его ресурсы памяти могли использоваться повторно после прекращения выполнения потока.

13.3.6. Инициализация потоков

Функция `pthread_once` позволяет инициализировать состояние, связанное с рутинной процедурой потока:

```
#include <pthread.h>

pthread_once_t once_control = PTHREAD_ONCE_INIT;

int pthread_once (pthread_once_t *once_control,
                  void (*init_routine) (void));
```

Переменная `once_control` является глобальной или статической, которая всегда инициализируется значением `PTHREAD_ONCE_INIT`. При первом вызове `pthread_once` с аргументом `once_control` она запускает `init_routine`, функцию без аргументов ввода, которая ничего не возвращает. Последующие обращения к `pthread_once` с аргументом `pthread_once` ни к чему не приводят. Функция `pthread_once` полезна всякий раз, когда необходимо динамически инициализировать глобальные переменные, совместно используемые множественными потоками. Один из примеров будет рассмотрен в разд. 13.6.

13.3.7. Параллельный сервер, основанный на потоках

В листинге 13.8 показан код для параллельного эхо-сервера, основанного на потоках. Общая структура сходна с проектом, основанным на процессах. Основной поток многократно сжидает запроса на подключение, после чего создает одноранговый поток для управления этим запросом. Код выглядит простым, однако в нем имеется пара общих и, в определенной степени, щекотливых моментов, на которые следует обратить особое внимание.

Листинг 13.8. Параллельный эхо-сервер, основанный на потоках

```
1 #include "csapp.h"
2
3 void echo (int connfd);
```

```
4 void *thread (void *vargp);
5
6 int main (int argc, char **argv)
7
8 int listenfd, *connfdp, port, clientlen=sizeof (struct sockaddr_in);
9 struct sockaddr_in clientaddr;
10 pthread_t tid
11
12 if (argc !=2) {
13 fprintf (stderr, "usage: &s <port>\n", argv [0]);
14 exit (0);
15 }
16 port = atoi (argv [1]);
17
18 listenfd = Open_listenfd (port);
19 while (1) {
20 connfdp = Malloc (sizeof (int));
21 * connfd = Accept (listenfd, (SA *) &clientaddr, &clientlen);
22 Pthread_create (&tid, NULL, thread, &connfd);
23 }
24 }
25
26 /* рутинная процедура потока */
27 void *thread (void *vargp)
28 {
29 int connfd = *((int *) vargp);
30 Pthread_detach (pthread_self ());
31 Free (vargp);
32 echo (connfd);
33 Close (connfd);
34 return NULL;
35}
```

Очевидным подходом является передача указателя на дескриптор следующим образом:

```
connfd = Accept (listenfd, (SA *) &clientaddr, &clientlen);
Pthread_create (&tid, NULL, thread, &connfd);
```

Затем одноранговый поток разыменовывает указатель и присваивает его локальной переменной следующим образом:

```
void *thread (void *vargp) {
int connfd = *( (int *)vargp);
...
}
```

Впрочем, это было бы неправильно, потому что таким образом представляется гонка между оператором присваивания в одноранговом потоке и оператором `accept` в основном потоке. Если оператор присваивания завершается до следующего оператора `accept`, тогда локальная переменная `connfd` в одноранговом потоке получает корректное значение дескриптора. Однако, если присваивание завершается после `accept`, тогда локальная переменная `connfd` в одноранговом потоке получает номер дескриптора следующего подключения. Ненужным результатом здесь становится то, что два потока теперь выполняют ввод и вывод на одном и том же дескрипторе. Во избежание потенциальной "гонки на выживание" необходимо присваивать каждый связный дескриптор, возвращенный `accept`, его собственному динамически выделенному блоку памяти, как показано в строках 20—21. К вопросу гонок авторы намерены вернуться в разд. 13.7.4.

Другим вопросом является избежание утечек памяти в рутинной процедуре потока. Поскольку потоки не собираются явно, каждый поток необходимо отделять так, чтобы его ресурсы памяти восстанавливались при прекращении выполнения потока (строка 30). Более того, необходимо следить за освобождением блока памяти, выделенного основным потоком (строка 31).

УПРАЖНЕНИЕ 13.5

В сервере, основанном на процессах (листинг 13.1), связный дескриптор был предусмотрен в порождающем и порожденном процессах. Однако в сервере, основанном на потоках (листинг 13.8), связный дескриптор закрывается только в одноранговом потоке. Почему?

13.4. Совместно используемые переменные в поточных программах

С точки зрения программиста, одним из привлекательных аспектов использования потоков является простота, с какой множественные потоки могут совместно использовать одни и те же программные переменные. Однако подобное совместное использование может быть коварным. Для корректного написания поточных программ необходимо четко понимать, что подразумевается под совместным использованием и как оно работает.

Для понимания того, используется ли совместно переменная в программе, необходимо рассмотреть несколько основополагающих вопросов: какова базовая модель памяти для потоков, каковы экземпляры переменной, отображенной в памяти, и сколько потоков обращается к каждому из этих экземпляров. Переменная используется совместно (является общей), если только потоки обращаются к определенному экземпляру этой переменной.

Для конкретизации обсуждения идеи совместного использования в качестве рабочего примера авторы используют программу, представленную листингом 13.9. Она выглядит несколько путаной, однако является полезной при изучении, потому что иллюстрирует несколько "тонких" положений понятия совместного использования пе-

ременных. Программа-пример состоит из основного потока, создающего два одноранговых потока. Основной поток передает каждому одноранговому потоку уникальный идентификатор, используемый для распечатки персонального сообщения вместе с подсчетом общего числа раз вызова рутинной процедуры потока.

Листинг 13.9. Различные аспекты совместного использования

```
1 #include "csapp.h"
2 #define N 2
3 void *thread (void *vargp);
4
5 char **ptr; /* глобальная переменная */
6
7 int main ()
8 {
9     int i;
10    pthread_t tid;
11    char *msgs [N] = {
12        "Hello from foo",
13        "Hello from bar"
14    };
15
16    ptr = msgs;
17    for (i = 0; i < N; i++)
18        Pthread_create (&tid, NULL, thread, (void *) i);
19    Pthread_exit (NULL);
20 }
21
22 void *thread (void *vargp)
23 {
24     int myid = (int) vargp;
25     static int cnt = 0;
26     printf ("%d: %s (cnt=%d)\n", myid, ptr [myid], ++cnt);
27 }
```

13.4.1. Модель памяти для потоков

Накопитель параллельных потоков выполняется в контексте процесса. Каждый поток имеет свой собственный отдельный контекст, включающий в себя идентификатор потока, указатель стека, счетчик команд, коды условий и значения регистров общего назначения. Каждый поток использует совместно с другими потоками остаток контекста процесса. Сюда входит полное виртуальное адресное пространство пользователя, состоящее из текста, предназначенного только для считывания (кода), данных для считывания или записи, динамической памяти, любого совместно используемого библиотечного кода и областей данных. Потоки также используют одно и то же множество открытых файлов.

В операционном смысле для одного потока невозможно считать или записать значения регистров другого потока. С другой стороны, любой поток может получить доступ к любой ячейке в совместно используемой виртуальной памяти. Если какой-либо поток модифицирует ячейку памяти, тогда любой другой поток, в конце концов, обнаружит это изменение, если он считает эту ячейку. Следовательно, регистры никогда не используются совместно, а виртуальная память — всегда.

Модель памяти для отдельных стеков потока не так прозрачна. Эти стеки содержатся в области стеков виртуального адресного пространства, и доступ к ней соответствующих потоков обычно осуществляется независимо. Здесь употреблено слово "обычно", а не "всегда", потому что разные стеки потока не защищены от других потоков. Поэтому, если какому-либо потоку удастся получить указатель на стек другого потока, тогда он может считать и записывать любую часть этого стека. В программе-примере это показано в строке 26, где одноранговые потоки косвенно обращаются к содержимому стека основного потока через переменную `ptr`.

13.4.2. Отображение переменных в память

Переменные в поточных программах С отображаются в виртуальную память в соответствии с их классами памяти.

- Глобальные переменные. Глобальной называется любая переменная, объявленная за пределами функции. Во время выполнения область считывания/записи виртуальной памяти содержит только один экземпляр каждой глобальной переменной, к которому может обратиться любой поток. Например, глобальная переменная `ptr`, объявленная в строке 5, имеет один экземпляр периода выполнения в области считывания/записи виртуальной памяти. При наличии только одного экземпляра переменной он обозначается просто именем переменной, в данном случае — `ptr`.
- Локальные автоматические переменные — это переменные, объявленные в рамках функции без атрибута `static`. Во время выполнения стек каждого потока содержит свои собственные экземпляры любых локальных автоматических переменных. Это так, даже если множественные потоки выполняют одну и ту же рутинную процедуру. Например, имеется один экземпляр локальной переменной `tid`; он расположен в стеке основного потока. Этот экземпляр будет обозначен `tid.m`. В качестве другого примера существует два экземпляра локальной переменной `myid`: один в стеке однорангового потока 0, а другой — в стеке однорангового потока 1. Эти экземпляры будут обозначены `myid.p0` и `myid.p1` соответственно.
- Локальные статические переменные. Локальная статическая переменная — это переменная, объявленная в рамках функции с атрибутом `static`. Как и в случае с глобальными переменными, область считывания/записи виртуальной памяти содержит только один экземпляр каждой локальной статической переменной, объявленной в программе. Например, несмотря на то, что каждый одноранговый поток в программе-примере объявляет `cnt` в строке 25, в период выполнения существует только один экземпляр, находящийся в области считывания/записи виртуальной памяти. Каждый одноранговый поток считывает и записывает этот экземпляр.

13.4.3. Совместно используемые переменные

Говорится, что переменная *v* используется совместно, если к одному из ее экземпляров обращается более одного потока. Например, переменная *cnt* в программе-примере является совместно используемой, потому что она имеет только один экземпляр периода выполнения, и к этому экземпляру обращаются оба одноранговых потока. С другой стороны, *myid* не является совместно используемой, потому что к каждому из ее двух экземпляров обращается только один поток. Важно понимать, что такие локальные автоматические переменные, как *msgs*, также могут использоваться совместно.

УПРАЖНЕНИЕ 13.6

- Используя анализ из разд. 13.4, заполните каждую графу следующей таблицы словами "Да" или "Нет" для программы-примера. В первом столбце запись *v.t* обозначает экземпляр переменной *v*, расположенный в локальном стеке потока *t*, где *t* — *m* (основной поток), *p0* (одноранговый поток 0) или *p1* (одноранговый поток 1).

Экземпляр переменной	Обращение основного потока?	Обращение однорангового потока 0?	Обращение однорангового потока 1?
<i>ptr</i>			
<i>cnt</i>			
<i>i.m</i>			
<i>msgs.m</i>			
<i>myid.p0</i>			
<i>myid.p1</i>			

- Какая из переменных *ptr*, *cnt*, *i*, *msgs* и *myid* является совместно используемой?

13.5. Синхронизация потоков с семафорами

Совместное использование переменных может быть удобным, однако они не исключают возможности появления ошибок синхронизации. Рассмотрим программу *badcnt.c*, показанную в листинге 13.10, создающую два потока, каждый из которых увеличивает совместно используемую переменную счетчика, называемую *cnt*.

Листинг 13.10. Некорректно синхронизированная программа-счетчик

```

1 #include "csapp.h"
2
3 #define NITERS 100000000

```

```
4 void *count (void *arg);
5
6 /* совместно используемая переменная счетчика */
7 unsigned int cnt = 0
8
9 int main ()
10 {
11     pthread_t tid1, tid2;
12
13     Pthread_create (&tid1, NULL, count, NULL);
14     Pthread_create (&tid2, NULL, count, NULL);
15     Pthread_join (&tid1, NULL);
16     Pthread_create (&tid2, NULL);
17
18     if (cnt != (unsigned)NITERS*2)
19         printf ("BOOM! cnt=%d\n", cnt);
20     else
21         printf ("OK! cnt=%d\n", cnt);
22     exit (0);
23 }
24
25 /* рутинная процедура потока */
26 void *count (void *arg)
27 {
28     int i;
29     for (i = 0; i < NITERS; i++)
30         cnt++;
31     return NULL;
32 }
```

Поскольку каждый поток увеличивает счетчик NITERS раз, можно ожидать, что конечное значение будет $2 \times \text{NITERS}$. Однако при запуске badcnt.c получаемые ответы не только неправильные; каждый раз ответы разные!

```
unix> . ./badcnt
BOOM! ctr=198841183
```

```
unix> . ./badcnt
BOOM! ctr=198261801
```

```
unix> . ./badcnt
BOOM! ctr=198269672
```

В чем же дело? Для четкого понимания проблемы необходимо изучить компонующий автокод для счетчика цикла (рис. 13.10). Будет полезно разбить код цикла для потока i на пять частей.

1. H_i — блок команд в начале цикла.
2. L_i — команда, загружающая совместно используемую переменную `cnt` в регистр `%eaxi`, где `%eaxi` обозначает значение регистра `%eax` в потоке i .
3. U_i — команда, обновляющая `%eaxi`.
4. S_i — команда, сохраняющая обновленное значение `%eaxi` в совместно используемой переменной `cnt`.
5. T_i — блок команд в конце цикла.

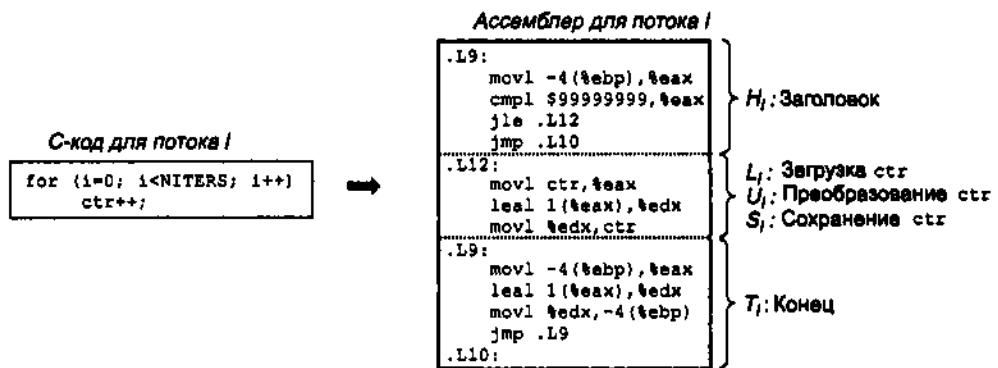


Рис. 13.10. Автокод для счетчика цикла

Упорядочение команд для первой итерации цикла

Обратите внимание, что заголовок и хвост манипулируют только локальными переменными стека, тогда как L_i , U_i и S_i манипулируют содержимым совместно используемой переменной счетчика.

Когда два одноранговых потока в `badcnt.c` выполняются параллельно на монопроцессоре, машинные команды завершаются одна за другой в том же порядке. Таким образом, каждое параллельное выполнение определяет некоторое общее упорядочение (или чередование) команд в этих двух потоках. К сожалению, одни из этих чередований произведут корректные результаты, а другие — нет.

Вообще, невозможно предсказать, выберет ли операционная система корректное упорядочение для потоков. Например, в табл. 13.1 показана пошаговая операция корректного упорядочения команд.

После того, как каждый поток обновит совместно используемую переменную `cnt`, его значение в памяти будет равно 2, что является ожидаемым результатом. С другой стороны, упорядочение, показанное в табл. 13.2, создает для `cnt` некорректное значение. Проблема возникает из-за того, что поток 2 загружает `cnt` в шаге 5, после того как поток 1 загружает `cnt` в шаге 2, но до того как поток 1 сохраняет обновленное значение в шаге 6. Таким образом, каждый поток заканчивает сохранением обновленного значения счетчика в 1.

Таблица 13.1. Корректное упорядочение

Шаг	Поток	Команда	%eax ₁	%eax ₂	cnt
1	1	H_1	-	-	0
2	1	L_1	0	-	0
3	1	U_1	1	-	0
4	1	S_1	1	-	1
5	2	H_1	-	-	1
6	2	L_1	-	1	1
7	2	U_1	-	2	1
8	2	S_1	-	2	2
9	2	T_1	-	2	2
10	1	T_1	1	-	2

Таблица 13.2. Некорректное упорядочение

Шаг	Поток	Команда	%eax ₁	%eax ₂	cnt
1	1	H_1	-	-	0
2	1	L_1	0	-	0
3	1	U_1	1	-	0
4	2	H_2	-	-	0
5	2	L_2	-	0	0
6	1	S_1	1	-	1
7	1	T_1	1	-	1
8	2	U_2	-	1	1
9	2	S_2	-	1	1
10	2	T_2	-	1	1

Данные понятия корректного и некорректного упорядочения команд можно прояснить с помощью схемы, называемой графиком продвижения.

УПРАЖНЕНИЕ 13.7

Заполните таблицу для следующего упорядочения команд badcnt.c.

Будет ли результатом такого упорядочения корректное значение для cnt?

Шаг	Поток	Команда	$\%eax_1$	$\%eax_2$	cnt
1	1	H_1	—	—	0
2	1	L_1			
3	2	H_2			
4	2	L_2			
5	2	U_2			
6	2	S_2			
7	1	U_1			
8	1	S_1			
9	1	T_1			
10	2	T_2			

13.5.1. Графы продвижения

Граф продвижения моделирует выполнение n параллельных потоков в виде траектории через n -мерное декартово пространство. Каждая ось k соответствует прогрессу потока k . Каждая точка (I_1, I_2, \dots, I_n) представляет состояния, где поток k имеет завершенную команду I_k . Начало графа соответствует первоначальному состоянию, в котором ни один из потоков не завершил выполнение команды.

На рис. 13.11 показан двумерный граф продвижения для первой итерации цикла программы badcnt.c. Горизонтальная ось соответствует потоку 1, вертикальная — потоку 2. Точка (L_1, S_2) соответствует состоянию, в котором поток 1 завершил L_1 , а поток 2 завершил S_2 .

Граф продвижения моделирует выполнение команд, как переход из одного состояния в другое. Переход изображен в виде ориентированного ребра из точки к смежной. Разрешенные переходы перемещаются вправо (завершается выполнение команды в потоке 1) или вверх (завершается выполнение команды в потоке 2). Выполнение двух команд не может завершаться одновременно: диагональные переходы не допускаются. Программы никогда не выполняются назад, поэтому переходы вниз или влево также недопустимы.

Протокол выполнения программы смоделирован в виде траектории через пространство состояний. На рис. 13.12 показана траектория, соответствующая следующему упорядочению команд:

$$H_1, L_1, U_1, H_2, L_2, S_1, T_1, U_2, S_2, T_2.$$

Для потока i команды (L_i, U_i, S_i) , манипулирующие содержимым совместно используемой переменной cnt, образуют критическую секцию (с учетом совместно используемой переменной cnt), которая не должна чередоваться с критической секцией другого потока. Пересечение двух критических секций определяет область простран-

ства состояний, называемую *небезопасной областью*. На рис. 13.13 показана небезопасная область для переменной c_{nt} . Обратите внимание на то, что небезопасная область примыкает к областям по своему периметру, но не включает их. Например, состояния (H_1, H_2) и (S_1, U_2) примыкают к небезопасной области, но не являются ее частью.

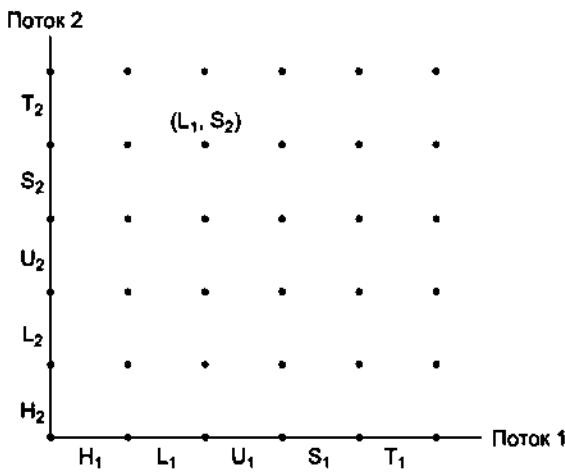


Рис. 13.11. Граф продвижения для первой итерации

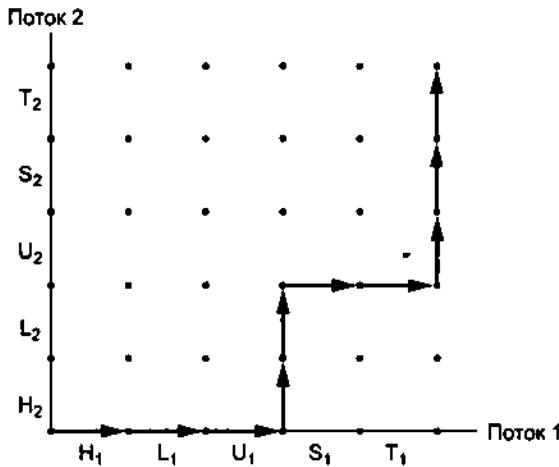


Рис. 13.12. Примерная траектория

Траектория, огибающая небезопасную область, называется *безопасной траекторией*, и наоборот, траектория, соприкасающаяся с любой частью небезопасной области, называется *небезопасной траекторией*. На рис. 13.14 показаны примеры безопасной и небезопасной траекторий, проходящих через пространство состояний рассматриваемой задачи.

ваемой программы-примера `badcnt.c`. Верхняя траектория огибает небезопасную область по левой и верхней сторонам и, следовательно, является безопасной. Нижняя траектория пересекает небезопасную область и, следовательно, является небезопасной.

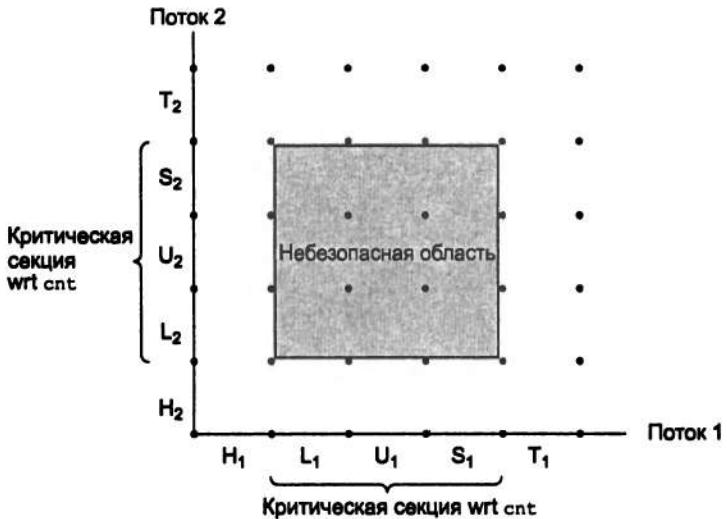


Рис. 13.13. Критические секции и небезопасные области

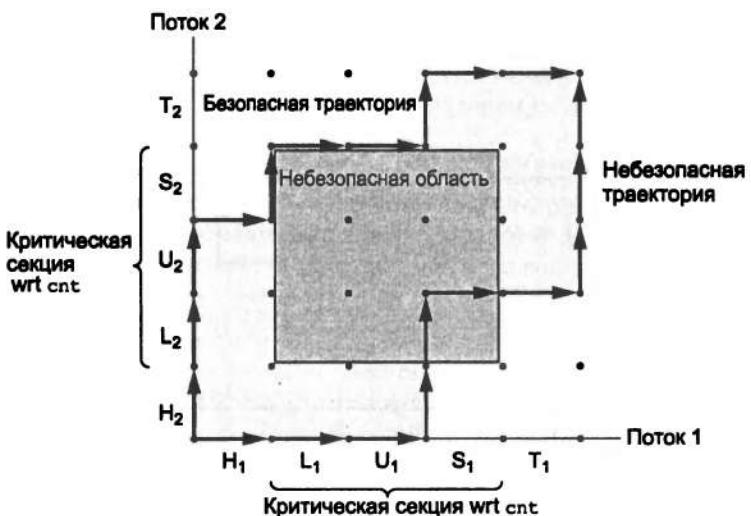


Рис. 13.14. Безопасная и небезопасная траектории

Любая безопасная траектория будет корректно обновлять совместно используемый счетчик. Для обеспечения гарантии корректного выполнения рассматриваемой при-

мерной поточной программы — да и любой параллельной программы, в которой совместно используются глобальные структуры данных, — так или иначе необходимо синхронизировать потоки так, чтобы они всегда имели безопасную траекторию.

13.5.2. Использование семафоров для доступа к совместно используемым переменным

Эдсгер Дейкстра (Edsger Dijkstra) — первый, кто изучил и сформулировал дисциплину параллельного программирования, предложил классическое решение проблемы синхронизации различных потоков выполнения, основанное на особом типе переменной, называемой *семафором*. Семафор s — это глобальная переменная с неотрицательным целочисленным значением, манипулировать которой могут только две специальные операции, называемые P и V .

- $P(s)$: Если s не равна нулю, тогда P уменьшает значение s и немедленно возвращается. Если s равна нулю, тогда процесс приостанавливается до того времени, пока s не примет ненулевое значение, и выполнение процесса возобновляется операцией V . После перезапуска операция P уменьшает значение s и возвращает управление вызывающему процессу.
- $V(s)$: Операция V увеличивает значение s на единицу. При наличии процессов, заблокированных в операции P , ожидающей для s ненулевого значения, операция V перезапускает только один из этих процессов, который затем завершает свою операцию P уменьшением значения s .

Операции тестирования и уменьшения значения в P происходят незаметно в том смысле, что как только предикат s принимает ненулевое значение, уменьшение значения s происходит без прерывания. Операция увеличения значения в V также происходит незаметно в том смысле, что она загружает, увеличивает и сохраняет семафор без прерывания.

Происхождение названий P и V

Эдсгер Дейкстра — уроженец Нидерландов. Названия P и V происходят из голландского языка, от слов *Proberen* (тестировать) и *Verhogen* (увеличивать значение).

Определения P и V гарантируют тот факт, что выполняющаяся программа никогда не может войти в состояние, где корректно инициализированный семафор имеет отрицательное значение. Это свойство, называемое *инвариантом семафора*, обеспечивает мощный инструмент управления траекториями параллельных программ, когда они избегают небезопасных областей.

Основной идеей является ассоциирование семафора s , первоначально равного 1, с каждой совместно используемой переменной (или со связанным множеством совместно используемых переменных) с последующим окружением соответствующей критической секции операциями $P(s)$ и $V(s)$. Семафор, используемый таким образом для защиты совместно используемых переменных, называется *двоичным семафором*, потому что его значение всегда равно 0 или 1.

Граф продвижения, показанный на рис. 13.15, представляет использование семафоров для корректной синхронизации примерной программы счетчика. Каждое состояние отмечено значением семафора s в этом состоянии. Принципиальной идеей здесь является то, что такая комбинация операций P и V создает набор состояний, называемый *запрещенной областью*, где $s < 0$. Из-за инварианта семафора ни одна из возможных траекторий не может включать в себя одно из состояний, находящихся в запрещенной области. И поскольку запрещенная область полностью включает в себя небезопасную область, ни одна из возможных траекторий не может касаться никакой части небезопасной области. Таким образом, каждая из возможных траекторий является безопасной, и независимо от упорядочения команд в период выполнения, программа корректно увеличивает значение счетчика.

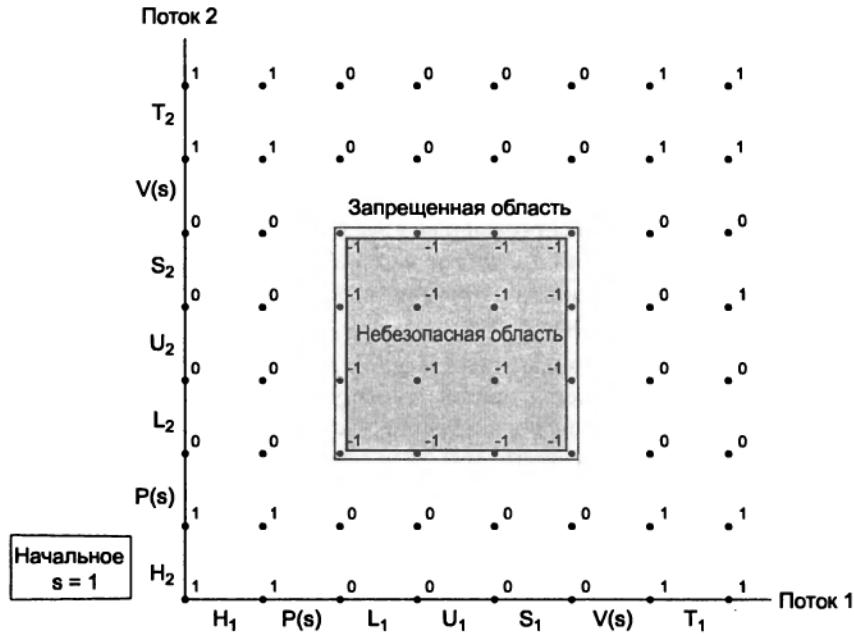


Рис. 13.15. Безопасное совместное использование с семафорами

В операционном смысле запрещенная область, созданная операциями P и V , делает невозможным выполнение команд множественными потоками в закрытой критической области в любой момент времени. Другими словами, операции семафора обеспечивают взаимоисключающий доступ к критической области. Общий феномен называется взаимным исключением.

Двоичные семафоры, целью которых является обеспечение взаимного исключения, часто называются *мьютексами* (MUTual EXclusion). Выполнение операции P на мьютексе называется блокированием последнего. Аналогично, выполнение операции V называется разблокированием мьютекса. Считается, что поток, который заблокировал, но еще не разблокировал мьютекс, удерживает его.

Ограничение графов продвижения

Графы продвижения обеспечивают прекрасный способ визуализации выполнения параллельных программ на монопроцессорах и понимания необходимости синхронизации. Однако они имеют свои ограничения, в частности в том, что касается параллельного выполнения на мультипроцессорах, где множество пар CPU/кэш совместно используют одну и ту же основную память. Поведение мультипроцессоров нельзя объяснить с помощью графов продвижения. Например, система памяти мультипроцессора может находиться в состоянии, не соответствующем никакой траектории на графике продвижения. Независимо от этого, сообщение остается тем же: доступ к совместно используемым переменным всегда нужно синхронизировать.

13.5.3. Семафоры Posix

Стандарт Posix определяет разнообразные функции для манипулирования семафорами. Три основные операции: `sem_init`, `sem_wait` (операция *P*) и `sem_post` (операция *V*).

```
#include <semaphore.h>

int sem_init (sem_t *sem, 0, unsigned int value);
int sem_wait (sem_t *s); /* P(s) */
int sem_post (sem_t *s); /* V(s) */
```

Программа инициализирует семафор вызовом функции `sem_init`. Функция `sem_init` инициализирует семафор `sem` значением `value`. Перед использованием каждый семафор должен быть инициализирован. Для описываемых здесь целей средний аргумент всегда равен 0. Программы выполняют операции *P* и *V* вызовом функций `sem_wait` и `sem_post` соответственно. Для краткости авторы предпочитают использовать следующие интерфейсные функции *P* и *V*:

```
#include "csapp.h"

void P (sem_t *s); /* Интерфейсная функция для sem_wait */
void V (sem_t *s); /* Интерфейсная функция для sem_post */
```

Например, для корректной синхронизации приводимого примера счетчика можно объявить семафор, называемый `mutex`:

```
sem_t mutex;
```

Далее он инициализируется единицей в основной рутинной операции:

```
Sem_init (&mutex, 0, 1);
```

Наконец, переменная `cnt` защищается окружением операциями *P* и *V* в `mutex`:

```
P (&mutex);
cnt++;
V (&mutex);
```

13.5.4. Использование семафоров для планирования совместно используемых ресурсов

В предыдущем разделе рассматривался вопрос об использовании семафоров для обеспечения взаимоисключающего доступа к совместно используемым переменным. Другим важным применением семафоров является планирование доступов к совместно используемым ресурсам. В данном сценарии поток использует операцию семафора для извещения другого потока о том, что некоторое условие в состоянии программы стало верным. Классический пример: модель производитель-потребитель на рис. 13.16. Поток-производитель и поток-потребитель совместно используют ограниченный буфер с *n* сегментами.

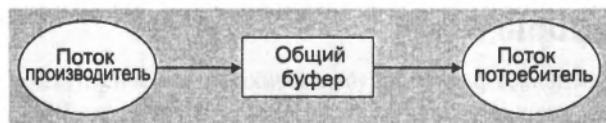


Рис. 13.16. Модель производитель-потребитель

Поток-производитель многократно создает новые элементы и вставляет их в буфер. Поток-потребитель многократно удаляет элементы из буфера и использует их. Варианты с разными количествами производителей и потребителей также возможны.

Поскольку вставка и удаление элементов связана с обновлением совместно используемых переменных, необходимо гарантированно обеспечить взаимоисключающий доступ к буферу. Однако гарантированного обеспечения взаимного исключения недостаточно. Здесь также необходимо запланировать доступы к буферу. Если буфер полон (отсутствуют пустые сегменты), тогда производитель должен дождаться освобождения сегмента. Точно так же, если буфер пуст (отсутствуют доступные элементы), тогда потребитель должен дождаться появления элементов.

Взаимодействие производителя и потребителя — обычное явление в реальных системах. Например, в мультимедийной системе задачей производителя является кодирование видеокадров, а потребителя — их раскодирование и отображение на экран. Цель буфера — снизить дрожание потока видеоданных, вызываемое различиями зависимостей по данным во времени кодирования и декодирования отдельных кадров. Буфер обеспечивает производителю накопитель сегментов, а потребителю — накопитель закодированных кадров. Другим расхожим примером является разработка графических пользовательских интерфейсов. Производитель выявляет сигналы (события) мыши и клавиатуры и вставляет их в буфер. Потребитель удаляет эти события из буфера в некой приоритетной последовательности и раскрашивает изображение на экране.

В данном разделе предполагается разработать простой пакет под названием SBUF для проектирования программ типа производитель-потребитель. В следующем разделе будут рассмотрены способы его использования для построения интересного параллельного сервера, основанного на предварительной организации поточной обработки. SBUF работает с буферами типа `sbuf_t` (листинг 13.11). Элементы хранятся в

динамически распределенном целочисленном массиве buf из n элементов. Индексы front и rear отслеживают первый и последний элемент в массиве. Три семафора управляют синхронизацией доступа к буферу. Семафор mutex обеспечивает взаимоисключающий доступ к буферу. Семафоры slots и items считают количество пустых сегментов и доступных элементов соответственно.

Листинг 13.11 Совместно используемый буфер

```

1 typedef struct {
2     int *buf; /* Буферный массив */
3     int n; /* Максимальное количество сегментов */
4     int front; /* buf [ (front+1) %n] — первый элемент */
5     int rear; /* buf [rear%n] — последний элемент */
6     sem_t mutex; /* Защищает доступы к buf */
7     sem_t slots; /* Считает доступные сегменты */
8     sem_t items; /* Считает доступные элементы */
9 } sbuf_t;

```

Функция sbuf_init (листинг 13.12) распределяет зарезервированную память для буфера, задает front и rear для указания пустого буфера и присваивает первичные значения трем семафорам. Эта функция вызывается один раз перед обращением к любой из трех других функций.

Листинг 13.12 Инициализация совместно используемого буфера

```

1 void sbuf_init (sbuf_t *sp, int n)
2 {
3     sp->buf = Calloc (n, sizeof (int));
4     sp->n = n; /* Буфер содержит максимальное количество n элементов */
5     sp->front = sp->rear = 0; /* Пустой буфер iff front == rear */
6     Sem_init (&sp->mutex, 0, 1); /* Двоичный семафор для блокирования */
7     Sem_init (&sp->slots, 0, n); /* Перв. buf имеет n пустых сегментов */
8     Sem_init (&sp->items, 0, 0); /* Перв. buf имеет 0 элементов данных */
9 }

```

Функция sbuf_deinit (не указана) освобождает буферное запоминающее устройство, когда программное приложение его использует. Функция sbuf_insert (листинг 13.13) ожидает появление доступного сегмента, блокирует мьютекс, добавляет элемент, разблокирует мьютекс, после чего объявляет о наличии нового элемента.

Листинг 13.13 Вставка элемента в честь совместно используемого буфера

```

1 void sbuf_insert (sbuf_t *sp, int item)
2 {
3     P (&sp->slots); /* Ожидание доступного сегмента */

```

```

4 P (&sp->mutex); /* Блокирование буфера */
5 sp->buf [++sp->rear) % (sp->n) ] = item; /* Вставка элемента */
6 V (&sp->mutex); /* Разблокирование буфера */
7 V (&sp->items); /* Объявление доступного элемента */
8 }

```

Функция `sbuf_remove` (листинг 13.14) симметрична. После ожидания доступного элемента буфера она блокирует мьютекс, удаляет элемент из передней части буфера, разблокирует мьютекс, после чего сигнализирует о наличии нового сегмента.

Листинг 13.14. Удаление элемента из совместно используемого буфера

```

1 int sbuf_remove (sbuf_t *sp)
2 {
3     int item;
4     P (&sp->slots); /* Ожидание доступного сегмента */
5     P (&sp->mutex); /* Блокирование буфера */
6     item = sp->buf [ (sp->front) % (sp->n) ]; /* Удаление элемента */
7     V (&sp->mutex); /* Разблокирование буфера */
8     V (&sp->slots); /* Объявление доступного элемента */
9     return item;
10 }

```

Другие механизмы синхронизации

Авторы уже демонстрировали синхронизацию потоков, используя семафоры, в основном потому, что они простые, классические и обладают понятной семантической моделью. Однако стоит знать, что существуют другие методики синхронизации. Например, потоки в Java синхронизированы с помощью механизма, называемого монитором Java [34], который обеспечивает более высокий уровень абстракции взаимного исключения и планирование возможностей семафора; фактически, мониторы можно реализовать с семафорами. Другим примером является интерфейс предварительной организации поточной обработки `Pthread`, определяющий множество операций синхронизации на переменных `mutex` и `condition`. Мьютексы `Pthreads` используются для взаимного исключения. Переменные условия `condition` применяются для планирования доступов к совместно используемым ресурсам, таким как ограниченный буфер в программе производитель-потребитель.

13.6. Параллельный сервер на базе предварительной организации поточной обработки

В этой главе рассмотрено использование семафоров для доступа к совместно используемым переменным, а также для планирования доступов к совместно используемым ресурсам. Для лучшего понимания этих вопросов применим их к параллельному серверу по методу предварительной организацией поточной обработки (`pthreading`).

В параллельном сервере (см. листинг 13.8) для каждого нового клиента создан новый поток. Недостаток данного подхода заключается в значительных затратах, связанных с созданием нового потока для каждого нового клиента. Сервер, основанный на предварительной организации поточной обработки, снижает эти издержки, используя модель производитель-потребитель, показанную на рис. 13.17.



Рис. 13.17. Организация параллельного сервера на базе предварительной организации поточной обработки

Сервер состоит из основного потока и множества рабочих потоков. Основной поток многократно принимает запросы на подключение от клиентов и помещает результирующие связные дескрипторы в буфер совместного использования. Каждый рабочий поток многократно удаляет дескриптор из буфера, обслуживает клиента, после чего ожидает следующего дескриптора.

В листинге 13.15 показано использование пакета SBUF для реализации параллельного эхо-сервера на базе предварительной организации поточной обработки. После инициализации буфера `sbuf` (строка 22) основной поток создает множество рабочих потоков (строки 25–26). Затем основной поток входит в бесконечный цикл сервера, принимая запросы на подключение и вставляя результирующие связные дескрипторы в `sbuf`. Каждый рабочий поток обладает очень простым поведением. Он ожидает возможности удаления связного дескриптора из буфера (строка 38), после чего обращается к функции `echo_cnt` для отражения клиентских данных ввода.

Листинг 13.15. Параллельный эхо-сервер на базе предварительной организации поточной обработки

```

1 #include "csapp.h"
2 #include "sbuf.h"
3 #define NTHREADS 4
4 #define SBUFSIZE 16
5
6 void echo_cnt(int connfd);
7 void *thread(void *vargp);
8

```

```

9 sbuf_t sbuf; /* совместно используемый буфер связных дескрипторов */
10
11 int main(int argc, char **argv)
12 {
13     int i, listenfd, connfd, port, clientlen=sizeof(struct sockaddr_in);
14     struct sockaddr_in clientaddr;
15     pthread_t tid;
16
17     if (argc != 2) {
18         fprintf(stderr, "usage: %s <port>\n", argv[0]);
19         exit(0);
20     }
21     port = atoi(argv[1]);
22     sbuf_init(&sbuf, SBUFSIZE);
23     listenfd = Open_listenfd(port);
24
25     for (i = 0; i < NTHREADS; i++) /* Создание рабочих потоков */
26         Pthread_create(&tid, NULL, thread, NULL);
27
28     while (1) {
29         connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
30         sbuf_insert(&sbuf, connfd); /* Вставка connfd в буфер */
31     }
32 }
33
34 void *thread(void *vargp)
35 {
36     Pthread_detach(pthread_self());
37     while (1) {
38         int connfd = sbuf_remove(&sbuf); /* Удаление connfd из буфера */
39         echo_cnt(connfd); /* Услуга клиента */
40         Close(connfd);
41     }
42 }
```

Функция `echo_cnt` (листинг 13.16) является версией функции `echo`, записывающей совокупное число байтов, полученных от всех клиентов, в глобальной переменной `byte_cnt`.

Листинг 13.16: Версия `echo`, считающая все полученные от клиентов байты

```

1 #include "csapp.h"
2
3 static int byte_cnt; /* счетчик байтов */
4 static sem_t mutex; /* и защищающий его mutex */
5
```

```

6 static void init_echo_cnt(void)
7 {
8     Sem_init(&mutex, 0, 1);
9     byte_cnt = 0;
10 }
11
12 void echo_cnt(int connfd)
13 {
14     int n;
15     char buf[MAXLINE];
16     rio_t rio;
17     static pthread_once_t once = PTHREAD_ONCE_INIT;
18
19     Pthread_once(&once, init_echo_cnt);
20     Rio_readinitb(&rio, connfd);
21     while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) (
22         P(&mutex);
23         byte_cnt += n;
24         printf("thread %d received %d (%d total) bytes on fd %d\n",
25                (int) pthread_self(), n, byte_cnt, connfd);
26         V(&mutex);
27         Rio_writen(connfd, buf, n);
28     )
29 }

```

Интересно изучить данный код, потому что он демонстрирует общую методику инициализации пакетов, вызываемых из рутинных процедур потоков. В рассматриваемом случае необходимо инициализировать счетчик `byte_cnt` и семафор `mutex`. Одним из методов, который использовался для пакетов SBUF и RIO, был запрос основного потока на явное обращение к функции инициализации. Другим представленным здесь подходом является использование функции `pthread_once` (строка 19) для вызова функции инициализации, как только какой-либо поток в первый раз вызовет функцию `echo_cnt`. Преимуществом данного подхода является то, что он упрощает пользование пакетом. Недостатком — то, что каждое обращение к `echo_cnt` вызывает функцию `pthread_once`, которая, по большей части, не делает ничего полезного.

Как только пакет инициализирован, функция `echo_cnt` инициализирует пакет RIO с буферизованным вводом-выводом (строка 20), после чего отражает каждую текстовую строку, полученную от клиента. Обратите внимание, что доступы к совместно используемой переменной `byte_cnt` в строках 23—24 защищены операциями `P` и `V`.

Событийно-управляемые программы, основанные на поточной обработке

Мультиплексирование (уплотнение) ввода-вывода — не единственный способ написания событийно-управляемых программ. К примеру, читатели, возможно, заметили, что спроектированный параллельный сервер на базе предварительной поточной обработки на самом деле является событийно-управляемым с простыми ко-

нечными автоматами для основного и рабочих потоков. Основной поток имеет два состояния (ожидание запроса на подключение и ожидание доступного сегмента буфера), два события ввода-вывода (поступление запроса на подключение и сегмент буфера доступен) и два перехода (прием запроса на подключение и вставка элемента буфера). Подобным же образом, каждый рабочий поток имеет одно состояние (ожидание доступного элемента буфера), одно событие ввода-вывода (элемент буфера доступен) и один переход (удаление элемента буфера).

13.7. Другие вопросы параллелизма

Читатели, возможно, заметили, что жизнь программиста значительно усложняется, как только от него требуется синхронизация доступа к совместно используемым данным. До сих пор авторы рассматривали методики взаимного исключения и синхронизацию по типу производитель-потребитель, однако это лишь верхушка айсберга. Синхронизация — это исключительно сложная и фундаментальная проблема, поднимающая вопросы, которые просто не могут возникнуть при рассмотрении обычных последовательных программ. Данный раздел представляет собой обзор (далеко не полный!) некоторых положений, которые следует обязательно учитывать при написании параллельных программ. Для конкретизации обсуждение будет строиться в переводе на потоки. Впрочем, не следует забывать, что это типично для положений, возникающих тогда, когда параллельные потоки любого типа манипулируют совместно используемыми ресурсами.

13.7.1. Безопасность потоков

Считается, что функция *безопасна по потокам*, если она всегда будет выдавать корректные результаты при многократном к ней обращении параллельных потоков. Различают четыре класса функций, небезопасных по потокам.

Класс 1: функции, не защищающие совместно используемые переменные

Эта проблема уже встречалась при рассмотрении функции `count` (листинг 13.10), увеличивающей незащищенную глобальную переменную счетчика. Данный класс небезопасных по потокам функций сравнительно просто сделать безопасным: достаточно защитить совместно используемые переменные операциями синхронизации, такими как `P` и `I`. Преимуществом является то, что он не требует изменений вызываемой программы. Недостаток — синхронизирующие операции замедляют выполнение функции.

Класс 2: функции, поддерживающие состояние при многократных вызовах

Генератор псевдослучайных чисел — простой пример данного класса небезопасной по потокам функции. Рассмотрим пакет генератора псевдослучайных чисел (листинг 13.17).

Функция `rand` является небезопасной по потокам, потому что результат текущего вызова зависит от промежуточного результата предыдущей итерации. При многократном вызове `rand` из одного потока после установки в нем обращения к `srand` можно ожидать повторяющуюся последовательность чисел. Однако данное предположение не оправдывается, если множественные потоки вызывают `rand`.

Единственным способом обезопасить по потокам такую функцию, как `rand` — это переписать ее так, чтобы она не использовала никаких данных `static`, чтобы вызывающая программа передала информацию о состоянии в аргументы. Недостатком является то, что теперь программист вынужден изменить код также и в вызывающей процедуре. В большой программе, где потенциально имеются сотни различных вызывающих местоположений, внесение таких изменений может оказаться сложной задачей, подверженной ошибкам [40].

Листинг 13.17. Небезопасный по потокам генератор псевдослучайных чисел

```
1 unsigned int next = 1;
2
3 /* rand - возврат псевдослучайного целого числа на 0..32767 */
4 int rand(void)
5 {
6     next = next*1103515245 + 12345;
7     return (unsigned int)(next/65536) % 32768;
8 }
9
10 /* srand - установка seed для rand() */
11 void srand(unsigned int seed)
12 {
13     next = seed;
14 }
```

Класс 3: функции, возвращающие указатель статической переменной

Некоторые функции, такие как `gethostbyname`, вычисляют результат в структуре `static`, после чего возвращают указатель в эту структуру. Вызов таких функций из параллельных потоков может закончиться катастрофой, потому что результаты, используемые одним потоком, без предупреждения переписываются другим потоком.

Существует два способа обработки данного класса небезопасных по потокам функций. Можно переписать функцию так, чтобы вызывающая программа передавала адрес структуры, в которой должен сохраняться результат. Это удаляет все совместно используемые данные, однако требует от программиста внесения изменений также в код вызывающей программы.

Если небезопасную по потокам функцию трудно или невозможно модифицировать (например, она из библиотеки), рекомендуем использовать альтернативный вариант,

называемый методикой блокировки и копирования. Идея заключается в ассоциировании мютекса с небезопасной по потокам функцией. На каждом местоположении вызова заблокируйте мютекс, вызовите небезопасную по потокам функцию, динамически распределите память для результата, скопируйте возвращенный функцией результат в эту память и разблокируйте мютекс. Другой довольно привлекательный способ: определите безопасную по потокам функцию-упаковщик, выполняющую блокировку и копирование, после чего замените все вызовы небезопасной функции на вызовы упаковщика. Например, в листинге 13.18 представлена безопасная по потокам версия `gethostbyname`, использующая методику блокировки и копирования.

Класс 4: функции, вызывающие небезопасные по потокам функции

Если функция f вызывает небезопасную по потокам функцию g , станет ли функция f также небезопасной по потокам? Как сказать... Если g — функция класса 2, полагающаяся на состояние в течение многократных вызовов, тогда f также будет небезопасной по потокам, и быстрого регрессивного способа перезаписи g не существует. Однако, если функция g принадлежит классу 1 или 3, тогда f может по-прежнему оставаться безопасной по потокам, если местоположение вызова и все результирующие совместно используемые данные будут защищены мютексом. Хороший пример такой ситуации показан в листинге 13.18, где для написания безопасной по потокам функции, вызывающей небезопасную по потокам функцию, использовалась методика блокировки и копирования.

Листинг 13.18. Безопасная по потокам программа-упаковщик

```

1 struct hostent *gethostbyname_ts(char *hostname)
2 {
3     struct hostent *sharedp, *unsharedp;
4
5     unsharedp = Malloc(sizeof(struct hostent));
6     P(&mutex);
7     sharedp = gethostbyname(hostname);
8     *unsharedp = *sharedp; /* копирование совместно используемой struct
                           в частную struct */
9     V(&mutex);
10    return unsharedp;
11 }
```

13.7.2. Реентерабельность

Существует важный класс безопасных по потокам функций, называемых *реентерабельными*, характеризующихся тем свойством, что они не запрашивают никаких совместно используемых данных при вызове потоками.

Несмотря на то, что термины "безопасная по потокам" и "реентерабельная" иногда (некорректно) используются как синонимы, между ними существует четкое разделение.

ние технического плана, которое следует учитывать. На рис. 13.18 показаны взаимоотношения между функциями как множества безопасных и небезопасных по потокам функций. Множество реентерабельных функций — это подмножество безопасных по потокам функций.

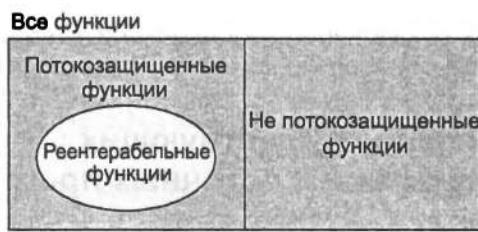


Рис. 13.18. Взаимоотношения между наборами реентерабельных, безопасных и небезопасных по потокам функций

Как правило, реентерабельные функции более эффективны, нежели нереентерабельные безопасные по потокам функции, потому что не требуют операций синхронизации. Более того, единственным способом преобразования небезопасной функции класса 2 в безопасную является ее перезапись таким образом, чтобы она стала реентерабельной. Например, в листинге 13.18 показана реентерабельная версия функции `rand`. Ключевой идеей является то, что статическая переменная `next` замещается указателем, передаваемым вызывающей программой.

Возможно ли изучить код какой-либо функции и заранее объявить ее реентерабельной? К сожалению, это зависит от разных факторов. Если все аргументы функции передаются значением (при отсутствии указателей), а все обращения к данным относятся к локальным автоматическим переменным стека (при отсутствии обращений к статическим или глобальным переменным), тогда такая функция называется явно реентерабельной в том смысле, что ее реентерабельность можно подтвердить независимо от способа обращения к ней.

Однако, если сделать допущения менее строгими и позволить передачу некоторых параметров рассматриваемой явно реентерабельной функции с помощью ссылки (т. е. позволить передачу указателей), тогда налицо наличие неявно реентерабельной функции в том смысле, что реентерабельной она будет только в том случае, если потоки аккуратно передают указатели не используемым совместно данным. К примеру, функция `rand_r`, показанная в листинге 13.19, является неявно реентерабельной.

Листинг 13.19. Реентерабельная версия функции

```

1 /* rand_r - реентерабельное псевдослучайное целое число на 0..32767 */
2 int rand_r(unsigned int *nextp)
3 {
4     *nextp = *nextp * 1103515245 + 12345;
5     return (unsigned int)(*nextp / 65536) % 32768;
6 }
  
```

Используемое в книге понятие относится как к явно, так и неявно реентерабельным функциям. Однако важно понимать, что реентерабельность иногда является свойством как вызывающей программы, так и вызываемой.

УПРАЖНЕНИЕ 13.8

Функция `gethostbyname_ts` (листинг 13.18) является безопасной по потокам, но не является реентерабельной. Объясните, почему?

13.7.3. Использование существующих библиотечных функций в поточных программах

Большинство функций Unix и функций, определенных в стандартной библиотеке C (`malloc`, `free`, `realloc`, `printf` и `scanf`), являются безопасными по потокам, с некоторыми исключениями. В табл. 13.3 перечислены общие исключения. Полный список представлен в [81].

Таблица 13.3. Общие небезопасные по потокам библиотечные функции

Небезопасная по потокам функция	Небезопасная по потокам функция	Безопасная версия Unix
<code>rand</code>	2	<code>rand_r</code>
<code>strtok</code>	2	<code>strtok_r</code>
<code>asctime</code>	3	<code>asctime_r</code>
<code>ctime</code>	3	<code>ctime_r</code>
<code>gethostbyaddr</code>	3	<code>gethostbyaddr_r</code>
<code>gethostbyname</code>	3	<code>gethostbyname_r</code>
<code>inet_ntoa</code>	3	(нет)
<code>localtime</code>	3	<code>localtimer_r</code>

Функции `asctime`, `ctime` и `localtime` обычно используются для прямых и обратных преобразований из разных форматов времени и дат. Функции `gethostbyname`, `gethostbyaddr` и `inet_ntoa` — типичные функции, используемые в сетевом программировании (см. главу 12). Функция `strtok` — второстепенная (ее использование не рекомендуется) для синтаксического анализа строк.

За исключением `rand` и `strtok`, все эти небезопасные по потокам функции являются разновидностью функций класса 3 и возвращают указатель статической переменной. При необходимости вызова одной из этих функций в поточной программе простейшим подходом будет использование методики блокировки и копирования. Недостатком ее является то, что дополнительная синхронизация замедляет выполнение программы. Более того, данный подход не сработает для небезопасной функции класса 2,

например, `rand`, полагающейся на статическое состояние на протяжении всех вызовов. Следовательно, Unix предоставляет реентерабельные версии большинства небезопасных по потокам функций. Названия реентерабельных функций всегда имеют окончание `_x`.

Например, реентерабельная версия `gethostbyname` называется `gethostbyname_x`. К сожалению, реентерабельные функции Unix описаны мало и на разных системах Unix имеют разные интерфейсы. По этой причине их рекомендуется избегать.

13.7.4. Гонки

Феномен *гонки* имеет место, когда корректность программы зависит от одного потока, достигающего точки *x* в его управляющей логике до того, как другой поток достигнет точки *y*. Гонки обычно происходят из-за того, что программисты предполагают выбор потоками каких-то особых траекторий в пространстве выполнения состояния, забывая правило, гласящее, что поточные программы должны работать корректно по любой вероятной траектории.

Природу гонок проще всего понять на примере. Рассмотрим простую программу (листинг 13.20). Основной поток создает четыре одноранговых потока и передает указатель уникальному целочисленному идентификатору каждого. Каждый одноранговый поток копирует переданный в его аргумент идентификатор в локальную переменную (строка 21), после чего распечатывает сообщение, содержащее идентификатор.

Программа выглядит довольно простой, однако результатом ее запуска будет следующий некорректный результат:

```
unix> ./race
Hello from thread 1
Hello from thread 3
Hello from thread 2
Hello from thread 3
```

Листинг 13.20. Программа с гонкой

```
1 #include "csapp.h"
2 #define N 4
3
4 void *thread(void *vargp);
5
6 int main()
7 {
8     pthread_t tid[N];
9     int i;
10
11    for (i = 0; i < N; i++)
12        Pthread_create(&tid[i], NULL, thread, &i);
```

```

13     for (i = 0; i < N; i++)
14         Pthread_join(tid[i], NULL);
15     exit(0);
16 }
17
18 /* рутинная процедура потока */
19 void *thread(void *vargp)
20 {
21     int myid = *((int *)vargp);
22     printf("Hello from thread %d\n", myid);
23     return NULL;
24 }
```

Проблема вызвана гонкой между каждым одноранговым потоком и основным потоком. Возможно ли проследить гонку? Вот что происходит: когда основной поток создает в строке 12 одноранговый поток, он передает указатель локальной переменной *i* стека. На этом этапе гонка происходит между следующим вызовом `pthread_create` в строке 12, разыменованием и присвоением аргумента в строке 21. Если одноранговый поток выполняет строку 21 до того, как основной поток выполнит строку 12, тогда переменная *myid* приобретает корректный идентификатор. В противном случае, она будет содержать идентификатор какого-то другого потока. Здесь страшно то, что получение корректного ответа зависит от того, как ядро планирует выполнение потоков. На описанной в книге системе корректный ответ получить невозможно, однако на других системах такая схема может сработать корректно, и программист окажется в блаженном неведении о наличии в его продукте серьезной ошибки.

Можно динамически распределить отдельный блок для каждого целочисленного идентификатора и передать рутинную процедуру указателя этим блокам, как показано в листинге 13.21 (строки 12—14). Обратите внимание, что процедура потоков должна освобождать блок во избежание утечек памяти.

Листинг 13.21. Корректная версия программы

```

1 #include "csapp.h"
2 #define N 4
3
4 void *thread(void *vargp);
5
6 int main()
7 {
8     pthread_t tid[N];
9     int i, *ptr;
10
11    for (i = 0; i < N; i++) {
12        ptr = Malloc(sizeof(int));
13        *ptr = i;
```

```

14     Pthread_create(&tid[i], NULL, thread, ptr);
15 }
16 for (i = 0; i < N; i++)
17     Pthread_join(tid[i], NULL);
18 exit(0);
19 }
20
21 /* рутинная процедура потока */
22 void *thread(void *vargp)
23 {
24     int myid = *((int *)vargp);
25     Free(vargp);
26     printf("Hello from thread %d\n", myid);
27     return NULL;
28 }
```

При запуске данной программы в нашей системе получаем корректный результат:

```

unix> ./norace
Hello from thread 0
Hello from thread 1
Hello from thread 2
Hello from thread 3
```

УПРАЖНЕНИЕ 13.9

В листинге 13.21 может возникнуть соблазн освободить распределенный блок памяти сразу после строки 15 в основном потоке, вместо его освобождения в одноранговом потоке. Но это будет неправильно. Почему?

УПРАЖНЕНИЕ 13.10

- Гонка была устранена распределением отдельного блока памяти для каждого це-
личисленного идентификатора. Предложите другой способ, при котором не нужно обращаться к функциям malloc или free.
- Каковы преимущества и недостатки данного подхода?

13.7.5. Взаимоблокировка (туниковые ситуации)

Семафоры обеспечивают потенциал для возникновения одной из ошибок периода исполнения, называемой *взаимоблокировкой*, когда набор потоков блокируется, ожидая условия, которое никогда не будет верным. Для понимания природы туниковых ситуаций ценнейшим инструментом является граф продвижения.

Например, на рис. 13.19 показан график продвижения для пары потоков, использующих два семафора для взаимного исключения. Из этого графа можно вывести несколько важнейших выводов, связанных с пониманием феномена взаимной блокировки потоков:

- Программист некорректно упорядочил операции P и V так, что запрещенные области двух семафоров перехлестываются. Если случится так, что какая-либо траектория выполнения достигнет состояния взаимоблокировки d , тогда дальнейшее продвижение невозможно, потому что перехлестывающиеся запрещенные области блокируют продвижение в каждом допустимом направлении. Другими словами, программа входит в тупиковую ситуацию из-за того, что каждый поток ожидает, пока другой выполнит операцию V , чего никогда не произойдет.
- Перехлестывающиеся запрещенные области вызывают набор состояний, называемый областью тупиковой ситуации (или взаимоблокировки). Если траектория вдруг соприкоснется с состоянием в области тупиковой ситуации, тогда последняя неизбежна. Траектории могут входить в области тупиковых ситуаций, но никогда не могут выходить из них.
- Взаимоблокировка — это особенно сложный вопрос, потому что ее не всегда можно спрогнозировать. Некоторые "удачные" траектории выполнения обогнут область тупиковой ситуации; другим же не повезет. На рис. 13.19 есть пример той и другой ситуации. Последствия для программиста могут оказаться, мягко говоря, не очень приятными. Программу можно без проблем запускать 1000 раз, а на 1001-й возникнет тупиковая ситуация. Либо программа будет прекрасно работать на одной машине, но "сбить" на другой. Хуже всего то, что эту повторить нельзя, потому что разные выполнения имеют разные траектории.

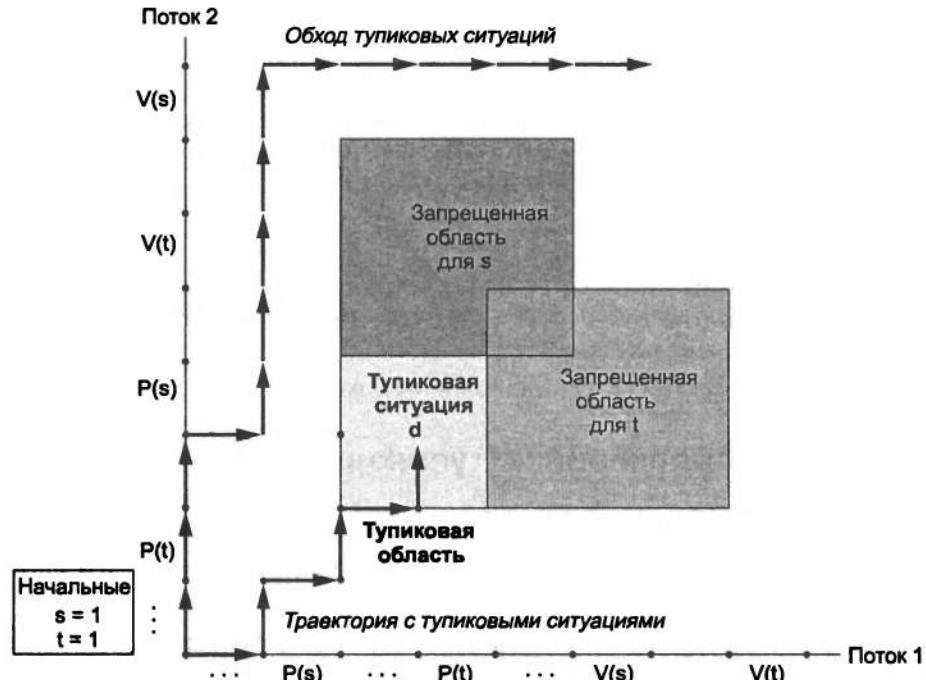


Рис. 13.19. Граф продвижения для программы, которая может попасть в тупиковую ситуацию

Программы входят в тупиковые ситуации по многим причинам, и избежать их — задача достаточно сложная. Впрочем, при использовании двоичных семафоров, как показано на рис. 13.19, можно применить следующее простое и эффективное правило, чтобы избежать тупиковых ситуаций: программа свободна от тупиковых ситуаций, если для каждой пары мьютексов (s, t) в программе каждый поток, содержащий s и t , одновременно блокирует их в том же порядке.

Например, можно исправить тупиковую ситуацию, показанную на рис. 13.19, сначала блокированием s , а затем t в каждом потоке. На рис. 13.20 показан получившийся в результате граф продвижения.

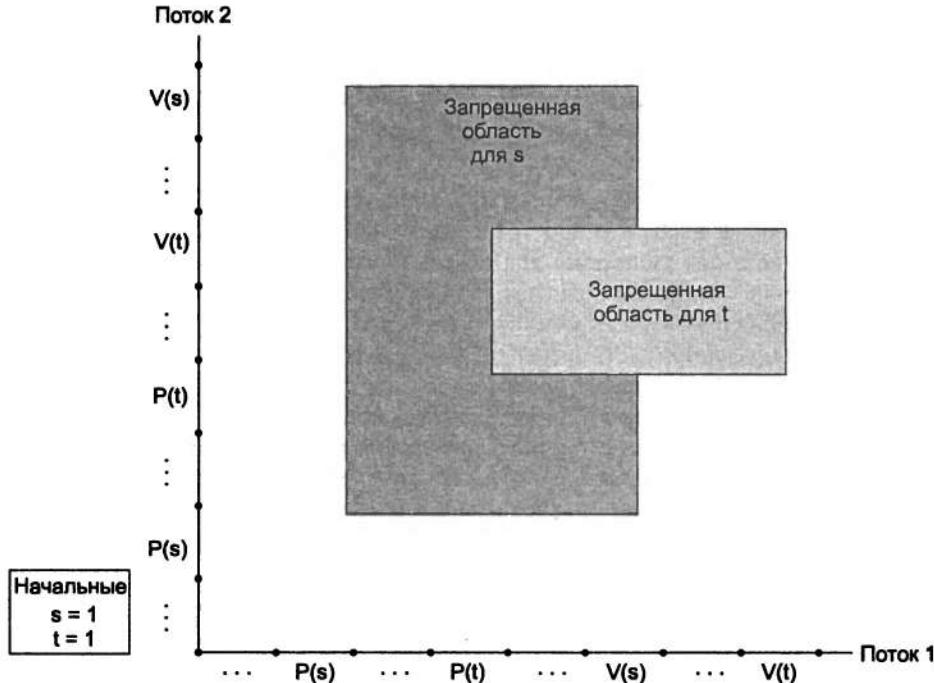


Рис. 13.20. Граф продвижения для программы, свободной от тупиковых ситуаций

УПРАЖНЕНИЕ 13.11

Рассмотрите следующую программу, в которой делается попытка использовать пару семафоров для взаимного исключения:

Initially: $s = 1$, $t = 0$.

Thread 1: Thread 2:

```
P(s); P(s);
V(s); V(s);
P(t); P(t);
V(t); V(t);
```

- Нарисуйте граф продвижения для данной программы.
- Всегда ли это будет тупиковой ситуацией?
- Если да, какое простое изменение первоначальных значений семафора устранит потенциальный риск тупиковой ситуации?
- Нарисуйте граф продвижения для получившейся программы, свободной от тупиковых ситуаций.

13.8. Резюме

Параллельная программа состоит из набора логических схем, перекрывающихся во времени. В данной главе были изучены три различных механизма построения параллельных программ: процессы, мультиплексирование ввода-вывода и потоки. На всем протяжении дискуссии параллельный сетевой сервер использовался как мотивирующее приложение.

Процессы автоматически планируются ядром, и по причине того, что они имеют раздельные виртуальные адресные пространства, они требуют явных механизмов IPC (межпроцессорное взаимодействие) для совместного использования данных. Событийно-управляемые программы создают свои собственные параллельные логические схемы, смоделированные в виде конечных автоматов, а также для явного планирования потоков используют мультиплексирование ввода-вывода. По причине того, что программы выполняются в одном процессе, совместное использование данных между схемами происходит быстро и просто. Потоки представляют собой симбиоз (гибрид) этих подходов. Подобно тому, как логические схемы основаны на процессах, потоки автоматически планируются ядром. Подобно тому, как логические схемы основаны на мультиплексировании ввода-вывода, потоки выполняются в контексте одного процесса и могут быстро и просто осуществлять совместное использование данных.

Независимо от механизма параллелизма, синхронизация параллельных доступов к совместно используемым данным представляет собой сложную проблему. В помощь были разработаны операции *P* и *V* на семафорах. Операции семафоров можно использовать для обеспечения взаимоисключающего доступа к совместно используемым данным, а также для планирования доступа к таким ресурсам, как совместно используемые буферы в программах, построенных по типу производитель-потребитель. Параллельный эхо-сервер с предварительной организацией поточной обработки предоставляет убедительный пример этих двух сценариев использования семафоров.

Параллелизм является источником и других сложностей. Функции, вызываемые потоками, должны обладать свойством, называемым безопасностью потока. В главе выделены четыре класса небезопасных по потокам функций вместе с предложениями по обеспечению их безопасности. Реентерабельные функции представляют собой необходимое подмножество безопасных по потокам функций, не имеющих доступа к каким бы то ни было совместно используемым данным. Реентерабельные функции часто более эффективны, нежели нереентерабельные, потому что они не требуют

синхронизации. Другими сложными моментами, возникающими при построении параллельных программ, являются гонки и тупиковые ситуации. Гонки возникают тогда, когда программисты делают некорректные допущения о том, как планируются логические схемы. Тупиковые ситуации возникают тогда, когда рабочий поток ожидает события, которое никогда не произойдет.

Библиографические примечания

Операции семафоров предложены Дейкстра (Dijkstra) [84]. Концепция графов продвижения представлена Коффманом (Coffman) [16] и позже формализована Карсоном (Carson) и Рейнольдсом (Reynolds) [10]. Книга Бутенхофа (Butenhof) [9] представляет собой подробное описание интерфейса переносимой операционной системы (POSIX). Материал Биррелла (Birrell) [4] — прекрасное введение в поточное. Пью (Pugh) распознает слабые места методов, с помощью которых потоки Java взаимодействуют посредством памяти, и предлагает модели замещения памяти [61].

Задачи для домашнего решения

УПРАЖНЕНИЕ 13.12 •

Напишите версию программы `hello.c`, создающую и "пожинающую" *n* объединяемых одноранговых потоков, где *n* — аргумент командной строки.

УПРАЖНЕНИЕ 13.13 •

1. Программа в следующем листинге имеет ошибку. Предполагается, что поток должен задержаться (заснуть) на одну секунду, после чего напечатать строку. Однако при ее запуске на нашей системе ничего не распечатывается. Почему?

```
1#include "csapp.h"
2 void *thread(void *vargp);
3
4 int main()
5 {
6     pthread_t tid;
7
8     Pthread_create(&tid, NULL, thread, NULL);
9     exit(0);
10}
11
12 /* рутинная процедура потока */
13 void *thread(void *vargp)
14 {
15     Sleep(1);
16     printf("Hello, world!\n");
17     return NULL;
18 }
```

2. Эту ошибку можно исправить замещением функции `exit` в строке 9 на вызов одной или двух разных функций `Pthread`. Каких?

УПРАЖНЕНИЕ 13.14 ++

Проверьте свое понимание функции `select` путем модификации программы сервера (листинг 13.2) так, чтобы он отражал максимум одну текстовую строку на одну итерацию цикла основного сервера.

УПРАЖНЕНИЕ 13.15 ++

Событийно-управляемый параллельный эхо-сервер (листинг 13.2) имеет недостатки, потому что клиент- злоумышленник может отказать в обслуживании других клиентов отправкой частичной текстовой строки. Напишите усовершенствованную версию программы сервера, который смог бы обрабатывать эти частичные текстовые строки без блокирования.

УПРАЖНЕНИЕ 13.16 +

Функции в пакете ввода-вывода RIO (см. разд. 11.4) безопасны по потокам. Являются ли они также реентерабельными?

УПРАЖНЕНИЕ 13.17 +

В параллельном эхо-сервере с предварительной организацией поточной обработки каждый поток вызывает функцию `echo_cnt`. Является ли функция `echo_cnt` безопасной по потокам? Является ли она реентерабельной? Почему да или почему нет?

УПРАЖНЕНИЕ 13.18 ++

В некоторых работах по сетевому программированию предлагается следующий подход к сокетам считывания и записи: перед взаимодействием с клиентом откройте два потока стандартного ввода-вывода на одном открытом связном сокете дескриптора: один поток для считывания, а другой — для записи:

```
FILE *fpin, *fpout;
fpin = fdopen (sockfd, "r");
fpout = fdopen (sockfd, "w");
```

Когда сервер закончит взаимодействие с клиентом, закройте оба потока следующим образом:

```
fclose (fpin);
fclose (fpout);
```

Однако, если попробовать применить данный подход на параллельном сервере, основанном на потоках, будет создано условие "гонки на выживание". Объясните, почему.

УПРАЖНЕНИЕ 13.19 ♦

Обусловит перемена мест двух операций V , показанных на рис 13.20, появление туниковых ситуаций в программе или нет? Обоснуйте ответ изображением графа продвижения для четырех возможных случаев:

Случай 1		Случай 2		Случай 3		Случай 4	
Поток 1	Поток 2						
P(s)	P(s)	P(s)	P(s)	P(s)	P(s)	P(s)	P(s)
P(t)	P(t)	P(t)	P(t)	P(t)	P(t)	P(t)	P(t)
V(s)	V(s)	V(s)	V(t)	V(t)	V(s)	V(t)	V(t)
V(t)	V(t)	V(t)	V(s)	V(s)	V(t)	V(s)	V(s)

УПРАЖНЕНИЕ 13.20 ♦

Может ли следующая программа попасть в туниковую ситуацию? Почему да или почему нет?

Initially: $a = 1$, $b = 1$, $c = 1$.

```
Thread 1: Thread 2:  
P(a); P(c);  
P(b); P(b);  
V(b); V(b);  
P(c); V(c);  
V(c);  
V(a);
```

УПРАЖНЕНИЕ 13.21 ♦

Рассмотрите следующую программу, которая попадает в туниковую ситуацию.

Initially: $a = 1$, $b = 1$, $c = 1$.

```
Thread 1: Thread 2: Thread 3:  
P(a); P(c); P(c);  
P(b); P(b); V(c);  
V(b); V(b); P(b);  
P(c); V(c); P(a);  
V(c); P(a); V(a);  
V(a); V(a); V(b);
```

- Перечислите для каждого потока пары мьютексов, которые он поддерживает одновременно.

- Если $a < b < c$, то какой поток нарушает правило упорядочивания блокировки мьютекса?
- Покажите для данных потоков новое упорядочивание блокировки, которое гарантирует освобождение от тупиковых ситуаций.

УПРАЖНЕНИЕ 13.22 +++

Реализуйте версию функции стандартного ввода-вывода `fgets` под названием `tfgets`, которая выходит за пределы отпущенного периода времени выполнения и возвраща-ет указатель `NULL`, если не получает строки ввода на стандартном вводе в течение 5 с. Функция должна быть реализована в пакете с названием `tfgets-proc.c` с исполь-зованием процессов, сигналов и нелокальных переходов. Она не должна использо-вать функцию Unix `alarm`. Протестируйте решение, используя программу драйвера в упр. 13.25.

УПРАЖНЕНИЕ 13.23 +++

Реализуйте версию функции `tfgets` из упр. 13.22, которая использует функцию `select`. Функция должна быть реализована в пакете с названием `tfgets-select.c`. Протестируйте решение, используя программу драйвера на рис. 13.22. Можно пред-положить, что стандартный ввод присваивается дескриптору 0.

УПРАЖНЕНИЕ 13.24 +++

Реализуйте поточную версию функции `tfgets` из упр. 13.22. Функция должна быть реализована в пакете с названием `tfgets-thread.c`. Протестируйте решение, исполь-зуя программу драйвера в упр. 13.25.

УПРАЖНЕНИЕ 13.25 +++

Реализуйте параллельную версию Web-сервера TINY, основанного на процессах. В процессе решения должен быть порожден новый процесс для каждого нового за-проса на подключение. Протестируйте решение, используя реальный Web-браузер.

```

1 #include "csapp.h"
2
3 char *tfgets(char *s, int size, FILE *stream);
4
5 int main()
6 {
7     char buf[MAXLINE];
8
9     if (tfgets(buf, MAXLINE, stdin) == NULL)
10         printf("BOOM!\n");
11     else
12         printf("%s", buf);
13
14     exit(0);
15 }
```

УПРАЖНЕНИЕ 13.26 ◆◆◆

Реализуйте параллельную версию Web-сервера TINY, основанного на мультиплексировании ввода-вывода. Протестируйте решение, используя реальный Web-браузер.

УПРАЖНЕНИЕ 13.27 ◆◆◆

Реализуйте параллельную версию Web-сервера TINY, основанного на потоках. Решение должно создать новый поток для каждого нового запроса на подключение. Протестируйте решение, используя реальный Web-браузер.

УПРАЖНЕНИЕ 13.28 ◆◆◆◆

Реализуйте параллельную версию Web-сервера TINY с предварительной организацией поточной обработки. Решение должно динамически увеличивать или уменьшать количество потоков в ответе на текущую загрузку. Одной стратегией будет удвоение числа потоков при заполнении буфера и сокращение числа потоков наполовину, когда буфер опустошается. Протестируйте решение, используя реальный Web-браузер.

УПРАЖНЕНИЕ 13.29 ◆◆◆◆

Web-агент — это программа, выполняющая роль посредника между Web-сервером и браузером. Вместо того, чтобы контактировать непосредственно с сервером для получения Web-страницы, браузер связывается с агентом (модулем доступа), который направляет запрос на сервер. Когда сервер отвечает агенту, последний отправляет ответ на браузер. Для данного упражнения необходимо написать простой модуль доступа, который бы отфильтровывал и регистрировал запросы:

1. В первой части лабораторной работы читатель настраивает агент для приема запросов, анализирует HTTP, отправляет запросы на сервер и возвращает результаты назад в браузер. Агент должен зарегистрировать URL (унифицированный указатель информационного ресурса) всех запросов в системном журнале на диске, а также заблокировать запросы любых URL, содержащихся в файле фильтра на диске.
2. Во второй части лабораторной работы студент сразу обновляет агента для работы с множественными открытыми подключениями путем порождения отдельного потока для работы с каждым запросом. Пока агент ждет ответа удаленного сервера на запрос для того, чтобы можно было обслужить один браузер, агент должен работать над незавершенным запросом с другого браузера.

Проверьте построенный агент, используя реальный Web-сервер.

Решение упражнений

РЕШЕНИЕ УПРАЖНЕНИЯ 13.1

Когда порождающий процесс создает новый процесс, он получает копию связного дескриптора и счетчик ссылок для ассоциированной таблицы файлов, увеличенный с 1 до 2. Когда порождающий процесс закрывает свою копию дескриптора, количест-

во ссылок уменьшается с 2 до 1. Поскольку ядро не закроет файл до тех пор, пока счетчик ссылок в его таблице файлов не примет нулевое значение, конец порожденного процесса подключения остается открытм.

РЕШЕНИЕ УПРАЖНЕНИЯ 13.2

Когда по какой-либо причине процесс прекращает выполнение, ядро закрывает все открытые дескрипторы. Таким образом, копия порожденного процесса связного файлового дескриптора будет закрыта автоматически при выходе порожденного процесса.

РЕШЕНИЕ УПРАЖНЕНИЯ 13.3

Вспомните, что дескриптор готов к считыванию, если запрос на считывание одного байта из этого дескриптора не будет блокирован. Если завершение файла (EOF) на дескрипторе верно, то дескриптор готов к считыванию, потому что операция считывания имеет нулевой код возврата, указывающий на завершение файла. Таким образом, ввод <Ctrl>+<D> заставляет функцию `select` осуществлять возврат с дескриптором 0 в множестве считывания.

РЕШЕНИЕ УПРАЖНЕНИЯ 13.4

Переменная `pool.ready_set` повторно инициализируется перед каждым вызовом `select`, потому что она служит аргументом ввода и вывода. При вводе она содержит множество считывания, при выводе — готовое множество.

РЕШЕНИЕ УПРАЖНЕНИЯ 13.5

Поскольку потоки выполняются в одном и том же процессе, все они совместно используют одну и ту же таблицу дескрипторов. Независимо от того, сколько потоков используют связный дескриптор, счетчик ссылок для таблицы файлов связного дескриптора равен единице. Таким образом, одной операции `close` достаточно для освобождения ресурсов памяти, ассоциированных со связным дескриптором, когда пользователь заканчивает с ним работу.

РЕШЕНИЕ УПРАЖНЕНИЯ 13.6

Основной идеей здесь является то, что переменные стека — частные, тогда как глобальная и статическая переменная являются совместно используемыми. Такие статические переменные, как `cnt`, довольно коварны, потому что совместное использование ограничено функциями, входящими в их границы, в данном случае — рутинной процедурой потоков.

1. Приведем таблицу и описание переменных:

- `ptr` — глобальная переменная, записанная основным потоком и считанная одноранговыми потоками;
- `cmt` — статическая переменная с одним экземпляром в памяти, который считывается и записывается двумя одноранговыми потоками;

- `i.m` — локальная автоматическая переменная, сохраненная в стеке основного потока. Даже несмотря на то, что ее значение передается одноранговым потокам, они никогда не обращаются к ней в стеке и, следовательно, она не является совместно используемой;
- `msgs.m` — локальная автоматическая переменная, сохраненная в стеке основного потока, обращение к которой осуществляется косвенно через `ptr` обоими одноранговыми потоками;
- `myid.0` и `myid.1` — экземпляры локальной автоматической переменной, расположенные в стеках одноранговых потоков 0 и 1 соответственно.

Экземпляр переменной	Обращение основного потока?	Обращение однорангового потока 0?	Обращение однорангового потока 1?
<code>ptr</code>	Да	Да	Да
<code>cnt</code>	Нет	Да	Да
<code>i.m</code>	Да	Нет	Нет
<code>msgs.m</code>	Да	Да	Да
<code>myid.p0</code>	Нет	Да	Нет
<code>myid.p1</code>	Нет	Нет	Да

2. К переменным `ptr`, `cnt` и `msgs` обращается более одного потока, следовательно, они являются совместно используемыми.

РЕШЕНИЕ УПРАЖНЕНИЯ 13.7

Важная идея заключается в том, что нельзя делать никаких предположений относительно того, какое упорядочивание выбирает ядро при планировании потоков.

Шаг	Поток	Команда	%eax ₁	%eax ₂	ctr
1	1	H_1	—	—	0
2	1	L_1	0	—	0
3	2	H_2	—	—	0
4	2	L_2	—	0	1
5	2	U_2	—	1	1
6	2	S_2	—	1	1
7	1	U_1	1	—	1
8	1	S_1	1	—	1

(окончание)

Шаг	Поток	Команда	%eax ₁	%eax ₂	ctr
9	1	T_1	1	-	1
10	2	T_2	1	-	1

Переменная `cnt` имеет окончательное некорректное значение, равное 1.

РЕШЕНИЕ УПРАЖНЕНИЯ 13.8

Функция `gethostbyname` не является реинтегрируемой, потому что каждый вызов совместно использует одно и то же переменную `static`, возвращенную функцией `gethostbyname`. Однако она является безопасной по потокам, потому что доступ к совместно используемой переменной защищен операциями P и V , следовательно, они взаимоисключающие.

РЕШЕНИЕ УПРАЖНЕНИЯ 13.9

Если освободить блок сразу после обращения к `pthread_create` в строке 15, тогда будет представлена новая гонка, на этот раз между вызовом `free` в основном потоке и оператором присваивания в строке 25 рутинной процедуры потоков.

РЕШЕНИЕ УПРАЖНЕНИЯ 13.10

1. Другим подходом является передача целого числа `i` напрямую, вместо передачи указателя в `i`:

```
for (i = 0; i < N; i++)
    Pthread_create (&tid [i], NULL, thread, (void *) i);
```

В рутинной процедуре потоков аргумент отбрасывается назад в `int` и присваивается функции `myid`:

```
int myid = (int) vargp;
```

2. Преимуществом является то, что снижается количество непроизводительных сигналов путем устранения вызовов `malloc` и `free`. Значительным недостатком является предположение, что указатели такие же большие, как и `ints`. Тогда как данное предположение верно для всех современных систем, оно может быть ложным для будущих систем.

РЕШЕНИЕ УПРАЖНЕНИЯ 13.11

1. Граф продвижения для программы-оригинала показан на рис. 13.21.
2. Программа всегда попадает в тупиковые ситуации, поскольку любая возможная траектория, в конце концов, окажется в состоянии тупика.
3. Для устранения потенциальной тупиковой ситуации инициализируйте двоичный семафор `t` значением 1 вместо 0.
4. Граф продвижения для исправленной программы показан на рис. 13.22.

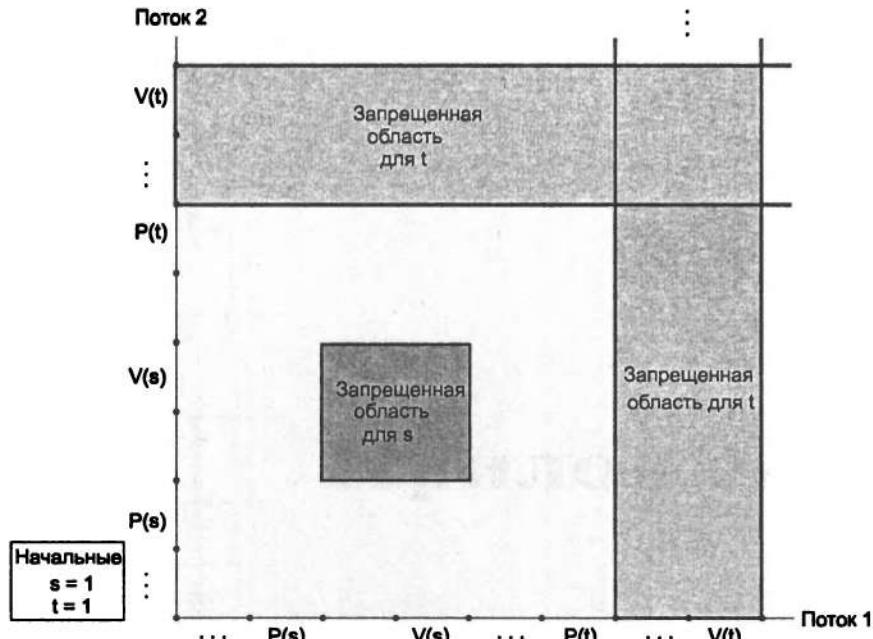


Рис. 13.21. Граф продвижения для программы-оригинала

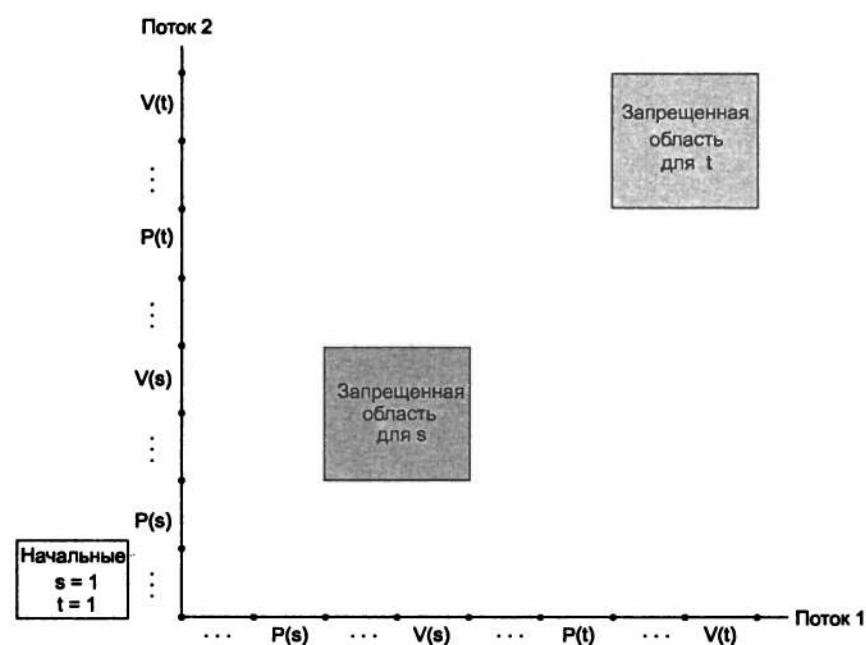
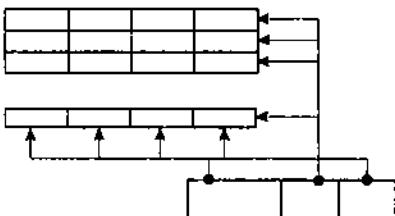


Рис. 13.22. Граф продвижения для исправленной программы, не попадающей в тупиковые ситуации

Приложения

ПРИЛОЖЕНИЕ 1



Описание управляемой логики процессоров с помощью HCL

Справочное руководство HCL

Для описания управляемой логики проектов процессоров использовался язык управления аппаратными средствами HCL (см. главу 4). HCL позволяет пользователям описывать булевые функции и операции выборки на уровне слова. С другой стороны, этому языку недостает многих особенностей HDL (Hardware Description Language, язык описания аппаратных средств), таких как способы объявления регистров и других элементов памяти, организация циклов и условных конструктивных компонентов, определение модуля и возможности реализации, а также извлечение отдельных битов и операции вставки.

В действительности, HCL — это всего лишь язык для генерирования очень стилизованной формы кода C. Все определения блоков в файле HCL преобразуются в функции C программой hcl2c (HCL для языка C). Затем эти функции компилируются и связываются с библиотечным кодом, реализующим другие функции для генерирования программы, как показано на рис. П1.1:

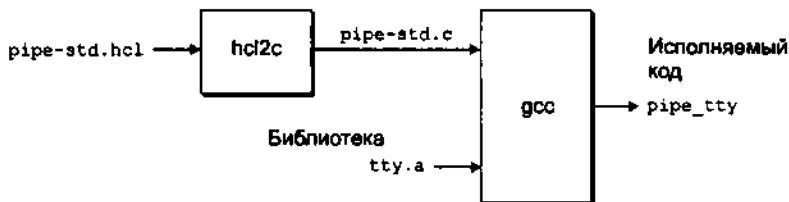


Рис. П1.1. Генерация текстовой версии программы

Поведение управляемой логики можно было бы описать непосредственно в C, не связываясь с HCL. Преимущество же HCL заключается в том, что функциональность аппаратных средств более четко отделяется от внутренних процессов моделирующие устройства или программы.

HCL поддерживает только два типа данных: `bool` представляют собой 0 или 1, а `int` эквивалентны значениям `int` в C. Тип данных `int` используется для всех типов сиг-

налов, таких как слова, идентификаторы регистров и коды команд. При преобразовании в С оба типа данных представляются как данные `int`, однако значение типа `bool` будет равно только 0 или 1.

Объявление сигналов

Выражения в HCL могут обращаться к именованным сигналам целочисленного или булева типа. Названия сигналов *name* должны начинаться с буквы, за которой следует некоторое количество букв, цифр или подчеркиваний. Названия сигналов восприимчивы к регистрам. Объявление сигнала также определяет связанное с ним выражение С. Объявление сигнала принимает одну из следующих форм:

<code>boolsig</code>	<code>name</code>	<code>'C-expr'</code>
<code>intsig</code>	<code>name</code>	<code>'C-expr'</code>

где *C-expr* может быть произвольным выражением С, за исключением того, что оно не может содержать одиночную кавычку ('') или символ новой строки (\n). При генерировании кода С HCL2C заменит любое название сигнала на соответствующее выражение С.

Текст в кавычках

Текст в кавычках обеспечивает механизм передачи текстовой информации непосредственно через HCL2C в сгенерированный файл С. Это может использоваться для вставки объявлений переменных; операторов `include` и других элементов, обычно имеющихся в файлах С. Общая форма такова:

<code>quote</code>	<code>'string'</code>
--------------------	-----------------------

где *string* может быть любой строкой, не содержащей одиночной кавычки или символа новой строки.

Выражения и блоки

Существует два типа выражений: булевые и целочисленные, которые в синтаксических описаниях называются *bool-expr* и *int-expr* соответственно. В табл. П1.1 показаны различные типы булевых выражений. Они перечислены в нисходящем приоритетном порядке, операции в рамках каждой группы имеют одинаковый приоритет. Для смены обычного приоритета оператора можно использовать круглые скобки.

На верхнем уровне расположены константные значения 0 и 1 и поименованные булевые сигналы. Далее в приоритетном порядке следуют выражения, имеющие целочисленный аргумент, но выдающие булев результат. Проверка на принадлежность множеству сравнивает целочисленные выражения, заключающие в себе множество $\{int\text{-}expr_1, \dots, int\text{-}expr_k\}$, дающее в результате 1, если найдено любое согласованное значение. Операторы отношения сравнивают два целочисленных выражения и генерируют 1 в случае совпадения выражений и 0 — в противном случае.

Таблица П1.1. Булевые выражения HCL

Синтаксис	Значение
0	Логическое значение 0
1	Логическое значение 1
название	Именованный булев сигнал
<i>int-expr</i> in { <i>int-expr</i> ₁ , <i>int-expr</i> ₂ , ..., <i>int-expr</i> _k }	Проверка на принадлежность множеству
<i>int-expr</i> ₁ == <i>int-expr</i> ₂	Проверка на равенство
<i>int-expr</i> ₁ != <i>int-expr</i> ₂	Не равная проверка
<i>int-expr</i> ₁ < <i>int-expr</i> ₂	Недостаточная проверка
<i>int-expr</i> ₁ <= <i>int-expr</i> ₂	Недостаточная или равная проверка
<i>int-expr</i> ₁ > <i>int-expr</i> ₂	Достаточная проверка
<i>int-expr</i> ₁ >= <i>int-expr</i> ₂	Достаточная или равная проверка
! <i>bool-expr</i>	NOT
<i>bool-expr</i> ₁ && <i>bool-expr</i> ₂	AND
<i>bool-expr</i> ₁ <i>bool-expr</i> ₂	OR

Существует всего три типа целочисленных выражений: числа, именованные целочисленные сигналы и выражения, осуществляющие выбор. Числа записываются в десятичном выражении и могут быть отрицательными. Именованные целочисленные сигналы используют правила именования, описанные ранее. Осуществляющие выбор выражения имеют следующую общую форму:

```
[  
    bool-expr1           int-expr1  
    bool-expr2           int-expr2  
  
    ...  
  
    bool-exprk           int-exprk  
]
```

Данное выражение содержит серию блоков, где каждый блок *i* состоит из булева выражения *bool-expr*_{*i*}, указывающего на то, должен ли быть выбран этот блок, и целочисленного выражения *int-expr*_{*i*}, указывающего на значение для этого блока. При оценке выбора выражения булевые выражения концептуально сравниваются в последовательности. Когда одно из них дает значение 1, значение соответствующего целочисленного выражения возвращается как значение выражения выбора. Если ни одно

булево выражение не имеет значения 1, тогда значения выражения выбора равно 0. Одним из хороших принципов программирования является значение 1 последнего булева выражения, что будет гарантировать минимум один совпадающий блок.

Выражения HCL используются для определения поведения блока управляющей логики. Определение блока имеет одну из следующих форм:

```
bool      name = bool-expr;
int       name = int-expr;
```

где первая форма определяет булев блок, а вторая — блок на уровне слова. Для блока, объявленного с именем *name*, HCL2C генерирует функцию *gen_name*. Эта функция не имеет аргументов и возвращает результат типа *int*.

Пример HCL

Следующий пример демонстрирует полный файл HCL. Его можно скомпилировать и выполнить, используя аргументы командной строки для сигналов ввода. Более типично, что файлы HCL определяют только управляющую часть имитационной модели. Затем сгенерированный код С компилируется и связывается с другим кодом для формирования программы. Этот пример приводится только для наглядной демонстрации HCL. Цикл основан на цикле MUX4, описанном в разд. 4.2.4, со следующей структурой (рис. П1.2), представленной листингом П1.1.

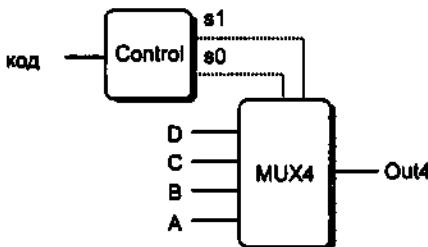


Рис. П1.2. Структура примера программы

Листинг П1.1. Пример файла

```

1      ## Простой пример файла HCL.
2      ## Его можно преобразовать в С, используя hcl2c, после чего
3      ## скомпилировать .
4
5      ## В этом примере будет сгенерирован цикл MUX4, показанный в
6      ## разделе \obey{<sect_arch_hclset>}. Он состоит из блока управления,
7      ## генерирующего сигналы на уровне битов s1 и s0 из кода сигнала
8      ## входа, после чего использует эти сигналы для управления 4-сторонним
9      ## мультиплексором с данными ввода A, B, C и D.
  
```

```
10      ## Данный код встроен в программу С, считающую
11      ## значения кода A, B, C и D из командной строки,
12      ## и распечатывающую вывод цикла
13
14      ## Информация, дословно вставленная в файл С
15      quote '#include <stdio.h>'
16      quote '#include <stdlib.h>'
17      quote 'int code_val, s0_val, s1_val;'
18      quote 'char **data_names;'
19
20      ## Объявления сигналов, использованных в HCL-описании и
21      ## соответствующие выражения С.
22      boolsig s0 's0_val'
23      boolsig s1 's1_val'
24      intsig code 'code_val'
25      intsig A 'atoi(data_names[0])'
26      intsig B 'atoi(data_names[1])'
27      intsig C 'atoi(data_names[2])'
28      intsig D 'atoi(data_names[3])'
29
30      ## HCL-описания логических блоков
31      bool s1 = code in { 2, 3 };
32
33      bool s0 = code in { 1, 3 };
34
35      int Out4 = [
36          !s1 && !s0 : A; # 00
37          !s1 : B; # 01
38          s1 && !s0 : C; # 10
39          1 : D; # 11
40      ];
41
42      ## Больше информации вставлено дословно в код С для
43      ## вычисления значений и распечатки данных вывода
44      quote 'int main(int argc, char *argv[]) {' 
45      quote '    data_names = argv+2; '
46      quote '    code_val = atoi(argv[1]); '
47      quote '    s1_val = gen_s1(); '
48      quote '    s0_val = gen_s0(); '
49      quote '    printf("Out = %d\n", gen_Out4()); '
50      quote '    return 0; '
51      quote '}'
```

Это определяет то, что булевые сигналы `s0` и `s1` и целочисленный сигнал `code` будут являться образами ссылок на глобальные переменные `s0_val`, `s1_val` и `code_val`.

Объявлены целочисленные сигналы A, B, C и D, где соответствующие выражения С применяют стандартную библиотечную функцию atoi к строкам, передаваемыми как аргументы командной строки.

Определение блока с именем s1 генерирует следующий код С:

```
int gen_s1 ()
{
    return ((code_val) == 2 || (code_val) == 3);
}
```

Здесь видно, что проверка принадлежности множеству реализуется как серия сравнений, и каждое обращение к сигналу code заменяется на выражение code_val.

Обратите внимание, что не существует прямого отношения между сигналом s1, объявленным в строке 23 файла HCL, и блоком с именем s1, объявленным в строке 31. Один является образом выражения С, тогда как другой генерирует функцию под называнием gen_s1.

Текст в кавычках в конце генерирует следующую основную функцию:

```
int main (int argc, char *argv[ ]) {
    data_names = argv+2;
    code_val = atoi (argv [1]);
    s1_val = gen_s1 ();
    s0_val = gen_s0 ();
    printf "Out = %d\n", gen_Out4 ());
    return 0;
}
```

Основная функция вызывает функции gen_s1 и gen_Out4, сгенерированные из определений блоков. Также видно, как код С должен определять последовательность оценок блоков и установку значений, использованных в выражениях С, представляющих разные значения сигналов.

Описание SEQ

```
1 ######
2 # HCL-описание управления одного цикла Y86 процессора SEQ #
3 # Copyright (C) Randal E. Bryant, David R. O'Hallaron, 2002 #
4 #####
5 #####
6 #####
7 # C Include's. Не изменяется #
8 #####
9 #####
10 quote '#include <stdio.h>'
11 quote '#include "isa.h"'
```

```

12  quote '#include "sim.h"'
13  quote 'int sim_main(int argc, char *argv[]){'#
14  quote 'int gen_pc(){return 0;}'#
15  quote 'int main(int argc, char *argv[])'#
16  quote '  (plusmode=0;return sim_main(argc,argv);)'#
17
18  ##### Объявления. Не изменять /перемещать/удалять #####
19
20
21
22  ##### Символическое представление командных кодов Y86 #####
23  intsig INOP          'I_NOP'
24  intsig IHALT         'I_HALT'
25  intsig IRRMOVL       'I_RRMOVL'
26  intsig IIRMOVL       'I_IRMOVL'
27  intsig IRMMOVL       'I_RMMOVL'
28  intsig IMRMOVL       'I_MRMOVL'
29  intsig IOPL           'I_ALU'
30  intsig IJXX           'I_JMP'
31  intsig ICALL          'I_CALL'
32  intsig IRET           'I_RET'
33  intsig IPUSHL         'I_PUSHL'
34  intsig IPOPL          'I_POPL'
35
36  ##### Символическое представление регистров Y86, на которые делаются явные ссылки #####
37  intsig RESP           'REG_ESP'          # Указатель стека
38  intsig RNONE          'REG_NONE'        # Особое значение, указывающее на отсутствие регистра
39
40  ##### Функции ALU, на которые делаются явные ссылки #####
41  intsig ALUADD          'A_ADD'           # ALU должна добавить свои аргументы
42
43  ##### Сигналы, к которым может обращаться управляющая логика #####
44
45  ##### Данные ввода этапа выборки #####
46  intsig pc 'pc'          # Счетчик команд
47  ##### Вычисления этапа выборки #####
48  intsig icode            'icode'          # Код управления командами
49  intsig ifun             'ifun'           # Функция команд
50  intsig rA               'ra'              # Поле rA из команды
51  intsig rB               'rb'              # Поле rB из команды
52  intsig valC             'valc'           # Константа из команды
53  intsig valP             'valp'           # Адрес следующей команды
54

```

```
55 ##### Вычисления этапа декодировки #####
56 intsig valA          'vala'           # Значение из порта регистра А
57 intsig valB          'valb'           # Значение из порта регистра В
58
59 ##### Вычисления этапа выполнения #####
60 intsig vale           'vale'            # Значение, вычисленное АЛУ
61 boolsig Bch           'bcond'          # Тестирование ветви
62
63 ##### Вычисления этапа памяти #####
64 intsig valM           'valm'            # Значение, считанное из памяти
65
66
67 ##### Определения управляющего сигнала #####
68 # Определения управляющего сигнала #
69
70
71 ##### Этап выборки #####
72
73 # Требует ли выбранная команда байт regid?
74 bool need_regids =
75     icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,
76                 TIRMOVL, IRMMOVL, IMRMOVL };
77
78 # Требует ли выбранная команда константного слова?
79 bool need_valC =
80     icode in { IIRMOVL, IRMMOVL, IMRMOVL, IJXX, ICALL };
81
82 bool instr_valid = icode in
83     { INOP, IHALT, IRRMOVL, IIRMOVL, IRMMOVL, IMRMOVL,
84       IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL };
85
86 ##### Этап декодирования #####
87
88 ## Какой регистр должен быть использован в качестве источника А?
89 int srcA =
90     icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL } : rA;
91     icode in { IPOPL, IRET } : RESP;
92     1 : RNONE; # Регистр не нужен
93 ];
94
95 ## Какой регистр должен быть использован в качестве источника В?
96 int srcB =
97     icode in { IOPL, IRMMOVL, IMRMOVL } : rB;
98     icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
99     1 : RNONE; # Регистр не нужен
100];
101
```

```
102    ## Какой регистр должен быть использован в качестве пункта назначения E?
103    int dstE = [
104        icode in { IRRMOVL, IIRMOVL, IOPL } : rB;
105        icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
106        1 : RNONE; # Регистр не нужен
107    ];
108
109    ## Какой регистр должен быть использован в качестве пункта назначения M?
110    int dstM = [
111        icode in { IMRMOVL, IPOPL } : rA;
112        1 : RNONE; # Регистр не нужен
113    ];
114
115    ##### Этап выполнения #####
116
117    ## Выбор данных ввода A на АЛУ
118    int aluA = [
119        icode in { IRRMOVL, IOPL } : valA;
120        icode in { IIRMOVL, IRMMOVL, IMRMOVL } : valC;
121        icode in { ICALL, IPUSHL } : -4;
122        icode in { IRET, IPOPL } : 4;
123        # Другие команды не требуют АЛУ
124    ];
125
126    ## Выбор данных ввода B на АЛУ
127    int aluB = [
128        icode in { IRMMOVL, IMRMOVL, IOPL, ICALL,
129                    IPUSHL, IRET, IPOPL } : valB;
130        icode in { IRRMOVL, IIRMOVL } : 0;
131        # Другие команды не требуют АЛУ
132    ];
133
134    ## Установка функции АЛУ
135    int alufun = [
136        icode == IOPL : ifun;
137        1 : ALUADD;
138    ];
139
140    # Нужно ли обновлять коды условий?
141    1 set_cc = icode in { IOPL };
142
143    ##### Этап памяти #####
144
145    Установка сигнала управления считыванием
146    1 mem_read = icode in { IMRMOVL, IPOPL, IRET };
147
```

```

148 Установка сигнала управления записью
149 1 mem_write = icode in { IRMMOVL, IPUSHL, ICALL };
150
151 Выбор адреса памяти
152 mem_addr = [
153     icode in { IRMMOVL, IPUSHL, ICALL, IMRMOVL } : valE;
154     icode in { IPOPL, IRET } : valA;
155     # Другие команды не требуют адреса
156 ];
157
158 ## Выбор данных ввода памяти
159     int mem_data = [
160         # Значение из регистра
161         icode in { IRMMOVL, IPUSHL } : valA;
162         # Возврат PC
163         icode == ICALL : valP;
164         # По умолчанию: Ничего не записывается
165 ];
166
167 ##### Обновление счетчика команд #####
168
169 ## На какой адрес должна быть выбрана команда
170
171 int new_pc = [
172     # Вызов. Использование константы команды
173     icode == ICALL : valC;
174     # Выбранная ветвь. Использование константы команды
175     icode == IJXX && Bch : valC;
176     # Завершение выполнения команды RET.
177     # Использование значения из стека
178     icode == IRET : valM;
179     # По умолчанию: Использование приращенного счетчика команд
180     1 : valP;
181 ];

```

Описание SEQ+

```

1 #####
2 # HCL-описание управления одного цикла Y86 процессора SEQ + #
3 # Copyright (C) Randal E. Bryant, David R. O'Hallaron, 2002 #
4 #####
5 #####
6 # C Include's. Не изменяется #
7 #####
8 #####
9 #####

```

```
10  quote '#include <stdio.h>'
11  quote '#include "isa.h"'
12  quote '#include "sim.h"'
13  quote 'int sim_main(int argc, char *argv[])'
14  quote 'int gen_new_pc(){return 0;}''
15  quote 'int main(int argc, char *argv[])'
16  quote '  (plusmode=1;return sim_main(argc,argv);)'
17
18  ##### Объявления. Не изменять/перемещать #####
19
20
21
22 ##### Символическое представление командных кодов Y86 #####
23 intsig INOP          'I_NOP'
24 intsig IHALT         'I_HALT'
25 intsig IRRMOVL       'I_RRMOVL'
26 intsig IIRMOVL       'I_IRMOVL'
27 intsig IRMMOVL       'I_RMMOVL'
28 intsig IMRMOVL       'I_MRMOVL'
29 intsig IOPL           'I_ALU'
30 intsig IJXX           'I_JMP'
31 intsig ICALL          'I_CALL'
32 intsig IRET           'I_RET'
33 intsig IPUSHL         'I_PUSHL'
34 intsig IPOPL          'I_POPL'
35
36 ##### Символическое представление регистров Y86, на которые делаются явные ссылки #####
37 intsig RESP           'REG_ESP'      # Указатель стека
38 intsig RNONE          'REG_NONE'    # Особое значение, указывающее на отсутствие регистра
39
40 ##### Функции АЛУ, на которые делаются явные ссылки #####
41 intsig ALUADD          'A_ADD'        # АЛУ должно добавить свои аргументы
42
43 ##### Сигналы, к которым может обращаться управляющая логика #####
44
45 ##### Данные ввода этапа счетчика команд #####
46
47 ## Все эти значения основаны на значениях из предыдущей команды
48 intsig     pIcode   'prev_icode'    # Код управления командами
49 intsig     pValC    'prev_valc'    # Константа из команды
50 intsig     pValM    'prev_valm'    # Значение, считанное из памяти
51 intsig     pValP    'prev_valp'    # Приращенный счетчик команд
52 boolsig   pBch     'prev_bcond'   # Флажок выбранной ветви
53
```

```
54 ##### Вычисления этапа выборки #####
55 intsig icode  'icode'          # Код управления командами
56 intsig ifun   'ifun'           # Функция команд
57 intsig rA     'ra'              # Поле rA из команды
58 intsig rB     'rb'              # Поле rB из команды
59 intsig valC   'valc'            # Константа из команды
60 intsig valP   'valp'            # Адрес следующей команды
61
62 ##### Вычисления этапа декодировки #####
63 intsig valA   'vala'            # Значение из порта регистра A
64 intsig valB   'valb'            # Значение из порта регистра B
65
66 ##### Вычисления этапа выполнения #####
67 intsig valE   'vale'             # Значение, вычисленное ALU
68 boolsig Bch    'bcond'           # Тестирование ветви
69
70 ##### Вычисления этапа памяти #####
71 intsig valM   'valm'             # Значение, считанное из памяти
72
73
74 ##### Определения управляющего сигнала #####
75 # Определения управляющего сигнала #
76
77
78 ##### Вычисления счетчика команд #####
79
80 # Вычислить место выборки для данной команды, на основании результатов
81 # из предыдущей команды
82
83 int pc = [
84     # Вызов. Использование константы команды
85     pIcode == ICALL : pValC;
86     # Выбранная ветвь. Использование константы команды
87     pIcode == IJXX && pBch : pValC;
88     # Завершение выполнения команды RET. Использование значения
89     # из стека
90     pIcode == IRET : pValM;
91     # По умолчанию: Использование приращенного счетчика команд
92     1 : pValP;
93 ];
94
95 ##### Этап выборки #####
96
97 # Требует ли выбранная команда байт regid?
98 bool need_regids =
99     icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,
100                 IIRMOVL, IRMMOVL, IMRMOVL };
```

```
100
101     # Требует ли выбранная команда константного слова?
102     bool need_valC =
103         icode in { IIRMOVL, IRMMOVL, IMRMOVL, IJXX, ICALL };
104
105     bool instr_valid = icode in
106         { INOP, IHALT, IIRMOVL, IIRMOVL, IRMMOVL,
107             IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL };
108
109 ##### Этап декодировки #####
110
111 ## Какой регистр должен быть использован в качестве источника A?
112 int srcA =
113     icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL } : rA;
114     icode in { IPOPL, IRET } : RESP;
115     1 : RNONE; # Регистр не нужен
116 ];
117
118 ## Какой регистр должен быть использован в качестве источника B?
119 int srcB =
120     icode in { IOPL, IRMMOVL, IMRMOVL } : rB;
121     icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
122     1 : RNONE; # Регистр не нужен
123 ];
124
125 ## Какой регистр должен быть использован в качестве пункта назначения E?
126 int dstE =
127     icode in { IRRMOVL, IIRMOVL, IOPL } : rB;
128     icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
129     1 : RNONE; # Регистр не нужен
130 ];
131
132 ## Какой регистр должен быть использован в качестве пункта назначения M?
133 int dstM =
134     icode in { IMRMOVL, IPOPL } : rA;
135     1 : RNONE; # Регистр не нужен
136 ];
137
138##### Этап выполнения #####
139
140 ## Выбор данных ввода A на АЛУ
141 int aluA =
142     icode in { IRRMOVL, IOPL } : valA;
143     icode in { IIRMOVL, IRMMOVL, IMRMOVL } : valC;
144     icode in { ICALL, IPUSHL } : -4;
145     icode in { IRET, IPOPL } : 4;
```

```
146          # Другие команды не требуют АЛУ
147      ];
148
149      ## Выбор данных ввода В на АЛУ
150      int aluB = [
151          icode in { IRMMOVL, IMRMOVL, IOPL, ICALL,
152                      IPUSHL, IRET, IPOPL } : valB;
153          icode in { IRRMOVL, IIRMOVL } : 0;
154          # Другие команды не требуют АЛУ
155      ];
156
157      ## Установка функции АЛУ
158      int alufun = [
159          icode == IOPL : ifun;
160          1 : ALUADD;
161      ];
162
163      ## Нужно ли обновлять коды условий?
164      bool set_cc = icode in { IOPL };
165
166      ##### Этап памяти #####
167
168      ## Установка сигнала управления считыванием
169      bool mem_read = icode in { IMRMOVL, IPOPL, IRET };
170
171      ## Установка сигнала управления записью
172      bool mem_write = icode in { IRMMOVL, IPUSHL, ICALL };
173
174      ## Выбор адреса памяти
175      int mem_addr = [
176          icode in { IRMMOVL, IPUSHL, ICALL, IMRMOVL } : valE;
177          icode in { IPOPL, IRET } : valA;
178          # Другие команды не требуют адреса
179      ];
180
181      ## Выбор данных ввода памяти
182      int mem_data = [
183          # Значение из регистра
184          icode in { IRMMOVL, IPUSHL } : valA;
185          # Возврат PC
186          icode == ICALL : valP;
187          # По умолчанию: Ничего не записывается
188      ];
```

Конвейер

```

1 ##### HCL-описание управления одного цикла конвейерного процессора Y86 #
2 # Copyright (C) Randal E. Bryant, David R. O'Hallaron, 2002 #
3 #####
4 #####
5 #####
6 # С Include's. Не изменяется #
7 #####
8 #####
9 #####
10 quote '#include <stdio.h>'
11 quote '#include "isa.h"'
12 quote '#include "pipeline.h"'
13 quote '#include "stages.h"'
14 quote '#include "sim.h"'
15 quote 'int sim_main(int argc, char *argv[]);'
16 quote 'int main(int argc, char *argv[]){return sim_main(argc,argv);}'
17 #####
18 # Объявления. Не изменять /перемещать/удалять #
19 #####
20 #####
21 #####
22 ##### Символическое представление командных кодов Y86 #####
23 intsig INOP          'I_NOP'
24 intsig IHALT         'I_HALT'
25 intsig IRRMOVL      'I_RRMOVL'
26 intsig IIRMOVL      'I_IRMOVL'
27 intsig IRMMOVL      'I_RMMOVL'
28 intsig IMRMOVL      'I_MRMOVL'
29 intsig IOPL          'I_ALU'
30 intsig IJXX          'I_JMP'
31 intsig ICALL         'I_CALL'
32 intsig IRET          'I_RET'
33 intsig IPUSHL        'I_PUSHL'
34 intsig IPOPL          'I_POPL'
35 #####
36 ##### Символическое представление регистров Y86, на которые делаются явные ссылки #####
37 intsig RESP          'REG_ESP'    # Указатель стека
38 intsig RNONE         'REG_NONE'   # Особое значение, указывающее на отсутствие регистра
39 #####
40 ##### Функции ALU, на которые делаются явные ссылки #####
41 intsig ALUADD        'A_ADD'      # ALU должно добавить свои аргументы

```

```

42
43      ##### Сигналы, к которым может обращаться управляющая логика #####
44
45      ##### Конвейерный регистр F #####
46
47  intsig F_predPC 'pc_curr->pc'          # Спрогнозированное значение PC
48
49      ##### Промежуточные значения на этапе выборки #####
50
51  intsig f_icode   'if_id_next->icode'    # Код выбранной команды
52  intsig f_ifun    'if_id_next->ifun'     # Функция выбранной команды
53  intsig f_valC    'if_id_next->valc'     # Константные данные выбранной
54                                # команды
55  intsig f_valP    'if_id_next->valp'     # Адрес следующей команды
56
57      ##### Конвейерный регистр D #####
58  intsig D_icode   'if_id_curr->icode'    # Код команды
59  intsig D_rA      'if_id_curr->ra'       # Поле rA из команды
60  intsig D_rB      'if_id_curr->rb'       # Поле rB из команды
61  intsig D_valP   'if_id_curr->valp'     # Приращенный счетчик команд
62
63      ##### Промежуточные значения на этапе декодировки #####
64
65  intsig d_srcA   'id_ex_next->srca'    # srcA из декодированной
66                                # команды
67  intsig d_srcB   'id_ex_next->srcb'    # srcB из декодированной команды
68  intsig d_rvalA  'd_regvala'           # Считывание valA
69                                # из регистрового файла
70  intsig d_rvalB  'd_regvalb'           # Считывание valB
71                                # из регистрового файла
72
73      ##### Конвейерный регистр E #####
74  intsig E_icode   'id_ex_curr->icode'    # Код команды
75  intsig E_ifun    'id_ex_curr->ifun'     # Функция команды
76  intsig E_valC    'id_ex_curr->valc'     # Константные данные
77  intsig E_srcA   'id_ex_curr->srca'    # Идентификатор регистра
78                                # источника A
79  intsig E_valA   'id_ex_curr->vala'     # Значение источника A
80  intsig E_srcB   'id_ex_curr->srcb'    # Идентификатор регистра
81                                # источника B
82  intsig E_valB   'id_ex_curr->valb'     # Значение источника B
83  intsig E_dstE   'id_ex_curr->deste'    # Идентификатор регистра места
84                                # назначения E
85  intsig E_dstM   'id_ex_curr->destm'    # Идентификатор регистра места
86                                # назначения M

```

```

80 ##### Промежуточные значения на этапе выполнения #####
81 intsig e_valE 'ex_mem_next->vale'      # valE, сгенерированное АЛУ
82 boolsig e_Bch 'ex_mem_next->takebranch' # Все уже готово для
83                                ветвления?

84 ##### Конвейерный регистр M #####
85 intsig M_icode 'ex_mem_curr->icode'    # Код команды
86 intsig M_ifun 'ex_mem_curr->ifun'       # Функция команды
87 intsig M_valA 'ex_mem_curr->vala'       # Значение источника A
88 intsig M_dstE 'ex_mem_curr->deste'     # Идентификатор регистра места
89                                назначения E
90                                # Значение АЛУ E
91 intsig M_valM 'ex_mem_curr->destm'     # Идентификатор регистра места
92                                назначения M
93                                # Флагок выбранной ветви

94 ##### Промежуточные значения на этапе памяти #####
95 intsig m_valM 'mem_wb_next->valm'      # valM, сгенерированный памятью

96 ##### Конвейерный регистр W #####
97 intsig W_icode 'mem_wb_curr->icode'    # Код команды
98 intsig W_dstE 'mem_wb_curr->deste'     # Идентификатор регистра места
99                                назначения E
100                                # Значение АЛУ E
101 intsig W_dstM 'mem_wb_curr->destm'     # Идентификатор регистра места
102                                назначения M
103                                # Значение памяти M

104 # Определения управляющего сигнала #
105 #####
106 #####
107 ##### Этап выборки #####
108 #####
109 ## В какой адрес должна быть выбрана команда?
110 int f_pc = [
111     # Неправильно спрогнозированная ветвь. Выборка на приращенный
112     # счетчик команд
113     M_icode == IJXX && !M_Bch : M_valA;
114     # Завершение выполнения команды RET.
115     W_icode == IRET : W_valM;
116     # По умолчанию: использование приращенного значения счетчика
117     # команд
118     1 : F_predPC;
119 ];

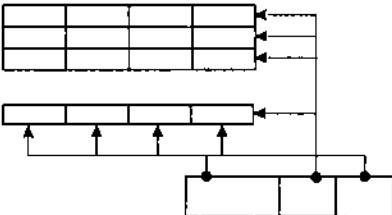
```

```
119 # Требует ли выбранная команда байт regid?
120 bool need_regids =
121         f_icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,
122                         IIRMOVL, IRMMOVL, IMRMOVL };
123
124 # Требует ли выбранная команда константного слова?
125 bool need_valC =
126         f_icode in { IIRMOVL, IRMMOVL, IMRMOVL, IJXX, ICALL };
127
128 bool instr_valid = f_icode in
129         { INOP, IHALT, IRRMOVL, IIRMOVL, IRMMOVL, IMRMOVL,
130             IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL };
131
132 # Предсказать следующее значение счетчика команд
133 int new_F_predPC = [
134     f_icode in { IJXX, ICALL } : f_valC;
135     1 : f_valP;
136 ];
137
138
139 ##### Этап декодировки #####
140
141
142 ## Какой регистр должен быть использован в качестве источника A?
143 int new_E_srcA = [
144     D_icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL } : D_rA;
145     D_icode in { IPOPL, IRET } : RESP;
146     1 : RNONE; # Регистр не нужен
147 ];
148
149 ## Какой регистр должен быть использован в качестве источника B?
150 int new_E_srcB = [
151     D_icode in { IOPL, IRMMOVL, IMRMOVL } : D_rB;
152     D_icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
153     1 : RNONE; # Регистр не нужен
154 ];
155
156 ## Какой регистр должен быть использован в качестве пункта назначения E?
157 int new_E_dstE = [
158     D_icode in { IRRMOVL, IIRMOVL, IOPL } : D_rB;
159     D_icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
160     1 : RNONE; # Регистр не нужен
161 ];
162
```

```
163    ## Какой регистр должен быть использован в качестве пункта назначения M?
164    int new_E_dstM = [
165        D_icode in { IMRMOVL, IPOPL } : D_rA;
166        1 : RNONE; # Регистр не нужен
167    ];
168
169    ## Каким должно быть значение A?
170    ## Передача на этап декодировки для valA
171    int new_E_valA = [
172        D_icode in { ICALL, IJXX } : D_valP; # Использование
173                                         приращенного счетчика команд
174        d_srcA == E_dstE : e_valE;      # Передача valE из этапа памяти
175        d_srcA == M_dstM : m_valM;    # Передача valM из памяти
176        d_srcA == M_dstE : M_valE;    # Передача valE из памяти
177        d_srcA == W_dstM : W_valM;    # Передача valM из обратной
                                         записи
178        d_srcA == W_dstE : W_valE;    # Передача valE из обратной
                                         записи
179        1 : d_rvalA;   # Использование значения считывания
180                                         из регистрового файла
181    ];
182
183    int new_E_valB = [
184        d_srcB == E_dstE : e_valE;      # Передача valE из этапа
                                         выполнения
185        d_srcB == M_dstM : m_valM;    # Передача valM из этапа памяти
186        d_srcB == M_dstE : M_valE;    # Передача valE из этапа памяти
187        d_srcB == W_dstM : W_valM;    # Передача valM из обратной
                                         записи
188        d_srcB == W_dstE : W_valE;    # Передача valE из обратной
                                         записи
189        1 : d_rvalB;   # Использование значения считывания
190                                         из регистрового файла
191
192    ##### Этап выполнения #####
193
194    ## Выбор данных ввода A на АЛУ
195    int aluA = [
196        E_icode in { IRRMOVL, IOPL } : E_valA;
197        E_icode in { IIRMOVL, IRMMOVL, IMRMOVL } : E_valC;
198        E_icode in { ICALL, IPUSHL } : -4;
199        E_icode in { IRET, IPOPL } : 4;
200        # Другие команды не требуют АЛУ
201    ];
202
```

```
201 ## Выбор данных ввода В на АЛУ
202 int aluB = [
203     E_icode in { IRMMOVL, IMRMOVL, IOPL, ICALL,
204                 IPUSHL, IRET, IPOPL } : E_valB,
205     E_icode in { IRRMOVL, IIIRMOVL } : 0;
206     # Другие команды не требуют АЛУ
207 ];
208
209 ## Установка функции АЛУ
210 int alufun = [
211     E_icode == IOPL : E_ifun,
212     1 : ALUADD;
213 ];
214
215 ## Нужно ли обновлять коды условий?
216 bool set_cc = E_icode == IOPL;
217
218
219 ##### Этап памяти #####
220
221 ## Выбор адреса памяти
222 int mem_addr = [
223     M_icode in { IRMMOVL, IPUSHL, ICALL, IMRMOVL } : M_valE;
224         M_icode in { IPOPL, IRET } : M_valA;
225         # Другие команды не требуют адреса
226 ];
227
228 ## Установка управления сигналом считывания
229 bool mem_read = M_icode in { IMRMOVL, IPOPL, IRET };
230
231 ## Установка управления сигналом записи
232 bool mem_write = M_icode in { IRMMOVL, IPUSHL, ICALL };
233
234
235 ##### Управление конвейерным регистром #####
236
237 # Остановить или вставить кружок в конвейерный регистр F?
238 # Максимум один из них будет верным
239 bool F_bubble = 0;
240 bool F_stall =
241     # Условия риска загрузки/использования
242     E_icode in { IMRMOVL, IPOPL } &&
243     E_dstM in { d_srcA, d_srcB } ||
244     # Останов на этапе выборки во время передачи ret через конвейер
245     IRET in { D_icode, E_icode, M_icode };
```

```
247 # Остановить или вставить кружок в конвейерный регистр D?
248 # Максимум один из них будет верным
249 bool D_stall =
250             # Условия риска загрузки/использования
251             E_icode in { IMRMOVL, IPOPL } &&
252             E_dstM in { d_srcA, d_srcB };
253
254 bool D_bubble =
255             # Mispredicted branch
256             (E_icode == IJXX && !e_Bch) ||
257             # Останов на этапе выборки во время передачи ret через
258             # конвейер
259             IRET in { D_icode, E_icode, M_icode };
260
261 # Остановить или вставить кружок в конвейерный регистр E?
262 # Максимум один из них будет верным
263 bool E_stall = 0;
264 bool E_bubble =
265             # Неправильно спрогнозированная ветвь
266             (E_icode == IJXX && !e_Bch) ||
267             # Условия риска загрузки/использования
268             E_icode in { IMRMOVL, IPOPL } &&
269             E_dstM in { d_srcA, d_srcB };
270
271 # Остановить или вставить кружок в конвейерный регистр M?
272 # Максимум один из них будет верным
273 bool M_stall = 0;
274 bool M_bubble = 0;
```



ПРИЛОЖЕНИЕ 2

Обработка ошибок

Программисты всегда должны проверять коды ошибок, возвращенные функциями на системном уровне. Смысл имеет использовать только информацию о состоянии, передаваемую программисту ядром. К сожалению, разработчики часто пренебрегают проверкой программ на предмет наличия ошибок, потому что это загромождает код, превращая, например, одну строку кода в многострочный условный оператор. Проверка ошибок также вносит в программу определенного рода путаницу, потому что разные функции по-разному указывают на ошибки.

При написании книги авторы сталкивались с подобными проблемами. С одной стороны, хотелось бы, чтобы примеры кодов были краткими и простыми для прочтения, а с другой — не стоит насаждать ложного мнения о том, что проверять программу на наличие ошибок не нужно вообще. Для решения связанных с этим задач на вооружение был принят принцип, основанный на интерфейсных программах обработки ошибок, впервые предложенных В. Ричардом Стивенсом (Richard Stevens) в его работе, посвященной сетевому программированию [81].

Идея заключается в том, что при наличии на системном уровне определенной функции `foo` определяется интерфейсная функция `Foo` с теми же аргументами. Упаковщик вызывает основную функцию и осуществляет проверку ошибок. При выявлении ошибки упаковщик выводит на печать информационное сообщение и прерывает выполнение процесса. В противном случае, он осуществляет возврат вызывающей программе. Обратите внимание на то, что при отсутствии ошибок поведение упаковщика ничем не отличается от поведения основной функции. И наоборот, если программа выполняется корректно с упаковщиками, она будет выполняться корректно, если ввести первую букву каждого упаковщика строчной и осуществить повторную компиляцию.

Упаковщики объединены в один исходный файл (`csapp.c`), скомпилированный и внедренный в каждую программу. Отдельный заголовочный файл (`csapp.h`) содержит прототипы функции для упаковщиков.

В данном приложении представлены учебные материалы по разным типам обработки ошибок в системах Unix, а также примеры различных стилей интерфейсных программ обработки ошибок. Для справок сюда же включены полные источники для файлов `csapp.h` и `csapp.c`.

Обработка ошибок в системе Unix

При вызовах функций системного уровня, с которыми предстоит столкнуться в книге, используется три разных стиля обработки ошибок: Unix, Posix и DNS.

Обработка ошибок в стиле Unix

Такие функции, как `fork` и `wait`, разработанные на заре появления систем Unix (как и некоторые старые функции Posix), перегружают возвращаемое значение функции кодами ошибок и полезными результатами. Например, когда функция `wait` Unix сталкивается с ошибкой (например, отсутствие порожденного процесса), она возвращает `-1` и устанавливает глобальную переменную `errno` в код ошибки, указывающий ее причину. Если выполнение функции `wait` успешно завершается, возвращается полезный результат, являющийся пропорционально-интегрально-дифференциальным регулированием собранного порожденного процесса. Код обработки ошибок стиля Unix, как правило, имеет следующую форму:

```

1     if ((pid = wait (NULL)) < 0) {
2         fprintf (stderr, "wait error: %s\n", strerror (errno));
3         exit (0);
4     }

```

Функция `strerror` возвращает текстовое описание конкретного значения `errno`.

Обработка ошибок в стиле Posix

Многие из новейших функций Posix, такие как `Pthread`, используют возвращаемое значение только для указания на успех (0) или на неудачу (ненулевое значение). Все полезные результаты возвращаются в аргументы функции, передаваемые по обращению. Данный подход называется *обработкой ошибок в стиле Posix*. Например, функция `pthread_create` стиля Posix указывает возвращаемым ею значением на успех или неудачу и возвращает идентификатор вновь созданного потока (полезный результат) по обращению в его первый аргумент. Код обработки ошибок стиля Posix, как правило, имеет следующую форму:

```

1 if ((restcode = pthread_create (&tid, NULL, thread, NULL)) != 0) {
2     fprintf (stderr, "pthread_create error: %s\n", strerror (restcode));
3     exit (0);
4 }

```

Обработка ошибок в стиле DNS

Функции `gethostbyname` и `gethostbyaddr`, получающие элементы записи хост-машины DNS, предлагают еще один вариант возвращения ошибок. Эти функции воз-

вращают указатель NULL при сбое (ошибке) и задают глобальную переменную h_errno. Как правило, обработка ошибок в стиле DNS имеет следующую форму:

```
1 if ((p = gethostbyname (name)) == NULL) {
2     fprintf (stderr, "gethostbyname error: %s\n:", hsterror (h_errno));
3     exit (0);
4 }
```

Резюме сообщающих об ошибках функций

На протяжении всей книги для представления различных стилей обработки ошибок используются следующие функции, сообщающие об ошибках:

```
#include "csapp.h"

void unix_error (char *msg);
void posix_error (int code, char *msg);
void dns_error (char *msg);
void app_error (char *msg);
```

По названиям функций unix_error, posix_error и dns_error видно, что данные функции сообщают об ошибках в стилях Unix, Posix и DNS, после чего прекращают выполнение. Функция app_error включена для удобства обнаружения ошибок приложений. Она просто распечатывает входные данные и прекращает работу. В листинге П2.1 показан код функций, сообщающих об ошибках.

Листинг П2.1. Обнаружение ошибок

```
1 void unix_error(char *msg) /* ошибка стиля Unix */
2 {
3     fprintf(stderr, "%s: %s\n", msg, strerror(errno));
4     exit(0);
5 }
6
7 void posix_error(int code, char *msg) /* ошибка стиля Posix */
8 {
9     fprintf(stderr, "%s: %s\n", msg, strerror(code));
10    exit(0);
11 }
12
13 void dns_error(char *msg) /* ошибка стиля DNS */
14 {
15     fprintf(stderr, "%s: DNS error %d\n", msg, h_errno);
16     exit(0);
17 }
```

```

19 void app_error(char *msg) /* ошибка приложения */
20 {
21     fprintf(stderr, "%s\n", msg);
22     exit(0);
23 }

```

Интерфейсные программы обработки ошибок

В этом разделе приведено несколько примеров различных интерфейсных программ обработки ошибок.

Программы обработки ошибок стиля Unix

В листинге П2.2 представлена функция `wait` стиля Unix. Если `wait` возвращается с ошибкой, тогда упаковщик распечатывает информационное сообщение и прекращает работу. В противном случае вызывающей программе возвращается PID.

Листинг П2.2. Упаковщик для функции `wait` стиля Unix

```

1 pid_t Wait(int *status)
2 {
3     pid_t pid;
4
5     if ((pid = wait(status)) < 0)
6         unix_error("Wait error");
7     return pid;
8 }

```

В листинге П2.3 представлена функция `kill` стиля Unix. Обратите внимание, что, в отличие от `wait`, данная функция при успешном выполнении возвращает `void`.

Листинг П2.3. Упаковщик для функции `kill` стиля Unix

```

1 void Kill(pid_t pid, int signum)
2 {
3     int rc;
4
5     if ((rc = kill(pid, signum)) < 0)
6         unix_error("Kill error");
7 }

```

Программы обработки ошибок стиля Posix

В листинге П2.4 представлена функция `pthread_detach` стиля Posix. Как большинство функций стиля Posix, она не перегружает полезные результаты кодами возврата ошибок, поэтому при успешном выполнении упаковщик возвращает `void`.

Листинг П2.4. Упаковщик для функции `pthread_detach` стиля Posix

```

1 void Pthread_detach(pthread_t tid) {
2     int rc;
3
4     if ((rc = pthread_detach(tid)) != 0)
5         posix_error(rc, "Pthread_detach error");
6 }
```

Программы обработки ошибок стиля DNS

В листинге П2.5 представлена интерфейсная программа обработки ошибок для функции `gethostbyname` стиля DNS.

Листинг П2.5. Упаковщик для функции `gethostbyname` стиля DNS

```

1 struct hostent *Gethostbyname(const char *name)
2 {
3     struct hostent *p;
4
5     if ((p = gethostbyname(name)) == NULL)
6         dns_error("Gethostbyname error");
7     return p;
8 }
```

Заголовочный файл csapp.h

```

1 #ifndef __CSAPP_H__
2 #define __CSAPP_H__
3
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <unistd.h>
7 #include <string.h>
8 #include <ctype.h>
9 #include <setjmp.h>
10 #include <signal.h>
11 #include <sys/time.h>
```

```
12 #include <sys/types.h>
13 #include <sys/wait.h>
14 #include <sys/stat.h>
15 #include <fcntl.h>
16 #include <sys/mman.h>
17 #include <errno.h>
18 #include <math.h>
19 #include <pthread.h>
20 #include <semaphore.h>
21 #include <sys/socket.h>
22 #include <netdb.h>
23 #include <netinet/in.h>
24 #include <arpa/inet.h>
25
26
27 /* Разрешение файлов по умолчанию: DEF_MODE & ~DEF_UMASK */
28 #define DEF_MODE    S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH
29 #define DEF_UMASK   S_IWGRP|S_IWOTH
30
31 /* Упрощает вызовы bind(), connect() и accept() */
32 typedef struct sockaddr SA;
33
34 /* Тупиковое состояние для устойчивого пакета ввода-вывода (RIO) */
35 #define RIO_BUFSIZE 8192
36 typedef struct {
37     int rio_fd;          /* дескриптор для данного внутреннего буфера */
38     int rio_cnt;         /* несчитанные байты во внутреннем буфере */
39     char *rio_bufptr;   /* следующий несчитанный байт */
40     char rio_buf[RIO_BUFSIZE]; /* внутренний буфер */
41 } rio_t;
42
43 /* Внешние переменные */
44 extern int h_errno;      /* определено BIND для ошибок DNS */
45 extern char **environ;   /* определено libc */
46
47 /* Различные константы */
48 #define MAXLINE 8192 /* максимальная длина текстовой строки */
49 #define MAXBUF 8192 /* максимальный размер буфера ввода-вывода */
50 #define LISTENQ 1024 /* второй аргумент на listen() */
51
52 /* Собственные функции обработки ошибок */
53 void unix_error (char *msg);
54 void posix_error (int code, char *msg);
55 void dns_error (char *msg);
56 void app_error (char *msg);
57
```

```
58 /* Упаковщики управления процессом */
59 pid_t Fork(void);
60 void Execve(const char *filename, char *const argv[], char *const envp[]);
61 pid_t Wait(int *status);
62 pid_t Waitpid(pid_t pid, int *iptr, int options);
63 void Kill(pid_t pid, int signum);
64 unsigned int Sleep(unsigned int secs);
65 void Pause(void);
66 unsigned int Alarm(unsigned int seconds);
67 void Setpgid(pid_t pid, pid_t pgid);
68 pid_t Getpgrp();
69
70 /* Упаковщики сигналов */
71 typedef void handler_t(int);
72 handler_t *Signal(int signum, handler_t *handler);
73 void Sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
74 void Sigemptyset(sigset_t *set);
75 void Sigfillset(sigset_t *set);
76 void Sigaddset(sigset_t *set, int signum);
77 void Sigdelset(sigset_t *set, int signum);
78 int Sigismember(const sigset_t *set, int signum);
79
80 /* Упаковщики ввода-вывода Unix */
81 int Open(const char *pathname, int flags, mode_t mode);
82 ssize_t Read(int fd, void *buf, size_t count);
83 ssize_t Write(int fd, const void *buf, size_t count);
84 off_t Lseek(int fildes, off_t offset, int whence);
85 void Close(int fd);
86 int Select(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
87     struct timeval *timeout);
88 int Dup2(int fd1, int fd2);
89 void Stat(const char *filename, struct stat *buf);
90 void Fstat(int fd, struct stat *buf);
91
92 /* Упаковщики отображения памяти */
93 void *Mmap(void *addr, size_t len, int prot, int flags, int fd, off_t
offset);
94 void Munmap(void *start, size_t length);
95
96 /* Упаковщики стандартного ввода-вывода */
97 void Fclose(FILE *fp);
98 FILE *Fdopen(int fd, const char *type);
99 char *Fgets(char *ptr, int n, FILE *stream);
100 FILE *Fopen(const char *filename, const char *mode);
101 void Fputs(const char *ptr, FILE *stream);
```

```
102 size_t Fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
103 void Fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
104
105 /* Упаковщики динамического распределения памяти */
106 void *Malloc(size_t size);
107 void *Realloc(void *ptr, size_t size);
108 void *Calloc(size_t nmemb, size_t size);
109 void Free(void *ptr);
110
111 /* Упаковщики интерфейсов сокетов */
112 int Socket(int domain, int type, int protocol);
113 void Setssockopt(int s, int level, int optname, const void *optval,
114                   int optlen);
115 void Bind(int sockfd, struct sockaddr *my_addr, int addrlen);
116 void Listen(int s, int backlog);
117 int Accept(int s, struct sockaddr *addr, int *addrlen);
118 void Connect(int sockfd, struct sockaddr *serv_addr, int addrlen);
119
120 /* Упаковщики DNS */
121 struct hostent *Gethostbyname(const char *name);
122 struct hostent *Gethostbyaddr(const char *addr, int len, int type);
123
124 /* Упаковщики управления потоком Pthreads */
125 void Pthread_create(pthread_t *tidp, pthread_attr_t *attrp,
126                      void * (*routine)(void *), void *argp);
127 void Pthread_join(pthread_t tid, void **thread_return);
128 void Pthread_cancel(pthread_t tid);
129 void Pthread_detach(pthread_t tid);
130 void Pthread_exit(void *retval);
131 void Pthread_once(pthread_once_t *once_control, void (*init_function)());
132
133 /* Упаковщики семафора POSIX */
134 void Sem_init(sem_t *sem, int pshared, unsigned int value);
135 void P(sem_t *sem);
136 void V(sem_t *sem);
137
138 /* Пакет Rio (устойчивый ввод-вывод) */
139 ssize_t rio_readn(int fd, void *usrbuf, size_t n);
140 ssize_t rio_writen(int fd, void *usrbuf, size_t n);
141 void rio_readinitb(rio_t *rp, int fd);
142 ssize_t rio_readnb(rio_t *rp, void *usrbuf, size_t n);
143 ssize_t rio_readlineb(rio_t *rp, void *usrbuf, size_t maxlen);
144
```

```
145 /* Упаковщики для пакета Rio */
146 ssize_t Rio_readn(int fd, void *usrbuf, size_t n);
147 void Rio_writen(int fd, void *usrbuf, size_t n);
148 void Rio_readinitb(rio_t *rp, int fd);
149 ssize_t Rio_readnb(rio_t *rp, void *usrbuf, size_t n);
150 ssize_t Rio_readlineb(rio_t *rp, void *usrbuf, size_t maxlen);
151
152 /* Функции системы помоши клиент/сервер */
153 int open_clientfd(char *hostname, int portno);
154 int open_listenfd(int portno);
155
156 /* Упаковщики для функция системы помоши клиент/сервер */
157 int Open_clientfd(char *hostname, int port);
158 int Open_listenfd(int port);
159
160 #endif /* __CSAPP_H__ */
```

Исходный файл csapp.c

```
1 #include "csapp.h"
2
3 /*****
4 * Функции обработки ошибок
5 *****/
6 void unix_error(char *msg) /* ошибка стиля Unix */
7 {
8     fprintf(stderr, "%s: %s\n", msg, strerror(errno));
9     exit(0);
10 }
11
12 void posix_error(int code, char *msg) /* ошибка стиля Posix */
13 {
14     fprintf(stderr, "%s: %s\n", msg, strerror(code));
15     exit(0);
16 }
17
18 void dns_error(char *msg) /* ошибка стиля DNS */
19 {
20     fprintf(stderr, "%s: DNS error %d\n", msg, h_errno);
21     exit(0);
22 }
23
24 void app_error(char *msg) /* ошибка приложения */
25 {
26     fprintf(stderr, "%s\n", msg);
```

```
27     exit(0);
28 }
29
30 ****
31 * Упаковщики для функций управления процессами Unix
32 ****
33
34 pid_t Fork(void)
35 {
36     pid_t pid;
37
38     if ((pid = fork()) < 0)
39         unix_error("Fork error");
40     return pid;
41 }
42
43 void Execve(const char *filename,char *const argv[],char *const envp[])
44 {
45     if (execve(filename, argv, envp) < 0)
46         unix_error("Execve error");
47 }
48
49 pid_t Wait(int *status)
50 {
51     pid_t pid;
52
53     if ((pid = wait(status)) < 0)
54         unix_error("Wait error");
55     return pid;
56 }
57
58 pid_t Waitpid(pid_t pid, int *iptr, int options)
59 {
60     pid_t retpid;
61
62     if ((retpid = waitpid(pid, iptr, options)) < 0)
63         unix_error("Waitpid error");
64     return(retpid);
65 }
66
67 void Kill(pid_t pid, int signum)
68 {
69     int rc;
```

```
71     if ((rc = kill(pid, signum)) < 0)
72         unix_error("Kill error");
73 }
74
75 void Pause()
76 {
77     (void)pause();
78     return;
79 }
80
81 unsigned int Sleep (unsigned int secs)
82 {
83     unsigned int rc;
84
85     if ((rc = sleep(secs)) < 0)
86         unix_error("Sleep error");
87     return rc;
88 }
89
90 unsigned int Alarm(unsigned int seconds) {
91     return alarm(seconds);
92 }
93
94 void Setpgid(pid_t pid, pid_t pgid) {
95     int rc;
96
97     if ((rc = setpgid (pid, pgid)) < 0)
98         unix_error ("Setpgid error");
99     return;
100 }
101
102 pid_t Getpgrp(void) {
103     return getpgrp();
104 }
105
106 ****
107 * Упаковщики для функций сигналов Unix
108 ****
109
110 handler_t *Signal(int signum, handler_t *handler)
111 {
112     struct sigaction action, old_action;
113
114     action.sa_handler = handler;
115     sigemptyset(&action.sa_mask); /* блокировочные сигналы обрабатываемого
типа */
```

```
116     action.sa_flags = SA_RESTART; /* по возможности перезапустить системные
117                                         вызовы */
118
119     if (sigaction(signum, &action, &old_action) < 0)
120         unix_error("Signal error");
121     return (old_action.sa_handler);
122 }
123
124 void Sigprocmask(int how, const sigset_t *set, sigset_t *oldset)
125 {
126     if (sigprocmask(how, set, oldset) < 0)
127         unix_error("Sigprocmask error");
128     return;
129 }
130
131 void Sigemptyset(sigset_t *set)
132 {
133     if (sigemptyset(set) < 0)
134         unix_error("Sigemptyset error");
135     return;
136 }
137
138 void Sigfillset(sigset_t *set)
139 {
140     if (sigfillset(set) < 0)
141         unix_error("Sigfillset error");
142     return;
143 }
144
145 void Sigaddset(sigset_t *set, int signum)
146 {
147     if (sigaddset(set, signum) < 0)
148         unix_error("Sigaddset error");
149     return;
150 }
151
152 void Sigdelset(sigset_t *set, int signum)
153 {
154     if (sigdelset(set, signum) < 0)
155         unix_error("Sigdelset error");
156     return;
157 }
158
159 int Sigismember(const sigset_t *set, int signum)
160 {
161     int rc;
```

```
161     if ((rc = sigismember(set, signum)) < 0)
162         unix_error("Sigismember error");
163     return rc;
164 }
165
166 ****
167 /* Упаковщики для рутинных процедур ввода-вывода Unix
168 ****/
169
170
171 int Open(const char *pathname, int flags, mode_t mode)
172 {
173     int rc;
174
175     if ((rc = open(pathname, flags, mode)) < 0)
176         unix_error("Open error");
177     return rc;
178 }
179
180 ssize_t Read(int fd, void *buf, size_t count)
181 {
182     ssize_t rc;
183
184     if ((rc = read(fd, buf, count)) < 0)
185         unix_error("Read error");
186     return rc;
187 }
188
189 ssize_t Write(int fd, const void *buf, size_t count)
190 {
191     ssize_t rc;
192
193     if ((rc = write(fd, buf, count)) < 0)
194         unix_error("Write error");
195     return rc;
196 }
197
198 off_t Lseek(int fildes, off_t offset, int whence)
199 {
200     off_t rc;
201
202     if ((rc = lseek(fildes, offset, whence)) < 0)
203         unix_error("Lseek error");
204     return rc;
205 }
206
```

```
207 void Close(int fd)
208 {
209     int rc;
210
211     if ((rc = close(fd)) < 0)
212         unix_error("Close error");
213 }
214
215 int Select(int n, fd_set *readfds, fd_set *writefds,
216             fd_set *exceptfds, struct timeval *timeout)
217 {
218     int rc;
219
220     if ((rc = select(n, readfds, writefds, exceptfds, timeout)) < 0)
221         unix_error("Select error");
222     return rc;
223 }
224
225 int Dup2(int fd1, int fd2)
226 {
227     int rc;
228
229     if ((rc = dup2(fd1, fd2)) < 0)
230         unix_error("Dup2 error");
231     return rc;
232 }
233
234 void Stat (const char *filename, struct stat *buf)
235 {
236     if (stat(filename, buf) < 0)
237         unix_error("Stat error");
238 }
239
240 void Fstat(int fd, struct stat *buf)
241 {
242     if (fstat(fd, buf) < 0)
243         unix_error("Fstat error");
244 }
245
246 ****
247 * Упаковщики для функций отображения памяти
248 ****
249 void *Mmap(void *addr, size_t len, int prot, int flags, int fd,
250             off_t offset)
251 {
252     void *ptr;
```

```
253 if ((ptr = mmap(addr, len, prot, flags, fd, offset)) == ((void *) -1))
254     unix_error("mmap error");
255 return(ptr);
256 }
257
258 void Munmap(void *start, size_t length)
259 {
260     if (munmap(start, length) < 0)
261         unix_error("munmap error");
262 }
263
264 /***** Улаковщики для функций динамического распределения памяти *****/
265
266 * Улаковщики для функций динамического распределения памяти
267 *****/
268 void *Malloc(size_t size)
269 {
270     void *p;
271
272     if ((p = malloc(size)) == NULL)
273         unix_error("Malloc error");
274     return p;
275 }
276
277 void *Realloc(void *ptr, size_t size)
278 {
279     void *p;
280
281     if ((p = realloc(ptr, size)) == NULL)
282         unix_error("Realloc error");
283     return p;
284 }
285
286 void *Calloc(size_t nmemb, size_t size)
287 {
288     void *p;
289
290     if ((p = calloc(nmemb, size)) == NULL)
291         unix_error ("Calloc error");
292     return p;
293 }
294
295 void Free(void *ptr)
296 {
297     free(ptr);
298 }
299
```

```
300 ****  
301 * Упаковщики для функций стандартного ввода-вывода  
302 ****  
303 void Fclose (FILE *fp)  
304 {  
305     if (fclose(fp) != 0)  
306         unix_error ("Fclose error");  
307 }  
308  
309 FILE *Fdopen (int fd, const char *type)  
310 {  
311     FILE *fp;  
312  
313     if ((fp = fdopen(fd, type)) == NULL)  
314         unix_error ("Fdopen error");  
315  
316     return fp;  
317 }  
318  
319 char *Fgets(char *ptr, int n, FILE *stream)  
320 {  
321     char *rptr;  
322  
323     if (((rptr = fgets(ptr, n, stream)) == NULL) && ferror(stream))  
324         app_error("Fgets error");  
325  
326     return rptr;  
327 }  
328  
329 FILE *Fopen(const char *filename, const char *mode)  
330 {  
331     FILE *fp;  
332  
333     if ((fp = fopen(filename, mode)) == NULL)  
334         unix_error ("Fopen error");  
335  
336     return fp;  
337 }  
338  
339 void Fputs(const char *ptr, FILE *stream)  
340 {  
341     if (fputs(ptr, stream) == EOF)  
342         unix_error ("Fputs error");  
343 }  
344
```

```
345 size_t Fread(void *ptr, size_t size, size_t nmemb, FILE *stream)
346 {
347     size_t n;
348
349     if (((n = fread(ptr, size, nmemb, stream)) < nmemb) && ferror(stream))
350         unix_error("Fread error");
351     return n;
352 }
353
354 void Fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)
355 {
356     if (fwrite(ptr, size, nmemb, stream) < nmemb)
357         unix_error("Fwrite error");
358 }
359
360
361 /***** ИНТЕРФЕЙС УПАКОВЩИКОВ СОКЕТОВ *****
362 * Интерфейс упаковщиков сокетов
363 *****/
364
365 int Socket(int domain, int type, int protocol)
366 {
367     int rc;
368
369     if ((rc = socket(domain, type, protocol)) < 0)
370         unix_error("Socket error");
371     return rc;
372 }
373
374 void Setsockopt(int s, int level, int optname, const void *optval, int optlen)
375 {
376     int rc;
377
378     if ((rc = setsockopt(s, level, optname, optval, optlen)) < 0)
379         unix_error("Setsockopt error");
380 }
381
382 void Bind(int sockfd, struct sockaddr *my_addr, int addrlen)
383 {
384     int rc;
385
386     if ((rc = bind(sockfd, my_addr, addrlen)) < 0)
387         unix_error("Bind error");
388 }
389
```

```
390 void Listen(int s, int backlog)
391 {
392     int rc;
393
394     if ((rc = listen(s, backlog)) < 0)
395         unix_error("Listen error");
396 }
397
398 int Accept(int s, struct sockaddr *addr, int *addrlen)
399 {
400     int rc;
401
402     if ((rc = accept(s, addr, addrlen)) < 0)
403         unix_error("Accept error");
404     return rc;
405 }
406
407 void Connect(int sockfd, struct sockaddr *serv_addr, int addrlen)
408 {
409     int rc;
410
411     if ((rc = connect(sockfd, serv_addr, addrlen)) < 0)
412         unix_error("Connect error");
413 }
414
415 /*****
416 * Упаковщики интерфейса DNS
417 *****/
418
419 struct hostent *Gethostbyname(const char *name)
420 {
421     struct hostent *p;
422
423     if ((p = gethostbyname(name)) == NULL)
424         dns_error("Gethostbyname error");
425     return p;
426 }
427
428 struct hostent *Gethostbyaddr(const char *addr, int len, int type)
429 {
430     struct hostent *p;
431
432     if ((p = gethostbyaddr(addr, len, type)) == NULL)
433         dns_error("Gethostbyaddr error");
434     return p;
435 }
436
```

```
437 /*****  
438 * Упакованный функций управления потоком Pthreads  
439 *****/  
440  
441 void Pthread_create(pthread_t *tidp, pthread_attr_t *attrp,  
442     void * (*routine)(void *), void *argp)  
443 {  
444     int rc;  
445  
446     if ((rc = pthread_create(tidp, attrp, routine, argp)) != 0)  
447         posix_error(rc, "Pthread_create error");  
448 }  
449  
450 void Pthread_cancel(pthread_t tid) {  
451     int rc;  
452  
453     if ((rc = pthread_cancel(tid)) != 0)  
454         posix_error(rc, "Pthread_cancel error");  
455 }  
456  
457 void Pthread_join(pthread_t tid, void **thread_return) {  
458     int rc;  
459  
460     if ((rc = pthread_join(tid, thread_return)) != 0)  
461         posix_error(rc, "Pthread_join error");  
462 }  
463  
464 void Pthread_detach(pthread_t tid) {  
465     int rc;  
466  
467     if ((rc = pthread_detach(tid)) != 0)  
468         posix_error(rc, "Pthread_detach error");  
469 }  
470  
471 void Pthread_exit(void *retval) {  
472     pthread_exit(retval);  
473 }  
474  
475 pthread_t Pthread_self(void) {  
476     return pthread_self();  
477 }  
478  
479 void Pthread_once(pthread_once_t *once_control, void (*init_function)())  
{  
480     pthread_once(once_control, init_function);  
481 }  
482
```

```
483 ****  
484 * Упаковщик семафоров Posix  
485 ****  
486  
487 void Sem_init(sem_t *sem, int pshared, unsigned int value)  
488 {  
489     if (sem_init(sem, pshared, value) < 0)  
490         unix_error("Sem_init error");  
491 }  
492  
493 void P(sem_t *sem)  
494 {  
495     if (sem_wait(sem) < 0)  
496         unix_error("P error");  
497 }  
498  
499 void V(sem_t *sem)  
500 {  
501     if (sem_post(sem) < 0)  
502         unix_error("V error");  
503 }  
504  
505 ****  
506 * Пакет Rio - устойчивые функции ввода-вывода  
*****  
508 /*  
509 * rio_readn - устойчиво считывает n байт (небуферизованных)  
510 */  
511 ssize_t rio_readn(int fd, void *usrbuf, size_t n)  
512 {  
513     size_t nleft = n;  
514     ssize_t nread;  
515     char *bufp = usrbuf;  
516  
517     while (nleft > 0) {  
518         if ((nread = read(fd, bufp, nleft)) < 0) {  
519             if (errno == EINTR) /* прервано возвратом обработчика сигналов */  
520                 nread = 0; /* очередной вызов считывания () */  
521             else  
522                 return -1; /* errno установлено read() */  
523         }  
524         else if (nread == 0)  
525             break; /* EOF */  
526         nleft -= nread;  
527         bufp += nread;  
528     }  
}
```

```
529     return (n - nleft);    /* возврат >= 0 */
530 }
531
532 /*
533 * rio_writen - устойчиво записывает n байт (небуферизованных)
534 */
535 ssize_t rio_writen(int fd, void *usrbuf, size_t n)
536 {
537     size_t nleft = n;
538     ssize_t nwritten;
539     char *bufp = usrbuf;
540
541     while (nleft > 0) {
542         if ((nwritten = write(fd, bufp, nleft)) <= 0) {
543             if (errno == EINTR) /* прервано возвратом обработчика сигналов */
544                 nwritten = 0; /* очередной вызов записи() */
545             else
546                 return -1; /* errno задано write() */
547         }
548         nleft -= nwritten;
549         bufp += nwritten;
550     }
551     return n;
552 }
553
554
555 /*
556 * rio_read - упаковщик для считывания Unix read(), переносящей
557 * минимум (n, rio_cnt) байт из внутреннего буфера в буфер
558 * пользователя, где n - количество байт, запрошенное пользователем, а
559 * rio_cnt - количество несчитанных байт во внутреннем буфере. При
560 * вводе записи rio_read() пополняет внутренний буфер по вызову
561 * функции read(), если внутренний буфер пуст
562 */
563 static ssize_t rio_read(rio_t *rp, char *usrbuf, size_t n)
564 {
565     int cnt;
566
567     while (rp->rio_cnt <= 0) { /* заполнить, если буфер пуст */
568         rp->rio_cnt = read(rp->rio_fd, rp->rio_buf,
569             sizeof(rp->rio_buf));
570         if (rp->rio_cnt < 0) {
571             if (errno != EINTR) /* прервано возвратом обработчика сигналов */
572                 return -1;
573         }
574     }
575 }
```

```
574 else if (rp->rio_cnt == 0) /* EOF */
575     return 0;
576 else
577     rp->rio_bufptr = rp->rio_buf; /* перезагрузка ptr буфера */
578 }
579
580 /* Скопировать минимум (n, rp->rio_cnt) байт из внутреннего буфера
   в буфер пользователя */
581 cnt = n;
582 if (rp->rio_cnt < n)
583     cnt = rp->rio_cnt;
584 memcpy(usrbuf, rp->rio_bufptr, cnt);
585 rp->rio_bufptr += cnt;
586 rp->rio_cnt -= cnt;
587 return cnt;
588 }
589
590 /*
591 * rio_readinitb - связать дескриптор с буфером считывания и перезагрузить
   буфер
592 */
593 void rio_readinitb(rio_t *rp, int fd)
594 {
595     rp->rio_fd = fd;
596     rp->rio_cnt = 0;
597     rp->rio_bufptr = rp->rio_buf;
598 }
599
600 /*
601 * rio_readnb - Robustly read n bytes (buffered)
602 */
603 ssize_t rio_readnb(rio_t *rp, void *usrbuf, size_t n)
604 {
605     size_t nleft = n;
606     ssize_t nread;
607     char *bufp = usrbuf;
608
609     while (nleft > 0) {
610         if ((nread = rio_read(rp, bufp, nleft)) < 0) {
611             if (errno == EINTR) /* interrupted by sig handler return */
612                 nread = 0; /* call read() again */
613             else
614                 return -1; /* errno set by read() */
615         }
616         else if (nread == 0)
617             break; /* EOF */
```

```
618     nleft -= nread;
619     bufp += nread;
620 }
621 return (n - nleft); /* return >= 0 */
622 }
623
624 /*
625 * rio_readlineb - robustly read a text line (buffered)
626 */
627 ssize_t rio_readlineb(rio_t *rp, void *usrbuf, size_t maxlen)
628 {
629     int n, rc;
630     char c, *bufp = usrbuf;
631
632     for (n = 1; n < maxlen; n++) {
633         if ((rc = rio_read(rp, &c, 1)) == 1) {
634             *bufp++ = c;
635             if (c == '\n')
636                 break;
637         } else if (rc == 0) {
638             if (n == 1)
639                 return 0; /* EOF, отсутствие считывания данных */
640             else
641                 break; /* EOF, некоторые данные были считаны */
642         } else
643             return -1; /* ошибка */
644     }
645     *bufp = 0;
646     return n;
647 }
648
649 ****
650 * Упаковщики для устойчивых рутинных процедур ввода-вывода
651 ****
652 ssize_t Rio_readn(int fd, void *ptr, size_t nbytes)
653 {
654     ssize_t n;
655
656     if ((n = rio_readn(fd, ptr, nbytes)) < 0)
657         unix_error("Rio_readn error");
658     return n;
659 }
660
661 void Rio_writen(int fd, void *usrbuf, size_t n)
662 {
```

```
663     if (rio_writen(fd, usrbuf, n) != n)
664         unix_error("Rio_writenb error");
665     }
666
667 void Rio_readinitb(rio_t *rp, int fd)
668 {
669     rio_readinitb(rp, fd);
670 }
671
672 ssize_t Rio_readnb(rio_t *rp, void *usrbuf, size_t n)
673 {
674     ssize_t rc;
675
676     if ((rc = rio_readnb(rp, usrbuf, n)) < 0)
677         unix_error("Rio_readnb error");
678     return rc;
679 }
680
681 ssize_t Rio_readlineb(rio_t *rp, void *usrbuf, size_t maxlen)
682 {
683     ssize_t rc;
684
685     if ((rc = rio_readlineb(rp, usrbuf, maxlen)) < 0)
686         unix_error("Rio_readlineb error");
687     return rc;
688 }
689
690 /*****
691 * Функции справки (помощи) клиента/сервера
692 *****/
693 /*
694 * open_clientfd - открытие подключения к серверу на <hostname, port>
695 * и возврат дескриптора сокета для считывания и записи.
696 * Возврат -1 и установка errno на ошибку Unix.
697 * Возврат -2 и установка h_errno на ошибку DNS (gethostbyname).
698 */
699 int open_clientfd(char *hostname, int port)
700 {
701     int clientfd;
702     struct hostent *hp;
703     struct sockaddr_in serveraddr;
704
705     if ((clientfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
706         return -1; /* проверка errno на причину ошибки */
707 }
```

```
708 /* Заполнение IP-адреса сервера и порта */
709 if ((hp = gethostbyname(hostname)) == NULL)
710     return -2; /* проверка h_errno на причину ошибки */
711 bzero((char *) &serveraddr, sizeof(serveraddr));
712 serveraddr.sin_family = AF_INET;
713 bcopy((char *)hp->h_addr,
714     (char *)&serveraddr.sin_addr.s_addr, hp->h_length);
715 serveraddr.sin_port = htons(port);
716
717 /* Установка подключения к серверу */
718 if (connect(clientfd, (SA *)&serveraddr, sizeof(serveraddr)) < 0)
719     return -1;
720 return clientfd;
721 }
722
723 /*
724 * open_listenfd - открытие и возврат прослушивающего сокета на порту
725 * Возврат -1 и установка errno на ошибку Unix
726 */
727 int open_listenfd(int port)
728 {
729     int listenfd, optval=1;
730     struct sockaddr_in serveraddr;
731
732     /* Создание дескриптора сокета */
733     if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
734         return -1;
735
736     /* Устранение ошибки "Address already in use" (адрес уже используется)
        из связывания. */
737     if (setsockopt (listenfd, SOL_SOCKET, SO_REUSEADDR,
738         (const void *)&optval, sizeof(int)) < 0)
739         return -1;
740
741     /* Listenfd будет конечной точкой порта на любом IP-адресе
        */
742     bzero((char *) &serveraddr, sizeof(serveraddr));
743     serveraddr.sin_family = AF_INET;
744     serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
745     serveraddr.sin_port = htons((unsigned short)port);
746     if (bind(listenfd, (SA *)&serveraddr, sizeof(serveraddr)) < 0)
747         return -1;
748
749
750     /* Подготовка сокета к приему запросов на подключение */
751     if (listen(listenfd, LISTENQ) < 0)
```

```
752     return -1;
753     return listenfd;
754 }
755
756 /***** Упаковщик для рутинных процедур справки клиента/сервера *****/
757 * Упаковщик для рутинных процедур справки клиента/сервера
758 ****
759 int Open_clientfd(char *hostname, int port)
760 {
761     int rc;
762
763     if ((rc = open_clientfd(hostname, port)) < 0) {
764         if (rc == -1)
765             unix_error("Open_clientfd Unix error");
766         else
767             dns_error("Open_clientfd DNS error");
768     }
769     return rc;
770 }
771
772 int Open_listenfd(int port)
773 {
774     int rc;
775
776     if ((rc = open_listenfd(port)) < 0)
777         unix_error("Open_listenfd error");
778     return rc;
779 }
```

Библиография

1. K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, 1996.
2. V. Bala, E. Duesterwald and S. Banerjiia. Dynamo: A transparent dynamic optimization system. In Proceedings of the 1995 ACM Conference on Programming Language Design and Implementation (PLDI), 1–12, June 2000.
3. T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext transfer protocol — HTTP/1.0. RFC 1945, 1996.
4. A. Birrell. An introduction to programming with threads. Technical Report Report 35, Digital Systems Research Center, 1989.
5. F. P. Brooks, Jr. *The Mythical Man-Month*, Second Edition. Addison-Wesley, 1995.
6. A. Demke Brown and T. Mowry. Taming the memory hogs: Using compiler-inserted releases to manage physical memory intelligently. In Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI), pages 31–44, October 2000.
7. R. E. Bryant and D. R. O'Hallaron. Introducing computer systems from a programmer's perspective. In Proceedings of the Technical Symposium on Computer Science Education (SIGCSE).ACM, February 2001.
8. B. R. Buck and J.K. Hollingsworth. An API for runtime code patching. *Journal of High Performance Computing Applications*, 14(4):317–324, June 2000.
9. D. Butenhof. *Programming with Posix Threads*. Addison-Wesley, 1997.
10. S. Carson and P. Reynolds. The geometry of semaphore programs. *ACM Transactions on Programming Languages and Systems*, 9(1):25–53, 1987.
11. J. B. Carter, W. C. Hsieh, L. B. Stoller, M. R. Swanson, L. Zhang, E. L. Brunvand, A. Davis, C.-C.Kuo, R. Kuramkote, M. A. Parker, L. Schaelicke, and T. Tateyama. Impulse: Building a smarter memory controller. In Proceedings of the Fifth International Symposium on High Performance Computer Architecture (HPCA), pages 70–79, January 1999.
12. P. Chen, E. Lee, G. Gibson, R. Katz, and D. Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2), June 1994.

13. S. Chen, P. Gibbons, and T. Mowry. Improving index performance through prefetching. In Proceedings of the 2001 ACM SIGMOD Conference. ACM, May 2001.
14. T. Chilimbi, M. Hill, and J. Larus. Cache-conscious structure layout. In Proceedings of the 1999 ACM Conference on Programming Language Design and Implementation (PLDI), pages 1–12. ACM, May 1999.
15. B. Cmelik and D. Keppel. Shade: A fast instruction set simulator for execution profiling. In Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, pages 128–137, May 1994.
16. E. Coffman, M. Elphick, and A. Shoshani. System deadlocks. ACM Computing Surveys, 3(2):67–78, June 1971.
17. Danny Cohen. On holy wars and a plea for peace. IEEE Computer, 14(10):48–54, October 1981.
18. Intel Corporation. Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture, 1999. Order Number 243190. Also available at <http://developer.intel.com/>.
19. Intel Corporation. Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference, 1999. Order Number 243191. Also available at <http://developer.intel.com/>.
20. C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In DARPA Information Survivability Conference and Expo (DISCEX), March 2000.
21. John H. Crawford. The i486 CPU: Executing instructions in one clock cycle. IEEE Micro, 10(1):27–36, February 1990.
22. V. Cuppu, B. Jacob, B. Davis, and T. Mudge. A performance comparison of contemporary DRAM architectures. In Proceedings of the Twenty-Sixth International Symposium on Computer Architecture (ISCA), Atlanta, GA, May 1999. IEEE.
23. B. Davis, B. Jacob, and T. Mudge. The new DRAM interfaces: SDRAM, RDRAM, and variants. In Proceedings of the Third International Symposium on High Performance Computing (ISHPC), Tokyo, Japan, October 2000.
24. E. W. Dijkstra. Cooperating sequential processes. Technical Report EWD-123, Technological University, Eindhoven, The Netherlands, 1965.
25. C. Ding and K. Kennedy. Improving cache performance of dynamic applications through data and computation reorganizations at run time. In Proceedings of the 1999 ACM Conference on Programming Language Design and Implementation (PLDI), pages 229–241. ACM, May 1999.
26. M. W. Eichen and J. A. Rochlis. With microscope and tweezers: An analysis of the Internet virus of November, 1988. In IEEE Symposium on Research in Security and Privacy, 1989.
27. R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol — HTTP/1.1. RFC 2616, 1999.

28. G. Gibson, D. Nagle, K. Amiri, J. Butler, F. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture. In Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). ACM, October 1998.
29. G. Gibson and R. Van Meter. Network attached storage architecture. Communications of the ACM, 43(11), November 2000.
30. L. Gwennap. Intel's P6 uses decoupled superscalar design. Microprocessor Report, 9(2), February 1995.
31. L. Gwennap. New algorithm improves branch prediction. Microprocessor Report, 9(4), March 1995.
32. S. P. Harbison and G. L. Steele, Jr. C, A Reference Manual. Prentice Hall, 1995.
33. J. L. Hennessy and D. A. Patterson. Computer Architecture: A Quantitative Approach, Third Edition. Morgan-Kaufmann, San Francisco, 2002.
34. C. A. R. Hoare. Monitors: An operating system structuring concept. Communications of the ACM, 17(10):549–557, October 1974.
35. Intel. Tool Interface Standards Portable Formats Specification, Version 1.1, 1993. Order number 241597. Also available at <http://developer.intel.com/>.
36. F. Jones, B. Prince, R. Norwood, J. Hartigan, W. Vogley, C. Hart and D. Bondurant. A new era of fast dynamic RAMs. IEEE Spectrum, pages 43–39, October 1992.
37. R. Jones and R. Lins. Garbage Collection: Algorithms for Automatic Dynamic Memory Management. Wiley, 1996.
38. M. Kaashoek, D. Engler, G. Ganger, H. Briceo, R. Hunt, D. Maziers, T. Pinckney, R. Grimm, J. Jannotti and K. MacKenzie. Application performance and flexibility on Exokernel systems. In Proceedings of the Sixteenth Symposium on Operating System Principles (SOSP), October 1997.
39. R. Katz. Contemporary Logic Design. Addison-Wesley, 1993.
40. B. Kernighan and D. Ritchie. The C Programming Language, Second Edition. Prentice Hall, 1988.
41. B. W. Kernighan and R. Pike. The Practice of Programming. Addison-Wesley, 1999.
42. T. Kilburn, B. Edwards, M. Lanigan and F. Sumner. One-level storage system. IRE Transactions on Electronic Computers, EC-11:223–235, April 1962.
43. D. Knuth. The Art of Computer Programming, Volume 1: Fundamental Algorithms, Second Edition. Addison-Wesley, 1973.
44. J. Kurose and K. Ross. Computer Networking: A Top-Down Approach Featuring the Internet. Addison-Wesley, 2000.
45. M. Lam, E. Rothberg and M. Wolf. The cache performance and optimizations of blocked algorithms. In Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). ACM, April 1991.

46. J. R. Larus and E. Schnarr. EEL: Machine-independent executable editing. In Proceedings of the 1995 ACM Conference on Programming Language Design and Implementation (PLDI), June 1995.
47. J. R. Levine. Linkers and Loaders. Morgan-Kaufmann, San Francisco, 1999.
48. Y. Lin and D. Padua. Compiler analysis of irregular memory accesses. In Proceedings of the 2000 ACM Conference on Programming Language Design and Implementation (PLDI), pages 157–168. ACM, June 2000.
49. J. L. Lions. Ariane 5 Flight 501 failure. Technical report, European Space Agency, July 1996.
50. S. Macguire. Writing Solid Code. Microsoft Press, 1993.
51. J. Markoff. Microsoft caught in 'dirty tricks' vs. AOL. New York Times, August 16 1999.
52. E. Marshall. Fatal error: How Patriot overlooked a Scud. Science, page 1347, March 13 1992.
53. J. Morris, M. Satyanarayanan, M. Conner, J. Howard, D. Rosenthal, and F. Smith. Andrew: A distributed personal computing environment. Communications of the ACM, March 1986.
54. T. Mowry, M. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). ACM, October 1992.
55. S. S. Muchnick. Advanced Compiler Design and Implementation. Morgan-Kaufmann, 1997.
56. M. Overton. Numerical Computing with IEEE Floating Point Arithmetic. SIAM, 2001.
57. D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks (RAID). In Proceedings of the 1998 ACM SIGMOD Conference. ACM, June 1988.
58. L. Peterson and B. Davies. Computer Networks: A Systems Approach, Third Edition. Morgan-Kaufmann, 1999.
59. S. Przybylski. Cache and Memory Hierarchy Design: A Performance-Directed Approach. Morgan-Kaufmann, 1990.
60. W. Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. Communications of the ACM, 35(8):102–114, August 1992.
61. W. Pugh. Fixing the Java memory model. In Proceedings of the Java Grande Conference, June 1999.
62. J. Rabaey. Digital Integrated Circuits: A Design Perspective. Prentice Hall, 1996.
63. D. Ritchie. The evolution of the Unix time-sharing system. AT&T Bell Laboratories Technical Journal, 63(6 Part 2):1577–1593, October 1984.
64. D. Ritchie. The development of the C language. In Proceedings of the Second History of Programming Languages Conference, Cambridge, MA, April 1993.
65. D. Ritchie and K. Thompson. The Unix timesharing system. Communications of the ACM, 17(7):365–367, July 1974.

66. T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. Bershad and B. Chen. Instrumentation and optimization of Win32/Intel executables using Etch. In Proceedings of the USENIXWindows NT Workshop, Seattle, Washington, August 1997.
67. M. Satyanarayanan, J. Kistler, P. Kumar, M. Okasaki, E. Siegel and D. Steere. Coda: A highly available file system for a distributed workstation environment. IEEE Transactions on Computers, 39(4):447–459, April 1990.
68. J. Schindler and G. Ganger. Automated disk drive characterization. Technical Report CMU-CS-99-176, School of Computer Science, Carnegie Mellon University, 1999.
69. B. Shriver and B. Smith. The Anatomy of a High-Performance Microprocessor: A Systems Perspective. IEEE Computer Society, 1998.
70. A. Silberschatz and P. Galvin. Operating Systems Concepts, Fifth Edition. John Wiley & Sons, 1998.
71. R. Skeel. Roundoff error and the Patriot missile. SIAM News, 25(4):11, July 1992.
72. A. Smith. Cache memories. ACM Computing Surveys, 14(3), September 1982.
73. E. H. Spafford. The Internet worm program: An analysis. Technical Report CSD-TR-823, Department of Computer Science, Purdue University, 1988.
74. A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In Proceedings of the 1994 ACM Conference on Programming Language Design and Implementation (PLDI), June 1994.
75. W. Stallings. Operating Systems: Internals and Design Principles, Fourth Edition. Prentice Hall, 2000.
76. W. Richard Stevens. Advanced Programming in the Unix Environment. Addison-Wesley, 1992.
77. W. Richard Stevens. TCP/IP Illustrated: The Protocols, volume 1. Addison-Wesley, 1994.
78. W. Richard Stevens. TCP/IP Illustrated: The Implementation, volume 2. Addison-Wesley, 1995.
79. W. Richard Stevens. TCP/IP Illustrated: TCP for Transactions, HTTP, NNTP and the Unix domain protocols, volume 3. Addison-Wesley, 1996.
80. W. Richard Stevens. Unix Network Programming: Interprocess Communications, Second Edition, volume 2. Prentice Hall, 1998.
81. W. Richard Stevens. Unix Network Programming: Networking APIs, Second Edition, volume 1. Prentice Hall, 1998.
82. T. Stricker and T. Gross. Global address space, non-uniform bandwidth: A memory system performance characterization of parallel systems. In Proceedings of the Third International Symposium on High Performance Computer Architecture (HPCA), pages 168–179, San Antonio, TX, February 1997. IEEE.
83. A. Tannenbaum. Modern Operating Systems, Second Edition. Prentice Hall, 2001.
84. A. Tannenbaum. Computer Networks, Third Edition. Prentice Hall, 1996.

85. K. P. Wadleigh and I. L. Crawford. Software Optimization for High-Performance Computing: Creating Faster Applications. Prentice Hall, 2000.
86. J. F. Wakerly. Digital Design Principles and Practices, Third Edition. Prentice Hall, 2000.
87. M. V. Wilkes. Slave memories and dynamic storage allocation. *IEEE Transactions on Electronic Computers*, EC-14(2), April 1965.
88. P. Wilson, M. Johnstone, M. Neely and D. Boles. Dynamic storage allocation: A survey and critical review. In International Workshop on Memory Management, Kinross, Scotland, 1995.
89. M. Wolf and M. Lam. A data locality algorithm. In Conference on Programming Language Design and Implementation (SIGPLAN), pages 30–44, June 1991.
90. J. Wylie, M. Bigrigg, J. Strunk, G. Ganger, H. Kiliccote and P. Khosla. Survivable information storage systems. *IEEE Computer*, August 2000.
91. X. Zhang, Z. Wang, N. Gloy, J. B. Chen and M.D. Smith. System support for automatic profiling and optimization. In Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP), pages 15–26, October 1997.

Предметный указатель

A

Absolute addressing 641
Address translation 786
Addressing modes См. Режимы адресации
Aggregate payload 835
Allocated bit 838
American National Standards Institute
 См. ANSI
American Standard Code for Information
 Interchange См. ASCII
Andreesen 940
ANSI 15
ASCII 15

B

Bell Labs 28
Berkeley Software Distribution 28
Berners-Lee 939
Binary translation 660
Boles 873
Brian Kernighan 28
Buddy system 858
Buffer overflow 244
Buffer overflow bug 866

C

Callee См. Вызываемая программа
Caller См. Вызывающая программа
Carry flag См. CF
Carson 1013
CAS 527
Central Processing Unit См. CPU

CF 267

CISC, система команд 418
Clock ticks 749
Coalescing 842
Code motion 288
COFF, формат 624
Coffman 1013
Column Access Strobe См. CAS
Compile time 619
Compiler См. Компилятор
Compiler driver 622
Concurrent servers 938
Coprocessor 250
Copy-on-write 824
CPE 442, 483, 741
CPI 410
◊ единица измерения 417
CPU См. Центральный процессор
Cycles per element См. CPI

D

Datagrams 920
DDR DRAM 530
Demand-zero pages 822
Dennis Ritchie 15
Dijkstra 993, 1013
Dirty bit 816
Dirty page 816
Disassembler См. Обратный ассемблер
DLL 650
DMA См. Прямой доступ к памяти
DNS 1048
Domain Naming System См. DNS
Dotted-decimal notation 922
DRAM 35

Dual Inline Memory Module *См.* DIMM
 Dynamic link libraries 650
 Dynamic memory allocation 828
 Dynamic Random Access Memory
См. DRAM

E

ECF 672
 EDO DRAM 530
 EEPROM 531
 Effective address 641
См. Исполнительный адрес
 Entry point 646
 Ephemerol port 927
 Ethernet segment 915
 EU 459
 Exceptional control flow *См.* ECF
 Explicit allocator 828
 Exploit code 247
 Extended precision 252

F

Fall through 197
 False fragmentation 841
 FIFO 461
 Flat addressing 151
 Floating point unit *См.* FPU
 Floating-point status word 262
 FPM DRAM 529
 FPU 264
 Fragmentation 835
 Frame pointer *См.* указатель фрейма
 Frederick Brooks 265

G

GAS 153
 GDB 154
 Gene Amdahl 510
 Global offset table *См.* GOT
 Global symbol 626
 Gordon Moore 150
 GOT 656
 GUI, интерфейс 419

H

Handler 713
 функция 713

Hardware Control Language, HCL 319
 Hardware Description Language 319
 HDL 1025
 Heap 647, 828
 Hennessy 601
 HTTP 939
 Hub 915
 Hypertext Transfer Protocol *См.* HTTP

I

IA32 149
 архитектура 151
 команда 420
 процессор 419
 IA-32 147
 ICANN, организация 923
 ICU 459
 ID группы процессов 710
 ID задания *См.* JID
 ID процесса *См.* PID
 Implicit allocator 829
 Implicit free list 838
 Intel Pentium 20
 Internet Software Consortium,
 организация 926
 Interprocess communication *См.* IPC
 Interval time 744
 IPC 967
 единица измерения 417
 ISA:
 модель 417
 система команд 418

J

JID 737
 Job 710
 Job list 725
 Joe Ossanna 28
 Johnstone 873

K

Ken Thompson 28
 Kernel 648
 Kernel mode 745
 Kilburn 872
 Knuth 840

L

Last-In First-Out *См.* LIFO
 Lawrence Roberts 928
 Lazy binding 657
 Levine 660
 LFU, стратегия 572
 LIFO 855
 Linear address space 787
 Linker 619. *См.* редактор связей
 Local symbol 626
 Locality 793
 Loopback address 925
 LRU, стратегия 555, 572

M

Marc Andreessen 940
 McCarthy 859
 Memory Management Unit *См.* MMU
 Memory mapping 821
 MIME 940
 Minimum block size 839
 Mockapetris 928
 Muchnick 274
 Multipurpose Internet Mail Extensions
См. MIME
 MUTual EXclusion 994

N

National Science Foundation *См.* NSF
 NCSA, организация 940
 Neely 873
 NSF 928

O

Object dump *См.* Дамп объектной программы
 Object file 624
 Object module 624

P

Padding 838
 Page Directory Entry *См.* PDE
 Page fault 791
 Page frames 788
 Page table 789

Page Table Entry *См.* PTE

Paging 792
 Patterson 601
 Paul Mockapetris 928
 Payload 835
 PC 826 *См.* счетчик команд
 PCI 539
 PC-relative 181
 PDE 814
 PE, формат 624
 Peak utilization 835
 Peripheral Component Interconnect
См. PCI
 Physical Address *См.* PA
 Physical address space 787
 Physical Pages *См.* PP
 PID 688
 PIPE-, конвейер 418
 PLT 657
 Point-to-point, соединение 927
 Pollute 687
 Position-independent code *См.* PIC
 POSIX, стандарт 722
 Preprocessor *См.* препроцессор
 Private 823
 Private copy-on-write 824
 Procedure linkage table *См.* PLT
 Process Identifier *См.* PID
 Program counter *См.* PC
 Prologue block 847
 Proxy chain 943
 Przybylski 601
 PTE 790, 814
 Pthreading 998
 Pugh 1013

R

RAS 527
 RDRAM 530
 Reachability graph 860
 Read-Only Memory *См.* ROM
 Reap 694
 Reference bit 816
 Relocation entry 640
 Response 914
 Return address *См.* адрес возврата
 Reverse engineering 146 *См.* обратное проектирование
 Reverse Polish Notation *См.* RPN

Reynolds 1013
 Richard Stallman 18
 Richard Stevens 1047
 RISC:
 ◊ процессор 417
 ◊ система команд 418
 Robert Kahn 928
 Robust I/O См. RIO
 ROM 530
 Row Access Strobe См. RAS
 RPN 252
 Run time 619

S

SDRAM 530
 Section header table 624
 Seek 889
 Segment header table 646
 Segregated storage 855
 Separate compilation 619
 SEQ:
 ◊ имитатор 419
 ◊ процессор 418
 SEQ+, процессор 418
 Shared 823
 Shared area 823
 Shared library См. Разделяемые библиотеки
 Single Inline Memory Module
 См. SIMM
 Smith 661
 Source host 918
 Spafford 248
 SPARC, процессор 417
 Stack frame См. стековый фрейм
 Startup code 648
 Static library 635
 Static linker 623
 Static Random Access Memory
 См. SRAM
 Stevens 722
 Swap area 822
 Swap file 822
 Swap space 822
 Swapping 792
 System clock 771

T

Tcl, язык 419
 test-expr См. условие продолжения цикла
 Thrashing 794
 Throughput 834
 TID 976
 Tim Berners-Lee 939
 Tk, библиотека 419
 TLB 416, 573, 803
 Trace 746
 Transaction 914
 Translation Lookaside Buffer См. TLB

U

Universal Resource Locator См. URL
 Universal Serial Bus См. USB
 URL 941, 1017
 USB 539
 User mode 745

V

valid bit 790
 VHDL 320
 Vinton Cerf 928
 Virtual address space 787
 Virtual Addressing См. VA
 Virtual machine 274
 Virtual Memory См. VM
 Virtual Pages См. VP
 Virus 248
 VM 785
 VRAM 530

W

W. Richard Stevens 899, 957
 Wilkes 601
 Working set 794

Y

Y86, система команд 418

А

- Автоматическое тестирование процессора 419
- Адрес:

 - ◊ возврата 203
 - ◊ команд назначения 180

- Адресный вход 329
- Аппаратные прерывания 677
- Арифметико-логическое устройство 327
- Архитектура:

 - ◊ Compaq Alpha 47
 - ◊ IA32 160
 - ◊ load/store 311

- Ассемблерный код 151, 158

Б

- Базовый регистр 307
- Бинарные операции 169
- Бит режима 684
- Блокиратор оптимизации 441, 438
- Блокировка загрузки 392
- Булева операция 56
- Булево кольцо 59
- Быстрое преобразование виртуальных адресов 416

В

- Виртуальная память 32, 36
- Виртуальное адресное пространство 31, 41
- Виртуальные адреса 305
- Виртуальная машина См. Virtual machine
- Восстановление синхронизации 359
- Время задержки 364
- Вставка конвейерных регистров 372
- Вызываемая программа 203
- Вызываемая процедура 205
- Вызывающая программа 203
- Вызывающая процедура 205
- Выравнивание данных 234

Г

- Генерирование оптимизированного кода 441
- Глобальные имена См. Global symbol

Д

- Дамп объектной программы 155
- Двухуровневая оптимизация 151
- Дескриптор сокета 907
- Диапазон виртуальных адресов 46
- Динамическая память См. Heap
- Динамически связанные библиотеки См. DLL

З

- Закон Эмдала 512.

И

- Идентификатор:

 - ◊ процесса См. PID
 - ◊ регистра 354

- Идиомы языка ассемблера 285
- Инициализация цикла 194
- Использование псевдонимов памяти 441

К

- Кодовая оптимизация 439
- Команда:

 - ◊ пор 156
 - ◊ перехода 180, 306
 - ◊ сравнения величин с плавающей запятой 263

- Компилятор 151, 154
- Конвейерная диаграмма 365
- Конвейерное проектирование 418
- Конвейерные регистры 368, 371
- Конвейерный процессор 381
- Контекстное переключение 29, 707
- Конфигурация восстановления 404
- Кэш:

 - ◊ данных 461
 - ◊ команд 459
 - ◊ прямого отображения 561
 - ◊ -память 27

Л

- Линус Торвальдс 33
- Логическая операция "И" 56
- Логическая операция "Исключающее ИЛИ" 56

Логическая операция "НЕ" 56
 Логический поток 682
 Локальные имена См. Local symbol
 Локальные переменные процедур 152

М

Медленные системные вызовы 716
 Методика избегания рисков 383
 Многозадачный режим 683
 Модульная арифметика 58
 Мультиплексирование ввода-вывода 1012
 Мультиплексор 321

О

Области видимости 620
 Обработка исключительных ситуаций 412
 Обработчик исключительных ситуаций 674
 Объединения 229
 Объектный код 154
 Операции:
 ◊ записи 349
 ◊ считывания 349
 ◊ тождественности 58
 Оптимизация компилятора 440
 Остроконечник 48
 Ошибки страницы диска 416

П

Память для хранения данных 349
 Передвижение загрузки, методика 422
 Перелив данных 225
 Переместимые объектные файлы 659
 Перехват сигнала 714
 Печать числа с плавающей точкой 51
 Подкачка страниц См. Paging
 Понятие параллелизма 30
 Порт записи регистрового файла 348
 Потенциал ошибочных вычислений конвейером 379
 Поток:
 ◊ пакета 146
 ◊ управления 671
 Представление логики декодировки 395

Препроцессор 151
 Префикс конвейерного регистра 376
 Признак переноса См. CF
 Принцип конвейерной обработки 364
 Программы:
 ◊ логического синтеза 320
 ◊ на машинном уровне 304
 Проект Multics 28
 Проектирование конвейерного регистра 405
 Промах кэша 416
 Прямой доступ к памяти 23
 Псевдонимы памяти 457
 Пятиэтапные конвейеры 417

Р

Развертка цикла 443
 Разделяемые библиотеки 620, 650
 Разделяемые объектные файлы 659
 Разрешение ссылок 630
 Разыменование указателя 167
 Регистр кода условия 343, 348
 Регистровый файл 307, 328, 329, 349
 Регулярное кодирование 311
 Редактирование связей 619
 Редактор связей 152, 619
 Реентерабельные функции 1012
 Риски по данным 379

С

Синхронизированные регистры 328
 Система виртуальной памяти процессора 328
 Система команд Y86 305
 Скалярный тип данных 152
 Событийно-управляемые программы 1012
 Совместно используемый См. Shared
 Сокеты Беркли 929
 Спрогнозированное значение счетчика 394
 Ссылка:
 ◊ PIC 656
 ◊ на ячейку памяти 161
 Стандарт ANSI C 223
 Статический библиотечный файл 636
 Стековый фрейм 203
 Страницчная организация памяти 620

Стратегии прогнозирования выбора 377

Структура:

◊ PIPE – 374

◊ аппаратных средств 347

◊ мультиплексора 324

Суперскалярные процессоры 417

Сценарии проверок, процессор 419

Счетчик команд 22, 152, 304

Т

Тело:

◊ процедуры 168

◊ цикла 287

Трассировка См. trace

Тупоконечник 49

У

Увеличение времени задержки 364

Указатель:

◊ стека 203, 304

◊ фрейма 203

Унарные операции 169

Условие продолжения цикла 287

Устройство:

◊ выполнения См. EU

◊ операций с плавающей точкой
См. FPU

◊ формирования команд См. ICU

Ф

Файл подкачки См. swap file

Физические адреса 305

Функции системного уровня 687

Ц

Циклов на элемент См. CPE

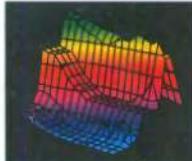
Э

Этап обратной записи 381

Эффективный конвейерный процессор
371

Я

Ядро Unix 28



КОМПЬЮТЕРНЫЕ СИСТЕМЫ АРХИТЕКТУРА И ПРОГРАММИРОВАНИЕ

Что говорят преподаватели об этой книге

"Материал освещается в книге так, как нигде больше, но так, как бы я хотел читать собственные лекции".

Джон Грайнер, Университет Райс

"Данный проект уникален в своем роде и имеет все шансы радикального изменения педагогических принципов в своей области".

Майкл Скотт, Университет Рочестера



Рэндал Э. Брайант, доктор наук, профессор, декан факультета информатики университета Карнеги-Меллона (г. Питтсбург). Автор более 100 технических работ. Результаты исследований профессора Брайанта используются ведущими производителями компьютерной техники, включая Intel, Motorola, IBM и Fujitsu. Является лауреатом многочисленных премий и наград. В течение 20 лет Р. Брайант читает курсы по архитектуре персонального компьютера, алгоритмам и программированию.



Дэвид Р. О'Халларон, доктор наук, профессор факультета вычислительной техники и электротехники университета Карнеги-Меллона (г. Питтсбург), где ведет курсы по компьютерной архитектуре, параллельному проектированию процессоров и др. Отмечен многочисленными наградами, в том числе медалью Алены Ньювелла (от университета Карнеги-Меллона) за исключительный вклад в исследования по компьютерной тематике.

За долгие годы преподавания Рэндал Брайант пришел к выводу, что программисты могут разрабатывать надежные программы только при более совершенном понимании всей компьютерной системы, под которой он понимает не только "стандартные элементы архитектуры", такие как центральный процессор, память, порты ввода-вывода и др., но также операционную систему, компилятор и сетевое окружение. Вместе с профессором О'Халлароном он разработал курс "Введение в компьютерные системы", положенный в основу данной книги и преподаваемый более чем в 90 университетах по всему миру.

Книга предназначена для программистов, которые стремятся к глубокому пониманию того, что происходит "под кожухом" системного блока при выполнении написанных ими приложений. Это позволяет, с одной стороны, разрабатывать компактный, надежный и эффективный программный код, а с другой – облегчает поиск и устранение возможных ошибок в работе программы.

В книге рассматриваются: представление данных и программ на машинном уровне, архитектура процессора, оптимизация программ, связывание, управление потоками, виртуальная память и управление памятью, ввод-вывод на системном уровне, сетевое и параллельное программирование. Описано, каким образом перечисленные выше аспекты необходимо учитывать программисту при разработке собственных приложений и систем. Например, при описании кэширования рассматривается, как организация циклов может влиять на производительность программы.

Не имея навыков программирования на ассемблере и языке C, можно получить академическое представление правил написания оптимального кода.

Приведенные в книге примеры для процессоров, совместимых с Intel (IA32), написаны на языке C и выполняются в операционной системе Unix или сходных, например Linux.

Полный набор ресурсов, включая лабораторные работы, выдержки из лекций и примеры кодов, представлены на сайте www.csapp.cs.cmu.edu.

ISBN 5-94157-433-9

БХВ-Петербург

194354, Санкт-Петербург,
ул. Есенина, 5б

E-mail: mail@bkhv.ru
Internet: www.bkhv.ru

Тел./факс: (812) 591-6243

