Оглавление

Предисловие от издательства 17
Вступление 18
Об авторах 34

Глава 1. Экскурс в компьютерные системы 36
1.1. Информация – это биты + контекст 38
1.2. Программы, которые переводятся другими программами в различные формы 39
1.3. Как происходит компиляция 41
1.4. Процессоры читают и интерпретируют инструкции, хранящиеся в памяти 42
1.4.1. Аппаратная организация системы 42
1.4.2. Выполнение программы hello 44
1.5. Различные виды кеш-памяти 46
1.6. Устройства памяти образуют иерархию 47
1.7. Операционная система управляет работой аппаратных средств 48
1.7.1. Процессы 49
1.7.2. Потоки 50
1.7.3. Виртуальная память 51
1.7.4. Файлы 53
1.8. Обмен данными в сетях 53
1.9. Важные темы 55
1.9.1. Закон Амдала 56
1.9.2. Конкуренция и параллелизм 57
1.9.3. Важность абстракций в компьютерных системах 60
1.10. Итоги 61
Библиографические заметки 61
Решения упражнений 61

Часть I
Структура программы и ее выполнение 63

Глава 2. Представление информации и работа с ней 64
2.1. Хранение информации 68
2.1.1. Шестнадцатеричная система счисления 68
2.1.2. Размеры данных 71
2.1.3. Адресация и порядок следования байтов 74
2.1.4. Представление строк 80
2.1.5. Представление программного кода 81
2.1.6. Введение в булеву алгебру 82
2.1.7. Битовые операции в C 85
2.1.8. Логические операции в C 87

2.1.9. Операции сдвига в C ... 88
2.2. Целочисленные представления ... 90
2.2.1. Целочисленные типы ... 91
2.2.2. Представление целых без знака ... 92
2.2.3. Представление в дополнительном коде ... 94
2.2.4. Преобразования между числами со знаком и без знака ... 99
2.2.5. Числа со знаком и без знака в C ... 104
2.2.6. Расширение битового представления числа ... 106
2.2.7. Усечение чисел ... 109
2.2.8. Советы по приемам работы с числами со знаком и без знака ... 111
2.3. Целочисленная арифметика ... 113
2.3.1. Сложение целых без знака ... 113
2.3.2. Сложение целых в дополнительном коде ... 118
2.3.3. Отрицание целых в дополнительном коде ... 123
2.3.4. Умножение целых без знака ... 124
2.3.5. Умножение целых в дополнительном коде ... 124
2.3.6. Умножение на константу ... 128
2.3.7. Деление на степень двойки ... 130
2.3.8. Заключительные размышления о целочисленной арифметике ... 134
2.4. Числа с плавающей точкой ... 135
2.4.1. Дробные двоичные числа ... 136
2.4.2. Представление значений с плавающей точкой в стандарте IEEE ... 139
2.4.3. Примеры чисел ... 141
2.4.4. Округление ... 146
2.4.5. Операции с плавающей точкой ... 148
2.4.6. Значения с плавающей точкой в C ... 150
2.5. Итоги ... 151
Библиографические заметки ... 152
Домашние задания ... 153
Правила представления целых чисел на битовом уровне ... 154
Правила представления чисел с плавающей точкой на битовом уровне ... 165
Решения упражнений ... 167

Глава 3. Представление программ на машинном уровне ... 184

3.1. Историческая перспектива ... 187
3.2. Программный код ... 190
3.2.1. Машинный код ... 190
3.2.2. Примеры кода ... 192
3.2.3. Замечание по форматированию ... 195
3.3. Форматы данных ... 197
3.4. Доступ к информации ... 198
3.4.1. Спецификаторы операндов ... 200
3.4.2. Инструкции перемещения данных ... 201
3.4.3. Примеры перемещения данных ... 205
3.4.4. Вталкивание данных в стек и выталкивание из стека ... 208
3.5. Арифметические и логические операции ... 209

3.5.1. Загрузка эффективного адреса 210
3.5.2. Унарные и бинарные операции 212
3.5.3. Операции сдвига 212
3.5.4. Обсуждение 213
3.5.5. Специальные арифметические операции 215
3.6. Управление 218
3.6.1. Флаги условий 218
3.6.2. Доступ к флагам 219
3.6.3. Инструкции перехода 222
3.6.4. Кодирование инструкций перехода 223
3.6.5. Реализация условного ветвления потока управления 225
3.6.6. Реализация условного ветвления потока данных 229
3.6.7. Циклы 235
3.6.8. Оператор switch 245
3.7. Процедуры 250
3.7.1. Стек времени выполнения 251
3.7.2. Передача управления 252
3.7.3. Передача данных 256
3.7.4. Локальные переменные на стеке 258
3.7.5. Локальные переменные в регистрах 260
3.7.6. Рекурсивные процедуры 262
3.8. Распределение памяти под массивы и доступ к массивам 264
3.8.1. Базовые принципы 264
3.8.2. Арифметика указателей 266
3.8.3. Вложенные массивы 267
3.8.4. Массивы фиксированных размеров 268
3.8.5. Массивы переменных размеров 271
3.9. Структуры разнородных данных 273
3.9.1. Структуры 273
3.9.2. Объединения 276
3.9.3. Выравнивание 279
3.10. Комбинирование инструкций управления потоком выполнения и передачи данных в машинном коде 282
3.10.1. Указатели 283
3.10.2. Жизнь в реальном мире: использование отладчика GDB 284
3.10.3. Ссылки на ячейки за границами выделенной памяти и переполнение буфера 286
3.10.4. Предотвращение атак методом переполнения буфера 290
3.10.5. Поддержка кадров стека переменного размера 295
3.11. Вычисления с плавающей точкой 298
3.11.1. Операции перемещения и преобразования данных 300
3.11.2. Операции с плавающей точкой в процедурах 305
3.11.3. Арифметические операции с плавающей точкой 305
3.11.4. Определение и использование констант с плавающей точкой 307
3.11.5. Поразрядные логические операции с числами с плавающей точкой 308
3.11.6. Операции сравнения значений с плавающей точкой 309

3.11.7. Заключительные замечания об операциях с плавающей точкой 312
3.12. Итоги 312
Библиографические заметки 313
Домашние задания 314
Решения упражнений 325

Глава 4. Архитектура процессора 349

4.1. Архитектура системы команд Y86-64 352
4.1.1. Состояние, видимое программисту 352
4.1.2. Инструкции Y86-64 353
4.1.3. Кодирование инструкций 355
4.1.4. Исключения в архитектуре Y86-64 360
4.1.5. Программы из инструкций Y86-64 361
4.1.6. Дополнительные сведения об инструкциях Y86-64 366
4.2. Логическое проектирование и язык HCL 368
4.2.1. Логические вентили 368
4.2.2. Комбинационные цепи и булевы выражения в HCL 369
4.2.3. Комбинационные цепи для слов и целочисленные выражения в HCL 371
4.2.4. Принадлежность множеству 375
4.2.5. Память и синхронизация 375
4.3. Последовательные реализации Y86-64 (SEQ) 378
4.3.1. Организация обработки в несколько этапов 378
4.3.2. Аппаратная реализация последовательной архитектуры SEQ 387
4.3.3. Синхронизация в последовательной реализации SEQ 391
4.3.4. Реализация этапов в последовательной версии SEQ 394
4.4. Общие принципы конвейерной обработки 402
4.4.1. Вычислительные конвейеры 402
4.4.2. Подробное описание работы конвейера 404
4.4.3. Ограничения конвейерной обработки 406
4.4.4. Конвейерная обработка с обратной связью 408
4.5. Конвейерные реализации Y86-64 409
4.5.1. SEQ+: переупорядочение этапов обработки 409
4.5.2. Добавление конвейерных регистров 411
4.5.3. Переупорядочение сигналов и изменение их маркировки 415
4.5.4. Прогнозирование следующего значения PC 416
4.5.5. Риски конвейерной обработки 418
4.5.6. Обработка исключений 431
4.5.7. Реализация этапов в PIPE 434
4.5.8. Управляющая логика конвейера 441
4.5.9. Анализ производительности 451
4.5.10. Незаконченная работа 454
4.6. Итоги 457
4.6.1. Имитаторы Y86-64 458
Библиографические заметки 458
Домашние задания 459
Решения упражнений 465

Глава 5. Оптимизация производительности программ 478
5.1. Возможности и ограничения оптимизирующих компиляторов 481
5.2. Выражение производительности программы 484
5.3. Пример программы 486
5.4. Устранение неэффективностей в циклах 490
5.5. Сокращение вызовов процедур 493
5.6. Устранение избыточных ссылок на память 495
5.7. Общее описание современных процессоров 498
5.7.1. Общие принципы функционирования 498
5.7.2. Производительность функционального блока 502
5.7.3. Абстрактная модель работы процессора 504
5.8. Развертывание циклов 510
5.9. Увеличение степени параллелизма 514
5.9.1. Несколько аккумуляторов 515
5.9.2. Переупорядочение операций 520
5.10. Обобщение результатов оптимизации комбинирующего кода 524
5.11. Некоторые ограничивающие факторы 525
5.11.1. Вытеснение регистров 525
5.11.2. Прогнозирование ветвлений и штрафы за ошибки предсказания 526
5.12. Понятие производительности памяти 530
5.12.1. Производительность операций загрузки 530
5.12.2. Производительность операций сохранения 531
5.13. Жизнь в реальном мире: методы повышения производительности 537
5.14. Выявление и устранение узких мест производительности 538
5.14.1. Профилирование программ 538
5.14.2. Использование профилировщика при выборе кода для оптимизации 540
5.15. Итоги 544
Библиографические заметки 545
Домашние задания 545
Решения упражнений 548

Глава 6. Иерархия памяти 553
6.1. Технологии хранения информации 554
6.1.1. Память с произвольным доступом 554
6.1.2. Диски 562
6.1.3. Твердотельные диски 572
6.1.4 Тенденции развития технологий хранения 574
6.2. Локальность 577
6.2.1. Локальность обращений к данным программы 577
6.2.2. Локальность выборки инструкций 579
6.2.3. В заключение о локальности 579
6.3. Иерархия памяти 581
6.3.1. Кеширование в иерархии памяти 582
6.3.2. В заключение об иерархии памяти 585
6.4. Кеш-память 586

6.4.1. Обобщенная организация кеш-памяти 586
6.4.2. Кеш с прямым отображением 588
6.4.3. Ассоциативные кеши 595
6.4.4. Полностью ассоциативные кеши 597
6.4.5. Проблемы с операциями записи 600
6.4.6. Устройство реальной иерархии кешей 601
6.4.7. Влияние параметров кеша на производительность 602
6.5. Разработка программ, эффективно использующих кеш 603
6.6. Все вместе: влияние кеша на производительность программ 608
6.6.1. Гора памяти 608
6.6.2. Переупорядочение циклов для улучшения пространственной локальности 612
6.6.3. Использование локальности в программах 615
6.7. Итоги 616
Библиографические заметки 616
Домашние задания 617
Решения упражнений 627

Часть II
Выполнение программ в системе 633

Глава 7. Связывание 634
7.1. Драйверы компиляторов 636
7.2. Статическое связывание 637
7.3. Объектные файлы 638
7.4. Перемещаемые объектные файлы 638
7.5. Идентификаторы и таблицы имен 640
7.6. Разрешение ссылок 643
7.6.1. Как компоновщик разрешает ссылки на повторяющиеся имена 644
7.6.2. Связывание со статическими библиотеками 648
7.6.3. Как компоновщики разрешают ссылки на статические библиотеки 651
7.7. Перемещение 652
7.7.1. Записи перемещения 653
7.7.2. Перемещение ссылок 654
7.8. Выполняемые объектные файлы 657
7.9. Загрузка выполняемых объектных файлов 659
7.10. Динамическое связывание с разделяемыми библиотеками 660
7.11. Загрузка и связывание с разделяемыми библиотеками из приложений 662
7.12. Перемещаемый программный код 665
7.13. Подмена библиотечных функций 668
7.13.1. Подмена во время компиляции 669
7.13.2. Подмена во время компоновки 670
7.13.3. Подмена во время выполнения 671
7.14. Инструменты управления объектными файлами 673
7.15. Итоги 673

Библиографические заметки...674
Домашние задания...674
Решения упражнений...677

Глава 8. Управление исключениями...680

8.1. Исключения...682
8.1.1. Обработка исключений...683
8.1.2. Классы исключений...685
8.1.3. Исключения в системах Linux/x86-64...687
8.2. Процессы...690
8.2.1. Логический поток управления...691
8.2.2. Конкурентные потоки управления...692
8.2.3. Изолированное адресное пространство...693
8.2.4. Пользовательский и привилегированный режимы...693
8.2.5. Переключение контекста...694
8.3. Системные вызовы и обработка ошибок...695
8.4. Управление процессами...696
8.4.1. Получение идентификатора процесса...697
8.4.2. Создание и завершение процессов...697
8.4.3. Утилизация дочерних процессов...701
8.4.4. Приостановка процессов...706
8.4.5. Загрузка и запуск программ...707
8.4.6. Запуск программ с помощью функций fork и execve...709
8.5. Сигналы...712
8.5.1. Терминология сигналов...714
8.5.2. Посылка сигналов...715
8.5.3. Получение сигналов...717
8.5.4. Блокировка и разблокировка сигналов...720
8.5.5. Обработка сигналов...721
8.5.6. Синхронизация потоков во избежание неприятных ошибок конкурентного выполнения...730
8.5.7. Явное ожидание сигналов...732
8.6. Нелокальные переходы...735
8.7. Инструменты управления процессами...739
8.8. Итоги...739
Библиографические заметки...740
Домашние задания...740
Решения упражнений...747

Глава 9. Виртуальная память...750

9.1. Физическая и виртуальная адресация...752
9.2. Пространства адресов...753
9.3. Виртуальная память как средство кеширования...754
9.3.1. Организация кеша DRAM...754
9.3.2. Таблицы страниц...755
9.3.3. Попадание в кеш DRAM...756
9.3.4. Промах кеша DRAM...757

9.3.5. Размещение страниц ... 758
9.3.6. И снова о локальности ... 759
9.4. Виртуальная память как средство управления памятью ... 759
9.5. Виртуальная память как средство защиты памяти ... 761
9.6. Преобразование адресов ... 762
9.6.1. Интегрирование кешей и виртуальной памяти ... 765
9.6.2. Ускорение трансляции адресов с помощью TLB ... 766
9.6.3. Многоуровневые таблицы страниц ... 767
9.6.4. Все вместе: сквозное преобразование адресов ... 769
9.7. Практический пример: система памяти Intel Core i7/Linux ... 773
9.7.1. Преобразование адресов в Core i7 ... 774
9.7.2. Система виртуальной памяти Linux ... 776
9.8. Отображение в память ... 780
9.8.1. И снова о разделяемых объектах ... 781
9.8.2. И снова о функции fork ... 783
9.8.3. И снова о функции execve ... 783
9.8.4. Отображение в память на уровне пользователя с помощью функции mmap ... 785
9.9. Динамическое распределение памяти ... 786
9.9.1. Функции malloc и free ... 787
9.9.2. Что дает динамическое распределение памяти ... 790
9.9.3. Цели механизмов распределения памяти и требования к ним ... 791
9.9.4. Фрагментация ... 792
9.9.5. Вопросы реализации ... 793
9.9.6. Неявные списки свободных блоков ... 794
9.9.7. Размещение распределенных блоков ... 796
9.9.8. Разбиение свободных блоков ... 796
9.9.9. Увеличение объема динамической памяти ... 797
9.9.10. Объединение свободных блоков ... 797
9.9.11. Объединение с использованием граничных тегов ... 798
9.9.12. Все вместе: реализация простого механизма распределения памяти ... 800
9.9.13. Явные списки свободных блоков ... 807
9.9.14. Раздельные списки свободных блоков ... 808
9.10. Сборка мусора ... 811
9.10.1. Основы сборки мусора ... 811
9.10.2. Алгоритм сборки мусора Mark&Sweep ... 813
9.10.3. Консервативный алгоритм Mark&Sweep для программ на C ... 814
9.11. Часто встречающиеся ошибки ... 815
9.11.1. Разыменование недопустимых указателей ... 815
9.11.2. Чтение неинициализированной области памяти ... 816
9.11.3. Переполнение буфера на стеке ... 816
9.11.4. Предположение о равенстве размеров указателей и объектов, на которые они указывают ... 816
9.11.5. Ошибки занижения или завышения на единицу ... 817
9.11.6. Ссылка на указатель вместо объекта ... 817

9.11.7. Неправильное понимание арифметики указателей 818
9.11.8. Ссылки на несуществующие переменные 818
9.11.9. Ссылка на данные в свободных блоках 818
9.11.10. Утечки памяти 819
9.12. Итоги 819
Библиографические заметки 820
Домашние задания 821
Решения упражнений 824

Часть III
Взаимодействие программ 829

Глава 10. Системный уровень ввода/вывода 830
10.1. Ввод/вывод в Unix 831
10.2. Файлы 832
10.3. Открытие и закрытие файлов 833
10.4. Чтение и запись файлов 835
10.5. Надежные чтение и запись с помощью пакета RIO 836
10.5.1. Функции RIO небуферизованного ввода/вывода 837
10.5.2. Функции RIO буферизованного ввода 838
10.6. Чтение метаданных файла 842
10.7. Чтение содержимого каталога 843
10.8. Совместное использование файлов 845
10.9. Переадресация ввода/вывода 848
10.10. Стандартный ввод/вывод 849
10.11. Все вместе: какие функции ввода/вывода использовать? 849
10.12. Итоги 851
Библиографические заметки 852
Домашние задания 852
Решения упражнений 853

Глава 11. Сетевое программирование 854
11.1. Программная модель клиент–сервер 854
11.2. Компьютерные сети 855
11.3. Всемирная сеть интернет 860
11.3.1. IP-адреса 861
11.3.2. Доменные имена интернета 863
11.3.3. Интернет-соединения 866
11.4. Интерфейс сокетов 867
11.4.1. Структуры адресов сокетов 868
11.4.2. Функция socket 869
11.4.3. Функция connect 869
11.4.4. Функция bind 870
11.4.5. Функция listen 870
11.4.6. Функция accept 870
11.4.7. Преобразование имен хостов и служб 872
11.4.8. Вспомогательные функции для интерфейса сокетов 876

11.4.9. Примеры эхо-клиента и эхо-сервера ... 879
11.5. Веб-серверы ... 881
11.5.1. Основные сведения о вебе ... 881
11.5.2. Веб-контент ... 882
11.5.3. Транзакции HTTP ... 884
11.5.4. Обслуживание динамического контента ... 886
11.6. Все вместе: разработка небольшого веб-сервера TINY ... 889
11.7. Итоги ... 896
Библиографические заметки ... 896
Домашние задания ... 897
Решения упражнений ... 898

Глава 12. Конкурентное программирование ... 901

12.1. Конкурентное программирование с процессами ... 903
12.1.1. Конкурентный сервер, основанный на процессах ... 904
12.1.2. Достоинства и недостатки подхода на основе процессов ... 905
12.2. Конкурентное программирование с мультиплексированием ввода/вывода ... 906
12.2.1. Конкурентный на основе мультиплексирования ввода/вывода, управляемый событиями ... 909
12.2.2. Достоинства и недостатки мультиплексирования ввода/вывода ... 913
12.3. Конкурентное программирование с потоками выполнения ... 914
12.3.1. Модель выполнения многопоточных программ ... 914
12.3.2. Потоки Posix ... 915
12.3.3. Создание потоков ... 916
12.3.4. Завершение потоков ... 916
12.3.5. Утилизация завершившихся потоков ... 917
12.3.6. Обособление потоков ... 917
12.3.7. Инициализация потоков ... 918
12.3.8. Конкурентный многопоточный сервер ... 918
12.4. Совместное использование переменных несколькими потоками выполнения ... 920
12.4.1. Модель памяти потоков ... 921
12.4.2. Особенности хранения переменных в памяти ... 921
12.4.3. Совместно используемые переменные ... 922
12.5. Синхронизация потоков выполнения с помощью семафоров ... 922
12.5.1. Граф выполнения ... 925
12.5.2. Семафоры ... 928
12.5.3. Использование семафоров для исключительного доступа к ресурсам ... 929
12.5.4. Использование семафоров для организации совместного доступа к ресурсам ... 930
12.5.5. Все вместе: конкурентный сервер на базе предварительно созданных потоков ... 935
12.6. Использование потоков выполнения для организации параллельной обработки ... 938
12.7. Другие вопросы конкурентного выполнения ... 944

12.7.1. Безопасность в многопоточном окружении 944
12.7.2. Реентерабельность 946
12.7.3. Использование библиотечных функций в многопоточных программах 947
12.7.4. Состояние гонки 948
12.7.5. Взаимоблокировка (тупиковые ситуации) 950
12.8. Итоги 953
Библиографические заметки 953
Домашние задания 954
Решения упражнений 958
Приложение А. Обработка ошибок 963
А.1. Обработка ошибок в системе Unix 963
А.2. Функции-обертки обработки ошибок 965
Библиография 968
Предметный указатель 975

Вступление

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

Целью авторов является попытка разъяснить устойчивые концепции, лежащие в основе всех компьютерных систем, а также демонстрация конкретных видов влияния этих идей на корректность, производительность и полезные свойства прикладных программ. Многие книги по компьютерным системам написаны *для создателей* таких систем и описывают, как скомпоновать оборудование или реализовать системное программное обеспечение, включая операционную систему, компилятор и сетевой интерфейс. Эта книга, напротив, написана *для программиста* и рассказывает, как прикладные программисты могут использовать свои знания о системах для создания более качественных программ. Конечно, знакомство с требованиями к системам является хорошим первым шагом в обучении их созданию, поэтому эта книга послужит также ценным введением для тех, кто продолжает заниматься созданием аппаратного и программного обеспечения систем. Большинство книг по компьютерным системам также имеют тенденцию сосредоточиваться только на одном аспекте системы, например на аппаратной архитектуре, операционной системе, компиляторе или сети. Эта книга охватывает все эти аспекты и рассматривает их с позиции программиста.

Доскональное изучение и освоение изложенных в книге концепций позволит читателю со временем превратиться в редкий тип профессионального программиста, понимающего саму суть происходящего и способного решить любую задачу. Вы сможете писать программы, которые эффективнее используют возможности операционной системы и системного программного обеспечения, действуют правильно в широком диапазоне рабочих условий и параметров, работают быстрее и не содержат уязвимостей для кибератак. При этом будет заложена основа для изучения таких специфических тем, как компиляторы, архитектура компьютерных систем, операционные системы, сети и кибербезопасность.

Что нужно знать перед прочтением

Эта книга посвящена системам с аппаратной архитектурой x86-64, являющиеся последним этапом на пути развития, который прошли Intel и ее конкуренты, начинавшие с микропроцессора 8086 в 1978 году. В соответствии с соглашениями об именовании, принятыми в Intel в отношении их линейки микропроцессоров, этот класс микропроцессоров в просторечии называется «x86». По мере развития полупроводниковых технологий, позволяющих размещать на одном кристалле все больше и больше транзисторов, производительность и объем внутренней памяти процессоров значительно увеличились. В ходе этого прогресса они перешли от 16-разрядных слов к 32-разрядным и выпустили процессор IA32, а совсем недавно произошел переход к 64-разрядным словам и появилась архитектура x86-64.

Мы рассмотрим, как машины с этой архитектурой выполняют программы на языке C в Linux. Linux – одна из операционных систем, ведущих свою родословную от операционной системы Unix, первоначально разработанной в Bell Laboratories. К другим членам этого класса операционных систем относятся Solaris, FreeBSD и MacOS X. В последние годы эти операционные системы сохраняли высокий уровень совместимости благодаря усилиям по стандартизации POSIX и Standard Unix Specification. То есть сведения, что приводятся в этой книге, почти напрямую применимы ко всем этим «Unix-подобным» операционным системам.

> **Еще незнакомы с C?** Совет по языку программирования C
>
> В помощь читателям, плохо знакомым или незнакомым с языком C, авторами предлагаются примечания, подобные данному, для подчеркивания функций, особенно важных для C. Предполагается, что читатели знакомы с C++ или Java.

В тексте содержится множество примеров программного кода, которые мы компилировали и опробовали в системах Linux. Мы предполагаем, что у вас есть доступ к такой системе, что вы можете входить в нее и умеете выполнять простые действия, такие как получение списка файлов или переход в другой каталог. Если ваш компьютер работает под управлением Microsoft Windows, то мы рекомендуем установить одну из множества виртуальных машин (например, VirtualBox или VMWare), которые позволяют программам, написанным для одной операционной системы (гостевой ОС), запускаться в другой (несущей ОС, или хост-ОС).

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 есть свои особенности (в частности, указатели, явное распределение динамической памяти и форматируемый ввод/вывод), которых нет в Java. К счастью, C – не очень сложный язык, он прекрасно описан в классической книге Брайана Кернигана и Денниса Ритчи [61]. Вне зависимости от «подкованности» читателя в области программирования, эта книга послужит ценным дополнением к его библиотеке. Если прежде вы использовали только интерпретируемые языки, такие как Python, Ruby или Perl, то вам определенно стоит посвятить некоторое время изучению C, прежде чем продолжить читать эту книгу.

В начальных главах книги рассматривается взаимодействие между программами на C и их аналогами на машинном языке. Все примеры на машинном языке были созданы с помощью компилятора GNU GCC на процессоре x86-64. Наличия какого бы то ни было опыта работы с аппаратными средствами, машинными языками или программирования в ассемблере не предполагается.

Как читать книгу

Изучение принципов работы компьютерных систем с точки зрения программиста – занятие очень увлекательное, потому что проходит в интерактивном режиме. Изучив что-то новое, вы тут же можете это проверить и получить результат, что называется, из первых рук. На самом деле авторы полагают, что единственным способом познания систем является их *практическое исследование*: 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 она будет сопровождаться иллюстрацией в форме одной или нескольких *практических задач*, которые нужно сразу же постараться решить, чтобы проверить правильность понимания изложенного. Решения упражнений приводятся в конце каждой главы. Пытайтесь сначала самостоятельно решать эти практические задачи и только потом проверяйте правильность выбранного пути. В конце каждой главы также представлены *домашние задания* различной степени сложности. Каждому домашнему заданию присвоен определенный уровень сложности:

- ♦ для решения достаточно нескольких минут; требуется минимальный объем программирования (или не требуется вообще);
- ♦♦ для решения потребуется до 20 минут. Часто нужно написать и опробовать программный код. Многие такие задания созданы на основе задач, приведенных в примерах;

- ♦♦♦ для решения потребуется приложить значительные усилия; по времени на решение может уйти до 2 часов. Как правило, для выполнения этих заданий требуется написать и опробовать значительный объем кода;
- ♦♦♦♦ лабораторная работа, на выполнение которой может уйти до 10 часов.

Все примеры кода в тексте книги отформатированы автоматически (без всякого ручного вмешательства) и получены из программ на C, скомпилированных с помощью GCC и протестированных в Linux. Конечно, в вашей системе может быть установлена другая версия gcc или вообще другой компилятор, генерирующий другой машинный код, но общее поведение примеров должно быть таким же. Весь исходный код доступен на веб-странице книги **csapp.cs.cmu.edu**. Имена файлов с исходным кодом документируются с использованием горизонтальных полос, окружающих отформатированный код. Например, программу из листинга 1 можно найти в файле `hello.c` в каталоге `code/intro/`. Мы советуем обязательно опробовать примеры программ у себя по мере их появления.

Листинг 1. Типичный пример кода

code/intro/hello.c

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("hello, world\n");
6     return 0;
7 }
```

code/intro/hello.c

Чтобы не раздувать объем и без того большой книги, мы создали несколько приложений, размещенных в интернете, которые содержат сведения, дополняющие основную книгу. Они отмечены в книге заголовками в форме «Приложение в интернете ТЕМА:ПОДТЕМА», где ТЕМА кратко описывает основную тему, а ПОДТЕМА – раздел темы. Например, «Приложение в интернете DATA:BOOL» содержит сведения по булевой алгебре, дополняющие описание представлений данных в главе 2, а «Приложение в интернете ARCH:VLOG» содержит описание приемов проектирования процессоров с использованием языка описания оборудования Verilog, дополняющее обсуждение конструкции процессора в главе 4. Все эти приложения доступны на веб-странице книги (**https://csapp.cs.cmu.edu/3e/waside.html**).

Обзор книги

Книга состоит из 12 глав, охватывающих основные принципы компьютерных систем:

Глава 1. Экскурс в компьютерные системы.

В этой главе описываются основные идеи и темы, относящиеся к компьютерным системам, на примере 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 и математические свойства арифметических операций. Для начинающих программистов часто оказывается откровением, что сложение (в дополнительном коде) или умножение двух положительных чисел может дать отрицательный результат. С другой стороны, арифметика дополнительного кода удовлетворяет многим алгебраическим свойствам целочисленной арифметики, благодаря чему компилятор может преобразовать операцию умножения на константу в последовательность сдвигов и сложений. Для иллюстрации принципов и применения булевой алгебры авторы используют поразрядные операции C. Формат IEEE с плавающей точкой описывается в терминах представления значений и математических свойств операций с плавающей точкой.

Абсолютное понимание компьютерной арифметики принципиально для создания надежных программ. Например, программисты и компиляторы не могут заменить выражение `(x < y)` на `(x - y < 0)` из-за возможности переполнения. Они не могут даже заменить его выражением `(-y < -x)` из-за асимметричности диапазона отрицательных и положительных чисел в дополнительном коде. Арифметическое переполнение является обычным источником ошибок программирования и уязвимостей в системе безопасности, однако мало в какой книге можно найти описание свойств компьютерной арифметики, сделанное с точки зрения самого программиста.

О чем рассказывается во врезках?

На протяжении всей книги вам будут встречаться подобные примечания. В них приводятся некоторые дополнительные сведения к обсуждаемой теме. Примечания преследуют несколько целей. Одни представляют собой небольшие исторические экскурсы. Например, как появились C, Linux и Internet? Другие разъясняют какие-либо понятия. Например, чем отличаются кеш, множество и блок? Третьи примечания описывают примеры «из жизни». Например, как ошибка в вычислениях с плавающей точкой уничтожила французскую ракету или какие геометрические и функциональные параметры имеют коммерческие жесткие диски. И наконец, некоторые примечания – это всего лишь забавные комментарии.

Глава 3. Представление программ на машинном уровне.

Авторы научат читать машинный код x86-64, созданный компилятором C. Здесь представлены основные шаблоны инструкций для различных управляющих структур, таких как условные операторы, циклы и операторы выбора. Также рассматривается реализация процедур, включая выделение места на стеке, условные обозначения использования реестров и передачу параметров. В главе рассматриваются различные структуры данных, например структуры, объединения и массивы, их размещение в памяти и доступ к ним. Здесь еще будет показано, как выглядят программы с точки зрения машины, что поможет понять распространенные уязвимости, такие как переполнение буфера, и шаги, которые программист, компилятор и операционная система могут предпринять для уменьшения этих угроз. Изучение данной главы поможет повысить профессиональный уровень, потому что при этом появится понимание, как компьютер воспринимает программы.

Глава 4. Архитектура процессора.

В этой главе описываются комбинаторные и последовательные логические элементы, после чего демонстрируется, как эти элементы можно объединить в

информационный канал, выполняющий упрощенный набор инструкций x86-64 с названием «Y86-64». Сначала будет рассматриваться однотактный тракт данных. Его архитектура проста, но не отличается высоким быстродействием. Затем будет представлено понятие *конвейерной обработки*, в которой различные шаги, необходимые для обработки инструкции, реализуются как отдельные этапы. В каждый конкретный момент этапы конвейера могут обрабатывать разные инструкции. Получившийся в результате пятиступенчатый процессорный конвейер намного более реалистичен. Управляющая логика процессора описывается с использованием простого языка описания аппаратных средств – HCL. Проекты аппаратного обеспечения, написанные на HCL, можно компилировать и объединять в симуляторы, а затем использовать для создания описания Verilog, пригодного для производства реального оборудования.

Глава 5. Оптимизация производительности программ.

В этой главе представлен ряд методов повышения производительности кода, при этом идея состоит в том, что программисты учатся писать код на C так, чтобы компилятор мог затем сгенерировать эффективный машинный код. Сначала рассматриваются преобразования, сокращающие объем работы, которую предстоит выполнить программе, и, следовательно, которые должны стать стандартной практикой при написании любых программ для любых машин. Затем мы перейдем к преобразованиям, повышающим степень параллелизма на уровне команд в сгенерированном машинном коде, чтобы повысить их производительность на современных «суперскалярных» процессорах. Для обоснования этих преобразований будет представлена простая модель работы современных процессоров и показано, как измерить потенциальную производительность программы с точки зрения критических путей с использованием графического представления программы. Вы будете удивлены, насколько можно ускорить программу, применив простые преобразования к коду на C.

Глава 6. Иерархия памяти.

Память является для программистов одной из самых «заметных» частей компьютерной системы. До этой главы читатели полагались на концептуальную модель памяти в форме одномерного массива с постоянным временем доступа. На практике память представляет собой иерархию запоминающих устройств разной емкости, стоимости и быстродействия. В главе рассматриваются разные типы памяти, такие как ОЗУ и ПЗУ, а также геометрические параметры и устройство существующих современных дисковых накопителей, организация этих запоминающих устройств в иерархию. Авторы показывают возможность иерархической организации посредством локальности ссылок. Данные идеи конкретизируются представлением уникального взгляда на систему памяти как на «гору памяти» со «скалами» временной локальности и «склонами» пространственной локальности. В заключение рассказывается, как повысить производительность программных приложений путем усовершенствования их временной и пространственной локальности.

Глава 7. Связывание.

В данной главе описывается статическое и динамическое связывание, включая такие понятия, как: перемещаемые и выполняемые объектные файлы, разрешение символов, перемещение, статические библиотеки, разделяемые библиотеки, перемещаемый код и подмена библиотечных функций (library interpositioning). Тема связывания редко рассматривается в книгах по компьютерным системам, но авторы решили включить ее в эту книгу по двум причи-

нам. Во-первых, некоторые типичные ошибки, с которыми сталкиваются программисты, как раз возникают на этапе связывания и особенно характерны для крупных программных пакетов. Во-вторых, объектные файлы, создаваемые компоновщиками, связаны с такими понятиями, как загрузка, виртуальная память и отображение памяти.

Глава 8. Управление исключениями.

В этой главе авторы отступают от однопрограммной модели, вводя общую концепцию потока управления исключениями (не совпадающего с обычным потоком управления путем ветвления в условных операторах и в точках вызова процедур). Мы рассмотрим примеры потоков управления исключениями, существующих на всех уровнях системы, от аппаратных исключений и прерываний низкого уровня до переключения контекста между параллельными процессами, внезапных изменений в потоке управления, вызванных передачей сигналов Linux, и нелокальных переходов в C, разрывающих стройную структуру стека.

В этой части книги будет представлено фундаментальное понятие *процесса* как абстракции выполняющейся программы. Здесь авторы расскажут, как работают процессы, как их создавать и как ими можно управлять из прикладных программ, и покажут, как прикладные программисты могут запустить несколько процессов с помощью системных вызовов Linux. По окончании этой главы вы сможете написать простую командную оболочку для Linux с поддержкой управления заданиями. Эта глава также станет первым знакомством с недетерминированным поведением, возникающим при параллельных вычислениях.

Глава 9. Виртуальная память.

Описывает представление системы виртуальной памяти с целью дать некоторое понимание ее особенностей и принципов работы. Здесь вы узнаете, как разные процессы, действующие одновременно, могут использовать один и тот же диапазон адресов, совместно использовать одни страницы памяти и иметь индивидуальные копии других. В этой главе также описываются вопросы, связанные с управлением виртуальной памятью и манипуляциями с ней. В частности, мы уделим большое внимание инструментам распределения памяти из стандартной библиотеки, таким как `malloc` и `free`. Обсуждение данной темы преследует несколько целей. Прежде всего оно подкрепляет концепцию о том, что пространство виртуальной памяти является всего лишь массивом байтов, который программа может разделить на блоки разного размера для хранения данных. Помогает понять последствия ошибок обращения к памяти в программах, такие как утечки и недействительные ссылки в указателях. Наконец, многие программисты реализуют свои инструменты распределения памяти, оптимизированные под требования и характеристики конкретного приложения. Эта глава в большей степени, чем любая другая, демонстрирует преимущества неразрывного освещения аппаратных и программных аспектов компьютерных систем. Книги о традиционных компьютерных архитектурах и операционных системах обычно представляют виртуальную память только с одной стороны.

Глава 10. Системный уровень ввода/вывода.

В этой главе рассматриваются основные концепции ввода/вывода в системе Unix, такие как файлы и дескрипторы. Авторы описывают совместное использование файлов, принципы работы переадресации ввода/вывода и доступ к метаданным файлов. Здесь также представлен пример разработки надежного пакета буферизованного ввода/вывода, прекрасно справляющегося с любопыт-

ным поведением подсистемы ввода/вывода, известным как *недостача*, когда библиотечные функции возвращают только часть ввода. В главе описывается стандартная библиотека ввода/вывода языка C и ее связь с подсистемой ввода/вывода в Linux с упором на ограничения стандартного ввода/вывода, делающие его непригодным для сетевого программирования. Вообще говоря, темы, охваченные в этой главе, служат основой для двух следующих глав, посвященных сетевому и параллельному программированию.

Глава 11. Сетевое программирование.

Сети являются своеобразными устройствами ввода/вывода для программ, объединяющими многие из понятий, описанных ранее: процессы, сигналы, порядок следования байтов, отображение памяти и динамическое распределение пространства запоминающих устройств. Сетевые программы также являются одними из первых кандидатов на применение приемов параллельного программирования, о котором рассказывается в следующей главе. Данная глава – лишь тонкий срез глобального предмета сетевого программирования, необходимый для создания простенького веб-сервера. Здесь будет представлена модель клиент–сервер, лежащая в основе всех сетевых приложений. Авторы представят взгляд программиста на сеть Интернет и покажут, как писать сетевые клиенты и серверы, используя интерфейс сокетов. И наконец, в главе будет представлен протокол HTTP и разработан простой веб-сервер.

Глава 12. Конкурентное программирование.

Эта глава описывает принципы конкурентного (параллельного) программирования на примере сетевого сервера. Авторы сравнивают и противопоставляют три основных механизма, используемых для создания конкурентных программ: процессы, мультиплексирование ввода/вывода и потоки выполнения – и показывают возможность их использования при создании серверов, способных обслуживать множество одновременных соединений. Здесь же описаны основные принципы синхронизации с использованием семафорных операций P и V, безопасность потоков выполнения и реентерабельность, а также состояние взаимоблокировки. Конкурентное программирование играет важную роль в большинстве сетевых приложений. Также в этой главе описывается конкурентное программирование на уровне потоков выполнения, что позволяет ускорить решение задач на многоядерных процессорах. Чтобы все ядра правильно и эффективно работали над одной вычислительной задачей, требуется тщательная координация потоков, выполняющихся конкурентно.

Что нового в этом издании

Первое издание этой книги было опубликовано в 2003 году, а второе – в 2011. Учитывая быстрое развитие компьютерных технологий, содержимое книги на удивление хорошо сохранило свою актуальность. Принципиальное устройство компьютеров Intel x86, на которых выполняются программы на C под управлением Linux (и других похожих операционных систем), мало изменилось за эти годы. Однако изменения в аппаратных технологиях, компиляторах, интерфейсах программных библиотек и накопленный опыт преподавания потребовали существенного пересмотра материала книги.

Самым большим изменением по сравнению со вторым изданием является переход с представления, основанного на сочетании IA32 и x86-64, к представлению, основанному исключительно на x86-64. Это смещение акцента повлияло на содержимое многих глав. Вот краткое перечисление основных значительных изменений.

Глава 1. Экскурс в компьютерные системы.

Мы переместили обсуждение закона Амдала из главы 5 в эту главу.

Глава 2. Представление информации и работа с ней.

В многочисленных отзывах читатели и рецензенты сообщают, что некоторые сведения в этой главе сложны для понимания. Поэтому мы постарались упростить форму подачи материала, поясняя моменты, которые обсуждаются в строгом математической стиле. Это позволит читателям сначала просмотреть математические детали, чтобы получить общее представление, а затем вернуться к подробному описанию.

Глава 3. Представление программ на машинном уровне.

Мы перешли от представления, основанного на комбинации IA32 и x86-64, к представлению только на основе x86-64. Мы также обновили примеры кода, генерируемые более свежими версиями gcc. Как результат глава претерпела существенные изменения, включая изменение порядка, в котором представлены некоторые концепции. Мы также добавили описание аппаратной поддержки вычислений с плавающей точкой. А для сохранения совместимости добавили приложение в интернете, описывающее машинный код для IA32.

Глава 4. Архитектура процессора.

Мы обновили описание архитектуры процессора, выполнив переход с 32-разрядной архитектуры на архитектуру с поддержкой 64-разрядных слов и операций.

Глава 5. Оптимизация производительности программ.

Мы обновили эту главу, отразив возможности последних поколений процессоров x86-64 в плане производительности. С введением большего количества функциональных модулей и более сложной логики управления разработанная нами модель производительности программ, основанная на представлении программ в форме потока данных, стала предсказывать производительность еще надежнее, чем раньше.

Глава 6. Иерархия памяти.

Мы обновили эту главу с учетом последних технологий.

Глава 7. Связывание.

Мы переписали эту главу, перейдя на архитектуру x86-64, расширили обсуждение использования глобальной таблицы смещений GOT и таблицы компоновки процедур PLT для создания перемещаемого кода и добавили новый раздел о мощном методе связывания, известном как *library interpositioning* (подмена библиотечных функций).

Глава 8. Управление исключениями.

Мы добавили более строгое описание обработчиков сигналов, включив функции, которые можно безопасно вызывать при обработке асинхронных сигналов, специальные рекомендации по написанию обработчиков сигналов и использование `sigsuspend` для приостановки обработчиков.

Глава 9. Виртуальная память.

Эта глава изменилась незначительно.

Глава 10. Системный уровень ввода/вывода.

Мы добавили новый раздел о файлах и иерархии файлов, но в остальном эта глава изменилась незначительно.

Глава 11. Сетевое программирование.

Мы представили новые методы протоколонезависимого и потокобезопасного сетевого программирования с использованием современных функций `getaddrinfo` и `getnameinfo`, пришедших на смену устаревшим и нереентерабельным функциям `gethostbyname` и `gethostbyaddr`.

Глава 12. Параллельное программирование.

Мы расширили обсуждение параллельного программирования, добавив потоки выполнения, использование которых позволяет программам быстрее решать свои задачи на многоядерных машинах.

Также мы добавили и пересмотрели множество практических и домашних заданий по всей книге.

Происхождение книги

Книга родилась из вводного курса, разработанного в университете Карнеги–Меллона (УКМ) осенью 1998 года и получившего название «15-213: Введение в компьютерные системы». С тех пор курс читается в каждом семестре, и в каждом семестре его слушают более 400 студентов, от второкурсников до аспирантов, самых разных специальностей. Данный курс стал основой для большинства других курсов по компьютерным системам в университете Карнеги–Меллона.

Идея этого курса заключалась в том, чтобы познакомить студентов с компьютерами, взглянув на них с другой стороны. 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 и умение программировать. Поэтому авторы данной книги решили начать знакомство с системами с точки зрения программиста и при выборе тем использовали следующий своеобразный фильтр: тема будет освещаться только в том случае, если она связана с производительностью, корректностью или с полезными свойствами пользовательских программ на C.

К примеру, исключены темы, связанные с аппаратными сумматорами и конструкцией шин. В курсе также имелись темы, посвященные машинному языку, но вместо подробного рассмотрения языка ассемблера мы предпочли сконцентрироваться на том, как компилятор транслирует конструкции языка C в машинный код, включая операции с указателями, циклы, вызовы процедур и возврат из них. Кроме того, мы решили более широко взглянуть на систему как на комплекс аппаратных и программных средств и включили в книгу такие темы, как связывание, загрузка, процессы, сигналы, оптимизация производительности, виртуальная память, ввод/вывод, а также сетевое и параллельное программирование.

Данный подход позволил сделать курс практичным, конкретным, наглядным и на редкость интересным для студентов. Ответная реакция с их стороны и со стороны коллег по факультету была незамедлительной и положительной, и авторы книги поняли, что преподаватели из других учебных заведений тоже смогут воспользоваться их наработками. Это и стало предпосылкой появления данной книги, написанной лекционным конспектом курса и которую мы теперь обновили, чтобы отразить изменения в технологиях и подходах к реализации систем.

Благодаря выходу новых изданий и переводам этой книги на разные языки она и многие ее варианты стали частью учебных программ по информатике и компьютерной инженерии в сотнях колледжей и университетов по всему миру.

Преподавателям: курсы на основе этой книги

Преподаватели могут использовать эту книгу для проведения нескольких видов курсов по компьютерным системам. В табл. 1 перечислены пять категорий таких курсов. Конкретный курс зависит от требований учебной программы, личного вкуса, а также опыта и способностей студентов. Курсы в табл. 1 перечислены слева направо в порядке близости ко взгляду программиста на систему. Вот их краткое описание.

УК. Курс «Устройство компьютеров» с традиционными темами, раскрытыми в нетрадиционном стиле. Охватываются такие традиционные темы, как логическая модель, архитектура процессора, язык ассемблера и системы памяти. При этом больше внимания должно уделяться программной стороне. Например, обсуждение данных должно быть связано с типами данных и операциями в программах на C, а обсуждение ассемблерного кода основываться не на рукописном машинном коде, а на сгенерированном компилятором C.

УК+. Курс УК с дополнительным упором на аппаратную сторону и производительность прикладных программ. По сравнению с УК, студенты, слушающие курс УК+, узнают больше об оптимизации кода и об улучшении производительности памяти своих программ на языке C.

ВКС. Базовый курс «Введение в компьютерные системы», разработанный для подготовленных программистов, которые понимают влияние оборудования, операционной системы и системы компиляции на производительность и правильность прикладных программ. Существенное отличие от УК+ – отсутствие охвата низкоуровневой архитектуры процессора. Вместо этого программисты учатся работать с высокоуровневой моделью современного процессора. Курс ВКС хорошо вписывается в 10-недельную четверть, но при необходимости может быть продлен до 15-недельного семестра, если проходить его в неторопливом темпе.

ВКС+. Базовый курс «Введение в компьютерные системы» с дополнительным охватом таких тем системного программирования, как ввод/вывод системного уровня, сетевое и параллельное программирование. В университете Карнеги–Меллона это семестровый курс, охватывающий все главы данной книги, кроме низкоуровневой архитектуры процессора.

СП. Курс «Системное программирование». Этот курс похож на ВКС+, но в нем не преподается оптимизация операций с плавающей запятой и производительности, а также уделяется больше внимания системному программированию, включая управление процессами, динамическое связывание, ввод/вывод на уровне системы, сетевое программирование и параллельное программирование. Преподаватели могут добавить информацию из других источников для обсуждения дополнительных тем, таких как демоны, управление терминалом и межпроцессные взаимодействия в Unix.

Как показывает табл. 1, эта книга дает студентам и преподавателям массу возможностей. Если вы хотите, чтобы ваши ученики познакомились с низкоуровневой архитектурой процессоров, то этот вариант доступен в курсах УК и УК+. С другой стороны, если вы хотите переключиться с текущего курса «Устройство компьютеров» на курс ВКС или ВКС+, но опасаетесь вносить радикальные изменения сразу, то можете организовать постепенный переход к ВКС. Вы можете начать с курса УК, который преподносит традиционные темы нетрадиционным способом, а освоившись с этим материалом – переходить к УК+ и в конечном итоге к ВКС. Если студенты не имеют опыта программирования на C (например, они программировали только на Java), то

вы можете потратить несколько недель на чтение лекций о C, а затем переходить к чтению курса УК или ВКС.

Таблица 1. Пять категорий курсов о компьютерных системах, основанных на книге «Компьютерные системы: архитектура и программирование». Курс ВКС+ – это курс 15-213 в университете Карнеги–Меллона.

Примечание: символ ⊙ означает частичный охват главы, как то: (1) только аппаратная часть; (2) без динамического распределения памяти; (3) без динамического связывания; (4) без представления данных с плавающей точкой

Глава	Тема	Курс				
		УК	УК+	ВКС	ВКС+	СП
1	Экскурс в компьютерные системы	•	•	•	•	•
2	Представление данных	•	•	•	•	⊙ (4)
3	Машинный язык	•	•	•	•	•
4	Архитектура процессора	•	•			
5	Оптимизация кода		•	•	•	
6	Иерархия памяти	⊙ (1)	•	•	•	⊙ (1)
7	Связывание			⊙ (3)	⊙ (3)	•
8	Управление исключениями			•	•	•
9	Виртуальная память	⊙ (2)	•	•	•	•
10	Ввод/вывод на уровне системы				•	•
11	Сетевое программирование				•	•
12	Параллельное программирование				•	•

Наконец, отметим, что курсы УК+ и СП могут образовать хорошую последовательность из двух семестров (или четверти и семестра). Или же можно подумать о чтении курса ВКС+ как состоящего из ВКС и СП.

Преподавателям: примеры лабораторных работ в классе

Курс ВКС+ в университете Карнеги–Меллона получил очень высокие оценки от студентов. Медианный балл 5,0/5,0 и средний балл 4,6/5,0 являются типичными оценками студентов курса. Основными достоинствами студенты называют забавные, увлекательные и актуальные лабораторные работы, которые доступны на веб-странице книги. Вот примеры лабораторных работ, которые поставляются с книгой.

Представление данных.

Эта лабораторная работа требует от студентов реализовать простые логические и арифметические функции с использованием строго ограниченного подмножества языка C. Например, они должны вычислить абсолютное значение числа, используя только битовые операции. Эта лабораторная работа помогает студентам понять представление типов данных в языке C на двоичном уровне и особенности побитовых операций.

Двоичные бомбы.

Двоичная бомба – это программа, которая передается студентам в виде скомпилированного файла. При запуске она предлагает пользователю ввести шесть разных строк. Если при вводе будет допущена ошибка, то бомба «взрывается» – выводит сообщение об ошибке и регистрирует событие на сервере оценки. Студенты должны «обезвредить» свои уникальные бомбы, дизассемблировав

программы и определив, как должны выглядеть эти шесть строк. Лабораторная работа учит студентов понимать язык ассемблера, а также заставляет их научиться пользоваться отладчиком.

Переполнение буфера.

Студенты должны изменить поведение двоичного выполняемого файла, используя уязвимость переполнения буфера. Эта лабораторная работа учит студентов осторожности обращения со стеком и показывает опасности кода, уязвимого для атак переполнения буфера.

Архитектура.

Некоторые домашние задания из главы 4 можно объединить в лабораторную работу, где студенты изменяют HCL-описание процессора, добавляя новые инструкции, изменяя политику прогнозирования ветвлений или добавляя и удаляя обходные пути и регистрируя порты. Полученные процессоры можно моделировать и запускать с помощью автоматических тестов, которые обнаруживают большинство возможных ошибок. Эта лабораторная работа позволяет студентам познакомиться с захватывающими аспектами проектирования процессоров, не требуя полного знания языков логического проектирования и описания оборудования.

Производительность.

Студенты должны оптимизировать производительность основных функций приложения, таких как свертка или транспонирование матриц. Эта лабораторная работа наглядно демонстрирует свойства кеш-памяти и дает студентам опыт низкоуровневой оптимизации программ.

Кеш.

Эта лабораторная работа является альтернативой лабораторной работе «Производительность». В ней студенты должны написать симулятор кеша общего назначения, а затем оптимизировать базовые функции программы транспонирования матрицы так, чтобы минимизировать количество промахов кеша. При этом мы используем инструмент Valgrind для трассировки реальных адресов.

Командная оболочка.

Студенты создают свою программу командной оболочки Unix с поддержкой управления заданиями, включая комбинации клавиш **Ctrl**+**C** и **Ctrl**+**Z**, а также команды `fg`, `bg` и `jobs`. В этой лабораторной работе учащиеся впервые знакомятся с параллельным программированием и получают четкое представление об управлении процессами в Unix, сигналах и их обработке.

Распределение памяти.

Студенты реализуют свои версии `malloc`, `free` и (необязательно) `realloc`. Эта лабораторная работа помогает студентам получить четкое представление о структуре и организации данных и требует от них оценки различных компромиссов между эффективностью потребления памяти и времени выполнения.

Прокси.

Студенты реализуют параллельный веб-прокси, находящийся между браузером и остальной частью Всемирной паутины. Эта лабораторная работа знакомит студентов с такими темами, как веб-клиенты и серверы, и связывает воедино многие концепции курса, такие как порядок следования байтов, файловый ввод/вывод, управление процессами, сигналы, обработка сигналов, отображение памяти, сокеты и параллельное выполнение. Студентам нравится видеть, как работают их программы, обслуживающие реальные веб-браузеры и веб-серверы.

Об авторах

Рэндал Э. Брайант (Randal E. Bryant) в 1973 г. получил степень бакалавра в Мичиганском университете, 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. Пять лет возглавлял факультет информатики и затем десять лет занимал пост декана этого же факультета. В настоящее время является профессором информатики в университете. Он также проводит встречи с представителями Департамента электротехники и вычислительной техники.

Вот уже 40 лет профессор Брайант преподает курсы по компьютерным системам как на уровне бакалавриата, так и на уровне магистратуры. За многие годы преподавания курсов компьютерной архитектуры он начал смещать акцент с проектирования компьютеров к тому, как программисты могли бы писать более эффективные и надежные программы, более полно понимая системы. Вместе с профессором О'Холлароном разработал в университете Карнеги–Меллон учебный курс 15-213 «Введение в компьютерные системы», легший в основу данной книги. Также преподавал курсы по алгоритмам, программированию, компьютерным сетям, распределенным системам и проектированию СБИС (сверхбольших интегральных схем).

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 150 технических работ. Результаты исследований профессора Брайанта используются ведущими производителями компьютерной техники, включая Intel, IBM, Fujitsu и Microsoft. Является лауреатом многих наград за исследования, в числе которых две награды за изобретения, а также награда за технические достижения от Semiconductor Research Corporation (SRC), премия Канеллакиса за теоретические и практические исследования от Association for Computer Machinery (ACM), премия Бейкера (W. R. G. Baker Award), премия Эммануэля Пиора (Emmanuel Piore Award), премия Фила Кауфмана (Phil Kaufman Award) и премия Ричарда Ньютона (A. Richard Newton Award) от Institute of Electrical and Electronics Engineers (IEEE). Является сотрудником как ACM, так и IEEE и членом Национальной академии США и Американской академии искусств и наук.

Дэвид Р. О'Холларон (David R. O'Hallaron) – профессор информатики, электротехники и вычислительной техники в университете Карнеги–Меллона. Получил докторскую степень в университете Вирджинии. С 2007 по 2010 год занимал должность директора Intel Labs в Питтсбурге.

В течение 20 лет преподавал курсы по компьютерным системам на уровне бакалавриата и магистратуры по таким темам, как компьютерная архитектура, введение в компьютерные системы, проектирование параллельных процессоров и сетевые службы. Вместе с профессором Брайантом разработал курс в университете Карнеги–Меллона, который привел к созданию этой книги. В 2004 году был награжден премией Герберта Саймона (Herbert Simon Award) за выдающиеся успехи в преподавании от школы компьютерных наук CMUSchool of Computer Science, лауреаты которой выбираются на основе опросов студентов.

Профессор О'Халларон занимается исследованиями в области компьютерных систем, уделяя особое внимание программным системам для научных вычислений, вычислений с интенсивным использованием данных и виртуализации. Одной из самых известных примеров его работ является проект Quake, в котором участвовала группа специалистов по информатике, инженеров-строителей и сейсмологов. Все вместе они реализовали возможность предсказывать движение земной коры во время сильных землетрясений. В 2003 году профессор О'Халларон и другие члены команды Quake получили премию Гордона Белла (Gordon Bell Prize) – высшую международную награду в области высокопроизводительных вычислений. В настоящее время он работает над проблемой автогрейдинга, то есть над программами, способными оценивать качество других программ.

Глава 1

Экскурс в компьютерные системы

1.1. Информация – это биты + контекст.
1.2. Программы, которые переводятся другими программами в различные формы.
1.3. Как происходит компиляция.
1.4. Процессоры читают и интерпретируют инструкции, хранящиеся в памяти.
1.5. Различные виды кеш-памяти.
1.6. Устройства памяти образуют иерархию.
1.7. Операционная система управляет работой аппаратных средств.
1.8. Обмен данными в сетях.
1.9. Важные темы.
1.10. Итоги.
Библиографические заметки.
Домашние задания.
Решения упражнений.

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, желающих повысить свою квалификацию за счет лучшего понимания того, как эти компоненты работают и какое влияние они оказывают на правильное функционирование их программ.

Вам предстоит увлекательное путешествие. Если вы не пожалеете времени для изучения идей, изложенных 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 ее влияния на ваши прикладные программы.

Вы приобретете специальные навыки, например будете знать, как избежать странных числовых ошибок, вызванных особенностями представления чисел конкретным

компьютером. Узнаете, как оптимизировать программный код на языке C путем применения специальных приемов, которые используют особенности современных процессоров и систем памяти. Получите представление о том, как компилятор реализует вызовы процедур и как использовать эти знания, чтобы избежать прорех в системе защиты, вызванных сбоями, возникающими в результате переполнения буферов, которые мешают работе сетевого программного обеспечения. Научитесь распознавать и избегать неприятных ошибок во время связывания, которые приводят в замешательство программистов средней руки. Узнаете, как написать свою командную оболочку, свой пакет процедур динамического распределения памяти и даже свой собственный веб-сервер. Познакомитесь с перспективами и подводными камнями параллельного выполнения – с темой, которая приобретает все большее значение с распространением процессоров, имеющих несколько ядер.

Происхождение языка программирования C

Язык C разрабатывался с 1969 года по 1973 год Деннисом Ритчи (Dennis Ritchie), сотрудником Bell Laboratories. Национальный институт стандартизации США (American National Standards Institute, ANSI) утвердил стандарт ANSI C в 1989 году. Этот стандарт дает определение языка программирования C и набора библиотечных функций, известных как стандартная библиотека C. Керниган и Ритчи описали язык в своей классической книге, которую в кругах программистов любовно называют не иначе как «K&R» [61]. По словам Ритчи [92], язык C – это «причудливый, порочный и в то же время безусловный успех». Так все-таки почему успех?

- *Язык C был тесно связан с операционной системой Unix*. Он разрабатывался с самого начала как язык системного программирования для Unix. Большая часть ядра, а также все вспомогательные инструменты и библиотеки были написаны на C. По мере роста популярности Unix во второй половине семидесятых и в начале восьмидесятых годов прошлого столетия многим пришлось столкнуться с языком C, и многим он понравился. Поскольку система Unix была практически полностью написана на C, она легко переносилась на новые машины, а это, в свою очередь, увеличивало круг пользователей и самого языка C, и операционной системы Unix.
- *C – простой, компактный язык*. Его разработкой занимался один человек, а не многочисленный комитет, и результатом явился четкий непротиворечивый язык с небольшим бременем прошлого. В книге K&R дается полное описание языка и стандартной библиотеки, приводятся многочисленные примеры и упражнения, и на это потребовалось всего 261 страница. Простота языка C облегчает его изучение и перенос на разные компьютеры.
- *C разрабатывался для решения практических задач*. Первоначально его целью была реализация операционной системы Unix. Потом обнаружилось, что на нем можно писать любые программы, потому что сам язык давал такую возможность.

C – это язык системного программирования, и в то же время в нем имеются все средства, обеспечивающие возможность написания прикладных программ. Он полностью удовлетворяет потребности некоторой части программистов, но все же подходит не для всех ситуаций. Указатели языка C часто оказываются причиной различных недоразумений и программных ошибок. Языку не хватает также прямой поддержки таких полезных абстракций, как классы, объекты и исключения. Более новые версии этого языка, такие как C++ и Java, позволяют решать подобного рода проблемы и для программ прикладного уровня.

В своих, ставших классическими книгах по программированию на языке C [61] Керниган (Kernighan) и Ритчи (Ritchie) начинают знакомить читателя с языком программирования на примере простой программы `hello` (которая всего лишь выводит текст приветствия), представленной в листинге 1.1. И хотя это очень простая программа, 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 вам понять, что происходит и как, когда вы запускаете программу `hello` в своей системе.

Листинг 1.1. Программа `hello` (источник: [60])

code/intro/hello.c

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("hello, world\n");
6     return 0;
7 }
```

code/intro/hello.c

Мы начнем изучение систем с того, что исследуем жизненный цикл программы `hello` от момента ее написания программистом до момента, когда она выполняется системой, выводит свое незатейливое послание и завершается. Двигаясь вперед по жизненному циклу этой программы, мы будем вводить основные понятия, терминологию и компоненты, вступающие в игру. В последующих главах мы подробнее остановимся на этих понятиях и идеях.

1.1. Информация – это биты + контекст

Наша программа `hello` начинает свой жизненный цикл как *исходная программа* (или *файл с исходным кодом*), которую программист создает с помощью текстового редактора и сохраняет в файле с именем `hello.c`. Исходный код программы представляет собой последовательность битов, каждый из которых принимает значение 0 или 1, организованных в 8-битные блоки, получившие название *байты*. Каждый байт в исходном коде представляет некоторый символ.

Большинство компьютерных систем представляют текстовые символы в стандарте ASCII (American Standard Code for Information Interchange – американский стандартный код обмена информацией), согласно которому каждый символ представляется уникальным однобайтным целым числом[1]. Например, в листинге 1.2 программа `hello.c` показана в кодах ASCII.

Листинг 1.2. Представление текстового файла `hello` 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  SP  p   r   i   n   t   f   (   "   h   e   l
 10  32  32  32  32 112 114 105 110 116 102  40  34 104 101 108

 l   o   ,   SP  w   o   r   l   d   \   n   "   )   ;  \n  SP
108 111  44  32 119 111 114 108 100  92 110  34  41  59  10  32

 SP  SP  SP  r   e   t   u   r   n   SP  0   ;  \n   }  \n
 32  32  32 114 101 116 117 114 110  32  48  59  10 125  10
```

[1] Для представления текстов на языках, отличных от английского, используются другие методы кодирования. См. врезку «Стандарт Юникода для представления текста» в главе 2.

Программа hello.c хранится в файле как последовательность байтов. Каждый байт принимает целочисленное значение, соответствующее некоторому символу. Например, первый байт имеет значение 35, которому соответствует символ «#». Второй байт имеет значение 105, которому соответствует символ «i», и т. д. Обратите внимание, что текстовая строка заканчивается невидимым символом *перевода строки* «\n», который представлен целым значением 10. Такие файлы, как hello.c, содержащие исключительно символы, называются *текстовыми файлами*, а все другие – *двоичными файлами*.

Представление файла hello.c иллюстрирует одну из фундаментальных идей. Вся информация в системе, включая файлы на дисках, программы и данные пользователей, хранящиеся в памяти, а также данные, передаваемые по сети, представляется в виде битовых блоков. Единственное, что отличает разные виды данных друг от друга, – контекст, в котором мы их рассматриваем. Например, в разных контекстах одна и та же последовательность байтов 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 вести себя непредсказуемым образом. Эта фундаментальная идея широко используется в главе 2.

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, чтобы запустить программу hello.c в системе, операторы на языке C должны быть преобразованы другими программами в некоторую последовательность инструкций на *машинном языке* низкого уровня. Эти инструкции затем упаковываются в *выполняемую объектную программу* и сохраняются в двоичном файле на диске. Объектные программы также называются *выполняемыми объектными файлами*.

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

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

Здесь драйвер компилятора GCC читает исходный файл hello.c и транслирует его в выполняемый объектный файл hello. Трансляция выполняется в четыре этапа, как показано на рис. 1.1. Совокупность программ, выполняющих эти четыре этапа (*препроцессор, компилятор, ассемблер и компоновщик*), называется *системой компиляции*.

- *Этап препроцессора* (или этап предварительной обработки). Препроцессор (cpp) изменяет исходную программу в соответствии с директивами, которые начинаются с символа «#». Например, директива #inciude <stdio.h> в строке 1 программы hello.c заставляет препроцессор прочитать содержимое системного заголовочного файла stdio.h и вставить его непосредственно в текст программы. В результате получается другая программа на языке C, обычно с расширением .i.
- *Этап компиляции*. Компилятор (ccl) транслирует текстовый файл hello.i в текстовый файл hello.s, который содержит *программу на языке ассемблера*. Эта программа включает следующее определение функции main:

```
1  main:
2    subq $8, %rsp
3    movl $.LC0, %edi
4    call puts
```

```
5       movl $0, %eax
6       addq $8, %rsp
7       ret
```

Каждый оператор в строках 2–7 этого определения точно описывает одну из инструкций низкоуровневого машинного языка в текстовой форме. Польза языка ассемблера прежде всего в том, что он представляет общий выходной язык для компиляторов разных языков высокого уровня. Например, компиляторы языка C и языка Fortran генерируют выходные файлы на одном и том же языке ассемблера.

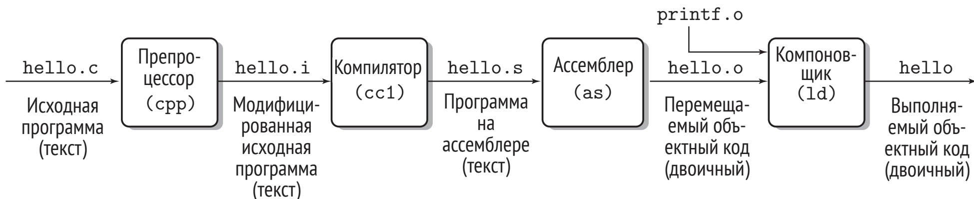

Рис. 1.1. Система компиляции

- *Этап ассемблирования*. Ассемблер (as) транслирует файл `hello.s` в машинные инструкции, упаковывает их в форму, известную как *перемещаемый объектный код*, и запоминает результат в объектном файле `hello.o`. Файл `hello.o` – это двоичный файл, содержащий 17 байт, которые кодируют машинные инструкции, составляющие функцию `main`. Если открыть `hello.o` в текстовом редакторе, то вы увидите совершенно непонятную абракадабру.
- *Этап компоновки*. Обратите внимание, что наша программа `hello` вызывает функцию `printf` из *стандартной библиотеки C*, которая поставляется в комплекте с любым компилятором языка C. Функция `printf` находится в отдельном предварительно скомпилированном объектном файле с именем `printf.o`, который тем или иным способом должен быть объединен с нашей программой `hello.o`. Это объединение осуществляет компоновщик (ld). В результате получается выполняемый объектный файл (или просто *выполняемый файл*), готовый к загрузке и выполнению системой.

О проекте GNU

GCC – один из множества полезных инструментов, созданных в рамках проекта GNU (сокращенно от «GNU's Not Unix» – «GNU – не Unix»). Проект GNU – это освобожденная от налогов благотворительная акция, основанная Ричардом Столлменом (Richard Stallman) в 1984 году с амбициозной целью разработать законченную Unix-подобную систему, исходный код которой не обременен ограничениями на его изменение и распространение. В рамках проекта GNU была разработана среда 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, компоновщик, утилиты для манипуляции двоичными файлами и другие компоненты. Компилятор GCC развился и ныне поддерживает множество разных языков и способен генерировать код для множества разных машин. В число поддерживаемых языков входят: C, C++, Fortran, Java, Pascal, Objective-C и Ada.

Проект GNU – замечательное достижение, однако сплошь и рядом ему не уделяют должного внимания. Современная мода на программные продукты с открытым исходным кодом (обычно ассоциируется с Linux) обязана своим интеллектуальным происхождением понятию свободное программное обеспечение, возникшему в рамках проекта GNU («свободное» – в смысле «свобода слова», но не в смысле «бесплатное пиво»). Более того, операционная система Linux во многом обязана своей популярности инструментальным средствам GNU, которые позволяют развернуть среду для ядра системы Linux.

1.3. Как происходит компиляция

В случае простых программ, таких как `hello.c`, мы можем рассчитывать, что система компиляции произведет правильный и эффективный машинный код. Однако есть несколько важных причин, почему программисты должны понимать, как работает система компиляции.

- *Оптимизация производительности программы*. Современные компиляторы – это сложные инструменты, которые обычно производят эффективный программный код. Однако мы, программисты, должны хотя бы немного понимать машинный код и знать, как компилятор транслирует разные инструкции на C, чтобы принимать осознанные решения при разработке своих программ. Например, всегда ли оператор `switch` является более эффективным, чем некоторая последовательность операторов `if-then-else`? Какие накладные расходы несет вызов функции? Является ли оператор `while` более эффективным, чем оператор `for`? Какой способ обращения к элементам массива эффективнее – по указателю или по индексам? Почему наш цикл выполняется намного быстрее, если накапливать сумму в локальной переменной вместо аргумента, который передается по ссылке? Можно ли ускорить функцию, если просто расставить круглые скобки в арифметическом выражении?

  В главе 3 мы представим машинный язык x86-64 последних поколений компьютеров с Linux, Macintosh и Windows. Там мы расскажем, как компиляторы транслируют различные конструкции языка C в этот язык. В главе 5 вы узнаете, как оптимизировать производительность своих программ, выполняя простые преобразования в коде на C, которые помогают компилятору лучше справляться со своей задачей. А в главе 6 вы будете изучать иерархическую природу системы памяти, узнаете, как компиляторы языка C хранят массивы данных и как можно использовать эти знания, чтобы ускорить работу программ на C.
- *Понимание ошибок времени компоновки*. Как показывает наш опыт, некоторые из наиболее запутанных программных ошибок порождаются компоновщиком, особенно при создании больших программных систем. Например, что означает сообщение компоновщика о том, что он не может разрешить ссылку? Чем отличаются статические и глобальные переменные? Что случится, если в разных файлах на C объявить две глобальные переменные с одинаковыми именами? Чем отличаются статические и динамические библиотеки? Почему важен порядок перечисления библиотек в командной строке? И самое неприятное: почему ошибки, источником которых является компоновщик, остаются незаметными до выполнения программы? Ответы на все эти вопросы вы получите в главе 7.
- *Как избежать прорех в системе защиты*. В течение многих лет *уязвимость*, порождаемая *переполнением буфера*, была причиной большей части проблем в сетевых серверах. Такие *уязвимости* существуют в силу того обстоятельства, что слишком мало программистов понимают необходимость тщательно ограничивать объемы и формы данных, которые поступают из ненадежных источников. Первым шагом в освоении приемов безопасного программирования является изучение особенностей хранения в программном стеке данных и управляющей информации. Мы расскажем, как правильно использовать стек и избежать уязвимостей, порождаемых переполнением буферов, в главе 3, когда приступим к изучению языка ассемблера. Там же мы познакомимся с методами, которые могут использовать программисты, компиляторы и операционные системы для снижения угрозы атаки.

1.4. Процессоры читают и интерпретируют инструкции, хранящиеся в памяти

Итак, наша исходная программа hello.c прошла через систему компиляции, которая преобразовала ее в выполняемый объектный файл с именем hello и сохранила на диске. Чтобы запустить выполняемый файл в системе Unix, нужно ввести его имя с клавиатуры в другой программе, известной как *командная оболочка*:

```
linux>./hello
hello, world
linux>
```

Командная оболочка – это интерпретатор командной строки, который выводит на экран приглашение к вводу, ждет, когда вы введете с клавиатуры команду, а затем выполняет ее. Если первое слово в команде не является именем какой-либо встроенной команды, то оболочка предполагает, что это слово является именем выполняемого файла, который она должна загрузить и выполнить. Поэтому в данном случае оболочка загружает и запускает программу hello, после чего ждет, когда та завершится. Программа hello выводит свое сообщение на экран и завершается, после чего оболочка выводит приглашение к вводу следующей команды и ждет, когда будет введена новая команда.

1.4.1. Аппаратная организация системы

Чтобы понять, что происходит с программой hello во время 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 отражает устройство современных систем Intel, однако примерно такое же устройство имеют все вычислительные системы. Пусть вас сейчас не беспокоит сложность этой блок-схемы – мы будем рассматривать различные ее детали на протяжении всей книги.

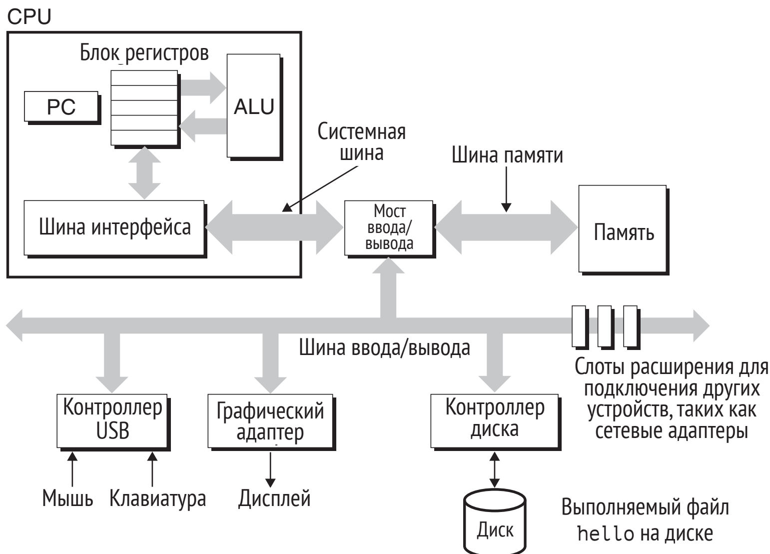

Рис. 1.2. Аппаратная организация типичной вычислительной системы. CPU: центральное процессорное устройство или просто процессор, ALU: арифметико-логическое устройство, PC (program counter): счетчик команд, USB (Universal Serial Bus): универсальная последовательная шина

Шины

Вычислительную систему пронизывает совокупность электрических проводников, так называемых *шин*, по которым байты информации циркулируют между компонентами системы. Обычно шины конструируются таким образом, чтобы по ним можно было передавать байты порциями фиксированного размера, которые называют *словами*. Число байтов в слове (*размер слова*) является одним из фундаментальных параметров, которые изменяются от системы к системе. В большинстве современных систем используются слова с размером 4 байта (32 бита) или 8 байт (64 бита). В этой книге мы будем использовать понятие «слово» без определения конкретного размера и уточнять его в случаях, когда это необходимо.

Устройства ввода/вывода

Устройства ввода/вывода – это средства связи с внешним миром. В нашем примере системы имеется четыре устройства ввода/вывода: клавиатура и мышь для ввода данных пользователем, устройство для отображения данных пользователю и дисковое устройство (или просто диск) для долгосрочного хранения данных и программ. В начальный момент выполняемый файл хранится на диске.

Каждое устройство ввода/вывода подключено к шине ввода/вывода посредством *контроллера* или *адаптера*. Различие между ними заключается в их конструктивных особенностях. Контроллеры – это платы с наборами микросхем, установленные в самом устройстве или на главной печатной плате (ее еще часто называют *материнской платой*). Адаптер – это плата, которая подключается через контактное гнездо на материнской плате. Независимо от конструкции таких устройств, их назначение заключается в том, чтобы передавать информацию между шиной и устройством ввода/вывода в обоих направлениях.

В главе 6 более подробно описана работа таких устройств ввода/вывода, как диск. В главе 10 вы узнаете, как пользоваться интерфейсом ввода/вывода системы Unix для доступа к устройствам из вашей прикладной программы. Мы же сосредоточим основное внимание на особо интересном классе устройств – сетевых адаптерах, принцип работы с которыми, впрочем, легко обобщить на любые другие устройства.

Основная память

Основная память – временное хранилище, в котором находятся сама программа и данные, которыми она манипулирует во время выполнения. Физически основная память состоит из совокупности микросхем *динамической памяти с произвольным доступом* (Dynamic Random Access Memory, DRAM). Логически основная память организована в виде линейного массива байтов, каждый из которых имеет свой уникальный адрес (индекс элемента массива); отсчет адресов начинается с нуля. Машинные инструкции, составляющие программу, могут состоять из разного числа байтов. Размер элементов данных, соответствующих переменным в программах на C, зависит от их типов. Например, на машине x86_64, работающей под Linux, тип данных `short` требует двух байт, типы `int` и `float` – четырех байт, а типы `long` и `double` – восьми байт.

В главе 6 мы более подробно рассмотрим, как работают технологии памяти, такие как DRAM, и как из них конструируется основная память.

Процессор

Центральный процессор (ЦП; Central Processing Unit, CPU), или просто процессор, – это механизм, который интерпретирует (или *выполняет*) инструкции, хранящиеся в основной памяти. Его ядро составляет устройство памяти с емкостью в одно слово (или *регистр*) – *счетчик команд* (Program Counter, PC). В любой конкретный момент времени он хранит адрес некоторой машинной инструкции в основной памяти[2].

[2] Аббревиатура PC также часто расшифровывается как Personal Computer (персональный компьютер). Различие между понятиями должно быть очевидно из контекста.

С момента включения системы и до ее выключения процессор снова и снова выполняет инструкцию, на которую указывает счетчик команд, и затем обновляет значение счетчика, выполняя переход к следующей инструкции. *Кажется*, что процессор работает в соответствии с очень простой моделью выполнения инструкций, определяемой его *архитектурным набором команд*. Согласно этой модели инструкции выполняются в строгой последовательности, а выполнение одной инструкции происходит в несколько этапов. Сначала процессор читает инструкцию из памяти по адресу в счетчике команд (PC), интерпретирует биты инструкции, выполняет простые операции, определяемые инструкцией, а затем обновляет значение счетчика, записывая в него адрес следующей инструкции, которая может размещаться за текущей или где-то в другом месте в памяти.

Существует всего несколько таких простых операций, и все они связаны с обслуживанием основной памяти, *блока регистров* и *арифметико-логического устройства* (Arithmetic/Logic Unit, ALU). Блок регистров – небольшое запоминающее устройство из совокупности регистров, каждый из которых имеет свое уникальное имя и может хранить одно слово. Устройство ALU вычисляет новые значения данных и адресов. Назовем лишь несколько примеров простых операций, которые процессор может выполнить по требованию той или иной инструкции:

- *загрузка*: копирует байт или слово из основной памяти в регистр, затирая при этом предыдущее содержимое этого регистра;
- *сохранение*: копирует байт или слово из регистра в некоторую ячейку основной памяти, затирая при этом предыдущее содержимое этой ячейки;
- *выполнение арифметико-логической операции*: копирует содержимое двух регистров в ALU, выполняет соответствующую операцию с этими словами и запоминает результат в одном из регистров, затирая при этом предыдущее содержимое этого регистра;
- *переход*: извлекает слово из самой инструкции и копирует его в счетчик команд (PC), затирая предыдущее его содержимое.

Мы говорим «кажется, что процессор работает в соответствии с очень простой моделью, определяемой его архитектурным набором», но на самом деле механика работы современных процессоров намного сложнее, чем было описано выше. Поэтому мы будем различать архитектурный набор команд процессора, определяющий эффект каждой машинной инструкции, и его микроархитектуру, определяющую фактическую реализацию процессора. В процессе знакомства с машинным кодом в главе 3 мы рассмотрим абстракцию архитектурного набора. В главе 4 вы узнаете больше о том, как в действительности устроены процессоры. В главе 5 мы опишем модель работы процессора с поддержкой предсказаний и оптимизации производительности программ на машинном языке.

1.4.2. Выполнение программы hello

Ознакомившись с простым описанием аппаратной организации и операций, мы начинаем понимать, что происходит, когда мы запускаем наш пример программы. Здесь мы должны опустить множество деталей, которые учтем позже, а пока нас вполне удовлетворит общая картина.

Сначала свои инструкции выполнит командная оболочка, ожидающая, когда мы введем команду с клавиатуры. Как только мы введем символы `./hello`, командная оболочка прочитает каждый из них в регистр и сохранит в основной памяти, как показано на рис. 1.3.

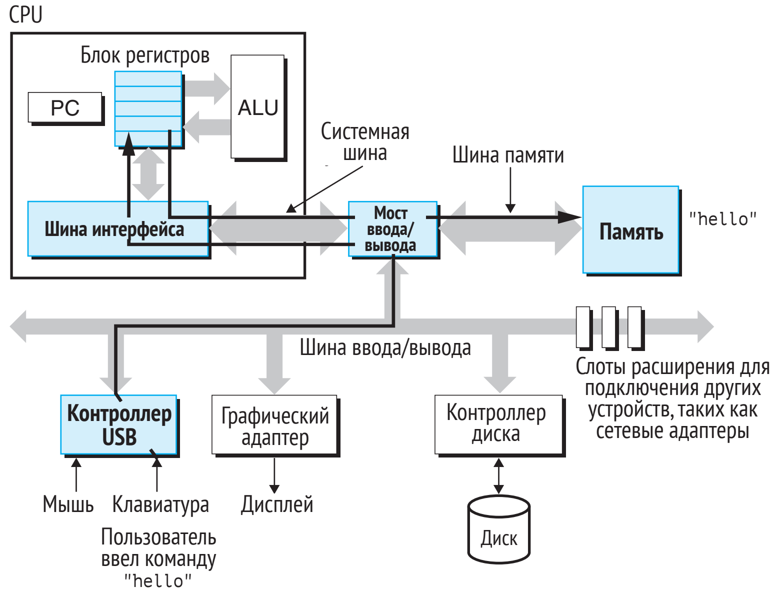

Рис. 1.3. Чтение команды hello с клавиатуры

Когда мы нажмем клавишу **Enter**, оболочка воспримет это как сигнал окончания ввода команды. После этого она загрузит выполняемый файл `hello`, осуществив последовательность инструкций, которая скопирует в основную память программные коды и данные, содержащиеся в объектном файле `hello` на диске. Данные включают строку символов `"hello, world\n"`, которая в конечном итоге будет выведена на экран.

Используя метод, известный как *прямой доступ к памяти* (Direct Memory Access, DMA; см. главу 6), данные перемещаются с диска непосредственно в оперативную память, минуя процессор. Этот шаг показан на рис. 1.4.

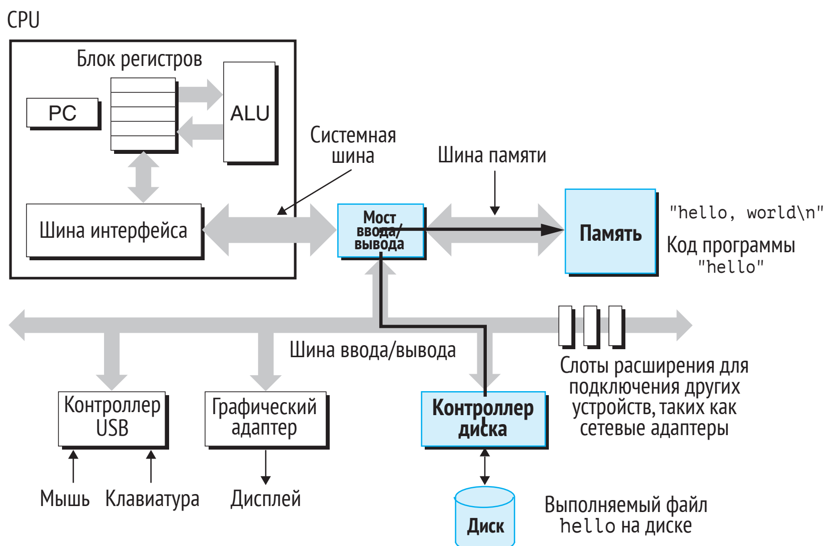

Рис. 1.4. Загрузка выполняемого файла в основную память

Как только код и данные из объектного файла `hello` будут загружены в память, процессор начинает выполнять машинные инструкции подпрограммы `main` в программе `hello`. Эти инструкции копируют байты строки `"hello, world\n"` из основной памяти в регистры, а оттуда – на дисплей, на экране которого они затем отображаются. Эти этапы показаны на рис. 1.5.

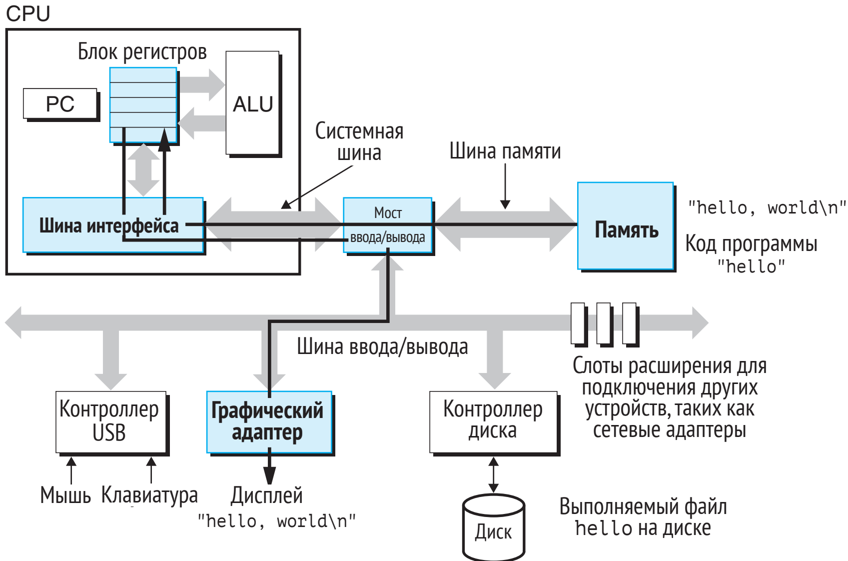

Рис. 1.5. Вывод строки из основной памяти на экран дисплея

1.5. Различные виды кеш-памяти

Важный урок из этого простого примера заключается в том, что система затрачивает уйму времени на перемещение информации из одного места в другое. Машинные инструкции в программе `hello` в начальный момент хранятся на диске. Когда производится загрузка программы, они копируются в основную память. По мере выполнения программы ее 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 на устройство отображения. С точки зрения программиста такой большой объем копирования ложится тяжким бременем на систему и существенно снижает «истинную производительность» программы. Следовательно, главная цель системных проектировщиков заключается в том, чтобы максимально ускорить операции копирования данных.

В силу физических законов чем больше запоминающее устройство, тем медленнее оно работает. И в то же время создание быстродействующих запоминающих устройств обходится дороже, чем более медленных. Например, емкость диска может оказаться в 1000 раз больше объема оперативной памяти, но, чтобы прочитать слово с диска, требуется в 10 млн раз больше времени, чем из основной памяти.

Аналогично типичный блок регистров может хранить лишь несколько сотен байтов информации, в то время как основная память – миллиарды. Однако процессор способен читать данные из регистров примерно в 100 раз быстрее, чем из основной памяти. Более того, по мере развития полупроводниковых технологий *расхождения в скорости доступа к регистрам и к основной памяти* продолжают углубляться. Увеличить быстродействие процессора намного проще, чем заставить основную память работать быстрее.

Чтобы уменьшить разрыв между процессором и основной памятью, создатели процессоров включили в них небольшие быстродействующие устройства хранения, получившие название *кеш-память* (или просто кеш) и служащие для временного хранения информации, которая, возможно, потребуется процессору в ближайшем будущем. На рис. 1.6 показана типичная система с кеш-памятью. *Кеш L1* (его еще называют кешем первого уровня) внутри процессора может хранить десятки тысяч байт, и доступ к ним осуществляется почти так же быстро, как к регистрам. Еще больший объем имеет кеш L2 (кеш второго уровня) – он может вместить от сотен тысяч до миллионов байт. Этот кеш связан с процессором специальной шиной. Скорость доступа к кешу L2 в 5 раз ниже скорости доступа к кешу L1, и все равно она в 5–10 раз выше скорости доступа к основной памяти. Кеши L1 и L2 построены по технологии, известной как *статическая память с произвольным доступом* (Static Random Access Memory, SRAM). Более новые и более мощные системы имеют три уровня кешей: L1, L2 и L3. Идея кеширования состоит в том, чтобы дать системе возможность получить положительный эффект от наличия большого объема памяти и очень короткого времени доступа за счет улучшения *локальности* – тенденции программ использовать данные и код, находящиеся в ограниченных областях памяти. Использование кешей для хранения данных, к которым программа, вероятно, будет часто обращаться, позволяет выполнять большинство операций с памятью, используя быстрые кеши.

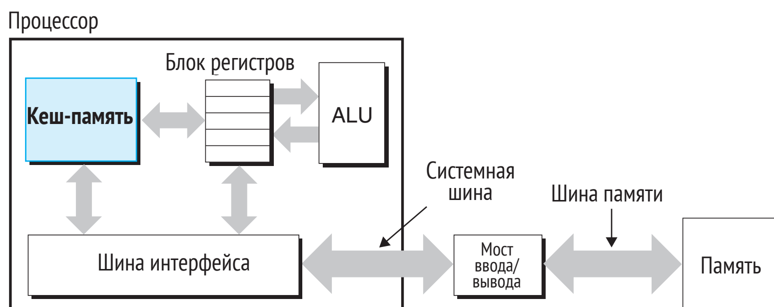

Рис. 1.6. Различные виды кеш-памяти

Один из самых важных уроков этой книги заключается в том, что прикладные программисты, которые знают о наличии кеш-памяти, могут воспользоваться ею и увеличить производительность своих программ на порядок. Мы будем изучать эти важные устройства и узнаем, как ими пользоваться, в главе 6.

1.6. Устройства памяти образуют иерархию

Идея поместить небольшое, зато более быстрое запоминающее устройство (кеш-память) между процессором и более емким, но с худшим быстродействием запоминающим устройством (основной памятью) оказалась весьма плодотворной. Фактически запоминающие устройства в любой вычислительной системе образуют *иерархию*, подобную изображенной на рис. 1.7. По мере движения по этой иерархии сверху вниз устройства становятся все медленнее, объемнее, а стоимость хранения одного байта уменьшается. Регистры находятся на вершине иерархии, которая обозначается как уровень 0 (L0). Кеш-память занимает уровни с 1 по 3 (L1–L3). Основная память находится на уровне 4 (L4) и т. д.

Основная идея иерархии памяти заключается в том, что память одного уровня служит кешем для следующего нижнего уровня. То есть блок регистров – это кеш для кеш-памяти L1. Кеши L1 и L2 служат кешами для кеш-памяти L2 и L3 соответственно.

Кеш L3 служит кешем для основной памяти, а та, в свою очередь, – кешем для диска. В некоторых сетевых системах с распределенными файловыми системами локальный диск служит кешем для данных, хранящихся на дисках других систем.

Рис. 1.7. Пример иерархии памяти

Подобно тому, как программисты используют знание структуры кешей LI 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 нашей программе `hello`. Когда оболочка загружала и запускала программу `hello` и когда программа `hello` отображала свое сообщение, ни та ни другая программа не обращалась напрямую к клавиатуре, диску или основной памяти. Для этого они пользовались услугами, предоставляемыми *операционной системой*. Операционную систему можно представить как некоторый слой программного обеспечения между прикладной программой и аппаратными средствами, как показано на рис. 1.8. Любые операции с аппаратными средствами прикладные программы должны выполнять через операционную систему.

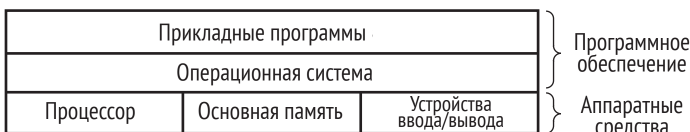

Рис. 1.8. Многослойная организация компьютерной системы

Операционная система прежде всего должна отвечать двум основным требованиям: (1) защищать аппаратные средства от катастрофических действий вышедшей из-под контроля программы и (2) предоставлять приложениям простые и единообраз-

ные механизмы манипулирования сложными и часто сильно отличающимися низкоуровневыми аппаратными средствами. Для этого операционная система предлагает набор фундаментальных абстракций, показанных на рис. 1.8: *процессы*, *виртуальную память* и *файлы*. Как видно по рис. 1.9, файлы являются абстракцией устройств ввода/вывода, виртуальная память является абстракцией как основной памяти, так и дисковых устройств ввода/вывода, а процессы – абстракцией процессора, основной памяти и устройств ввода/вывода. Обсудим эти абстракции по очереди.

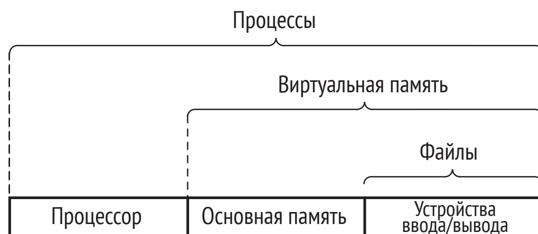

Рис. 1.9. Абстракции, предоставляемые операционной системой

1.7.1. Процессы

Когда программа, такая как `hello`, запускается в современной системе, операционная система создает иллюзию, что она – единственная программа, выполняющаяся в системе. С точки зрения программы создается впечатление, что только она распоряжается процессором, основной памятью и устройствами ввода/вывода. Процессор как бы выполняет 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, одной из наиболее важных и успешных идей в теории вычислительных машин и систем.

Процесс – это абстракция операционной системы, представляющая выполняемую программу. В одной и той же системе одновременно может выполняться множество процессов, и в то же время каждому процессу кажется, что только он пользуется аппаратными средствами. Под *одновременным* (или параллельным) выполнением мы понимаем поочередное выполнение инструкций то одного, то другого процесса. В большинстве систем существует больше процессов, выполняющихся одновременно, чем процессоров, на которых они выполняются.

Традиционные системы могут выполнять только одну программу в каждый конкретный момент времени, тогда как новые *многоядерные* процессоры способны одновременно выполнять программный код нескольких программ. В любом случае может показаться, что один процессор реализует несколько процессов одновременно, если будет переключаться между ними очень быстро. Операционная система проделывает такое чередование с помощью механизма *переключения контекста*. Чтобы упростить остальную часть этого обсуждения, мы будем рассматривать только *однопроцессорную систему* с единственным процессором. К обсуждению *многопроцессорных* систем мы вернемся в разделе 1.9.2.

Операционная система хранит всю информацию о состоянии, необходимую для правильного выполнения процесса. Состояние, известное как *контекст*, содержит такие сведения, как текущее значение счетчика команд PC, состояние блока регистров и содержимое основной памяти. В любой конкретный момент времени в системе выполняется только один процесс. Когда операционная система принимает решение передать управление некоторому другому процессу, она производит *переключение контекста*, запоминая контекст текущего процесса и восстанавливая контекст нового, после чего передает управление новому процессу. Новый процесс возобновляет выполнение точ-

но с того места, в котором он был прерван. Эту идею иллюстрирует рис. 1.10 на примере нашей программы `hello`.

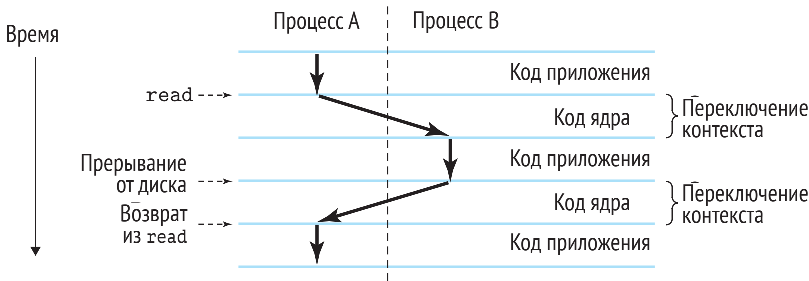

Рис. 1.10. Переключение контекстов процессов

В рассматриваемом нами примере существует два процесса: процесс командной оболочки и процесс `hello`. Первоначально система выполняет только один процесс, а именно процесс командной оболочки, которая ожидает ввода команды. Когда мы обращаемся к нему с требованием запустить программу `hello`, оболочка выполняет наше требование, вызывая специальную функцию, так называемый *системный вызов*, который передает управление операционной системе. Операционная система сохраняет контекст процесса оболочки, создает новый процесс `hello` с его контекстом, а затем передает управление новому процессу `hello`. После завершения `hello` операционная система восстанавливает контекст процесса оболочки и возвращает ему управление, а он затем ждет ввода следующей команды.

Как показано на рис. 1.10, переключение с одного процесса на другой производится *ядром* операционной системы. Ядро – это часть кода операционной системы, которая всегда находится в памяти. Когда прикладная программа требует от операционной системы какого-либо действия, например чтения из файла или записи в файл, она выполняет специальную инструкцию *системного вызова*, передавая управление ядру. Затем ядро выполняет запрошенную операцию и возвращает управление прикладной программе. Обратите внимание, что ядро – это не отдельный процесс, а блок кода и структур данных, которые система использует для управления всеми процессами.

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

1.7.2. Потоки

Обычно мы представляем себе процесс как единственный поток управления, однако в современных системах процесс может состоять из множества единиц выполнения, которые называют *потоками выполнения* (или нитями), каждый поток выполняется в контексте процесса, использует тот же программный код и глобальные данные, что и сам процесс. Роль потоков как программных моделей непрерывно возрастает в связи с требованиями к поддержке параллельных вычислений, предъявляемыми сетевыми серверами, поскольку организовать совместное использование данных несколькими потоками намного проще, чем несколькими процессами, а также в силу того, что потоки обычно значительно эффективнее процессов. Кроме того, многопоточность является одним из способов ускорить работу программ, когда в системе имеется несколько процессоров, о чем будет рассказываться в разделе 1.9.2. С базовыми понятиями параллельных вычислений, включая создание многопоточных программ, вы познакомитесь в главе 12.

Unix, Poslx и Standard Unix Specification

В шестидесятые годы прошлого столетия господствовали большие и 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 влачила 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 его чрезмерной сложности и ввиду отсутствия положительных результатов. Полученный при разработке системы отрицательный опыт сподвиг группу исследователей компании – Кена Томпсона (Ken Thompson), Денниса Ритчи (Dennis Ritchie), Дуга Макилроя (Doug Mcllroy) и Джо Оссанну (Joe Ossanna) – приступить в 1969 году к работе над более простой операционной системой для компьютера PDP-7 компании DEC, написанной исключительно на машинном языке. Многие из идей новой системы, такие как иерархическая файловая система и понятие командной оболочки как процесса пользовательского уровня, были заимствованы из системы Multics, но реализованы в виде более простого и компактного пакета программ. В 1970 году Брайан Керниган (Brian Kernighan) выбрал для новой системы название «Unix» как противовес названию «Multics», тем самым подчеркнув неповоротливость и тяжеловесность системы Multics (в некотором приближении Multics можно перевести как «многогранный», в том же контексте Unix можно перевести как «одногранный». – *Прим. перев.*). Ядро Unix было переписано на языке C в 1973 году, а сама операционная система была представлена широкой публике в 1974 году [93].

Поскольку компания Bell Labs предоставила высшим учебным заведениям исходные коды на очень выгодных условиях, у операционной системы Unix появилось множество сторонников 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 сетевые протоколы в последовательность выпусков, 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 оригинальных версий BSD и System V.

Осложнения возникли в середине восьмидесятых годов, когда производители операционной системы предприняли попытку выбрать собственные направления в развитии, добавляя новые свойства, часто нарушающие совместимость с прежними версиями. Чтобы пресечь 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, командные оболочки и утилиты, потоки и сетевое программирование. Запущенный вслед за этим отдельный проект по стандартизации, известный как стандартная спецификация Unix (Standard Unix Specification, SUS), объединил свои усилия с Posix для создания единого унифицированного стандарта систем Unix. В результате этого сотрудничества различия между разными версиями Unix почти исчезли.

1.7.3. Виртуальная память

Виртуальная память – это 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 прост-*

ранство. Виртуальное адресное пространство для процессов операционной системы Linux показано на рис. 1.11. (Другие Unix-системы используют ту же топологию.) В системе Linux верхняя часть адресного пространства резервируется для программного кода и данных операционной системы, которые являются общими для всех процессов. В нижней части адресного пространства находятся программный код и данные, принадлежащие процессам пользователей. Обратите внимание на тот факт, что адреса на схеме увеличиваются снизу вверх.

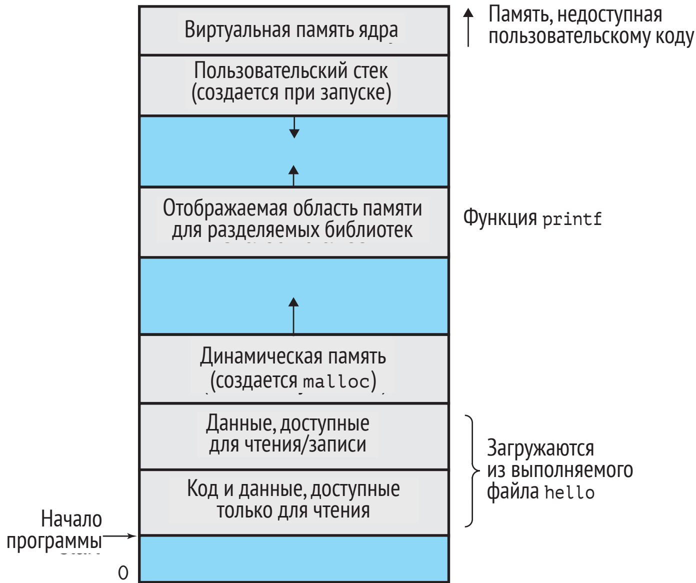

Рис. 1.11. Виртуальное адресное пространство процесса (области нарисованы без соблюдения масштаба)

Виртуальное адресное пространство, с точки зрения каждого процесса, состоит из некоторого числа четко определенных областей, каждая из которых выполняет свою функцию. Эти области будут подробно описаны далее в книге, тем не менее давайте кратко рассмотрим каждую из них прямо сейчас, начав с меньших адресов и продвигаясь в сторону их увеличения.

- *Программный код и данные*. Программный код каждого процесса всегда начинается с одного и того же адреса, за ним следуют ячейки памяти, соответствующие глобальным переменным. Область с кодом и данными инициализируется непосредственно содержимым выполняемого объектного файла, в нашем случае `hello`. Более подробно эта часть адресного пространства будет описана в главе 7, где мы исследуем связывание и загрузку.
- *Динамическая память (куча)*. Непосредственно за кодом и данными следует область *динамической памяти* (кучи) программы. В отличие от областей с программным кодом и глобальными данными, размеры которых фиксируются, как только процесс начнет выполняться, динамическая память может расширяться и сокращаться в размерах во время выполнения программы, при вызове некоторых функций из стандартной библиотеки C, таких как `malloc` и `free`. Мы продолжим подробное изучение динамической памяти после того, как в главе 9 рассмотрим вопросы управления виртуальной памятью.

- *Совместно используемые (разделяемые) библиотеки*. Ближе к середине адресного пространства находится область с программным кодом и данными *совместно используемых библиотек*, таких как стандартная библиотека C или библиотека математических функций. Понятие совместно используемой библиотеки является мощным, но в то же время несколько трудным для понимания. Вы узнаете, как с ними работать, когда мы приступим к изучению динамического связывания в главе 7.
- *Стек*. В верхней части виртуального адресного пространства находится *стек пользователя*, который применяется компилятором для реализации вызовов функций. Как и динамическая память, стек пользователя может динамически расширяться и сокращаться в размерах во время выполнения программы. В частности, каждый раз, когда программа вызывает какую-либо функцию, размер стека увеличивается. Каждый раз, когда функция возвращает управление, стек сокращается. В главе 3 вы узнаете, как компилятор использует стек.
- *Виртуальная память ядра*. Верхняя часть адресного пространства зарезервирована для ядра. Прикладным программам запрещено читать содержимое этой области, записывать в нее данные или напрямую вызывать функции ядра.

Для правильной работы виртуальной памяти требуется сложное взаимодействие аппаратных и программных средств операционной системы, включая аппаратное преобразование каждого адреса, генерируемого процессором. Основная идея заключается в том, чтобы сохранить содержимое виртуальной памяти процесса на диске, а затем использовать основную память как кеш диска. В главе 9 мы покажем вам, как работает этот механизм и почему это так важно для функционирования современных систем.

1.7.4. Файлы

Файл – это всего лишь последовательность байтов, не более и не менее. Каждое устройство ввода/вывода, в том числе диски, клавиатуры, устройства отображения и даже сети, моделируется соответствующим файлом. Все операции ввода/вывода в системе выполняются путем чтения и записи в файлы посредством нескольких системных вызовов, образующих *подсистему ввода/вывода Unix*.

Это простое и элегантное понятие файла, тем не менее, обладает глубоким смыслом, поскольку обеспечивает унифицированное представление всего разнообразия файлов, которые могут входить в состав системы. Например, прикладные программисты, манипулирующие содержимым дискового файла, могут быть абсолютно не знакомы с дисковыми технологиями и при этом чувствовать себя вполне комфортно. Более того, одна и та же программа будет прекрасно выполняться в разных системах, использующих разные дисковые технологии. Подсистема ввода/вывода Unix будет рассматриваться в главе 10.

1.8. Обмен данными в сетях

До этого момента в нашем экскурсе мы рассматривали систему как изолированную совокупность аппаратных и программных средств. На практике современные системы часто соединены с другими системами посредством компьютерных сетей. С точки зрения отдельной системы, сеть можно рассматривать как еще одно устройство ввода/вывода, как показано на рис. 1.12. Когда система копирует некоторую последовательность байтов из основной памяти в сетевой адаптер, поток данных устремляется через сеть в другую машину, а не, скажем, в локальный дисковый накопитель. Аналогично система может читать данные, отправленные другими машинами, и сохранять в своей основной памяти.

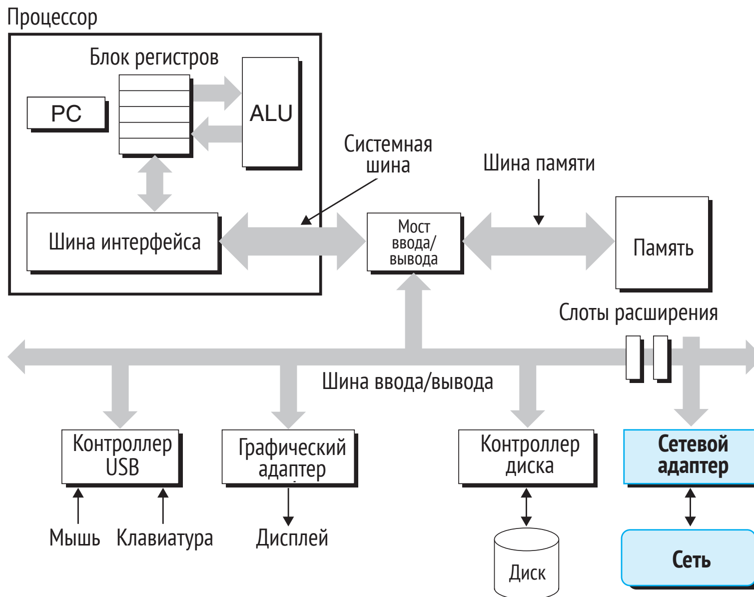

Рис. 1.12. Сеть – это еще одно устройство ввода/вывода

С пришествием глобальных сетей, таких как интернет, копирование информации между машинами стало одним из наиболее важных применений компьютерных систем. Например, такие приложения, как электронная почта, обмен мгновенными сообщениями, Всемирная паутина, FTP (File Transfer Protocol – протокол передачи файлов) и telnet, основаны на копировании информации по сети.

Возвращаясь к нашему примеру `hello`, мы можем воспользоваться знакомым приложением telnet, чтобы запустить программу `hello` на удаленной машине. Предположим, что мы решили воспользоваться *клиентом* на нашей машине для подключения к telnet-*серверу* на удаленной машине. После регистрации на удаленной машине запустится удаленная командная оболочка, которая будет ждать от нас ввода команд. С этого момента процесс дистанционного запуска программы `hello` требует выполнения пяти простых действий, представленных на рис. 1.13.

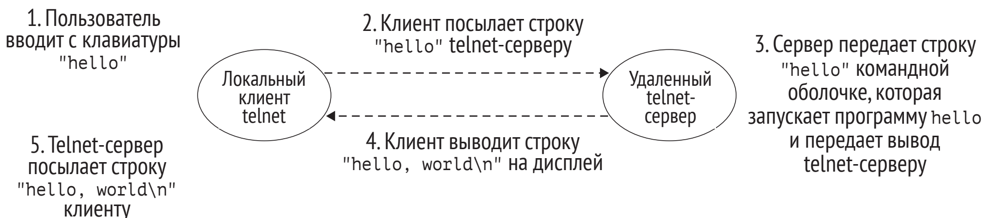

Рис. 1.13. Использование telnet для запуска программы `hello` на удаленной машине

После ввода строки `hello` на стороне клиента и нажатия клавиши **Enter** клиент отправит эту строку telnet-серверу. Когда сервер получит эту строку из сети, он передаст ее своей командной оболочке. Затем удаленная командная оболочка запустит программу `hello` и передаст выходную строку telnet-серверу. Наконец, telnet-сервер перешлет полученную строку клиенту по сети, а тот отобразит ее на локальном дисплее.

Такой тип обмена между клиентами и серверами характерен для всех сетевых приложений. В главе 11 вы узнаете, как создавать сетевые приложения, и примените полученные знания для создания простого веб-сервера.

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 телеконференция: 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 вызревал 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)
```

Как указывает Торвальдс, отправной точкой для создания Linux стала операционная система Minix, разработанная Эндрю С. Таненбаумом (Andrew S. Tanenbaum) в образовательных целях [113].

Все остальное, как говорится, уже история. Операционная система Linux превратилась в техническое и культурное явление. 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 на самых разных компьютерах, от карманных до мейнфреймов. Группа разработчиков компании IBM умудрилась впихнуть ее даже в наручные часы!

1.9. Важные темы

На этом мы завершаем наш начальный 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 действовать сообща для достижения окончательной цели – выполнения прикладных программ. Остальная часть книги представит некоторые дополнительные подробности об аппаратном и программном обеспечении и покажет, как, зная эти подробности, можно писать более быстрые, надежные и безопасные программы.

В заключение этой главы выделим несколько важных концепций, которые затрагивают все аспекты компьютерных систем. Мы будем отмечать их важность на протяжении всей книги.

1.9.1. Закон Амдала

Джин Амдал (Gene Amdahl), один из пионеров компьютерных вычислений, сделал простое, но далеко идущее наблюдение об эффективности повышения производительности одной части системы, которое стало известно как *закон Амдала*. Согласно этому закону, когда мы ускоряем одну часть системы, прирост общей производительности системы зависит не только от величины ускорения этой части системы, но и насколько значимой она является. Рассмотрим систему, в которой для выполнения некоторого приложения требуется время T_{old}. Предположим, что некоторая часть системы потребляет долю α этого времени, и мы повысили ее производительность в k раз. То есть компоненту изначально требовалось время αT_{old}, а теперь $(\alpha T_{old})/k$. Таким образом, общее время выполнения будет

$$T_{new} = (1 - \alpha)T_{old} + (\alpha T_{old})/k$$

$$= T_{old}[(1 - \alpha) + \alpha/k].$$

Отсюда прирост скорости $S = T_{old}/T_{new}$ можно вычислить как

$$S = \frac{1}{(1 - \alpha) + \alpha/k}. \quad (1.1)$$

Рассмотрим пример, когда часть системы, которая изначально потребляла 60 % времени ($\alpha = 0,6$), ускоряется в 3 раза ($k = 3$). В этом случае общее ускорение составит $1/[0,4 + 0,6/3] = 1,67\times$. Несмотря на значительное ускорение основной части системы, общее ускорение оказалось значительно меньше ускорения одной части. Это основная идея закона Амдала – чтобы значительно ускорить работу всей системы, нужно повысить скорость очень большой части всей системы.

Упражнение 1.1 (решение в конце главы)

Представьте, что вы работаете водителем грузовика и вас наняли для перевозки груза картофеля из города Бойсе, штат Айдахо, в город Миннеаполис, штат Миннесота, на расстояние 2500 километров. По вашим оценкам, вы ездите со средней скоростью 100 км/ч, то есть на выполнение рейса потребуется в общей сложности 25 часов.

1. Вы услышали в новостях, что в штате Монтана, на который приходится 1500 км пути, только что отменили ограничение скорости. Ваш грузовик может двигаться со скоростью 150 км/ч. Каким могло бы быть общее ускорение рейса?
2. Вы можете купить новый турбокомпрессор для своего грузовика на сайте **www.fasttrucks.com**. У них есть множество разных моделей, но чем эффективнее турбокомпрессор, тем дороже он стоит. Насколько быстро вы должны проехать штат Монтана, чтобы получить общее ускорение в 1,67 раза?

Упражнение 1.2 (решение в конце главы)

Отдел маркетинга вашей компании пообещал вашим клиентам, что производительность следующей версии программного обеспечения улучшится в 2 раза. Вам поручили выполнить это обещание. Вы определили, что можно ускорить только 80 % системы. Насколько (то есть какое значение k) вам нужно ускорить эту часть, чтобы достичь общего увеличения производительности в 2 раза?

Выражение относительной производительности

Лучший способ выразить увеличение производительности – это отношение вида T_{old}/T_{new}, где T_{old} – время, необходимое для выполнения в исходной версии системы, а T_{new} – время, необходимое для выполнения в измененной версии. Если улучшение действительно имеет место быть, то это число будет больше 1,0. Для обозначения таких отношений мы используем окончание «×», то есть запись «2,2×» читается как «в 2,2 раза».

Также часто используется более традиционный способ выражения относительного изменения в процентах, особенно в случаях, когда изменение небольшое, но определение этого способа неоднозначно. Как должен вычисляться процент? Как $100 \cdot (T_{old} - T_{new})/T_{new}$? Или, может быть, как $100 \cdot (T_{old} - T_{new})/T_{old}$? Или как-то иначе? К тому же этот способ менее наглядный для больших изменений. Понять фразу «производительность улучшилась на 120 %» сложнее, чем «производительность улучшилась в 2,2 раза».

Один интересный частный случай закона Амдала – рассмотреть эффект выбора k равным ∞. То есть можно взять некоторую часть системы и ускорить ее до такой степени, что на ее работу будет тратиться ничтожно мало времени. В результате получаем

$$S_{\infty} = \frac{1}{(1 - \alpha)}. \tag{1.2}$$

Так, например, если мы сможем ускорить 60 % системы до такой степени, что она практически не будет потреблять времени, то чистое ускорение все равно составит только 1/0,4 = 2,5×.

Закон Амдала описывает общий принцип ускорения любого процесса. Однако он может применяться не только к ускорению компьютерных систем, но также может помочь компании, пытающейся снизить стоимость производства бритвенных лезвий, или студенту, желающему улучшить свой средний балл. И все же он наиболее актуален в мире компьютеров, где производительность нередко улучшается в 2 или более раз. Таких высоких значений можно достичь только путем оптимизации больших частей системы.

1.9.2. Конкуренция и параллелизм

На протяжении всей истории развития цифровой вычислительной техники постоянными движущими силами, толкающими к совершенствованию, были два требования: компьютеры должны делать больше и работать быстрее. Оба этих параметра улучшаются, когда процессор одновременно может выполнять большее количество задач. Для обозначения общей идеи одновременного выполнения множества действий мы используем термин *конкуренция*, а для обозначения использования конкуренции для ускорения работы системы – термин *параллелизм*. Параллелизм может использоваться в компьютерной системе на нескольких уровнях абстракции. Мы выделяем здесь три уровня, от самого верхнего до самого нижнего в системной иерархии.

Конкуренция на уровне потоков

Основываясь на абстракции процессов, можно разрабатывать системы, в которых несколько программ выполняются одновременно, что приводит к *конкуренции*. Используя механизм потоков, можно даже запустить несколько потоков управления в рамках одного процесса. Поддержка конкурентного выполнения появилась в компьютерных системах с момента появления механизма разделения времени в начале 1960-х годов. В ту пору конкурентное выполнение только моделировалось – компьютер просто быстро переключался между выполняемыми процессами, подобно тому, как жонглер держит в воздухе сразу несколько шаров. Эта форма конкуренции позволяет нескольким пользователям одновременно взаимодействовать с системой, например получать

страницы с одного веб-сервера. Она также дает возможность одному пользователю одновременно выполнять несколько задач, например открыть веб-браузер в одном окне, текстовый процессор в другом и одновременно воспроизводить потоковую музыку. До недавнего времени в большинстве случаев все вычисления выполнялись одним процессором, даже если ему приходилось переключаться между несколькими задачами. Эта конфигурация известна как *однопроцессорная система*.

Когда в системе имеется несколько процессоров, все они управляются одним ядром операционной системы, и мы получаем *многопроцессорную систему*. Такие системы были доступны для крупномасштабных вычислений начиная с 1980-х годов, но с появлением *многоядерных* процессоров и технологии *гиперпоточности* они стали обычным явлением. На рис. 1.14 показана классификация этих различных типов процессоров.

Многоядерные процессоры состоят из нескольких процессоров (называемых «ядрами»), интегрированных на один кристалл. На рис. 1.15 показана организация типичного многоядерного процессора, имеющего в одной микросхеме четыре ядра, каждое со своими кешами L1 и L2, причем каждый кеш L1 разделен на две части: одна предназначена для хранения недавно выбиравшихся инструкций, а другая – данных. Ядра совместно используют кеш-память более высокого уровня (L3), а также интерфейс с основной памятью. Эксперты прогнозируют, что вскоре на одном кристалле будут размещаться десятки, а то и сотни ядер.

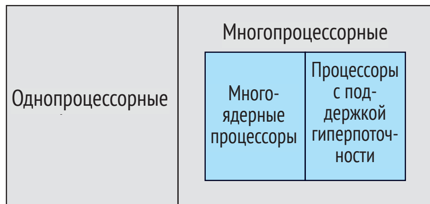

Рис. 1.14. Классификация систем в зависимости от количества процессоров. Многопроцессорные системы становятся все более распространенными с появлением многоядерных процессоров и технологии гиперпоточности

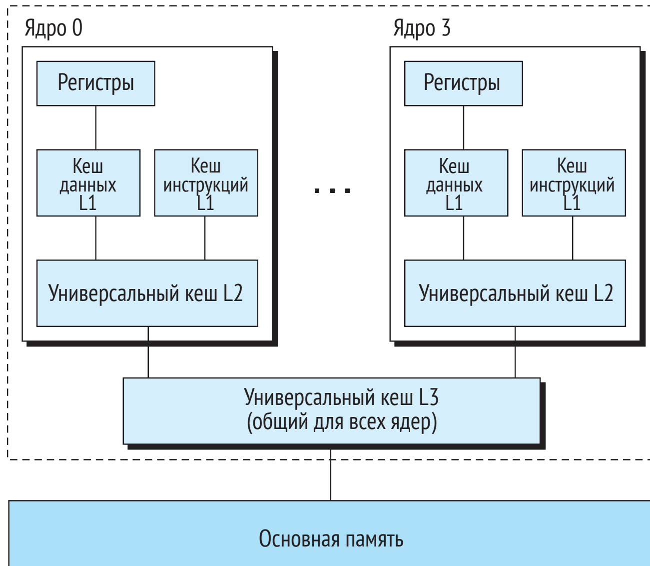

Рис. 1.15. Организация многоядерного процессора. Четыре процессорных ядра размещены на одном кристалле

Гиперпоточность, которую иногда называют *одновременной многопоточностью*, – это технология, позволяющая одному процессору выполнять сразу несколько потоков управления. Это предполагает наличие нескольких копий определенных аппаратных средств процессора, таких как счетчики инструкций и блоки регистров, при этом другие аппаратные компоненты, такие как блок арифметических операций с плавающей точкой, наличествуют в единственном числе. В отличие от обычного процессора, которому требуется около 20 000 тактов для переключения между потоками, гиперпотоковый процессор выбирает потоки для выполнения на циклической основе. Это позволяет процессору полнее использовать свои ресурсы. Например, если один поток должен дождаться загрузки некоторых данных в кеш, то процессор может продолжить выполнение другого потока. Например, каждое ядро в процессоре Intel Core i7 может выполнять два потока, поэтому четырехъядерная система фактически способна параллельно выполнять восемь потоков.

Использование технологий многопроцессорной обработки позволяет повысить производительность системы, предлагая два преимущества. Во-первых, уменьшает необходимость моделирования конкуренции при выполнении нескольких задач. Как уже упоминалось, даже персональный компьютер, используемый одним человеком, может выполнять множество действий одновременно. Во-вторых, дает возможность выполнять каждую отдельную прикладную программу быстрее, правда при условии, что в ней имеется несколько потоков управления, способных эффективно выполняться параллельно. То есть даже притом что принципы параллелизма формировались и изучались на протяжении более 50 лет, только появление многоядерных и гиперпоточных систем способствовало появлению желания использовать приемы разработки прикладных программ, которые могут применять параллелизм на уровне потоков, реализованный на аппаратном уровне. Более подробно поддержка конкурентного выполнения и ее использование рассматриваются в главе 12.

Параллелизм на уровне инструкций

На гораздо более низком уровне абстракции современные процессоры могут выполнять несколько инструкций одновременно. Это свойство известно как *параллелизм на уровне инструкций*. Например, ранним микропроцессорам, таким как Intel 8086 1978 года выпуска, для выполнения одной инструкции требовалось несколько тактов (обычно 3–10). Более современные процессоры могут выполнять по 2–4 инструкции за такт. Для выполнения любой конкретной инструкции требуется намного больше времени, например 20 тактов или больше, но процессор использует ряд хитрых приемов для одновременной обработки до 100 инструкций. В главе 4 мы рассмотрим использование *конвейерной обработки*, в которой действия, необходимые для выполнения инструкции, разделены на этапы, а аппаратное обеспечение процессора организовано в виде последовательности стадий, каждая из которых выполняет один из этапов. Стадии могут работать параллельно, выполняя разные части разных инструкций. Мы увидим, что для поддержания скорости выполнения, близкой к 1 инструкции на такт, не требуется ничего особенно сложного.

Процессоры, которые могут выполнять более одной инструкции за такт, называют *суперскалярными*. Большинство современных процессоров поддерживают суперскалярные операции. В главе 5 мы опишем высокоуровневую модель таких процессоров и покажем, как прикладные программисты могут использовать эту модель для прогнозирования производительности своих программ, а затем писать программы так, чтобы сгенерированный код достигал более высокой степени параллелизма на уровне инструкций и, следовательно, работал быстрее.

Одиночный поток команд, множественный поток данных

На самом низком уровне многие современные процессоры имеют специальное оборудование, позволяющее одной инструкции параллельно выполнять несколько опера-

ций. Этот режим известен как *одиночный поток команд, множественный поток данных* (Single-Instruction, Multiple-Data, SIMD). Например, в последних поколениях процессоров Intel и AMD есть инструкции, которые могут параллельно складывать 8 пар чисел с плавающей точкой одинарной точности (тип данных `float` в языке C).

Эти инструкции SIMD предоставляются в основном для ускорения приложений, обрабатывающих изображения, звук и видео. Некоторые компиляторы пытаются автоматически использовать параллелизм этого вида в программах на C, но все же более надежным методом является разработка программ с использованием специальных типов *векторных* данных, поддерживаемых компиляторами, такими как GCC. Мы кратко опишем этот стиль программирования в приложении в интернете OPT:SIMD в рамках общего описания способов оптимизации программ в главе 5.

1.9.3. Важность абстракций в компьютерных системах

Абстракции – одна из важнейших концепций информатики. Например, одной из рекомендуемых практик программирования является создание простого прикладного программного интерфейса (Application Program Interface, API) – набора функций, позволяющих программистам использовать код, не вникая в особенности его работы. Разные языки программирования предоставляют разные формы и уровни поддержки абстракции, такие как объявления классов в Java и прототипов функций в C.

Мы уже познакомились с некоторыми абстракциями, изображенными на рис. 1.16, которые встречаются в компьютерных системах. *Архитектурный набор команд* обеспечивает абстракцию физического устройства процессора. Согласно этой абстракции, программа в машинном коде ведет себя так, будто она выполняется на процессоре, который обрабатывает инструкции по одной. Физический процессор устроен намного сложнее и может выполнять несколько инструкций параллельно, но всегда в соответствии с этой простой последовательной моделью. Придерживаясь одной и той же модели выполнения, разные реализации процессоров могут обрабатывать один и тот же машинный код, предлагая различные затраты и производительность.

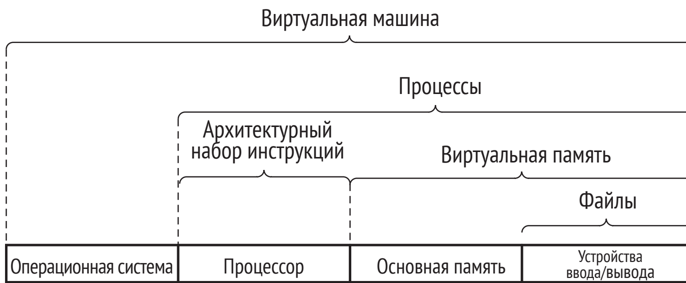

Рис. 1.16. Некоторые абстракции, предоставляемые компьютерной системой. Основная задача компьютерных систем – обеспечить абстрактные представления на разных уровнях, чтобы скрыть сложность фактических реализаций

На уровне операционной системы мы ввели три абстракции: *файлы* как абстракцию устройств ввода/вывода, *виртуальную память* как абстракцию памяти программ и *процессы* как абстракцию выполняющихся программ. К этим абстракциям мы можем добавить еще одну: *виртуальную машину*, предоставляющую абстракцию всего компьютера, включая операционную систему, процессор и программы. Идея виртуальной машины была представлена компанией IBM еще в 1960-х годах, но лишь не так давно она стала привлекать особое внимание как способ управления компьютерами, которые должны запускать программы, разработанные для разных операционных систем (таких как

Microsoft Windows, Mac OS X и Linux) или разных версий одной и той же операционной системы.

Мы вернемся к этим абстракциям в следующих разделах книги.

1.10. Итоги

Компьютерные системы состоят из аппаратных и программных средств, 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 исходного текста ASCII, затем преобразуются компиляторами и компоновщиками в выполняемые файлы.

Процессоры читают и интерпретируют двоичные инструкции, находящиеся в основной памяти. Поскольку большую часть времени компьютеры тратят на копирование данных между основной памятью, устройствами ввода/вывода и регистрами процессора, память системы образует некоторую иерархию, на вершине которой находятся регистры процессора, далее следуют несколько уровней аппаратной кеш-памяти, затем основная DRAM-память и дисковая память. Чем выше находится устройство памяти в иерархии, тем выше его быстродействие и стоимость в пересчете на один бит. Кроме того, устройства памяти, находящиеся выше в иерархии, служат кешем для устройств памяти, находящихся ниже. Программисты могут оптимизировать производительность своих программ на языке C, изучив и воспользовавшись особенностями иерархии памяти.

Ядро операционной системы играет роль посредника между прикладными программами и аппаратными средствами. Оно реализует три фундаментальные абстракции:

1) файлы, абстрагирующие устройства ввода/вывода;
2) виртуальную память, абстрагирующую как основную память, так и дисковую;
3) процессы, абстрагирующие процессоры, основную память и устройства ввода/вывода.

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

Библиографические заметки

Ритчи (Ritchie) написал интересные заметки о первых шагах языка C и системы Unix [91, 92]. Ритчи и Томпсон (Ritchie and Thompson) впервые опубликовали отчет о системе [93]. Зильбершатц, Галвин и Ганье (Silberschatz, Galvin and Gagne) представили исчерпывающую историю появления разных версий Unix [102]. Веб-страницы проектов GNU (**www.gnu.org**) и Linux (**www.linux.org**) содержат текущую и историческую информацию разного характера. Информация о стандартах Posix тоже доступна онлайн (**www.unix.org**).

Решения упражнений

Решение упражнения 1.1

Эта задача показывает, что закон Амдала применим не только к компьютерным системам.

1. В терминах уравнения 1.1 мы имеем $\alpha = 0{,}6$ и $k = 1{,}5$. Если говорить более конкретно, преодоление 1500 километров через штат Монтана займет 10 часов.

Остальная часть пути также займет 10 часов. Несложные вычисления дают нам ускорение 25/(10 + 10) = 1,25×.

2. В терминах уравнения 1.1 мы имеем $\alpha = 0{,}6$, и требуется определить k, чтобы на выходе получить $S = 1{,}67$. Если говорить более конкретно, то чтобы ускорить поездку в 1,67 раза, мы должны уменьшить общее время до 15 часов. На преодоление части пути за пределами штата Монтана по-прежнему потребуется 10 часов, поэтому мы должны пересечь Монтану за 5 часов. Для этого нужно ехать со скоростью 300 км/ч, что довольно много для грузовика!

Решение упражнения 1.2

Лучше всего действие закона Амдала исследовать на наглядных примерах. Эта задача требует взглянуть на уравнение 1.1 с необычной точки зрения.

Для решения данной задачи нужно просто применить уравнение. Итак, дано: $S = 2$ и $\alpha = 0{,}8$, мы должны найти k:

$$2 = \frac{1}{(1 - 0{,}8) + 0{,}8/k}$$

$$0{,}4 + 1{,}6/k = 1{,}0$$

$$k = 2{,}67$$