Низкоуровневое программирование

Ассемблер Intel 64 и язык C

Игорь Жирков

Перевод на русский язык выполнен для личного чтения.
Оригинал: Igor Zhirkov. Low-Level Programming. Apress, 2017.

Содержание

Введение 2

Язык ассемблера и архитектура компьютера 5

Основы архитектуры компьютера 6

1.1 Базовая архитектура 6

1.2 Эволюция 9

1.3 Регистры 12

1.4 Кольца защиты 20

1.5 Аппаратный стек 20

1.6 Итоги 23

Язык ассемблера 25

2.1 Настройка окружения 25

2.2 Пишем «Hello, world» 26

2.3 Пример: вывод содержимого регистра 31

2.4 Вызовы функций 36

2.5 Работа с данными 40

2.6 Пример: вычисление длины строки 45

2.7 Задание: библиотека ввода/вывода 47

2.8 Итоги 51

Наследие 55

3.1 Реальный режим 55

3.2 Защищённый режим 57

3.3 Минимальная сегментация в длинном режиме 61

3.4 Доступ к частям регистров 62

3.5 Итоги 64

Виртуальная память 65

4.1 Кэширование 65

4.2 Мотивация 65

4.3 Адресные пространства 67

4.4 Возможности 67

4.5 Пример: обращение к запрещённому адресу 69

4.6 Эффективность 71

4.7 Реализация 71

4.8 Отображение памяти 76

4.9 Пример: отображение файла в память 76

4.10 Итоги 81

Конвейер компиляции 84

5.1 Препроцессор 85

5.2 Трансляция 97

5.3 Компоновка 97

5.4 Задание: словарь 114

5.5 Итоги 116

Прерывания и системные вызовы 119

6.1 Ввод и вывод 119

6.2 Прерывания 122

6.3 Системные вызовы 126

6.4 Итоги 128

Модели вычислений 128

7.1 Конечные автоматы 130

7.2 Машина Форта 140

7.3 Задание: компилятор и интерпретатор Форта 151

7.4 Итоги 160

Язык программирования C 163

Основы 164

8.1 Введение 164

8.2 Структура программы 166

8.3 Управляющие конструкции 170

8.4 Операторы и выражения 177

8.5 Функции 181

8.6 Препроцессор 184

8.7 Итоги 185

Система типов 187

9.1 Базовая система типов C 187

9.2 Помеченные типы 212

9.3 Типы данных в языках программирования 218

9.4 Полиморфизм в C 222

9.5 Итоги 227

Структура кода 230

10.1 Объявления и определения 230

10.2 Доступ к коду из других файлов 233

10.3 Стандартная библиотека 239
10.4 Препроцессор 242
10.5 Пример: сумма динамического массива 248
10.6 Задание: связный список 250
10.7 Ключевое слово static 252
10.8 Компоновка 254
10.9 Итоги 255

Память 255

11.1 Указатели снова 257
11.2 Модель памяти 263
11.3 Массивы и указатели 267
11.4 Строковые литералы 270
11.5 Модели данных 273
11.6 Потоки данных 275
11.7 Задание: функции высшего порядка и списки 278
11.8 Итоги 282

Синтаксис, семантика и прагматика 285

12.1 Что такое язык программирования? 285
12.2 Синтаксис и формальные грамматики 286
12.3 Семантика 298
12.4 Прагматика 302
12.5 Выравнивание в C11 307
12.6 Итоги 308

Практики написания хорошего кода 309

13.1 Принятие решений 311
13.2 Элементы кода 313
13.3 Файлы и документация 318
13.4 Инкапсуляция 319
13.5 Неизменяемость 323
13.6 Утверждения 324
13.7 Обработка ошибок 325
13.8 О выделении памяти 328
13.9 О гибкости 328
13.10 Задание: поворот изображения 331
13.11 Задание: собственный аллокатор памяти 334
13.12 Итоги 337

Между C и ассемблером 338

Детали трансляции 339
14.1 Последовательность вызова функции 339
14.2 volatile 347
14.3 Нелокальные переходы — setjmp 350
14.4 inline 355
14.5 restrict 356
14.6 Строгий алиасинг 358
14.7 Проблемы безопасности 359
14.8 Механизмы защиты 364
14.9 Итоги 365

Разделяемые объекты и модели кода 368
15.1 Динамическая загрузка 368
15.2 Перемещения и PIC 370
15.3 Пример: динамическая библиотека на C 370
15.4 GOT и PLT 373
15.5 Предзагрузка 381
15.6 Итоги по адресации символов 383
15.7 Примеры 384
15.8 Какие объекты компонуются? 392
15.9 Оптимизации 396
15.10 Модели кода 399
15.11 Итоги 411

Производительность 411
16.1 Оптимизации 414
16.2 Кэширование 432
16.3 Класс инструкций SIMD 445
16.4 Расширения SSE и AVX 446
16.5 Итоги 453

Многопоточность 455
17.1 Процессы и потоки 455
17.2 Что делает многопоточность сложной? 456
17.3 Порядок выполнения 457
17.4 Строгая и слабая модели памяти 458
17.5 Пример переупорядочивания 459
17.6 Что является volatile, а что нет 462
17.7 Барьеры памяти 463
17.8 Введение в pthreads 465
17.9 Семафоры 488

17.10 Насколько строг Intel 64? 491
17.11 Что такое неблокирующее программирование? 495
17.12 Модель памяти C11 498
17.13 Итоги 502

Приложение A. Использование gdb 508

Приложение B. Использование Make 518

19.1 Простой Makefile 518
19.2 Добавляем переменные 519
19.3 Автоматические переменные 521

Приложение C. Системные вызовы 523

20.1 read 523
20.2 write 524
20.3 open 525
20.4 close 526
20.5 mmap 526
20.6 munmap 528
20.7 exit 529

Приложение D. Информация о тестах производительности 530

Библиография 533

Сноски 536

Низкоуровневое программирование

Ассемблер Intel 64 и язык C · Игорь Жирков

Введение

Эта книга призвана помочь вам выработать целостное представление о предметной области низкоуровневого программирования. Мы хотим дать внимательному читателю возможность

- Свободно писать на языке ассемблера.
- Понимать модель программирования Intel 64.
- Писать поддерживаемый и надёжный код на C11.
- Понимать процесс компиляции и разбираться в листингах ассемблера.
- Отлаживать ошибки в скомпилированном ассемблерном коде.
- Использовать подходящие модели вычислений, чтобы значительно снизить сложность программ.
- Писать код, критичный к производительности.

Существует два вида технических книг: те, что используют как справочник, и те, по которым учатся. Эта книга, без сомнения, относится ко второму виду. Она намеренно довольно плотная, и чтобы успешно усвоить информацию, мы настоятельно рекомендуем читать её последовательно, без пропусков. Чтобы быстро запоминать новую информацию, стоит связывать её с тем, что вам уже знакомо. Именно поэтому мы, где это было возможно, старались строить объяснение каждой темы на информации, полученной из предыдущих тем. Эта книга написана для студентов, изучающих программирование, программистов среднего и продвинутого уровня, а также энтузиастов низкоуровневого программирования. Необходимые предварительные знания — базовое понимание двоичной и шестнадцатеричной систем счисления и базовое знание команд Unix.

> **Вопросы и ответы** На протяжении этой книги вам встретится множество вопросов. Большинство из них призваны заставить вас ещё раз обдумать то, что вы только что изучили, но некоторые побуждают провести дополнительное исследование, указывая на ключевые слова, с которых стоит начать.

Ответы на эти вопросы мы предлагаем на нашей странице на GitHub, где также размещены все листинги и исходный код для заданий, обновления и другие полезные материалы. За дополнительной информацией обращайтесь к странице книги на сайте Apress: `http://www.apress.com/us/book/9781484224021.` Там же можно найти несколько предварительно настроенных виртуальных машин с установленным Debian Linux, с графическим интерфейсом (GUI) и без него, что позволяет сразу приступить к практике, не тратя время на настройку системы. Подробнее об этом можно узнать в разделе 2.1. Мы начинаем с самых простых базовых идей о том, что такое компьютер, объясняя понятия модели вычислений и архитектуры компьютера. Мы расширяем базовую модель дополнениями, пока она не станет достаточно адекватной, чтобы описывать современный процессор

так, как его видит программист. Начиная с Главы 2 мы начинаем программировать на настоящем языке ассемблера для Intel 64, не прибегая к более старым 16-битным архитектурам, которые часто преподают по историческим причинам. Это позволяет нам увидеть взаимодействие между приложениями и операционной системой через интерфейс системных вызовов, а также специфические детали архитектуры, такие как порядок байтов. После краткого обзора устаревших возможностей архитектуры, некоторые из которых используются до сих пор, мы подробно изучаем виртуальную память и иллюстрируем её использование с помощью procfs и примеров применения системного вызова mmap в ассемблере. Затем мы погружаемся в процесс компиляции, рассматривая препроцессирование, статическую и динамическую компоновку. После более подробного изучения механизмов прерываний и системных вызовов мы завершаем первую часть главой о различных моделях вычислений, изучая примеры конечных автоматов, стековых машин и реализуя полностью функциональный компилятор языка Форт на чистом ассемблере. Вторая часть посвящена языку C. Мы начинаем с обзора языка, выстраивая базовое понимание его модели вычислений, необходимое для того, чтобы начать писать программы. В следующей главе мы изучаем систему типов C и иллюстрируем различные виды типизации, заканчивая обсуждением полиморфизма и приводя примеры реализации разных видов полиморфизма в C. Затем мы изучаем способы правильной структуризации программы путём разбиения её на несколько файлов, а также рассматриваем, как это влияет на процесс компоновки. Следующая глава посвящена управлению памятью, вводу и выводу. После этого мы разбираем три аспекта любого языка: синтаксис, семантику и прагматику, сосредотачиваясь на первом и третьем. Мы увидим, как языковые конструкции преобразуются в абстрактные синтаксические деревья, в чём разница между неопределённым и неуточнённым поведением в C, и как прагматика языка влияет на ассемблерный код, порождаемый компилятором. В конце второй части мы посвящаем главу практикам написания хорошего кода, чтобы дать читателям представление о том, каким должен быть код в зависимости от конкретных требований к нему. Последовательность заданий этой части завершается поворотом растрового изображения и созданием собственного аллокатора памяти. Заключительная часть — мост между двумя предыдущими. Она погружается в детали трансляции, такие как соглашения о вызовах и кадры стека, а также в продвинутые возможности языка C, требующие определённого понимания ассемблера, такие как ключевые слова `volatile` и `restrict`. Мы даём обзор нескольких классических низкоуровневых ошибок, таких как переполнение буфера на стеке, которое может быть использовано для того, чтобы вызвать нежелательное поведение программы. Следующая глава подробно рассказывает о разделяемых объектах и изучает их на уровне ассемблера, приводя минимальные рабочие примеры разделяемых библиотек, написанных на C и ассемблере. Затем мы обсуждаем сравнительно редкую тему моделей кода. Глава изучает оптимизации, на которые способны современные компиляторы, и то, как это знание можно использовать для получения читаемого и быстрого кода. Мы также даём обзор техник повышения производительности, таких как использование специализированных ассемблерных инструкций и оптимизация использования кэша. За этим следует задание, в котором вы реализуете сепия-фильтр для изображения с использованием специализированных инструкций SSE и из-

мерите его производительность. Последняя глава знакомит с многопоточностью через использование библиотеки `pthreads`, модели памяти и переупорядочивания, о которых должен знать каждый, кто занимается многопоточным программированием, и разбирает необходимость барьеров памяти. Приложения включают короткие руководства по gdb (отладчик), make (система автоматической сборки) и таблицу наиболее часто используемых системных вызовов для справки, а также информацию о системе, упрощающую воспроизведение тестов производительности, приведённых в книге. Их стоит читать по мере необходимости, но мы рекомендуем начать привыкать к gdb сразу, как только вы приступите к программированию на ассемблере в Главе 2. Большинство иллюстраций были созданы с помощью библиотеки VSVG, предназначенной для создания сложной интерактивной векторной графики, написанной Алексеем Великим (`http://www.corpglory.com`). Исходный код библиотеки и иллюстраций к книге доступен на странице VSVG на GitHub: `https://github.com/corpglory/vsvg`. Мы надеемся, что эта книга окажется вам полезной, и желаем приятного чтения!

Язык ассемблера и архитектура компьютера

Основы архитектуры компьютера

Эта глава даст вам общее представление об основах работы компьютера. Мы опишем базовую модель вычислений, перечислим её расширения и подробнее рассмотрим два из них — регистры и аппаратный стек. Это подготовит вас к тому, чтобы начать программировать на ассемблере в следующей главе.

1.1 Базовая архитектура

1.1.1 Модель вычислений

Чем занимается программист? Первое, что приходит в голову — «построением алгоритмов и их реализацией». Итак, мы улавливаем идею, затем пишем код — так обычно и рассуждают. Можем ли мы построить алгоритм, описывающий какое-нибудь повседневное действие, например прогулку или поход за покупками? Вопрос не кажется особенно сложным, и многие с радостью предложат своё решение. Однако все эти решения будут принципиально разными. Один будет оперировать такими действиями, как «открыть дверь» или «взять ключ»; другой скорее скажет «выйти из дома», опустив детали. Третий же пойдёт до конца и подробно опишет движения рук и ног или даже паттерны сокращения мышц. Причина, по которой эти ответы так сильно различаются — *неполнота исходного вопроса*. Любой идее (включая алгоритмы) нужен способ быть выраженной. Чтобы описать новое понятие, мы используем другие, более простые понятия. Мы также хотим избежать порочных кругов, поэтому объяснение будет иметь форму пирамиды. Каждый уровень объяснения будет расти по горизонтали. Мы не можем строить эту пирамиду бесконечно, поскольку объяснение должно быть конечным, поэтому мы останавливаемся на уровне базовых, примитивных понятий, которые мы сознательно решили дальше не раскрывать. Таким образом, выбор базовых понятий — фундаментальное требование для того, чтобы вообще что-то выразить. Это означает, что построение алгоритма невозможно, пока мы не зафиксировали набор базовых действий, служащих его строительными блоками. *Модель вычислений* — это набор базовых операций и их соответствующих затрат. Затраты обычно представляют собой целые числа и используются для рассуждений о сложности алгоритмов путём вычисления суммарной стоимости всех их операций. Мы не будем обсуждать вычислительную сложность в этой книге. Большинство моделей вычислений являются также *абстрактными машинами*. Это означает, что они описывают гипотетический компьютер, чьи инструкции соответствуют базовым операциям модели. Другой тип моделей, деревья решений, выходит за рамки этой книги.

1.1.2 Архитектура фон Неймана

Теперь представим, что мы живём в 1930-х годах, когда сегодняшних компьютеров ещё не существовало. Люди хотели каким-то образом автоматизировать вычисления, и разные исследователи предлагали разные способы добиться такой автоматизации. Распространённые примеры — лямбда-исчисление Чёрча или машина Тьюринга. Это типичные абстрактные машины, описывающие воображаемые компьютеры.

Один тип машины вскоре стал доминирующим: компьютер с архитектурой фон Неймана. *Архитектура компьютера* описывает функциональность, организацию и реализацию компьютерных систем. Это относительно высокоуровневое описание по сравнению с моделью вычислений, которая не опускает ни малейшей детали. Архитектура фон Неймана обладала двумя решающими преимуществами: она была надёжной (в мире, где электронные компоненты были крайне нестабильными и недолговечными) и простой в программировании. Коротко говоря, это компьютер, состоящий из одного процессора и одного банка памяти, соединённых общей шиной. *Центральный процессор (CPU)* может выполнять инструкции, извлекаемые из *памяти устройством управления*. *Арифметико-логическое устройство (АЛУ)* выполняет необходимые вычисления. Память также хранит данные. См. рисунки 1-1 и 1-2. Ниже перечислены ключевые особенности этой архитектуры:

- Память хранит только биты (единицу информации, значение равное 0 или 1).
- Память хранит и закодированные инструкции, и данные, над которыми они оперируют. Нет никакого способа

отличить данные от кода: и то, и другое на самом деле — битовые строки.

- Память организована в ячейки, помечаемые своими соответствующими индексами

естественным образом (например, ячейка №43 следует за ячейкой №42). Индексы начинаются с 0. Размер ячейки может быть разным (Джон фон Нейман считал, что у каждого бита должен быть свой адрес); современные компьютеры используют один байт (восемь бит) в качестве размера ячейки памяти. Так, 0-й байт хранит первые восемь бит памяти и т. д.

- Программа состоит из инструкций, которые извлекаются одна за другой. Их

выполнение последовательно, если не выполняется специальная инструкция перехода.

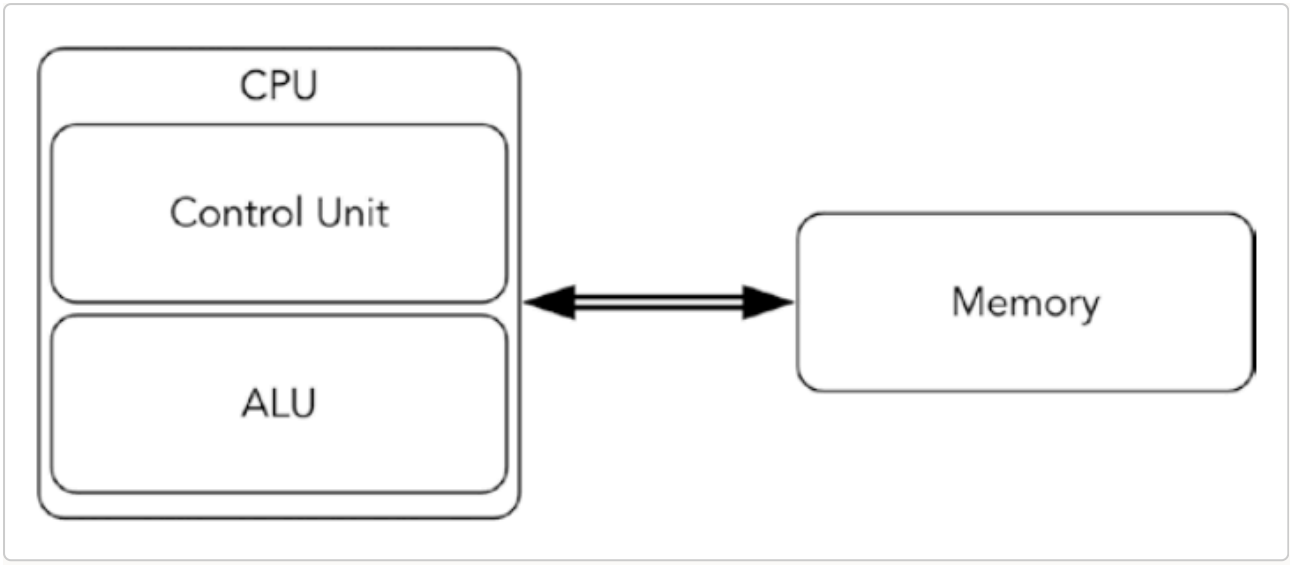


Рисунок 1-1. Архитектура фон Неймана — обзор

Язык ассемблера для выбранного процессора — это язык программирования, состоящий из мнемоник для каждой возможной двоично-кодированной инструкции (машинного кода). Он значительно упрощает программирование в машинных кодах, потому что программисту больше не нужно запоминать двоичное кодирование инструкций — только их имена и параметры. Заметьте, что инструкции могут иметь параметры разных размеров и форматов. *Архитектура не всегда определяет точную систему команд, в отличие от модели вычислений.* Обычный современный персональный компьютер произошёл от старых компьютеров архитектуры фон Неймана, поэтому мы рассмотрим эту эволюцию и увидим, чем современный компьютер отличается от простой схемы на рисунке 1-2.

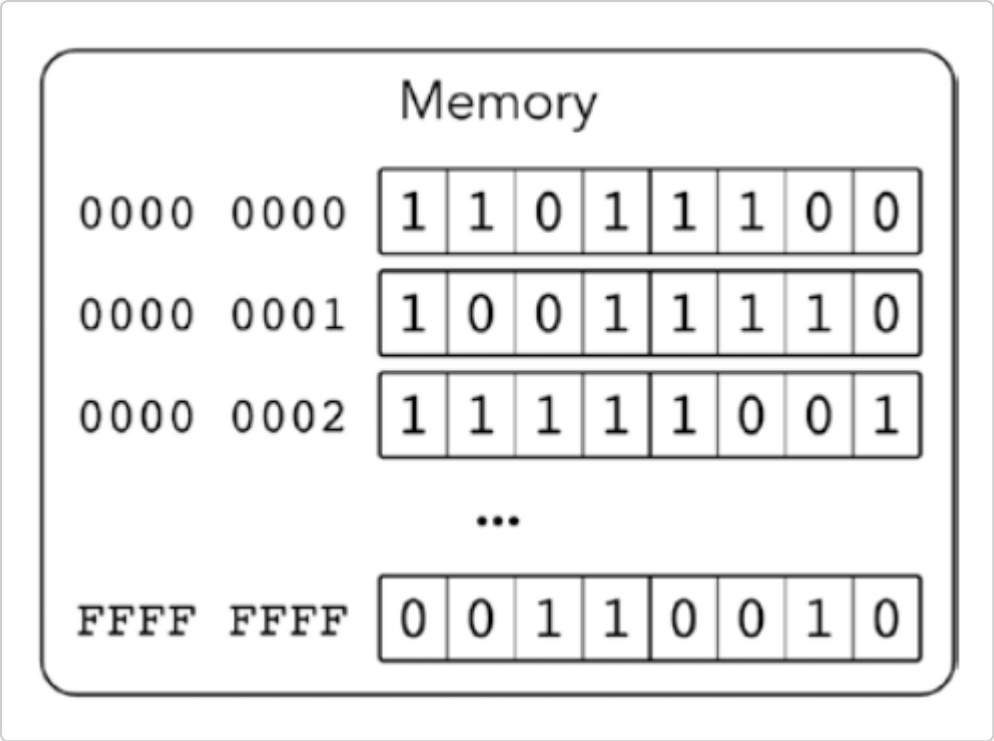


Рисунок 1-2. Архитектура фон Неймана — память

> **Примечание** Состояние памяти и значения регистров полностью описывают состояние CPU (с точки зрения программиста). Понимание инструкции означает понимание её воздействия на память и регистры.

1.2 Эволюция

1.2.1 Недостатки архитектуры фон Неймана

У описанной ранее простой архитектуры есть серьёзные недостатки. Прежде всего, эта архитектура совсем не интерактивна. Программист ограничен ручным редактированием памяти и каким-либо способом визуализации её содержимого. В ранние дни компьютеров это было довольно просто, потому что схемы были большими, и биты можно было переключать буквально голыми руками. Более того, эта архитектура не приспособлена для многозадачности. Представьте, что ваш компьютер выполняет очень медленную задачу (например, управляет принтером). Она медленная, потому что принтер намного медленнее самого медленного CPU. Тогда CPU приходится ждать реакции устройства почти 99% времени, что является растратой ресурсов (а именно, времени CPU). Далее, когда кто угодно может выполнить любую инструкцию, могут происходить всевозможные непредвиденные ситуации. Назначение операционной системы (ОС) состоит (среди прочего) в управлении ресурсами (такими как внешние устройства), чтобы пользовательские приложения не устраивали хаос, взаимодействуя с одними и теми же устройствами одновременно. Из-за этого мы хотим запретить всем пользовательским приложе-

ниям выполнять некоторые инструкции, связанные с вводом/выводом или управлением системой. Другая проблема заключается в том, что производительность памяти и CPU сильно различается. В старые времена компьютеры не только были проще: они проектировались как единое целое. Память, шина, сетевые интерфейсы — всё создавалось одной и той же инженерной командой. Каждая часть была специализирована для использования именно в этой конкретной модели. Так что части не предназначались для взаимозаменяемости. В этих условиях никто не пытался создать деталь, способную работать быстрее остальных, потому что это никак не могло повысить общую производительность компьютера. Но по мере того как архитектуры становились более или менее стабильными, разработчики оборудования начали работать над разными частями компьютеров независимо друг от друга. Естественно, они старались улучшить их производительность в маркетинговых целях. Однако не все части было легко *и дёшево*[1] ускорить. Именно поэтому CPU вскоре стали *намного* быстрее памяти. Ускорить память, выбрав другой тип используемых схем, возможно, но это обошлось бы намного дороже [12].

Когда система состоит из разных частей, и их характеристики производительности сильно различаются, самая медленная часть может стать *узким местом*. Это означает, что если заменить самую медленную часть более быстрым аналогом, общая производительность заметно возрастёт. Именно поэтому архитектуру пришлось серьёзно модифицировать.

1.2.2 Архитектура Intel 64

В этой книге мы описываем только архитектуру Intel 64.[2] Intel разрабатывает своё семейство основных процессоров с 1970-х годов. Каждая модель была призвана сохранять двоичную совместимость со старыми моделями. Это означает, что даже современные процессоры могут выполнять код, написанный и скомпилированный для старых моделей. Это приводит к огромному объёму наследия. Процессоры могут работать в ряде режимов: реальном, защищённом, виртуальном и т. д. Если не указано иное, мы будем описывать работу CPU в новейшем, так называемом *длинном режиме*.

1.2.3 Расширения архитектуры

Intel 64 включает в себя несколько расширений архитектуры фон Неймана. Наиболее важные из них перечислены здесь для беглого обзора. *Регистры* Это ячейки памяти, расположенные непосредственно на кристалле CPU. С точки зрения схемотехники они гораздо быстрее, но также сложнее и дороже. Обращения к регистрам не используют шину. Время отклика довольно мало и обычно равно паре тактов CPU. См. раздел 1.3 «Регистры». *Аппаратный стек* Стек в общем смысле — это структура данных. Она поддерживает две операции: *проталкивание (push)* элемента на её вершину и *снятие (pop)* верхнего элемента. Аппаратный стек реализует эту абстракцию поверх памяти с помощью специальных инструкций и регистра, указывающего на последний элемент стека. Стек используется не только в вычислениях, но и для хранения локальных переменных

и реализации последовательности вызова функций в языках программирования. См. раздел 1.5 «Аппаратный стек». *Прерывания* Эта возможность позволяет изменять порядок выполнения программы на основе событий, внешних по отношению к самой программе. После того как сигнал (внешний или внутренний) пойман, выполнение программы приостанавливается, некоторые регистры сохраняются, и CPU начинает выполнять специальную процедуру для обработки ситуации. Ниже приведены примеры ситуаций, в которых возникает прерывание (и выполняется соответствующий код для его обработки):

- Сигнал от внешнего устройства.
- Деление на ноль.
- Недопустимая инструкция (когда CPU не удалось распознать инструкцию по её двоичному

представлению).

- Попытка выполнить привилегированную инструкцию в непривилегированном режиме.

Подробнее см. раздел 6.2 «Прерывания». *Кольца защиты* CPU всегда находится в состоянии, соответствующем одному из так называемых колец защиты. Каждое кольцо определяет набор разрешённых инструкций. Нулевое кольцо разрешает выполнение любой инструкции из всей системы команд CPU, и потому оно наиболее привилегированное. Третье разрешает только самые безопасные из них. Попытка выполнить привилегированную инструкцию приводит к прерыванию. Большинство приложений работают внутри третьего кольца, чтобы гарантировать, что они не изменяют критически важные системные структуры данных (такие как таблицы страниц) и не работают с внешними устройствами в обход ОС. Два других кольца (первое и второе) промежуточные, и современные операционные системы их не используют. Подробнее см. раздел 3.2 «Защищённый режим». *Виртуальная память* Это абстракция над физической памятью, которая помогает распределять её между программами более безопасным и эффективным способом. Она также изолирует программы друг от друга.

Подробнее см. раздел 4.2 «Мотивация». Некоторые расширения не доступны программисту напрямую (например, кэши или теневые регистры). Мы упомянем и некоторые из них. Таблица 1-1 суммирует информацию о некоторых расширениях архитектуры фон Неймана, встречающихся в современных компьютерах.

Таблица 1-1. Архитектура фон Неймана: современные расширения

ПРОБЛЕМА	РЕШЕНИЕ
Ничего невозможно без обращения к медленной памяти	Регистры, кэши
Отсутствие интерактивности	Прерывания
Нет поддержки изоляции кода в процедурах или сохранения контекста	Аппаратный стек
Многозадачность: любая программа может выполнить любую инструкцию	Кольца защиты
Многозадачность: программы не изолированы друг от друга	Виртуальная память

> **Источники информации** Ни одна книга не должна полностью охватывать систему команд и архитектуру процессора. Многие книги пытаются включить исчерпывающую информацию о системе команд. Она довольно быстро устаревает; более того, это неоправданно раздувает книгу. Мы будем часто отсылать вас к Intel® 64 and IA-32 Architectures Software Developer's Manual, доступному онлайн: см. [15]. Скачайте его прямо сейчас! Нет никакой добродетели в переписывании описаний инструкций из «оригинального» места, где они приведены; гораздо более зрело — научиться работать с первоисточником. Второй том полностью охватывает систему команд и имеет очень удобное оглавление. Пожалуйста, всегда используйте его для получения информации о системе команд: это не только очень хорошая практика, но и весьма надёжный источник. Заметьте, что многие обучающие материалы по языку ассемблера в интернете часто сильно устарели (поскольку сегодня мало кто программирует на ассемблере) и вовсе не охватывают 64-битный режим. У инструкций, присутствующих в более старых режимах, часто есть обновлённые аналоги в длинном режиме, и работают они по-другому. По этой причине мы настоятельно не рекомендуем искать описания инструкций через поисковые системы, каким бы соблазнительным это ни казалось.

1.3 Регистры

Обмен данными между CPU и памятью — важнейшая часть вычислений в компьютере фон Неймана. Инструкции нужно извлекать из памяти, операнды нужно извлекать из памяти; некоторые инструкции также сохраняют результаты в память. Это создаёт узкое место и приводит к потере времени CPU, пока он ожидает ответа с данными от микросхемы памяти. Чтобы избежать постоянного ожидания, процессор оснастили собственными ячейками памяти, называемыми регистрами. Их мало, но они быстрые. Программы обычно пишутся таким образом, что в большинстве случаев рабочее множество ячеек памяти достаточно мало. Этот факт наводит на мысль, что программы можно писать так, чтобы большую часть времени CPU работал с регистрами. Регистры основаны на транзисторах, тогда как основная память использует конденсаторы. Мы могли

бы реализовать основную память на транзисторах и получить гораздо более быструю схему. Есть несколько причин, по которым инженеры предпочитают другие способы ускорения вычислений.

- Регистры дороже.
- Инструкции кодируют номер регистра как часть своего кода. Чтобы адресовать больше

регистров, инструкциям пришлось бы расти в размере.

- Регистры добавляют сложности схемам, которые их адресуют. Более сложные схемы

труднее разгонять. Непросто заставить большой регистровый файл работать на 5 ГГц. Естественно, использование регистров в худшем случае замедляет компьютеры. Если всё нужно сначала загрузить в регистры перед вычислениями, а затем сбросить обратно в память, в чём же выгода? Программы обычно пишутся таким образом, что обладают одним особым свойством. Это результат использования распространённых шаблонов программирования, таких как циклы, функции и повторное использование данных, а не какой-то закон природы. Это свойство называется *локальностью обращений*, и у неё есть два основных вида: временная и пространственная. *Временная локальность* означает, что обращения к одному и тому же адресу, вероятно, будут близки по времени. *Пространственная локальность* означает, что после обращения к адресу X следующее обращение к памяти, вероятно, будет близко к X (например, $X - 16$ или $X + 28$). Эти свойства не бинарны: можно написать программу с более сильной или более слабой локальностью. Типичные программы используют следующий шаблон: рабочее множество данных мало и может поместиться в регистрах. Загрузив данные в регистры один раз, мы будем работать с ними довольно долго, а затем результаты будут сброшены в память. Данные, хранящиеся в памяти, программа будет использовать редко. Если нам всё же нужно работать с этими данными, мы потеряем производительность, потому что

- Нам нужно загрузить данные в регистры.
- Если все регистры заняты данными, которые нам всё ещё понадобятся позже, нам придётся вытеснить некоторые из

них, что означает сохранить их содержимое во временно выделенные ячейки памяти.

> **Примечание** Распространённая для инженера ситуация: снижать производительность в худшем случае, чтобы улучшить её в среднем. Это довольно часто работает, но недопустимо при построении систем реального времени, которые накладывают ограничения на наихудшее время реакции системы. Такие системы обязаны выдавать реакцию на события не более чем за определённое время, поэтому снижение производительности в худшем случае ради улучшения в остальных случаях недопустимо.

1.3.1 Регистры общего назначения

Чаще всего программист работает с *регистрами общего назначения*. Они взаимозаменяемы и могут использоваться во многих различных командах. Это 64-битные регистры с именами `r0`, `r1`, ..., `r15`. Первые восемь из них можно называть иначе; эти имена отражают смысл, который они несут для некоторых специальных инструкций. Например, `r1` альтернативно называется `rcx`, где `c` означает «cycle» (цикл). Есть инструкция `loop`, которая использует `rcx` в качестве счётчика цикла, но *не принимает операндов явно*. Разумеется, такое специальное значение регистра отражено в документации к соответствующим командам (например, как счётчик для инструкции `loop`). Таблица 1-2 перечисляет их все; см. также рисунок 1-3.

> **Примечание** В отличие от аппаратного стека, реализованного поверх основной памяти, регистры представляют собой совершенно другой вид памяти. Поэтому у них нет адресов, в отличие от ячеек основной памяти!

Альтернативные имена на самом деле более распространены по историческим причинам. Мы приведём оба варианта для справки и дадим подсказку для каждого. Эти смысловые описания даны для справки; запоминать их прямо сейчас не обязательно.

Таблица 1-2. 64-битные регистры общего назначения

ИМЯ	ПСЕВДОНИМ	ОПИСАНИЕ
`r0`	`rax`	Своего рода «аккумулятор», используемый в арифметических инструкциях. Например, инструкция `div` используется для деления двух целых чисел. Она принимает один операнд и неявно использует `rax` в качестве второго. После выполнения `div rcx` большое 128-битное число, хранящееся по частям в двух регистрах `rdx` и `rax`, делится на `rcx`, и результат снова сохраняется в `rax`.
`r3`	`rbx`	Базовый регистр. Использовался для базовой адресации в ранних моделях процессоров.
`r1`	`rcx`	Используется для циклов (например, в `loop`).
`r2`	`rdx`	Хранит данные во время операций ввода/вывода.
`r4`	`rsp`	Хранит адрес верхнего элемента аппаратного стека. См. раздел 1.5 «Аппаратный стек».
`r5`	`rbp`	База кадра стека. См. раздел 14.1.2 «Соглашение о вызовах».
`r6`	`rsi`	Индекс источника в командах работы со строками (таких как `movsd`)
`r7`	`rdi`	Индекс назначения в командах работы со строками (таких как `movsd`)
r8		
`r9` … `r15`	нет	Появились позже. Используются в основном для хранения временных переменных (но иногда используются неявно, как `r10`, который сохраняет флаги CPU при выполнении инструкции `syscall`. См. Главу 6 «Прерывания и системные вызовы»).

Обычно вы *не хотите* использовать регистры `rsp` и `rbp` из-за их особого назначения (позже мы увидим, как они портят стек и кадр стека). Однако вы можете выполнять над ними арифметические операции напрямую, что делает их регистрами общего назначения. Таблица 1-3 показывает регистры, отсортированные по именам согласно индексному соглашению.

Таблица 1-3. 64-битные регистры общего назначения — разные соглашения об именовании

`R0`	`R1`	`R2`	`R3`	`R4`	`R5`	`R6`	`R7`
`rax`	`rcx`	`rdx`	`rbx`	`rsp`	`rbp`	`rsi`	`rdi`

Можно адресовать часть регистра. Для каждого регистра можно обратиться к его младшим 32 битам, младшим 16 битам или младшим 8 битам. При использовании имён `r0,...,r15` это делается добавлением соответствующего суффикса к имени регистра:

- `d` для `double word` — младшие 32 бита;

- `w` для `word` — младшие 16 бит;
- `b` для `byte` — младшие 8 бит.

Например,

- `r7b` — младший байт регистра `r7`;
- `r3w` состоит из младших двух байт `r3`; и
- `r0d` состоит из младших четырёх байт `r0`.

Альтернативные имена также позволяют адресовать меньшие части. Рисунок 1-4 показывает разложение широких регистров общего назначения на более мелкие. Соглашение об именовании для доступа к частям `rax`, `rbx`, `rcx` и `rdx` следует одному и тому же шаблону; меняется только средняя буква (a для `rax`). Остальные четыре регистра не позволяют обращаться к своему второму младшему байту (как это делает `rax` под именем ah). Наименование младшего байта немного отличается для `rsi`, `rdi`, `rsp` и `rbp`.

- Самые маленькие части `rsi` и `rdi` — это `sil` и `dil` (см. рисунок 1-5).
- Самые маленькие части `rsp` и `rbp` — это `spl` и `bpl` (см. рисунок 1-6).

На практике имена `r0–r7` встречаются редко. Обычно программисты придерживаются альтернативных имён для первых *восьми* регистров общего назначения. Это сделано как по историческим, так и по смысловым причинам: `rsp` несёт гораздо больше информации, чем `r4`. Остальные восемь (`r8–r15`) можно называть только с помощью индексного соглашения.

> **Несогласованность в записи** Все чтения из меньших регистров работают очевидным образом. Однако запись в 32-битные части заполняет старшие 32 бита полного регистра битами знака. Например, обнуление `eax` обнулит весь `rax`, а запись -1 в `eax` заполнит старшие 32 бита единицами. Другие записи (например, в 16-битные части) работают как ожидается: они не затрагивают остальные биты. См. раздел 3.4.2 «CISC и RISC» для объяснения.

1.3.2 Другие регистры

Остальные регистры имеют особое назначение. Некоторые регистры имеют общесистемную важность и потому не могут изменяться никем, кроме ОС.



Рисунок 1-3. Приближённое представление Intel 64: регистры общего назначения

Программисту доступен регистр `rip`. Это 64-битный регистр, который всегда хранит адрес следующей выполняемой инструкции. Инструкции ветвления (например, `jmp`) фактически изменяют его. Так, каждый раз, когда выполняется какая-либо инструкция, `rip` хранит адрес следующей выполняемой инструкции.

> **Примечание** Все инструкции имеют разный размер!

Ещё один доступный регистр называется `rflags`. Он хранит флаги, отражающие текущее состояние программы — например, каким был результат последней арифметической инструкции: был ли он отрицательным, произошло ли переполнение и т. д. Его меньшие части называются `eflags` (32 бита) и `flags` (16 бит).

> **Вопрос 1** Настало время провести предварительное исследование на основе документации [15]. Обратитесь к разделу 3.4.3 первого тома, чтобы узнать о регистре `rflags`. Что означают флаги `CF`, `AF`, `ZF`, `OF`, `SF`? В чём разница между `OF` и `CF`?

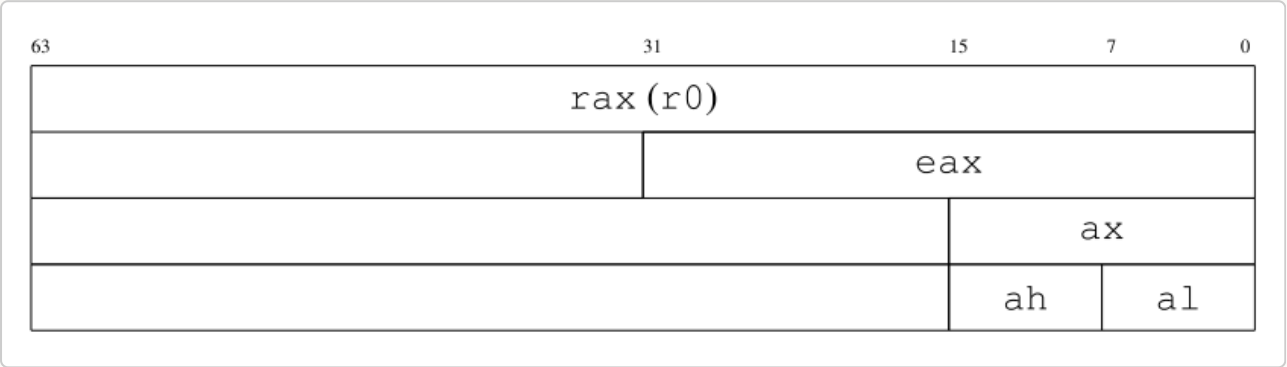


Рисунок 1-4. Разложение `rax`

В дополнение к этим основным регистрам есть также регистры, используемые инструкциями для работы с числами с плавающей точкой или специальными параллелизованными инструкциями, способными выполнять сходные действия сразу над несколькими парами операндов. Такие инструкции часто используются в мультимедийных целях (они помогают ускорить алгоритмы декодирования мультимедиа). Соответствующие регистры имеют ширину 128 бит и называются `xmm0 – xmm15`. Мы поговорим о них позже. Некоторые регистры появились как нестандартные расширения, но вскоре стали стандартизированными. Это так называемые *модельно-специфичные регистры*. Подробнее см. раздел 6.3.1 «Модельно-специфичные регистры».

1.3.3 Системные регистры

Некоторые регистры специально предназначены для использования ОС. Они не хранят значения, используемые в вычислениях. Вместо этого они хранят информацию, необходимую общесистемным структурам данных. Таким образом, их роль — поддерживать каркас, рождённый симбиозом ОС и CPU. Все приложения работают внутри этого каркаса. Последний обеспечивает хорошую изолированность приложений от самой системы и друг от друга; он также управляет ресурсами способом, более или менее прозрачным для программиста. Крайне важно, чтобы эти регистры были недоступны самим приложениям (по крайней мере, приложения не должны иметь возможности их изменять). Это цель привилегированного режима (см. раздел 3.2). Перечислим здесь некоторые из этих регистров. Их назначение будет подробно объяснено позже.

- `cr0`, `cr4` хранят флаги, связанные с различными режимами процессора и виртуальной памятью;
- `cr2`, `cr3` используются для поддержки виртуальной памяти (см. разделы 4.2 «Мотивация», 4.7.1

«Структура виртуального адреса»);

- `cr8` (с псевдонимом `tpr`) используется для тонкой настройки механизма прерываний (см. раздел 6.2 «Прерывания»).
- `efer` — ещё один регистр флагов, используемый для управления режимами процессора и расширениями (например, длинным режимом и обработкой системных вызовов).
- `idtr` хранит адрес таблицы дескрипторов прерываний (см. раздел 6.2 «Прерывания»).
- `gdtr` и `ldtr` хранят адреса таблиц дескрипторов (см. раздел 3.2 «Защищённый режим»).
- `cs`, `ds`, `ss`, `es`, `gs`, `fs` — так называемые *сегментные регистры*. Механизм сегментации, который они предоставляют, уже много лет считается устаревшим, но часть его всё ещё используется для реализации привилегированного режима. См. раздел 3.2 «Защищённый режим».

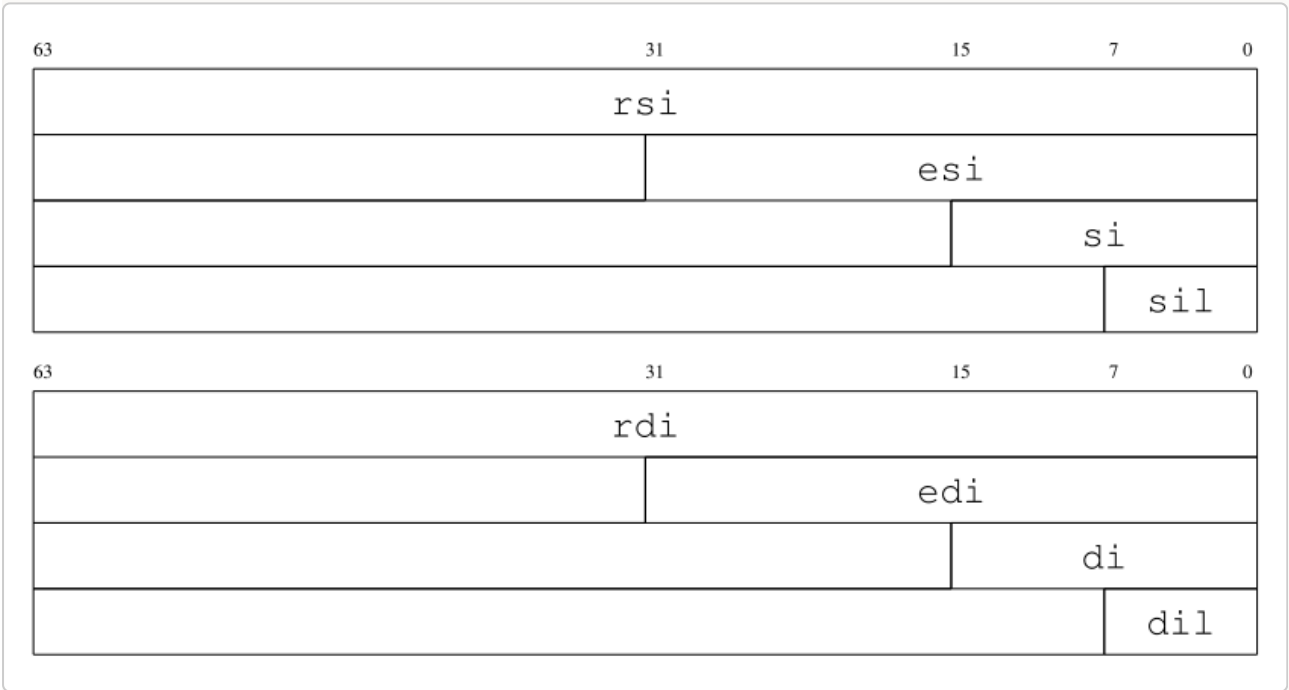


Рисунок 1-5. Разложение `rsi` *и* `rdi`

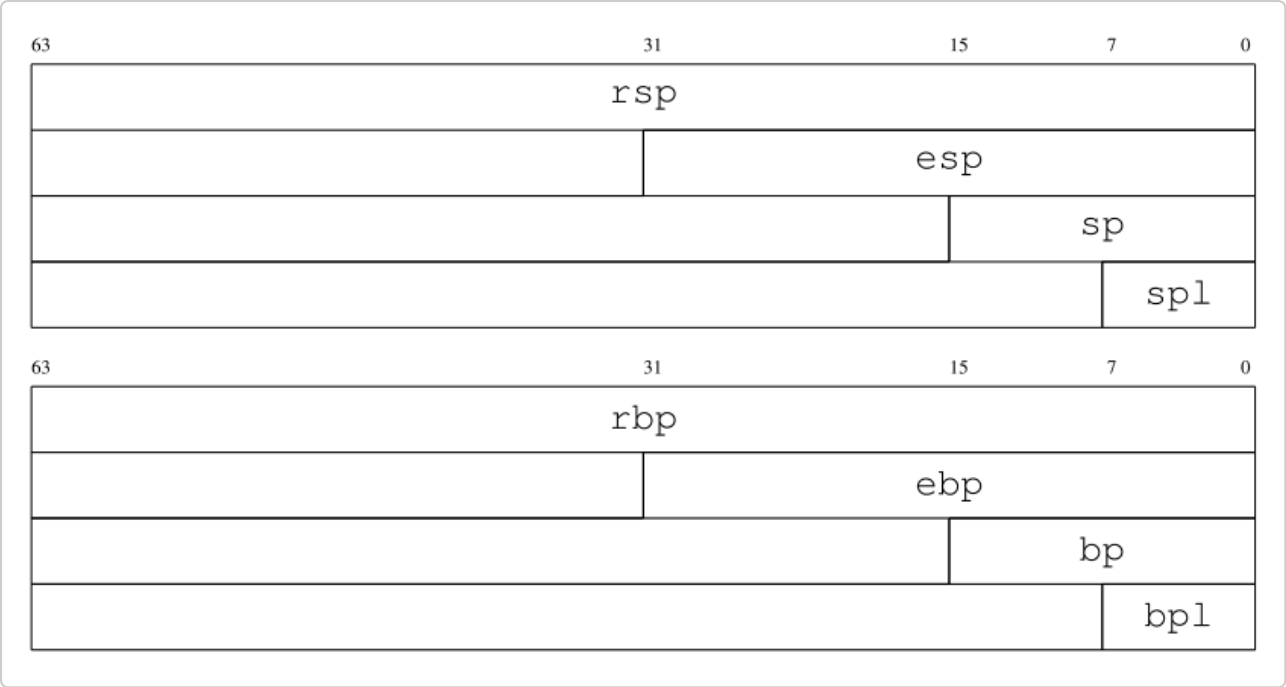


Рисунок 1-6. Разложение `rsp` *и* `rbp`

1.4 Кольца защиты

Кольца защиты — один из механизмов, призванных ограничить возможности приложений по соображениям безопасности и надёжности. Они были изобретены для ОС Multics, прямого предшественника Unix. Каждое кольцо соответствует определённому уровню привилегий. Каждый тип инструкций связан с одним или несколькими уровнями привилегий и не может выполняться на других. Текущий уровень привилегий каким-то образом хранится (например, в специальном регистре). Intel 64 имеет четыре уровня привилегий, из которых *на практике используются только два*: кольцо 0 (наиболее привилегированное) и кольцо 3 (наименее привилегированное). Средние кольца планировалось использовать для драйверов и служб ОС, но популярные ОС не восприняли этот подход. В длинном режиме номер текущего кольца защиты хранится в двух младших битах регистра `cs` (и дублируется в тех же битах `ss`). Изменить его можно только при обработке прерывания или системного вызова. Так что приложение не может выполнить произвольный код с повышенными привилегиями: оно может только вызвать обработчик прерывания или выполнить системный вызов. Подробнее см. Главу 3 «Наследие».

1.5 Аппаратный стек

Если говорить о структурах данных в целом, стек — это структура данных, контейнер с двумя операциями: новый элемент можно поместить на вершину стека (`push`); верхний элемент можно снять со стека (`pop`). Для такой структуры данных существует аппа-

ратная поддержка. Это не значит, что существует также отдельная память под стек. Это просто своего рода эмуляция, реализованная двумя машинными инструкциями (`push` и `pop`) и регистром (`rsp`). Регистр `rsp` хранит адрес верхнего элемента стека. Инструкции работают следующим образом:

- push аргумент

1. В зависимости от размера аргумента (допускаются 2, 4 и 8 байт), значение `rsp` уменьшается на 2, 4 или 8.
2. Аргумент сохраняется в памяти начиная с адреса, взятого из изменённого `rsp` .

- аргумент `pop`

1. Верхний элемент стека копируется в регистр/память.
2. `rsp` увеличивается на размер своего аргумента. Дополненная архитектура представлена на рисунке 1-7.

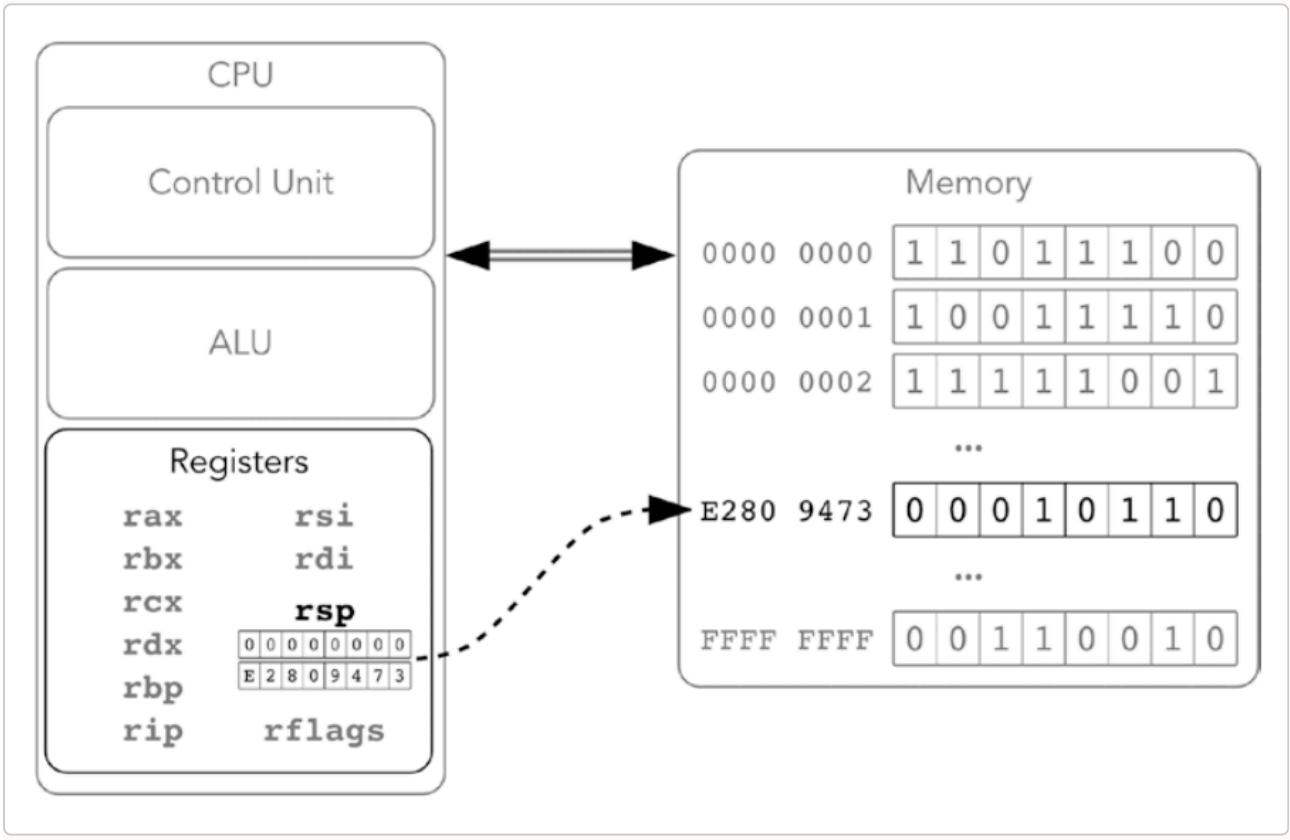


Рисунок 1-7. Intel 64, регистры и стек

Аппаратный стек особенно полезен для реализации вызовов функций в языках более высокого уровня. Когда функция *A* вызывает другую функцию *B*, она использует стек, чтобы сохранить контекст вычислений и вернуться к нему после завершения *B*. Вот некоторые важные факты об аппаратном стеке, большинство из которых следуют из его описания:

1. Не существует такой ситуации, как пустой стек, даже если мы ни разу не выполнили `push`. Алгоритм `pop` в любом случае может быть выполнен, вероятно, вернув мусорный «верхний» элемент стека.
2. Стек растёт в сторону нулевого адреса.
3. Почти все виды его операндов считаются знаковыми целыми числами и потому могут расширяться битом знака. Например, выполнение push с аргументом $B9_{16}$ приведёт к сохранению на стеке следующей единицы данных:

```
0xff b9, 0xffffffb9 or 0xff ff ff ff ff ff ff b9.
```

По умолчанию `push` использует размер операнда 8 байт. Таким образом, инструкция `push -1`

```
store 0xff ff ff ff ff ff ff ff on the stack.
```

4. Большинство архитектур, поддерживающих стек, используют тот же принцип, при котором его вершина определяется каким-либо регистром. Однако различается смысл соответствующего адреса. В одних архитектурах это адрес *следующего элемента*, который будет записан при следующем push. В других это адрес *последнего элемента*, уже помещённого в стек.

> **Работа с документацией Intel: как читать описания инструкций** Откройте второй том [15]. Найдите страницу, соответствующую инструкции `push`. Она начинается с таблицы. Для наших целей мы рассмотрим только столбцы OPCODE, INSTRUCTION, 64-BIT MODE и DESCRIPTION. Поле OPCODE определяет машинную кодировку инструкции (код операции). Как видите, есть несколько вариантов, и каждый вариант соответствует своему DESCRIPTION. Это означает, что иногда различаются не только операнды, но и сами коды операций. INSTRUCTION описывает мнемоники инструкции и допустимые типы операндов. Здесь R означает любой регистр общего назначения, M означает ячейку памяти, IMM означает непосредственное значение (например, целочисленную константу вроде 42 или 1337). Число определяет размер операнда. Если допустимы только конкретные регистры, они называются явно. Например:

> • `push r/m16` — поместить в стек 16-битный регистр общего назначения или 16-битное число, взятое из памяти.

> • `push CS` — поместить в стек сегментный регистр `cs`. Столбец DESCRIPTION даёт краткое объяснение эффектов инструкции. Часто этого достаточно, чтобы понять и использовать инструкцию.

> • Прочитайте дальнейшее объяснение `push` . Когда операнд не расширяется знаком?

> • Объясните все эффекты инструкции `push rsp` на память и регистры.

1.6 Итоги

В этой главе мы дали краткий обзор архитектуры фон Неймана. Мы начали добавлять к этой модели возможности, чтобы сделать её более адекватной для описания современных процессоров. Пока что мы подробнее рассмотрели регистры и аппаратный стек. Следующий шаг — начать программировать на ассемблере, и именно этому посвящена следующая глава. Мы рассмотрим несколько примеров программ, отметим несколько новых архитектурных особенностей (таких как порядок байтов и режимы адресации) и спроектируем простую библиотеку ввода/вывода для *nix, чтобы упростить взаимодействие с пользователем.

> **Вопрос 2** Каковы ключевые принципы архитектуры фон Неймана?

> **Вопрос 3** Что такое регистры?

> **Вопрос 4** Что такое аппаратный стек?

> **Вопрос 5** Что такое прерывания?

> **Вопрос 6** Какие основные проблемы пытаются решить современные расширения модели фон Неймана?

> **Вопрос 7** Какие основные регистры общего назначения есть у Intel 64?

> **Вопрос 8** Какова цель указателя стека?

> **Вопрос 9** Может ли стек быть пустым?

Вопрос 10 Можем ли мы подсчитать элементы в стеке?

Язык ассемблера

В этой главе мы начнём практиковаться в языке ассемблера, постепенно переходя к написанию всё более сложных программ для Linux. Мы рассмотрим некоторые архитектурные детали, влияющие на написание любых программ (например, порядок байтов). Мы выбрали *nix-систему для этой книги, потому что программировать на ассемблере в ней намного проще, чем в Windows.

2.1 Настройка окружения

Невозможно научиться программировать, не пытаясь программировать. Поэтому мы начнём программировать на ассемблере прямо сейчас. Для выполнения заданий по ассемблеру и C мы используем следующую конфигурацию:

- Debian GNU\Linux 8.0 в качестве операционной системы.
- NASM 2.11.05 в качестве компилятора языка ассемблера.
- GCC 4.9.2 в качестве компилятора языка C. Именно эта версия используется для получения ассемблера из программ на C. Также можно использовать компилятор Clang.
- GNU Make 4.0 в качестве системы сборки.
- GDB 7.7.1 в качестве отладчика.
- Любой текстовый редактор по вкусу (предпочтительно с подсветкой синтаксиса). Мы рекомендуем использовать ViM.

Если вы хотите настроить собственную систему, установите любой понравившийся вам дистрибутив Linux и убедитесь, что установили перечисленные выше программы. Насколько нам известно, Windows Subsystem for Linux также хорошо подходит для выполнения всех заданий. Вы можете установить его, а затем установить необходимые пакеты с помощью `apt-get`. Обратитесь к официальному руководству

```
located at: https://msdn.microsoft.com/en-us/commandline/wsl/install_guide.
```

На сайте Apress для этой книги, `http://www.apress.com/us/book/9781484224021`, вы можете найти следующее:

- Две предварительно настроенные виртуальные машины со всем установленным набором инструментов. У одной из них есть окружение рабочего стола; другая — минимальная система, доступ к которой возможен только через SSH (Secure Shell). Инструкции по установке и другая информация об использовании находятся в файле `README.txt` в скачанном архиве.

- Ссылку на страницу GitHub со всеми листингами книги, ответами на вопросы и решениями.

2.1.1 Работа с примерами кода

На протяжении этой главы вам встретится множество примеров кода. Компилируйте их, и если вам сложно понять их логику, попробуйте выполнить их пошагово с помощью `gdb`. Это очень помогает при изучении кода. См. Приложение A с кратким руководством по `gdb`. Приложение D содержит больше информации о системе, использованной для тестов производительности.

2.2 Пишем «Hello, world»

2.2.1 Базовый ввод и вывод

Идеология Unix постулирует, что «всё есть файл». *Файл* в широком смысле — это всё, что выглядит как поток байтов. Через файлы можно абстрагировать такие вещи, как

- доступ к данным на жёстком диске/SSD;
- обмен данными между программами; и
- взаимодействие с внешними устройствами.

Мы последуем традиции и для начала напишем простую программу «Hello, world!». Она выводит на экран приветственное сообщение и завершается. Однако такая программа должна выводить символы на экран, что невозможно сделать напрямую, если программа не выполняется на голом железе, без присмотра операционной системы за её деятельностью. Назначение операционной системы, среди прочего, — абстрагировать ресурсы и управлять ими, и экран, безусловно, один из них. Она предоставляет набор процедур для обработки взаимодействия с внешними устройствами, другими программами, файловыми системами и так далее. Программа обычно *не может обойти* операционную систему и напрямую взаимодействовать с ресурсами, которые она контролирует. Она ограничена *системными вызовами* — процедурами, предоставляемыми операционной системой пользовательским приложениям. Unix отождествляет файл с его *дескриптором*, как только он открыт программой. Дескриптор — это не что иное, как целочисленное значение (например, 42 или 999). Файл открывается явно с помощью системного вызова *open*; однако три важных файла открываются сразу при запуске программы, и потому ими не нужно управлять вручную. Это *stdin*, *stdout* и *stderr*. Их дескрипторы равны 0, 1 и 2 соответственно. *stdin* используется для обработки ввода, *stdout* — для обработки вывода, а *stderr* используется для вывода информации о процессе выполнения программы, но не о её результатах (например, ошибки и диагностика). По умолчанию ввод с клавиатуры связан со *stdin*, а вывод в терминал связан со *stdout*. Это означает, что «Hello, world!» должна *писать в stdout*. Таким образом, нам нужно вызвать систем-

ный вызов *write*. Он записывает *заданное количество байт* из памяти, начиная с *заданного адреса*, в *файл с заданным дескриптором* (в нашем случае, 1). Байты будут кодировать символы строки с помощью заранее определённой таблицы (таблицы ASCII). Каждая запись — это символ; индекс в таблице соответствует его коду в диапазоне от 0 до 255. См. листинг 2-1 — наш первый полноценный пример программы на ассемблере.

Листинг 2-1. `hello.asm`

```
global _start
section .data
message: db 'hello, world!', 10
section .text
_start:
    mov     rax, 1           ;system call number should be stored in rax
    mov     rdi, 1           ; argument #1 in rdi: where to write (descriptor)?
    mov     rsi, message     ; argument #2 in rsi: where does the string start?
    mov     rdx, 14          ; argument #3 in rdx: how many bytes to write?
    syscall                  ; this instruction invokes a system call
```

Эта программа вызывает системный вызов *write* с правильными аргументами на строках 6-9. На самом деле это единственное, что она делает. Следующие разделы объяснят этот пример подробнее.

2.2.2 Структура программы

Как мы помним из описания машины фон Неймана, есть только одна память — и для кода, и для данных; они неразличимы. Однако программист хочет их разделять. Программа на ассемблере обычно делится на *секции*. У каждой секции своё назначение: например, `.text` хранит инструкции, `.data` предназначена для *глобальных переменных* (данных, доступных в любой момент выполнения программы). Можно переключаться между секциями туда и обратно; в итоговой программе все данные, относящиеся к каждой секции, будут собраны в одном месте. Чтобы избавиться от числовых значений адресов, программисты используют *метки*. Это просто читаемые имена и адреса. Они могут предшествовать любой команде и обычно отделяются от неё двоеточием. В этой программе есть одна метка на строке 5 — `_start`. Понятие переменной типично для языков более высокого уровня. В языке ассемблера понятия переменных и процедур на самом деле довольно тонкие. Удобнее говорить о метках (или адресах). Программа на ассемблере может быть разбита на несколько файлов. Один из них должен содержать метку `_start`. Это *точка входа*; она отмечает первую инструкцию, которая будет выполнена. Эта метка должна быть объявлена `global` (см. строку 1). Смысл этого станет ясен позже. *Комментарии* начинаются с точки с запятой и продолжаются до конца строки. Язык ассемблера состоит из команд, которые напрямую отображаются в машинный код.

Однако не все языковые конструкции являются командами. Другие управляют процессом трансляции и обычно называются *директивами*.[3] В примере «Hello, world!» есть три директивы: `global`, `section` и `db`.

> **Примечание** Язык ассемблера в целом регистронезависим, но имена меток — нет! `mov`, `mOV`, `Mov` — это одно и то же, но `global _start` и `global _START` — нет! Имена секций тоже регистрозависимы: `section .DATA` и `section .data` различаются!

Директива `db` используется для создания байтовых данных. Обычно данные определяются с помощью одной из этих директив, которые различаются форматом данных:

- `db` — байты;
- `dw` — так называемые слова, каждое равно 2 байтам;
- `dd` — двойные слова, равные 4 байтам; и
- `dq` — учетверённые слова, равные 8 байтам.

Рассмотрим пример в листинге 2-2.

Листинг 2-2. data_decl.asm

```
section .data
   example1: db 5, 16, 8, 4, 2, 1
   example2: times 999 db 42
   example3: dw 999
```

Директива `times` *n* cmd повторяет `cmd` *n* раз в коде программы. Как если бы вы скопировали и вставили её *n* раз. Это также работает с инструкциями центрального процессора (CPU). Заметьте, что вы можете создавать данные внутри любой секции, включая `.text`. Как мы уже говорили ранее, для CPU данные и инструкции совершенно одинаковы, и CPU будет пытаться интерпретировать данные как закодированные инструкции, если его об этом попросят. Эти директивы позволяют определять несколько объектов данных один за другим, как в листинге 2-3, где за последовательностью символов следует один байт, равный 10.

Листинг 2-3. `hello.asm`

```
message: db 'hello, world!', 10
```

Буквы, цифры и другие символы кодируются в ASCII. Программисты договорились о таблице, в которой каждому символу присвоено уникальное число — его *ASCII-код*. Мы начинаем с адреса, соответствующего метке message. Мы сохраняем ASCII-коды всех букв

строки `"hello, world!"`, затем добавляем байт, равный 10. Почему 10? По соглашению, чтобы начать новую строку, мы выводим специальный символ с кодом 10.

> **Терминологический хаос** Довольно распространено называть целочисленный формат, наиболее естественный для компьютера,

> **машинным словом**. Поскольку мы программируем 64-битный компьютер, где адреса 64-битные, регистры общего назначения 64-битные, вполне удобно считать размер машинного слова равным 64 битам, или 8 байтам.

> **Примечание.** В программировании на ассемблере для архитектуры Intel термин **слово** действительно использовался для описания 16-битной единицы данных, потому что на старых машинах это и было машинное слово в точности. К сожалению, по историческим причинам он до сих пор используется по-старому. Именно поэтому 32-битные данные называются двойными словами, а 64-битные данные называются учетверёнными словами.

2.2.3 Базовые инструкции

Инструкция `mov` используется для записи значения в регистр или память. Значение может быть взято из другого регистра, из памяти, или может быть непосредственным операндом. Однако

1. `mov` не может копировать данные из памяти в память;
2. операнды источника и назначения должны быть одного размера.

Инструкция `syscall` используется для выполнения *системных вызовов* в *nix-системах. Операции ввода/вывода зависят от оборудования (которое также может использоваться несколькими программами одновременно), поэтому программистам не разрешено управлять ими напрямую, в обход операционной системы. У каждого системного вызова есть уникальный номер. Чтобы его выполнить,

1. Регистр `rax` должен содержать номер системного вызова;
2. Следующие регистры должны содержать его аргументы: `rdi`, `rsi`, `rdx`, `r10`, `r8` и `r9`.

Системный вызов не может принять больше шести аргументов.

3. Выполнить инструкцию `syscall`.

Не важно, в каком порядке инициализированы регистры. Заметьте, что инструкция `syscall` изменяет `rcx` и `r11`! Причину мы объясним позже. Когда мы писали программу «Hello, world!», мы использовали простой системный вызов `write`. Он принимает

1. *Дескриптор файла*;
2. Адрес буфера. Отсюда мы начинаем брать последовательные байты для записи;
3. Количество байт для записи.

Чтобы скомпилировать нашу первую программу, сохраните код в `hello.asm`[4], а затем выполните эти команды в оболочке:

```
> nasm -felf64 hello.asm -o hello.o
> ld -o hello hello.o
> chmod u+x hello
```

Детали процесса компиляции вместе с этапами компиляции будут обсуждаться в Главе 5. Давайте запустим «Hello, world!»

```
> ./hello
hello, world!
Segmentation fault
```

Мы отчётливо вывели то, что хотели. Однако программа, похоже, вызвала ошибку. Что мы сделали не так? После выполнения системного вызова программа продолжает свою работу. Мы не написали никаких инструкций после `syscall`, но память в следующих ячейках действительно хранит какие-то случайные значения.

> **Примечание** Если вы ничего не поместили по какому-то адресу памяти, там наверняка будет храниться какой-то мусор, а не нули или какие-либо корректные инструкции.

Процессор понятия не имеет, предназначались ли эти значения для кодирования инструкций или нет. Поэтому, следуя своей природе, он пытается их интерпретировать, потому что на них указывает регистр `rip`. Крайне маловероятно, что эти значения кодируют корректные инструкции, поэтому произойдёт прерывание с кодом 6 (недопустимая инструкция).[5] Так что же нам делать? Нам нужно использовать системный вызов `exit`, который корректно завершает программу, как показано в листинге 2-4.

Листинг 2-4. hello_proper_exit.asm

```
section .data
message: db 'hello, world!', 10
section .text
global _start
_start:
    mov     rax, 1           ; 'write' syscall number
    mov     rdi, 1           ; stdout descriptor
    mov     rsi, message     ; string address
    mov     rdx, 14          ; string length in bytes
    syscall
    mov     rax, 60          ; 'exit' syscall number
    xor     rdi, rdi
    syscall
```

> **Вопрос 11** Что делает инструкция `xor rdi, rdi`?

> **Вопрос 12** Каков код возврата программы?

> **Вопрос 13** Каков первый аргумент системного вызова `exit`?

2.3 Пример: вывод содержимого регистра

Время попробовать что-то чуть более сложное. Давайте выведем значение `rax` в шестнадцатеричном формате, как показано в листинге 2-5.

Листинг 2-5. Вывод значения rax: `print_rax.asm`

```
section .data
codes:
    db      '0123456789ABCDEF'
section .text
global _start
_start:
    ; number 1122... in hexadecimal format
    mov rax, 0x1122334455667788
    mov rdi, 1
    mov rdx, 1
    mov rcx, 64
   ; Each 4 bits should be output as one hexadecimal digit
   ; Use shift and bitwise AND to isolate them
   ; the result is the offset in 'codes' array
.loop:
    push rax
    sub rcx, 4
   ; cl is a register, smallest part of rcx
   ; rax -- eax -- ax -- ah + al
   ; rcx -- ecx -- cx -- ch + cl
    sar rax, cl
    and rax, 0xf
    lea rsi, [codes + rax]
    mov rax, 1
   ; syscall leaves rcx and r11 changed
    push rcx
    syscall
    pop rcx
    pop rax
   ; test can be used for the fastest 'is it a zero?' check
   ; see docs for 'test' command
    test rcx, rcx
    jnz .loop
    mov     rax, 60 ;          invoke 'exit' system call
    xor      rdi, rdi
    syscall
```

Сдвигая значение `rax` и применяя к нему логическое И с маской `0xF`, мы преобразуем всё число в одну из его шестнадцатеричных цифр. Каждая цифра — это число от 0 до 15. Используйте его как индекс и прибавьте к адресу метки `codes`, чтобы получить представляющий её символ.

Например, при `rax = 0x4A` мы используем индексы `0x4 =` 4_{10} и `0xA =` 10 [6] Первый даст нам

символ `'4'`, чей код `0x34`. Второй даст символ `'a'`, чей код `0x61`.

> **Вопрос 14** Проверьте, что ASCII-коды, упомянутые в последнем примере, верны.

Мы можем использовать аппаратный стек для сохранения и восстановления значений регистров, например вокруг инструкции `syscall`.

> **Вопрос 15** В чём разница между `sar` и `shr`? Проверьте по документации Intel.

> **Вопрос 16** Как записать числа в разных системах счисления понятным для NASM способом? Проверьте документацию NASM.

> **Примечание** Когда программа запускается, значение большинства регистров не определено чётко (оно может быть абсолютно случайным). Это частый источник ошибок новичков, так как есть склонность предполагать, что они обнулены.

2.3.1 Локальные метки

Обратите внимание на необычное имя метки `.loop`: оно начинается с точки. Эта метка локальная. Мы можем повторно использовать имена меток, не вызывая конфликтов имён, пока они локальные. Последняя использованная глобальная метка без точки — базовая для всех последующих локальных меток (до следующей глобальной метки). Полное имя метки `.loop` — `_start.loop`. Мы можем использовать это имя, чтобы обратиться к ней из любого места программы, даже после появления других глобальных меток.

2.3.2 Относительная адресация

Это демонстрирует, как адресовать память более сложным способом, чем просто по непосредственному адресу.

Листинг 2-6. Относительная адресация: `print_rax.asm`

```
lea rsi, [codes + rax]
```

Квадратные скобки обозначают *косвенную адресацию*; адрес записывается внутри них.

- mov rsi, rax — копирует rax в rsi
- `mov rsi, [rax]` — копирует содержимое памяти (8 последовательных байт), начиная с адреса, хранящегося в `rax`, в `rsi`. Как мы узнаём, что нужно скопировать именно 8 байт? Как мы знаем, операнды `mov` должны быть одного размера, а размер `rsi` — 8 байт. Зная эти факты, ассемблер способен вывести, что из памяти нужно взять ровно 8 байт.

Между инструкциями `lea` и `mov` есть тонкое различие в их значении. `lea` означает «load effective address» (загрузить исполнительный адрес). Она позволяет вычислить адрес ячейки памяти и сохранить его куда-либо. Это не всегда тривиально, потому что существуют хитрые режимы адресации (как мы увидим позже): например, адрес может быть суммой нескольких операндов. Листинг 2-7 даёт быструю демонстрацию того, что делают `lea` и `mov`.

Листинг 2-7. `lea_vs_mov.asm`

```
; rsi <- address of label 'codes', a number
mov rsi, codes
; rsi <- memory contents starting at 'codes' address
; 8 consecutive bytes are taken because rsi is 8 bytes long
mov rsi, [codes]
; rsi <- address of 'codes'
; in this case it is equivalent of mov rsi, codes
; in general the address can contain several components
lea rsi, [codes]
; rsi <- memory contents starting at (codes+rax)
mov rsi, [codes + rax]
; rsi <- codes + rax
; equivalent of combination:
; -- mov rsi, codes
; -- add rsi, rax
; Can't do it with a single mov!
lea rsi, [codes + rax]
```

2.3.3 Порядок выполнения

Все команды выполняются последовательно, кроме случаев, когда встречаются специальные инструкции перехода. Существует инструкция безусловного перехода `jmp addr`. Её можно рассматривать как замену `mov rip, addr`.[7] Условные переходы опираются на содержимое регистра `rflags`. Например, `jz address` переходит по адресу, только если установлен флаг нуля. Обычно для настройки нужных флагов вместе с инструкцией условного перехода используют инструкцию `test` или `cmp`. `cmp` вычитает второй операнд из первого; результат нигде не сохраняется, но на его основе устанавливаются со-

ответствующие флаги (например, если операнды равны, будет установлен флаг нуля). `test` делает то же самое, но использует логическое И вместо вычитания. Пример, показанный в листинге 2-8, записывает 1 в `rbx`, если `rax` < 42, и 0 в противном случае.

Листинг 2-8. `jumps_example.asm`

```
    cmp rax, 42
    jl yes
    mov rbx, 0
    jmp ex
yes:
    mov rbx, 1
ex:
```

Распространённый (и быстрый) способ проверить значение регистра на равенство нулю — инструкция `test reg,reg`. Для каждого арифметического флага *F* существует как минимум две команды: `jF` и `jnF`. Например, для флага знака: `js` и `jns`. Другие полезные команды включают

1. `ja` (переход, если выше)/ `jb` (переход, если ниже) для перехода после сравнения *беззнаковых чисел* с помощью `cmp`.
2. `jg` (переход, если больше)/ `jl` (переход, если меньше) для *знаковых*.
3. `jae` (переход, если выше или равно), `jle` (переход, если меньше или равно) и подобные. Некоторые распространённые инструкции перехода показаны в листинге 2-9.

Листинг 2-9. Инструкции перехода: `jumps.asm`

```
mov rax, -1
mov rdx, 2
cmp rax, rdx
jg location
ja location           ; different logic!
cmp rax, rdx
je  location          ; if rax equals rdx
jne location          ; if rax is not equal to rdx
```

> **Вопрос 17** В чём разница между `je` и `jz`?

2.4 Вызовы функций

Процедуры (функции) позволяют изолировать часть логики программы и использовать её как чёрный ящик. Это необходимый механизм для обеспечения абстракции. Абстракция позволяет строить более сложные системы, инкапсулируя сложные алгоритмы под непрозрачными интерфейсами. Для выполнения вызовов используется инструкция `call <address>`. Она делает именно следующее:

```
push rip
jmp <address>
```

Адрес, теперь хранящийся в стеке (бывшее содержимое `rip`), называется *адресом возврата*. Любая функция может принимать неограниченное число аргументов. Первые шесть аргументов передаются соответственно в `rdi`, `rsi`, `rdx`, `rcx`, `r8` и `r9`. Остальные передаются через стек в обратном порядке. Что мы считаем концом процедуры — не вполне ясно. Проще всего сказать, что конец функции обозначает инструкция `ret`. Её семантика полностью эквивалентна `pop rip`. Судя по всему, хрупкий механизм `call` и `ret` работает только тогда, когда состояние стека тщательно контролируется. Не следует вызывать `ret`, если стек не находится точно в том же состоянии, в котором он был на начало функции. Иначе процессор возьмёт то, что лежит на вершине стека, в качестве адреса возврата и использует это как новое содержимое `rip`, что наверняка приведёт к выполнению мусора. Теперь поговорим о том, как функции используют регистры. Очевидно, что выполнение функции *может изменить регистры*. Есть два типа регистров.

- *Регистры, сохраняемые вызываемой функцией*, должны быть восстановлены вызываемой процедурой. Так что, если ей

нужно их изменить, она должна вернуть их обратно.

Эти регистры сохраняются вызываемой функцией: `rbx`, `rbp`, `rsp`, `r12–r15`, всего семь регистров.

- *Регистры, сохраняемые вызывающей функцией*, должны быть сохранены перед вызовом функции и восстановлены после. Не

обязательно сохранять и восстанавливать их, если их значение не будет важно впоследствии.

Все остальные регистры сохраняются вызывающей функцией. Эти две категории *являются соглашением*. То есть программист должен следовать этому соглашению, а именно

- Сохранять и восстанавливать регистры, сохраняемые вызываемой функцией.

- Всегда помнить, что регистры, сохраняемые вызывающей функцией, могут быть изменены во время выполнения функции.

> **Источник ошибок** Распространённая ошибка — не сохранять регистры, сохраняемые вызывающей функцией, перед `call` и использовать их после возврата из функции. Запомните:

1. Если вы меняете `rbx`, `rbp`, `rsp` или `r12–r15`, верните их обратно!
2. Если вам нужно, чтобы какой-либо другой регистр пережил вызов функции, сохраните его сами перед вызовом!

Некоторые функции могут *возвращать значение*. Это значение обычно и есть сама суть того, зачем функция написана и выполняется. Например, мы можем написать функцию, которая принимает число в качестве аргумента и возвращает его квадрат. С точки зрения реализации мы возвращаем значения, сохраняя их в `rax` перед завершением выполнения функции. Если вам нужно вернуть два значения, для второго можно использовать `rdx`. Итак, шаблон вызова функции выглядит так:

- Сохраните все регистры, сохраняемые вызывающей функцией, которые должны пережить вызов функции

(для этого можно использовать `push`).

- Поместите аргументы в соответствующие регистры (`rdi`, `rsi` и т. д.).
- Вызовите функцию с помощью `call`.
- После возврата из функции `rax` будет содержать возвращаемое значение.
- Восстановите регистры, сохраняемые вызывающей функцией, сохранённые перед вызовом функции.

> **Зачем нужны соглашения?** Функция используется для абстрагирования части логики, полностью забывая о её внутренней реализации и меняя её при необходимости. Такие изменения должны быть полностью прозрачны для внешней программы. Описанное ранее соглашение позволяет вызвать любую функцию из любого места и быть уверенным в её эффектах (может изменить любой регистр, сохраняемый вызывающей функцией; оставит нетронутыми регистры, сохраняемые вызываемой функцией).

Некоторые системные вызовы также возвращают значения — будьте внимательны и читайте документацию! Вы никогда не должны использовать `rbp` и `rsp`. Они неявно используются во время выполнения. Как вы уже знаете, `rsp` используется как указатель стека.

> **Об аргументах системных вызовов** Аргументы для системных вызовов хранятся в другом наборе регистров, отличном от того, что используется для функций. Четвёртый аргумент хранится в `r10`, тогда как функция принимает четвёртый аргумент в `rcx`!

Причина в том, что инструкция `syscall` неявно использует `rcx`. Системные вызовы не могут принимать больше шести аргументов. Если вы не будете следовать описанному соглашению, вы не сможете менять свои функции, не внося ошибок в местах их вызова. Теперь настало время написать ещё две функции: `print_newline` выведет символ новой строки; `print_hex` примет число и выведет его в шестнадцатеричном формате (см. листинг 2-10).

Листинг 2-10. `print_call.asm`

```
section .data
newline_char: db 10
codes: db '0123456789abcdef'
section .text
global _start
print_newline:
    mov rax, 1            ; 'write' syscall identifier
    mov rdi, 1            ; stdout file descriptor
    mov rsi, newline_char ; where do we take data from
    mov rdx, 1            ; the amount of bytes to write
    syscall
   ret
print_hex:
    mov rax, rdi
    mov rdi, 1
    mov rdx, 1
    mov rcx, 64           ; how far are we shifting rax?
iterate:
    push rax              ; Save the initial rax value
    sub rcx, 4
    sar rax, cl           ; shift to 60, 56, 52, ... 4, 0
                          ; the cl register is the smallest part of rcx
    and rax, 0xf          ; clear all bits but the lowest four
    lea rsi, [codes + rax]; take a hexadecimal digit character code
    mov rax, 1            ;
    push rcx              ; syscall will break rcx
    syscall               ; rax = 1 (31) -- the write identifier,
                          ;   rdi = 1 for stdout,
                          ; rsi = the address of a character, see line 29
    pop rcx
    pop rax               ; ^ see line 24 ^
    test rcx, rcx         ; rcx = 0 when all digits are shown
    jnz iterate
    ret
_start:
    mov rdi, 0x1122334455667788
    call print_hex
    call print_newline
    mov rax, 60
    xor rdi, rdi
    syscall
```

2.5 Работа с данными

2.5.1 Порядок байтов

Давайте попробуем вывести значение, хранящееся в памяти, с помощью только что написанной функции. Мы сделаем это двумя разными способами: сначала мы перечислим все его байты по отдельности, а затем наберём его как обычно (см. листинг 2-11).

Листинг 2-11. `endianness.asm`

```
section .data
demo1: dq 0x1122334455667788
demo2: db 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88
section .text
_start:
    mov rdi, [demo1]
    call print_hex
    call print_newline
    mov rdi, [demo2]
    call print_hex
    call print_newline
    mov rax, 60
    xor rdi, rdi
    syscall
```

Когда мы запускаем её, к нашему удивлению, мы получаем совершенно разные результаты для `demo1` и `demo2`.

```
> ./main
1122334455667788
8877665544332211
```

Как мы видим, многобайтовые числа хранятся в *обратном* порядке! Биты внутри каждого байта хранятся привычным образом, но байты хранятся от наименее значимого к наиболее значимому. Это применимо только к операциям с памятью: в регистрах байты хранятся естественным образом. У разных процессоров разные соглашения о том, как хранятся байты.

- При *обратном порядке байтов (big-endian)* многобайтовые числа хранятся в памяти начиная с *наиболее* значимых байтов.
- При *прямом порядке байтов (little-endian)* многобайтовые числа хранятся в памяти начиная с *наименее* значимых байтов.

Как показывает пример, Intel 64 следует соглашению little-endian. В общем случае выбор одного соглашения вместо другого — вопрос решения, принятого инженерами оборудования. Эти соглашения не касаются массивов и строк. Однако, если каждый символ кодируется 2 байтами вместо 1, эти байты будут храниться в обратном порядке. Преимущество little-endian в том, что можно отбросить наиболее значимые байты, эффективно преобразуя число из более широкого формата в более узкий, например 8 байт. Например, `demo3: dq 0x1234`. Тогда, чтобы преобразовать это число в `dw`, нужно прочитать число `dword`, начиная с того же адреса `demo3`. Полную раскладку по памяти см. в таблице 2-1.

Таблица 2-1. Little endian и big endian для учетверённого слова 0x1234

АДРЕС	ЗНАЧЕНИЕ – LE	ЗНАЧЕНИЕ – BE
demo3	0x34	0x00
demo3 + 1	0x12	0x00
demo3 + 2	0x00	0x00
demo3 + 3	0x00	0x00
demo3 + 4	0x00	0x00
demo3 + 5	0x00	0x00
demo3 + 6	0x00	0x12
demo3 + 7	0x00	0x34

Big endian — родной формат, часто используемый внутри сетевых пакетов (например, TCP/IP). Это также внутренний формат чисел для виртуальной машины Java. *Middle endian* — не очень известное понятие. Предположим, мы хотим создать набор процедур для выполнения арифметики над 128-битными числами. Тогда байты можно хранить следующим образом: сначала 8 наименее значимых байт в обратном порядке, а затем 8 наиболее значимых байт тоже в обратном порядке:

7 6 5 4 3 2 1 0, 16 15 14 13 12 11 10 9 8

2.5.2 Строки

Как мы уже знаем, символы кодируются с помощью таблицы ASCII. Каждому символу присвоен код. Строка, очевидно, представляет собой последовательность кодов символов. Однако это ничего не говорит о том, как определить её длину.

1. Строки начинаются со своей явной длины.

```
db 27, 'Selling England by the Pound'
```

2. Специальный символ обозначает конец строки. По традиции используется нулевой код.

Такие строки называются *null-terminated* (завершающиеся нулём).

```
db 'Selling England by the Pound', 0
```

2.5.3 Предварительное вычисление констант

Нередко можно встретить такой код:

```
lab: db 0
...
   mov rax, lab + 1 + 2*3
```

NASM поддерживает арифметические выражения со скобками и битовыми операциями. Такие выражения могут включать только константы, известные компилятору. Таким образом он может заранее вычислить все такие выражения и вставить результаты вычислений (в виде числовых констант) в исполняемый код. Так что такие выражения НЕ вычисляются во время выполнения. Аналог во время выполнения потребовал бы использования таких инструкций, как `add` или `mul`.

2.5.4 Указатели и разные типы адресации

Указатели — это адреса ячеек памяти. Они могут храниться в памяти или в регистрах. Размер указателя — 8 байт. Данные обычно занимают несколько ячеек памяти (то есть несколько последовательных адресов). Указатели не содержат никакой информации о длине данных, на которые они указывают. При попытке записать куда-либо значение,

чей размер не указан и не может быть выведен (например, `mov [myvariable], 4`), можно получить ошибки компиляции. В таких случаях нужно явно указать размер, как показано ниже:

```
section .data
test: dq -1
section .text
mov byte[test], 1 ;1
mov word[test], 1 ;2
mov dword[test], 1 ;4
mov qword[test], 1 ;8
```

> **Вопрос 18** Чему равен `test` после каждой из перечисленных выше команд?

Посмотрим, как можно закодировать операнды в инструкциях.

1. Непосредственно:

Инструкция сама по себе содержится в памяти. Операнды в некоторой форме — её части; у этих частей есть собственные адреса. Многие инструкции могут содержать сами значения операндов.

Вот так число 10 перемещается в `rax`.

```
mov rax, 10
```

2. Через регистр: эта инструкция переносит значение `rbx` в `rax`.

```
mov rax, rbx
```

3. Прямой адресацией памяти: эта инструкция переносит 8 байт, начиная с десятого адреса, в `rax`:

```
mov rax, [10]
```

Мы также можем взять адрес из регистра:

```
mov r9, 10
mov  rax, [r9]
```

Мы можем использовать предварительные вычисления:

```
buffer: dq 8841, 99, 00
...
mov rax, [buffer+8]
```

Адрес внутри этой инструкции был вычислен заранее, потому что и база, и смещение — константы, известные компилятору. Теперь это просто число.

4. База-индекс с масштабом и смещением Большинство режимов адресации обобщаются этим режимом. Адрес здесь вычисляется на основе следующих компонентов: *Адрес = база + индекс * масштаб + смещение*

- База — либо непосредственное значение, либо регистр;
- Масштаб может быть только непосредственным значением, равным 1, 2, 4 или 8;
- Индекс — непосредственное значение или регистр; и
- Смещение всегда непосредственное значение.

В листинге 2-12 показаны примеры разных типов адресации.

Листинг 2-12. `addressing.asm`

```
mov rax, [rbx + 4* rcx + 9]
mov rax, [4*r9]
mov rdx, [rax + rbx]
lea rax, [rbx + rbx * 4]     ; rax = rbx * 5
add r8, [9 + rbx*8 + 7]
```

Общая картина Можно думать о `byte`, `word` и т. д. как о спецификаторах типа. Например, можно поместить в стек 16-, 32- или 64-битные числа. Инструкция `push 1` не даёт понять, сколько бит занимает операнд. Точно так же `mov word[test]`, `1` означает, что `[test]` — это `word;` в `push word 1` закодирована информация о формате числа.

2.6 Пример: вычисление длины строки

Начнём с написания функции для вычисления длины строки, завершающейся нулём. Поскольку у нас пока нет процедуры для вывода чего-либо в стандартный вывод, единственный способ вывести значение — вернуть его как код возврата через системный вызов `exit`. Чтобы увидеть код возврата последнего процесса, используйте переменную `$?`.

```
> true
> echo $?
0
> false
> echo $?
1
```

Давайте напишем программу на ассемблере, имитирующую команду оболочки `false`, как показано в листинге 2-13.

Листинг 2-13. `false.asm`

```
global _start
section .text
_start:
    mov rdi, 1
    mov rax, 60
    syscall
```

Теперь у нас есть всё необходимое для вычисления длины строки. В листинге 2-14 показан код.

Листинг 2-14. Длина строки: `strlen.asm`

```
global _start
section .data
test_string: db "abcdef", 0
section .text
strlen:                   ; by our convention, first and the only argument
                          ; is taken from rdi
    xor rax, rax          ; rax will hold string length. If it is not
                          ; zeroed first, its value will be totally random
.loop:                    ; main loop starts here
    cmp byte [rdi+rax], 0 ; Check if the current symbol is null-terminator.
                          ; We absolutely need that 'byte' modifier since
                          ; the left and the right part of cmp should be
                          ; of the same size. Right operand is immediate
                          ; and holds no information about its size,
                          ; hence we don't know how many bytes should be
                          ; taken from memory and compared to zero
    je .end               ; Jump if we found null-terminator
    inc rax               ; Otherwise go to next symbol and increase
                          ; counter
    jmp .loop
.end:
    ret                   ; When we hit 'ret', rax should hold return value
_start:
    mov rdi, test_string
    call strlen
    mov rdi, rax
    mov rax, 60
    syscall
```

Важная часть (и единственная, которую мы оставим) — функция `strlen`. Заметьте, что

1. `strlen` изменяет регистры, так что после выполнения `call strlen` регистры *могут* изменить свои значения.
2. `strlen` не изменяет `rbx` или любые другие регистры, сохраняемые вызываемой функцией.

> **Вопрос 19** Можете ли вы найти одну или две ошибки в листинге 2-15? Когда они возникнут?

Листинг 2-15. Альтернативная версия strlen: `strlen_bug1.asm`

```
global _start
section .data
test_string: db "abcdef", 0
section .text
strlen:
.loop:
    cmp byte [rdi+r13], 0
    je .end
    inc r13
    jmp .loop
.end:
    mov rax, r13
    ret
_start:
    mov rdi, test_string
    call strlen
    mov rdi, rax
    mov rax, 60
    syscall
```

2.7 Задание: библиотека ввода/вывода

Прежде чем заняться чем-то по-настоящему интересным, нам нужно убедиться, что не придётся писать одни и те же базовые процедуры снова и снова. Пока что у нас нет ничего; даже получение ввода с клавиатуры — целая морока. Так что давайте построим небольшую библиотеку базовых функций ввода и вывода. Сначала вам нужно прочитать документацию Intel [15] по следующим инструкциям (помните, все они подробно описаны во втором томе):

- `xor`
- `jmp`, `ja` и подобные
- cmp
- mov
- inc, dec
- add, imul, mul, sub, idiv, div
- neg
- call, ret
- push, pop

Эти команды для нас основные, и вы должны хорошо их знать. Как вы могли заметить, Intel 64 поддерживает тысячи команд. Разумеется, нам нет нужды в них погружаться. Использование системных вызовов вместе с перечисленными выше инструкциями позволит нам добраться практически куда угодно. Вам также нужно прочитать документацию по системному вызову `read`. Его код — 0; в остальном он похож на `write`. Обратитесь к Приложению C в случае затруднений. Отредактируйте `lib.inc` и предоставьте определения функций вместо заглушек — инструкций `xor rax, rax`. Обратитесь к таблице 2-2, чтобы узнать требуемую семантику функций. Мы рекомендуем реализовывать их в указанном порядке, потому что иногда вы сможете повторно использовать свой код, вызывая уже написанные вами функции.

Таблица 2-2. Функции библиотеки ввода/вывода

ФУНКЦИЯ	ОПРЕДЕЛЕНИЕ
`exit`	Принимает код возврата и завершает текущий процесс.
`string_length`	Принимает указатель на строку и возвращает её длину.
`print_string`	Принимает указатель на строку, завершающуюся нулём, и выводит её в *stdout*.
`print_char`	Принимает код символа непосредственно как свой первый аргумент и выводит его в *stdout*.
`print_newline`	Выводит символ с кодом 0xA.
`print_uint`	Выводит беззнаковое 8-байтовое целое число в десятичном формате. Мы предлагаем создать буфер на стеке[8] и сохранять в нём результаты деления. Каждый раз вы делите последнее значение на 10 и сохраняете соответствующую цифру в буфере. Не забудьте, что каждую цифру нужно преобразовать в её ASCII-код (например, `0x04` становится `0x34`) .
`print_int`	Выводит знаковое 8-байтовое целое число в десятичном формате.
`read_char`	Читает один символ из *stdin и возвращает его*. Если поток ввода закончился, возвращает 0.
`read_word`	Принимает адрес буфера и его размер в качестве аргументов. Читает следующее слово из *stdin* (пропуская пробельные символы[9]) в буфер. Останавливается и возвращает 0, если слово не помещается в указанный буфер; иначе возвращает адрес буфера. Эта функция должна завершить принятую строку нулём.
`parse_uint`	Принимает строку, завершающуюся нулём, и пытается разобрать беззнаковое число с её начала. Возвращает разобранное число в `rax` , количество его символов в `rdx` .
`parse_int`	Принимает строку, завершающуюся нулём, и пытается разобрать знаковое число с её начала. Возвращает разобранное число в `rax` ; количество его символов в `rdx` (включая знак, если он есть). Пробелы между знаком и цифрами не допускаются.
`string_equals`	Принимает два указателя на строки и сравнивает их. Возвращает 1, если они равны, иначе 0.
`string_copy`	Принимает указатель на строку, указатель на буфер и длину буфера. Копирует строку в место назначения. Возвращает адрес назначения, если строка помещается в буфер; иначе возвращает ноль.

Используйте `test.py` для автоматизированной проверки корректности. Просто запустите его, и он сделает всё остальное. Помните, что строке из n символов нужно $n + 1$ байт для хранения в памяти из-за нуль-терминатора. Прочитайте Приложение A, чтобы узнать, как выполнить программу пошагово, наблюдая за изменениями значений регистров и состояния памяти.

2.7.1 Самопроверка

Перед тестированием или при получении неожиданного результата проверьте следующий короткий список:

1. Метки, обозначающие функции, должны быть глобальными; остальные — локальными.
2. Не предполагайте, что регистры «по умолчанию» содержат ноль.
3. Сохраняйте и восстанавливайте регистры, сохраняемые вызываемой функцией, если вы их используете.
4. Сохраняйте нужные вам регистры, сохраняемые вызывающей функцией, перед `call` и восстанавливайте их после.
5. Не используйте буферы в `.data`. Вместо этого выделяйте их на стеке, что

позволяет при необходимости адаптироваться к многопоточности.

6. Ваши функции принимают аргументы в `rdi`, `rsi`, `rdx`, `rcx`, `r8` и `r9`.
7. Не выводите числа по одной цифре. Вместо этого преобразуйте их в строки

символов и используйте `print_string`.

8. `parse_int` и `parse_uint` корректно устанавливают `rdx`. Это будет действительно важно в

следующем задании.

9. Все функции разбора и `read_word` работают, когда ввод завершается через Ctrl-D.

Сделанный правильно, код не займёт больше 250 строк.

> **Вопрос 20** Попробуйте переписать `print_newline` без вызова `print_char` или копирования его кода. Подсказка: почитайте про оптимизацию хвостового вызова.

Вопрос 21 Попробуйте переписать `print_int` без вызова `print_uint` или копирования его кода. Подсказка: почитайте про оптимизацию хвостового вызова.

Вопрос 22 Попробуйте переписать `print_int` без вызова `print_uint`, копирования его кода или использования `jmp`. Вам понадобится только одна инструкция и аккуратное размещение кода. Почитайте про сопрограммы.

2.8 Итоги

В этой главе мы начали делать настоящие вещи и применять наши базовые знания о языке ассемблера. Мы надеемся, что вы преодолели весь возможный страх перед ассемблером. Несмотря на крайнюю многословность, это не сложный в использовании язык. Мы научились делать ветвления и циклы, выполнять базовую арифметику и системные вызовы; мы также увидели разные режимы адресации, прямой и обратный порядок байтов. В следующих заданиях по ассемблеру будет использоваться небольшая библиотека, которую мы построили, чтобы упростить взаимодействие с пользователем.

Вопрос 23 Какая связь между `rax`, `eax`, `ax`, `ah` и `al`?

Вопрос 24 Как получить доступ к частям `r9`?

Вопрос 25 Как можно работать с аппаратным стеком? Опишите инструкции, которые можно использовать.

Вопрос 26 Какие из этих инструкций некорректны и почему?

```
mov [rax], 0
cmp [rdx], bl
mov bh, bl
mov al, al
add bpl, 9
add [9], spl
mov r8d, r9d
mov r3b, al
mov r9w, r2d
mov rcx, [rax + rbx + rdx]
mov r9, [r9 + 8*rax]
mov [r8+r7+10], 6
mov [r8+r7+10], r6
```

Вопрос 27 Перечислите регистры, сохраняемые вызываемой функцией

Вопрос 28 Перечислите регистры, сохраняемые вызывающей функцией

Вопрос 29 Каково назначение регистра `rip` ?

Вопрос 30 Что такое флаг SF?

Вопрос 31 Что такое флаг ZF?

Вопрос 32 Опишите эффекты следующих инструкций:

- sar
- shr
- xor
- jmp
- `ja, jb` и подобные.
- cmp
- mov
- inc,dec
- add

- imul, mul
- sub
- idiv, div
- call, ret
- push, pop

Вопрос 33 Что такое метка и есть ли у неё размер?

Вопрос 34 Как проверить, содержится ли целое число в определённом диапазоне (x, y)?

Вопрос 35 В чём разница между `ja/jb` и `jg/jl`?

Вопрос 36 В чём разница между `je` и `jz`?

Вопрос 37 Как проверить, равен ли `rax` нулю, без команды `cmp`?

Вопрос 38 Каков код возврата программы?

Вопрос 39 Как умножить `rax` на 9, используя ровно одну инструкцию?

Вопрос 40 Используя ровно две инструкции (первая — `neg`), возьмите модуль целого числа, хранящегося в `rax`.

Вопрос 41 В чём разница между прямым и обратным порядком байтов?

Вопрос 42 Какой тип адресации самый сложный?

Вопрос 43 Где начинается выполнение программы?

Вопрос 44 `rax = 0x1122334455667788`. Мы выполнили `push rax`. Каким будет содержимое

> **Примечание.** байта по адресу `[rsp+3]` ?

Наследие

Эта глава познакомит вас с устаревшими режимами процессора, которые больше не используются, и с их *в основном* устаревшими возможностями, которые всё ещё актуальны сегодня. Вы увидите, как эволюционировали процессоры, и узнаете детали реализации колец защиты (привилегированный и пользовательский режим). Вы также поймёте назначение глобальной таблицы дескрипторов. Хотя эта информация помогает лучше понять архитектуру, она не критична для программирования на ассемблере в пользовательском пространстве. По мере эволюции процессоров каждый новый режим увеличивал длину машинного слова и добавлял новые возможности. Процессор может работать в одном из следующих режимов:

- Реальный режим (самый древний, 16-битный);
- Защищённый (обычно называемый 32-битным);
- Виртуальный (для эмуляции реального режима внутри защищённого);
- Режим управления системой (для спящего режима, управления питанием и т. д.);
- Длинный режим, с которым мы уже немного знакомы.

Мы подробнее рассмотрим реальный и защищённый режимы.

3.1 Реальный режим

Реальный режим — самый древний. В нём нет виртуальной памяти; физическая память адресуется напрямую, а регистры общего назначения 16-битные. Так что ни `rax`, ни `eax` пока не существуют, а вот `ax`, `al` и `ah` — существуют. Такие регистры могут хранить значения от 0 до 65535, поэтому с помощью одного из них можно адресовать 65536 байт. Такая область памяти называется *сегментом*. Не путайте его с сегментами защищённого режима или секциями файлов формата ELF (Executable and Linkable Format)! Вот регистры, доступные в реальном режиме:

- ip, flags;
- ax, bx, cx, dx, sp, bp, si, di;
- Сегментные регистры: `cs`, `ds`, `ss`, `es` (позже также `gs` и `fs`).

Поскольку адресовать больше 64 килобайт памяти было не так просто, инженеры придумали решение — использовать специальные *сегментные регистры* следующим образом:

- Каждый физический адрес состоит из 20 бит (то есть 5 шестнадцатеричных цифр).
- Каждый логический адрес состоит из двух компонентов. Один берётся из сегментного регистра

и кодирует начало сегмента. Другой — смещение внутри этого сегмента. Оборудование вычисляет физический адрес из этих компонентов следующим образом:

физический адрес = база сегмента *** 16 + смещение

Часто можно увидеть адреса, записанные в форме `segment:offset`, например:

```
4a40:0002, ds:0001, 7bd3:ah.
```

Как мы уже говорили, программисты хотят отделять код от данных (и стека), поэтому они предполагают использовать разные сегменты для этих секций кода. Сегментные регистры специализированы для этого: `cs` хранит начальный адрес сегмента кода, `ds` соответствует сегменту данных, а `ss` — сегменту стека. Остальные сегментные регистры используются для хранения дополнительных сегментов данных. Заметьте, что строго говоря, сегментные регистры хранят не начальные адреса сегментов, а скорее их части (четыре старшие шестнадцатеричные цифры). Добавив ещё одну нулевую цифру, то есть умножив на 16_{10}, мы получаем настоящий начальный адрес сегмента. Каждая инструкция, обращающаяся к памяти, неявно предполагает использование одного из сегментных регистров. Документация уточняет сегментные регистры по умолчанию для каждой инструкции. Впрочем, здравый смысл тоже помогает. Например, `mov` используется для работы с данными, поэтому адрес относителен сегменту данных.

```
mov al, [0004]   ; === mov al, ds:0004
```

Можно явно переопределить сегмент:

```
mov al, cs:[0004]
```

Когда программа загружается, загрузчик настраивает регистры `ip`, `cs`, `ss` и `sp` так, чтобы `cs:ip` соответствовал точке входа, а `ss:sp` указывал на вершину стека. Центральный процессор (CPU) всегда стартует в реальном режиме, а затем главный загрузчик обычно выполняет код, чтобы явно переключить его в защищённый режим, а затем в длинный режим. У реального режима есть множество недостатков.

- Он сильно затрудняет многозадачность. Одно и то же адресное пространство разделяется между всеми

программами, поэтому их приходится загружать по разным адресам. Их относительное расположение обычно приходится определять во время компиляции.

- Программы могут перезаписывать код друг друга или даже код операционной системы, поскольку все они живут в

одном адресном пространстве.

- Любая программа может выполнить любую инструкцию, включая те, что используются для настройки

состояния процессора. Некоторые инструкции должны использоваться только операционной системой (например, те, что настраивают виртуальную память, управляют питанием и т. д.), поскольку их некорректное использование может обрушить всю систему. Защищённый режим был призван решить эти проблемы.

3.2 Защищённый режим

Intel 80386 был первым процессором, реализовавшим защищённый 32-битный режим. Он предоставляет более широкие версии регистров (`eax`, `ebx`, ..., `esi`, `edi`), а также новые механизмы защиты: кольца защиты, виртуальную память и улучшенную сегментацию. Эти механизмы изолировали программы друг от друга, так что аварийное завершение одной из них не вредило остальным. Более того, программы больше не могли повреждать память других процессов. *Способ получения начального адреса сегмента* изменился по сравнению с реальным режимом. Теперь начало вычисляется на основе записи в специальной таблице, а не прямым умножением содержимого сегментного регистра.

Линейный адрес = база сегмента (взятая из системной таблицы) + смещение

Каждый из сегментных регистров `cs`, `ds`, `ss`, `es`, `gs` и `fs` хранит так называемый *селектор сегмента*, содержащий индекс в специальной таблице дескрипторов сегментов и немного дополнительной информации. Есть два типа таблиц дескрипторов сегментов: возможно, множественные *LDT (Local Descriptor Table, локальная таблица дескрипторов)* и только одна *GDT (Global Descriptor Table, глобальная таблица дескрипторов)*. LDT предназначались для аппаратного механизма переключения задач; однако производители операционных систем не восприняли этот подход. Сегодня программы изолируются виртуальной памятью, и LDT не используются. `GDTR` — регистр для хранения адреса и размера GDT. Селекторы сегментов устроены так, как показано на рисунке 3-1.

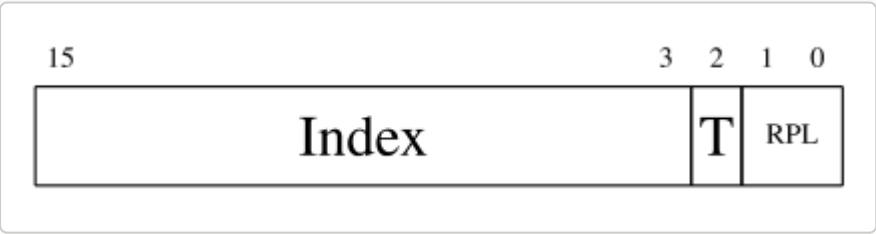


Рисунок 3-1. Селектор сегмента (содержимое любого сегментного регистра)

Индекс обозначает позицию дескриптора либо в GDT, либо в LDT. Бит `T` выбирает либо LDT, либо GDT. Поскольку LDT больше не используются, этот бит всегда будет равен нулю. Записи таблицы в GDT/LDT также хранят информацию о том, какой уровень при-

вилегий назначен описываемому сегменту. При обращении к сегменту через селектор сегмента выполняется проверка значения *Request Privilege Level* (RPL, запрашиваемого уровня привилегий) (хранящегося в селекторе = сегментном регистре) относительно *Descriptor Privilege Level* (DPL, уровня привилегий дескриптора) (хранящегося в таблице дескрипторов). Если RPL недостаточно привилегирован для доступа к высокопривилегированному сегменту, произойдёт ошибка. Таким образом мы могли бы создать множество сегментов с разными правами доступа и использовать значения RPL в селекторах сегментов, чтобы определить, какие из них доступны нам прямо сейчас (с учётом нашего уровня привилегий). *Уровни привилегий — это то же самое, что и кольца защиты!* Можно смело сказать, что текущий уровень привилегий (то есть текущее кольцо) хранится в двух младших битах `cs` или `ss` (эти числа должны совпадать). Именно это влияет на возможность выполнения определённых критических инструкций (например, изменения самой GDT). Легко заключить, что для `ds` изменение этих битов позволяет нам переопределить текущий уровень привилегий на менее привилегированный конкретно для доступа к данным выбранного сегмента. Например, мы сейчас находимся в кольце 0, а `ds= 0x02`. Хотя два младших бита `cs` и `ss` равны 0 (поскольку мы внутри кольца 0), мы не можем получить доступ к данным в сегменте с уровнем привилегий выше 2 (например, 1 или 0). Другими словами, поле `RPL` хранит, насколько мы привилегированы, когда запрашиваем доступ к сегменту. Сегменты, в свою очередь, приписаны к одному из четырёх колец защиты. При запросе доступа с определённым уровнем привилегий этот уровень должен быть выше уровня привилегий, приписанного самому сегменту.

> **Примечание** Нельзя изменить `cs` напрямую.

Рисунок 3-2 показывает формат дескриптора GDT[10].

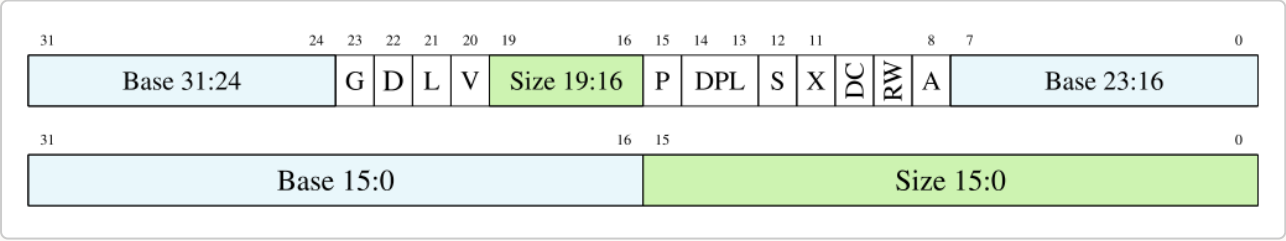


Рисунок 3-2. Дескриптор сегмента (внутри GDT или LDT)

G — Granularity (гранулярность), то есть единица размера: 0 = байты, 1 = страницы по 4096 байт каждая. D — размер операнда по умолчанию (0 = 16 бит, 1 = 32 бита). L — является ли сегмент сегментом 64-битного режима? V — доступен для использования системным ПО. P — присутствует ли сейчас в памяти. S — данные/код (1) или это просто носитель некоторой системной информации (0). X — данные (0) или код (1). RW — для сегмента данных: разрешена ли запись? (чтение разрешено всегда); для сегмента кода: разрешено ли чтение? (запись всегда запрещена). DC — направление роста: в сто-

рону меньших или больших адресов? (для сегмента данных); можно ли его выполнять с более высоких уровней привилегий? (если сегмент кода) A — обращались ли к нему? DPL — Descriptor Privilege Level (уровень привилегий дескриптора) (к какому кольцу он привязан?) Процессор всегда (даже сегодня) стартует в реальном режиме. Чтобы войти в защищённый режим, нужно создать GDT и настроить `gdtr`; установить специальный бит в `cr0` и выполнить так называемый *дальний переход*. Дальний переход означает, что сегмент (или селектор сегмента) указан явно (и потому может отличаться от используемого по умолчанию), как показано ниже:

```
jmp 0x08:addr
```

Листинг 3-1 показывает небольшой фрагмент того, как можно включить защищённый режим (предполагая, что `start32` — метка на начало 32-битного кода).

Листинг 3-1. Включение защищённого режима `loader_start32.asm`

```
lgdt cs:[_gdtr]
mov eax, cr0                ; !! Privileged instruction
or al, 1                    ; this is the bit responsible for protected mode
mov cr0, eax                ; !! Privileged instruction
    jmp (0x1 << 3):start32  ; assign first seg selector to cs
align 16
_gdtr:                      ; stores GDT's last entry index + GDT address
dw 47
dq _gdt
align 16
_gdt:
; Null descriptor (should be present in any GDT)
dd 0x00, 0x00
; x32 code descriptor:
db 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x9A, 0xCF,     0x00 ; differ by exec bit
; x32 data descriptor:
db 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x92, 0xCF,     0x00 ; execution off (0x92)
;  size  size  base  base  base  util  util|size  base
```

Директивы align управляют выравниванием, суть которого мы объясним позже в этой книге.

> **Вопрос 45** Расшифруйте этот селектор сегмента: `0x08`.

Можно подумать, что теперь каждая транзакция с памятью требует ещё одной для чтения содержимого GDT. Это не так: для каждого сегментного регистра существует так называемый *теневой регистр*, на который нельзя сослаться напрямую. Он служит кэшем содержимого GDT. Это означает, что как только селектор сегмента изменяется, в соответствующий теневой регистр загружается соответствующий дескриптор из GDT. Теперь этот регистр будет служить источником всей необходимой информации об этом сегменте. Флаг `D` требует небольшого пояснения, поскольку его смысл зависит от типа сегмента.

- Это сегмент кода: размеры адреса и операнда по умолчанию. Единица означает 32-битные адреса и 32-битные или 8-битные операнды; ноль соответствует 16-битным адресам и 16-битным или 8-битным операндам. Здесь мы говорим о кодировании машинных инструкций. Это поведение можно изменить, поставив перед инструкцией префикс `0x66` (чтобы изменить размер операнда) или `0x67` (чтобы изменить размер адреса).
- Сегмент стека (это сегмент данных, И мы говорим о том, что выбран через ss).[11] Здесь снова речь о размере операнда по умолчанию для `call`, `ret`, `push/pop` и т. д. Если флаг установлен, операнды 32-битные, и инструкции влияют на `esp`; иначе операнды 16-битные, и затрагивается `sp`.
- Для сегментов данных, растущих в сторону меньших адресов, он обозначает их пределы (0 для 64 КБ, 1 для 4 ГБ). Этот бит всегда должен быть установлен в длинном режиме.

Как видите, сегментация — довольно громоздкий зверь. Есть причины, по которым она не получила широкого распространения ни среди операционных систем, ни среди программистов (и теперь практически заброшена).

- Программистам проще без сегментации;
- Ни один широко используемый язык программирования не включает сегментацию в свою модель памяти. Память всегда плоская. Так что настройка сегментов — задача компилятора (которую сложно реализовать).
- Сегменты превращают фрагментацию памяти в катастрофу.
- Таблица дескрипторов может содержать до 8192 дескрипторов сегментов. Как эффективно использовать это небольшое количество?

После введения длинного режима сегментация была вычищена из процессора, но не полностью. Она всё ещё используется для колец защиты, и потому программист должен её понимать.

3.3 Минимальная сегментация в длинном режиме

Даже в длинном режиме каждый раз при выборке инструкции процессор использует сегментацию. Она предоставляет нам плоский линейный виртуальный адрес, который затем превращается в физический с помощью механизмов виртуальной памяти (см. раздел 4.2). LDT — часть аппаратного механизма переключения задач, который на самом деле никто не стал использовать; по этой причине в длинном режиме он полностью отключён. Вся адресация памяти через *основные* сегментные регистры (`cs`, `ds`, `es` и `ss`) больше не учитывает значения базы и смещения из GDT. База сегмента всегда зафиксирована на 0x0 независимо от содержимого дескриптора; размеры сегментов вообще не ограничены. Однако остальные поля дескриптора не игнорируются. Это означает, что в длинном режиме *в GDT должно присутствовать как минимум три дескриптора*: нулевой дескриптор (должен присутствовать в любой GDT всегда), сегмент кода и сегмент данных. Если вы хотите использовать кольца защиты для реализации привилегированного и пользовательского режимов, вам также понадобятся дескрипторы кода и данных для кода пользовательского уровня.

> **Зачем нужны отдельные дескрипторы для кода и данных?** Никакая комбинация флагов дескриптора не позволяет программисту одновременно настроить права на чтение/запись и право на выполнение.

Даже с тем совсем небольшим опытом в языке ассемблера, который у нас уже есть, несложно расшифровать этот фрагмент загрузчика, показывающий пример GDT. Он взят из Pure64, загрузчика операционной системы с открытым исходным кодом. Поскольку он выполняется до операционной системы, он не содержит дескрипторов кода и данных пользовательского уровня (см. листинг 3-2).

Листинг 3-2. Пример GDT `gdt64.asm`

```
align 16  ; This ensures that the next command or data element is
; stored starting at an address divisible by 16 (even if we need
; to skip some bytes to achieve that).
; The following will be copied to GDTR via LGDTR instruction:
GDTR64:                 ; Global Descriptors Table Register
    dw gdt64_end - gdt64 - 1 ; limit of GDT (size minus one)
    dq 0x0000000000001000    ; linear address of GDT
; This structure is copied to 0x0000000000001000
gdt64:
SYS64_NULL_SEL equ $-gdt64      ; Null Segment
    dq 0x0000000000000000
; Code segment, read/exec, nonconforming
SYS64_CODE_SEL equ $-gdt64
    dq 0x0020980000000000       ; 0x00209A0000000000
; Data segment, read/write, expand down
SYS64_DATA_SEL equ $-gdt64
    dq 0x0000900000000000       ; 0x0020920000000000
gdt64_end:
; Dollar sign denotes the current memory address, so
; $-gdt64 means an offset from `gdt64` label in bytes
```

3.4 Доступ к частям регистров

3.4.1 Неожиданное поведение

Мы обычно думаем о `eax`, `rax`, `ax` и т. д. как о частях одного и того же физического регистра. Наблюдаемое поведение в основном подтверждает эту гипотезу, если только мы не записываем в 32-битную часть 64-битного регистра. Давайте посмотрим на пример, показанный в листинге 3-3.

Листинг 3-3. Страна регистровых чудес `risc_cisc.asm`

```
mov rax, 0x1122334455667788      ; rax = 0x1122334455667788
mov eax, 0x42                    ; !rax = 0x00 00 00 00 00 00 00 42
                                 ; why not rax = 0x1122334400000042 ??
mov rax, 0x1122334455667788      ; rax = 0x1122334455667788
mov ax, 0x9999                   ; rax = 0x1111222233339999, as expected
                                 ; this works as expected
mov rax, 0x1122334455667788      ; rax = 0x1122334455667788
xor eax, eax                     ; rax = 0x0000000000000000
                                 ; why not rax = 0x1122334400000000?
```

Как видите, запись в 8-битные или 16-битные части оставляет остальные биты нетронутыми. Однако запись в 32-битные части заполняет старшую половину широкого регистра битом знака! Причина в том, что привычное для программистов восприятие процессора сильно отличается от того, как всё устроено внутри на самом деле. На самом деле регистры `rax`, `eax` и все остальные не существуют как фиксированные физические сущности. Чтобы объяснить это несоответствие, нам сначала нужно разобрать два типа систем команд: CISC и RISC.

3.4.2 CISC и RISC

Одна из возможных классификаций процессоров делит их на основе системы команд. При проектировании системы команд есть две крайности.

- Сделать множество специализированных высокоуровневых инструкций. Это соответствует архитектурам *CISC (Complete Instruction Set Computer, компьютер с полным набором команд)*.
- Использовать лишь несколько примитивных инструкций, получая архитектуру *RISC (Reduced Instruction Set Computer, компьютер с сокращённым набором команд)*.

Инструкции CISC обычно медленнее, но и делают больше; иногда сложные инструкции удаётся реализовать лучше, чем комбинацией примитивных инструкций RISC (мы увидим пример этого позже в книге, когда будем изучать SSE (Streaming SIMD Extensions) в Главе 16). Однако большинство программ пишутся на языках высокого уровня и потому зависят от компиляторов. Очень трудно написать компилятор, который хорошо использует богатую систему команд. RISC облегчает работу компиляторов, а также лучше поддаётся оптимизациям на более низком, микрокодовом уровне, таком как конвейеры.

> **Вопрос 46** Почитайте о микрокоде в целом и о конвейерах процессора.

Система команд Intel 64 действительно относится к CISC. В ней тысячи инструкций — просто взгляните на второй том [15]! Однако эти инструкции декодируются и переводятся в поток более простых микрокодовых инструкций. Здесь вступают в действие различные оптимизации; микрокодовые инструкции переупорядочиваются, а некоторые из них могут даже выполняться одновременно. Это не исходная особенность процессоров, а скорее адаптация, нацеленная на лучшую производительность вместе с обратной совместимостью со старым программным обеспечением. К сожалению, доступно не так много информации о деталях микрокодового уровня современных процессоров. Читая технические обзоры вроде [17] и руководства по оптимизации вроде тех, что предоставляет Intel, можно развить определённую интуицию на этот счёт.

3.4.3 Объяснение

Теперь вернёмся к примеру, показанному в листинге 3-3. Подумаем о декодировании инструкций. Часть CPU, называемая декодером инструкций, постоянно переводит команды из более старой системы CISC в более удобную RISC. Конвейеры позволяют одновременно выполнять до шести более мелких инструкций. Однако для этого понятие регистров должно быть виртуализировано. Во время выполнения микрокода декодер выбирает доступный регистр из большого банка физических регистров. Как только более крупная инструкция завершается, эффекты становятся видны программисту: значение некоторых физических регистров может быть скопировано в те, что в данный момент назначены, скажем, `rax`. Взаимозависимости данных между инструкциями блокируют конвейер, снижая производительность. Худшие случаи возникают, когда один и тот же регистр читается и изменяется несколькими последовательными инструкциями (вспомните про `rflags!`). Если изменение `eax` означает сохранение старших битов `rax` нетронутыми, это вносит дополнительную зависимость между текущей инструкцией и любой инструкцией, которая изменяла `rax` или его части раньше. Отбрасывая старшие 32 бита при каждой записи в `eax`, мы устраняем эту зависимость, потому что нас больше не волнует предыдущее значение `rax` или его частей. Такое новое поведение было введено вместе с недавним ростом регистров общего назначения до 64 бит и не затрагивает операции с их меньшими частями ради совместимости. Иначе большинство старых бинарных файлов перестали бы работать, поскольку присвоение, например, `bl`, изменило бы весь `ebx`, что было неверно во времена, когда 64-битных регистров ещё не существовало.

3.5 Итоги

Эта глава была кратким историческим экскурсом в эволюцию процессоров за последние 30 лет. Мы также разобрали, для чего изначально предназначались сегменты в эпоху 32-бит, а также то, какие пережитки сегментации остались с нами по историческим причинам. В следующей главе мы подробнее рассмотрим механизм виртуальной памяти и его взаимодействие с кольцами защиты.

Виртуальная память

Эта глава посвящена виртуальной памяти в том виде, в котором она реализована в Intel 64. Мы начнём с обоснования абстракции над физической памятью, а затем получим общее представление о том, как она выглядит с точки зрения программиста. Наконец, мы углубимся в детали реализации, чтобы получить более полное понимание.

4.1 Кэширование

Начнём с поистине вездесущего понятия — кэширования. Интернет — это огромное хранилище данных. Вы можете получить доступ к любой его части, но задержка после запроса может быть значительной. Чтобы сделать просмотр более комфортным, веб-браузер кэширует веб-страницы и их элементы (изображения, таблицы стилей и т. д.). Так ему не приходится каждый раз заново скачивать одни и те же данные. Другими словами, браузер сохраняет данные на жёстком диске или в ОЗУ (оперативной памяти), чтобы обеспечить гораздо более быстрый доступ к локальной копии. Однако скачать весь интернет не вариант, потому что хранилище на вашем компьютере очень ограничено. Жёсткий диск намного больше ОЗУ, но и значительно медленнее. Именно поэтому вся работа с данными выполняется после их предварительной загрузки в ОЗУ. Таким образом, основная память используется как кэш для данных из внешнего хранилища. Впрочем, у жёсткого диска тоже есть собственный кэш... На кристалле CPU есть несколько уровней кэшей данных (обычно три: L1, L2, L3). Их размер намного меньше размера основной памяти, но зато они намного быстрее (уровень, ближайший к CPU, находится почти так же близко, как регистры). Кроме того, у CPU есть как минимум *кэш инструкций* (очередь, хранящая инструкции) и *буфер ассоциативной трансляции (TLB)* для повышения производительности виртуальной памяти. Регистры ещё быстрее кэшей (и меньше), так что они тоже своего рода кэш. Почему эта ситуация так распространена? В информационной системе, которой не нужно давать строгих гарантий относительно уровня производительности, введение кэшей часто *уменьшает среднее время доступа* (время между запросом и ответом). Чтобы это работало, нам нужен наш старый знакомый — *локальность*: в каждый момент времени у нас есть лишь небольшое рабочее множество данных. Механизм виртуальной памяти позволяет нам, среди прочего, использовать физическую память как кэш для частей кода программы и данных.

4.2 Мотивация

Естественно, в однозадачной системе, где в любой момент времени выполняется только одна программа, разумно просто разместить её прямо в физической памяти, начиная с некоторого фиксированного адреса. Другие компоненты (драйверы устройств, библиотеки) также можно разместить в памяти в некотором фиксированном порядке.

Однако в системе, дружественной к многозадачности, мы предпочитаем каркас, поддерживающий параллельное (или псевдопараллельное) выполнение нескольких программ. В этом случае операционной системе нужен некоторый вид управления памятью, чтобы справиться со следующими задачами:

- Выполнение программ произвольного размера (возможно, даже больше размера

физической памяти). Это требует способности загружать только те части программы, которые понадобятся в ближайшем будущем.

- Наличие нескольких программ в памяти одновременно.

Программы могут взаимодействовать с внешними устройствами, время отклика которых обычно медленное. Во время запроса к медленному аппаратному устройству, который может длиться тысячи тактов, мы хотим одолжить драгоценный CPU другим программам. Быстрое переключение между программами возможно только тогда, когда они уже находятся в памяти; иначе нам придётся тратить много времени на их извлечение из внешнего хранилища.

- Хранение программ в любом месте физической памяти.

Если мы этого добьёмся, мы сможем загружать части программ в любую свободную часть памяти, даже если они используют абсолютную адресацию.

В случае абсолютной адресации, как в `mov rax, [0x1010ffba]`, все адреса, включая начальный, становятся фиксированными: все точные значения адресов записываются в машинный код.

- Максимально возможное освобождение программистов от задач управления памятью.

Во время программирования мы не хотим думать о том, как могут функционировать разные микросхемы памяти на наших целевых архитектурах, каков объём доступной физической памяти и т. д. Вместо этого программисты должны уделять больше внимания логике программы.

- Эффективное использование общих данных и кода.

Когда несколько программ хотят получить доступ к одним и тем же файлам данных или кода (библиотекам), дублировать их в памяти для каждого дополнительного пользователя — это расточительство. Использование виртуальной памяти решает эти задачи.

4.3 Адресные пространства

Адресное пространство — это диапазон адресов. Мы видим два типа адресных пространств:

- Физический адрес, используемый для доступа к байтам на реальном оборудовании. Естественно,

существует определённый объём памяти, который процессор не может превысить. Он определяется адресными возможностями. Например, 32-битная система не может адресовать больше 4 ГБ памяти на процесс, потому что 2^{32} различных адресов примерно соответствуют 4 ГБ адресуемой памяти. Однако мы могли бы поставить в машину, способную адресовать 4 ГБ, меньше памяти, скажем, 1 ГБ или 2 ГБ. В этом случае некоторые адреса физического адресного пространства станут запрещёнными, поскольку за ними нет реальных ячеек памяти.

- Логический адрес — это адрес, каким его видит приложение. В инструкции `mov rax, [0x10bfd]` есть логический адрес: `0x10bfd`. У программиста есть иллюзия, что он *единственный пользователь памяти*. К какой бы ячейке памяти он ни обращался, он никогда не видит данные или инструкции других программ, которые выполняются параллельно с его собственной. Однако физическая память одновременно хранит несколько программ. В наших обстоятельствах *виртуальный адрес* — синоним логического адреса.

Трансляцию между этими двумя типами адресов выполняет аппаратная сущность, называемая *блоком управления памятью (MMU)*, с помощью нескольких таблиц трансляции, находящихся в памяти.

4.4 Возможности

Виртуальная память — это абстракция над физической памятью. Без неё нам пришлось бы работать напрямую с адресами физической памяти. При наличии виртуальной памяти мы можем притворяться, что каждая программа — единственный потребитель памяти, потому что она изолирована от остальных в своём собственном адресном пространстве. Адресное пространство одного процесса разбивается на *страницы* равной длины (обычно 4 КБ). Затем эти страницы динамически управляются. Некоторые из них могут быть выгружены во внешнее хранилище (в *файл подкачки*) и возвращены обратно при необходимости. Виртуальная память предлагает несколько полезных возможностей, придавая необычный смысл операциям с памятью (чтение, запись) на определённых страницах памяти.

- Мы можем взаимодействовать с внешними устройствами с помощью отображаемого на память ввода/вывода (Memory Mapped Input/Output) (например, записывая по адресам, назначенным некоторому устройству, и читая из них).

- Некоторые страницы могут соответствовать файлам, взятым из внешнего хранилища с помощью операционной системы и файловой системы.
- Некоторые страницы могут быть общими для нескольких процессов.
- Большинство адресов *запрещены* — их значение не определено, и попытка обратиться к ним приводит к ошибке.[12] Такая ситуация обычно приводит к аварийному завершению программы. Linux и другие Unix-подобные системы используют механизм *сигналов* для уведомления приложений об исключительных ситуациях. Почти для всех типов сигналов можно назначить обработчик. Обращение к запрещённому адресу будет перехвачено операционной системой, которая пошлёт приложению сигнал `SIGSEGV`. В этой ситуации часто можно увидеть сообщение об ошибке `Segmentation fault`.
- Некоторые страницы соответствуют файлам, взятым из хранилища (сам исполняемый файл, библиотеки и т. д.), а некоторые — нет. Эти *анонимные страницы* соответствуют областям памяти стека и *кучи* — динамически выделяемой памяти. Они так называются, потому что в файловой системе нет имён, которым они соответствуют. Напротив, у образа выполняемого исполняемого файла, файлов данных и устройств (которые тоже абстрагируются как файлы) есть имена в файловой системе.

Непрерывная область памяти называется *регионом*, если:

- Она начинается с адреса, кратного размеру страницы (например, 4 КБ).
- Все её страницы имеют одинаковые права доступа.

Если свободная физическая память заканчивается, некоторые страницы могут быть вытеснены во внешнее хранилище и сохранены в файле подкачки, либо просто отброшены (например, если они соответствуют каким-то файлам в файловой системе и не были изменены). В Windows такой файл называется `PageFile.sys`, в *nix-системах для этого обычно выделяется отдельный раздел на диске. Выбор страниц для вытеснения описывается одной из *стратегий замещения*, например:

- Наименее давно использовавшаяся (Least recently used).
- Последняя недавно использовавшаяся (Last recently used).
- Случайная (просто выбрать случайную страницу).

У любой системы с кэшированием есть стратегия замещения.

> **Вопрос 47** Почитайте о разных стратегиях замещения. Какие ещё стратегии существуют?

У каждого процесса есть *рабочее множество* страниц. Оно состоит из его исключительных страниц, присутствующих в физической памяти.

> **Выделение памяти** Что происходит, когда процессу нужно больше памяти? Он не может получить больше страниц самостоятельно, поэтому запрашивает у операционной системы дополнительные страницы. Система предоставляет ему дополнительные адреса. Динамическое выделение памяти в языках более высокого уровня (C++, Java, C# и т. д.) в конечном счёте сводится к запросу страниц у операционной системы, использованию выделенных страниц до тех пор, пока у процесса не закончится память, а затем к запросу новых страниц.

4.5 Пример: обращение к запрещённому адресу

Теперь мы своими глазами увидим *карту памяти* одного процесса. Она показывает, какие страницы доступны и чему они соответствуют. Мы рассмотрим разные виды областей памяти:

1. Соответствующие самому исполняемому файлу, загруженному в память.
2. Соответствующие библиотекам кода.
3. Соответствующие стеку и куче (*анонимные страницы*).
4. Просто пустые области запрещённых адресов.

Linux предлагает нам удобный в использовании механизм для изучения различной полезной информации о процессах, называемый `procfs`. Он реализует файловую систему специального назначения, где, перемещаясь по каталогам и просматривая файлы, можно получить доступ к памяти любого процесса, переменным окружения и т. д. Эта файловая система монтируется в каталог `/proc`. В первую очередь, файл `/proc/PID/maps` показывает карту памяти процесса с идентификатором `PID`.[13]

Давайте напишем простую программу, которая входит в цикл (и потому не завершается) (листинг 4-1). Она позволит нам увидеть её раскладку памяти, пока она выполняется.

Листинг 4-1. `mappings_loop.asm`

```
section .data
correct: dq -1
section .text
global _start
_start:
jmp _start
```

Теперь нам нужно открыть файл `/proc/?/maps`, где `?` — идентификатор процесса. Полное содержимое терминала см. в листинге 4-2.

Листинг 4-2. `mappings_loop`

```
> nasm -felf64 -o main.o mappings_loop.asm
> ld -o main main.o
> ./main &
[1] 2186
> cat /proc/2186/maps
00400000-00401000 r-xp 00000000 08:01 144225 /home/stud/main
00600000-00601000 rwxp 00000000 08:01 144225 /home/stud/main
7fff11ac0000-7fff11ae1000 rwxp 00000000 00:00 0 [stack]
7fff11bfc000-7fff11bfe000 r-xp 00000000 00:00 0 [vdso]
7fff11bfe000-7fff11c00000 r--p 00000000 00:00 0 [vvar]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
```

Левый столбец определяет диапазон области памяти. Как вы могли заметить, все области заключены между адресами, оканчивающимися тремя шестнадцатеричными нулями. Причина в том, что они состоят из страниц размером 4 КБ каждая (= 0x1000 байт). Мы видим, что разные секции, определённые в файле ассемблера, были загружены как разные области. Первая область соответствует секции кода и хранит закодированные инструкции; вторая соответствует данным. Как видите, адресное пространство огромно и охватывает от 0-го до 2^{64}*−*1-го байта. Однако выделено лишь несколько адресов; остальные запрещены. Второй столбец показывает права на чтение, запись и выполнение для страниц. Он также указывает, является ли страница *общей* для нескольких процессов или *приватной* для этого конкретного процесса.

> **Вопрос 48** Прочитайте о значении четвёртого (`08:01`) и пятого (`144225`) столбца в `man procfs`.

Пока мы не сделали ничего плохого. Теперь попробуем записать в запрещённое место.

Листинг 4-3. Получение segfault: `segfault_badaddr.asm`

```
section .data
correct: dq -1
section .text
global _start
_start:
mov rax, [0x400000-1]
; exit
mov rax, 60
xor rdi, rdi
syscall
```

Мы обращаемся к памяти по адресу `0x3fffff`, который находится на один байт раньше начала сегмента кода. Этот адрес запрещён, и потому попытка записи приводит к ошибке сегментации, как и говорит сообщение.

```
> ./main Segmentation fault
```

4.6 Эффективность

Загрузка отсутствующей страницы в физическую память из файла подкачки — очень затратная операция, требующая огромного объёма работы от операционной системы. Как же получилось, что этот механизм оказался не только эффективным по памяти, но и работает с адекватной производительностью? Ключевые факторы успеха:

1. Благодаря локальности необходимость загружать дополнительные страницы возникает редко. В худшем

случае доступ действительно очень медленный; однако такие случаи крайне редки. Среднее время доступа остаётся низким.

Другими словами, мы редко пытаемся обратиться к странице, которая не загружена в физическую память.

2. Понятно, что эффективность не была бы достигнута без помощи специального

оборудования. Без кэша транслированных адресов страниц — *TLB (Translation Lookaside Buffer, буфера ассоциативной трансляции)* — нам пришлось бы постоянно использовать механизм трансляции. TLB хранит начальные физические адреса для некоторых страниц, с которыми мы, вероятно, будем работать. Если мы транслируем виртуальный адрес внутри одной из этих страниц, начало страницы будет немедленно получено из TLB.

Другими словами, мы редко пытаемся транслировать адрес страницы, которую недавно не находили в физической памяти. Помните, что программа, использующая меньше памяти, может быть быстрее, потому что она вызывает меньше страничных отказов.

> **Вопрос 49** Что такое ассоциативный кэш? Почему TLB — это он?

4.7 Реализация

Теперь мы углубимся в детали и посмотрим, как именно происходит трансляция.

> **Примечание** Пока мы говорим только о доминирующем случае страниц по 4 КБ. Размер страницы можно настраивать, и остальные параметры будут меняться соответственно; подробности см. в разделе 4.7.3 и [15].

4.7.1 Структура виртуального адреса

Каждый виртуальный 64-битный адрес (например, те, что мы используем в наших программах) состоит из нескольких полей, как показано на

Рисунке 4-1.

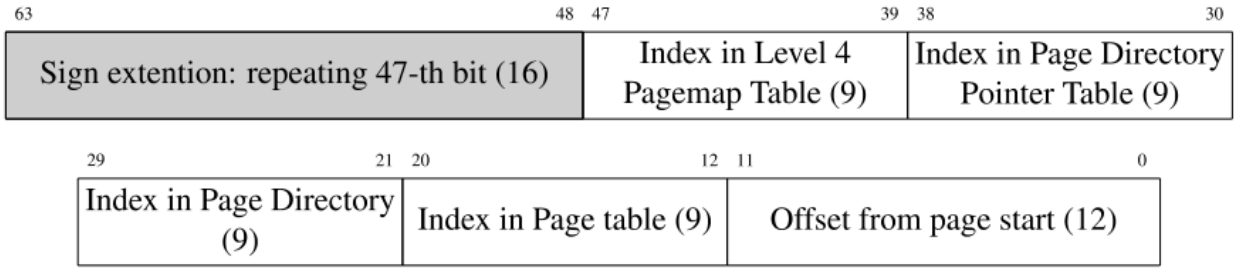


Рисунок 4-1. Структура виртуального адреса

Сам адрес на самом деле имеет ширину только 48 бит; он расширяется знаком до 64-битного *канонического адреса*. Его особенность в том, что его 17 левых бит равны. Если это условие не выполняется, адрес немедленно отклоняется при использовании. Затем 48 бит виртуального адреса преобразуются в 52 бита физического адреса с помощью специальных таблиц.[14]

> **Ошибка шины** При случайном использовании неканонического адреса вы увидите ещё одно сообщение об ошибке:

```
Bus error.
```

Физическое адресное пространство разделено на слоты, заполняемые виртуальными страницами. Эти слоты называются *страничными кадрами*. Между ними нет промежутков, так что они всегда начинаются с адреса, оканчивающегося 12 нулевыми битами. Младшие значащие 12 бит виртуального адреса и физической страницы соответствуют смещению адреса внутри страницы, поэтому они равны. Остальные четыре части виртуального адреса представляют индексы в таблицах трансляции. Каждая таблица занимает ровно 4 КБ, чтобы заполнить целую страницу памяти. Каждая запись имеет ширину 64 бита; она хранит часть начального адреса следующей таблицы, а также некоторые служебные флаги.

4.7.2 Трансляция адреса в деталях

Рисунок 4-2 отражает процесс трансляции адреса.

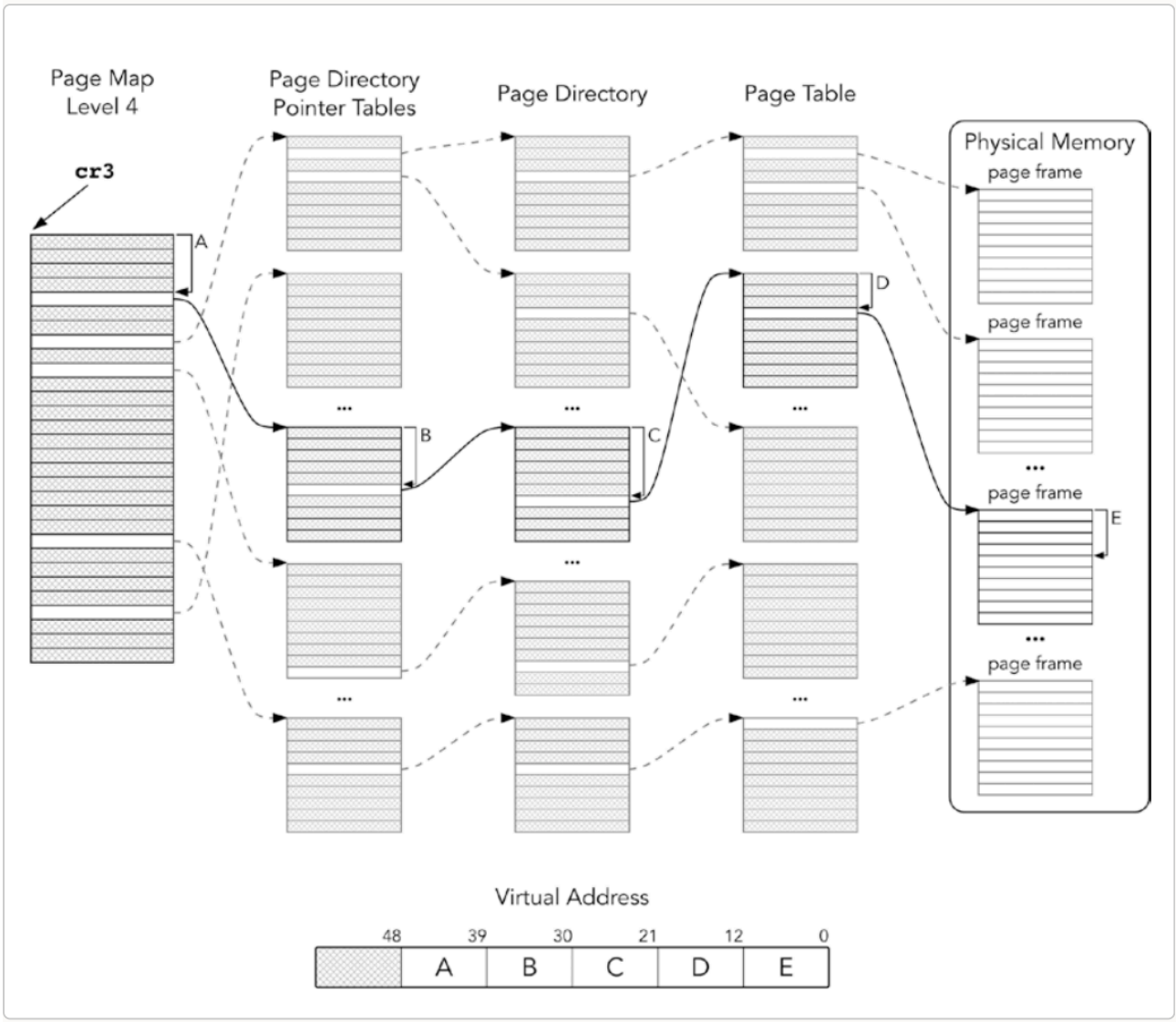


Рисунок 4-2. Схема трансляции виртуального адреса

Сначала мы берём начальный адрес первой таблицы из `cr3`. Эта таблица называется *Page Map Level 4 (PML4)*. Извлечение элементов из PML4 выполняется следующим образом:

- Биты 51:12 предоставляются `cr3`.
- Биты 11:3 — это биты 47:39 виртуального адреса.
- Последние три бита — нули.

Записи PML4 называются PML4E. Следующий шаг — извлечение записи из таблицы Page Directory Pointer — повторяет предыдущий:

- Биты 51:12 предоставляются выбранной PML4E.
- Биты 11:3 — это биты 38:30 виртуального адреса.

- Последние три бита — нули.

Процесс повторяется ещё через две таблицы, пока наконец мы не извлечём адрес страничного кадра (точнее, его биты 51:12). Физический адрес будет использовать их, а 12 бит будут взяты напрямую из виртуального адреса. *Неужели теперь нам приходится выполнять столько чтений памяти вместо одного?* Да, это выглядит громоздко. Однако благодаря кэшу адресов страниц, *TLB*, мы обычно обращаемся к памяти на уже транслированных и запомненных страницах. Нужно лишь добавить правильное смещение внутри страницы, что делается очень быстро. Поскольку TLB — ассоциативный кэш, он быстро предоставляет нам транслированные адреса страниц по начальному виртуальному адресу страницы. Заметьте, что страницы трансляции могут кэшироваться для более быстрого доступа. Рисунок 4-3 определяет формат записи таблицы страниц.

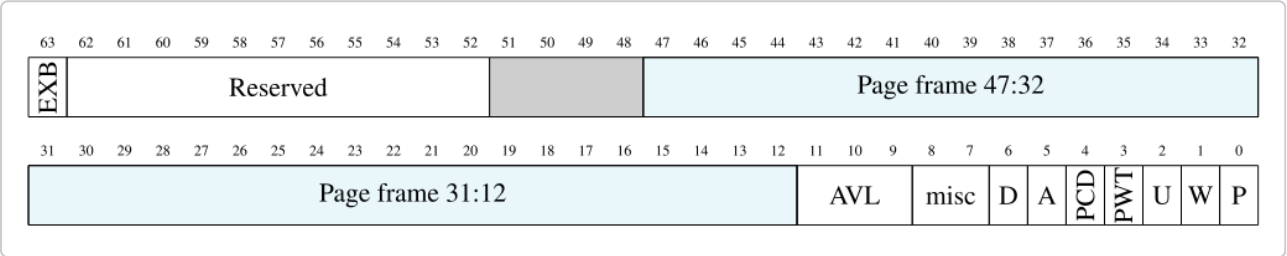


Рисунок 4-3. Запись таблицы страниц

P Present (присутствует в физической памяти) W Writable (запись разрешена) U User (может быть доступна из кольца 3) A Accessed (была затронута) D Dirty (страница была изменена после загрузки — например, с диска) EXB Execution-Disabled Bit (запрещает выполнение инструкций на этой странице) AVL Available (доступно для разработчиков операционных систем) PCD Page Cache Disable (отключение кэша для страницы) PWT Page Write-Through (обход кэша при записи на страницу)

Если `P` не установлен, попытка обратиться к странице приводит к прерыванию с кодом #PF (Page fault, страничный отказ). Операционная система может обработать его и загрузить соответствующую страницу. Это также можно использовать для реализации ленивого отображения файлов в память. Части файла будут загружаться в память по мере необходимости. Операционная система использует бит `W`, чтобы защитить страницу от изменения. Это нужно, когда мы хотим разделять код или данные между процессами, избегая ненужного дублирования. Общие страницы, помеченные `W`, можно использовать для обмена данными между процессами. У страниц операционной системы бит `U` снят. Если мы попытаемся обратиться к ним из кольца 3, произойдёт прерывание. *В отсутствие защиты сегментов виртуальная память — конечный механизм защиты памяти.*

> **Об ошибках сегментации** В общем случае ошибки сегментации возникают при попытке обратиться к памяти с недостаточными правами доступа (например, запись в память, доступную только для чтения). В случае запрещённых адресов можно считать, что у них вообще нет действительных прав доступа, так что обращение к ним — просто частный случай обращения к памяти с недостаточными правами.

Бит `EXB` (также называемый `NX`) запрещает выполнение кода. На нём основана технология *DEP* (Data Execution Prevention, предотвращение выполнения данных). Пока программа выполняется, часть её входных данных может храниться в стеке или в её секции данных. Злоумышленник может воспользоваться её уязвимостями, чтобы подмешать закодированные инструкции во входные данные, а затем выполнить их. Однако если страницы секции данных и стека помечены `EXB`, инструкции с них выполнить нельзя. Секция `.text`, однако, останется исполняемой, но обычно она в любом случае защищена от каких-либо изменений битом `W`.

4.7.3 Размеры страниц

Структура таблиц разных уровней иерархии очень похожа. Размер страницы можно настроить на 4 КБ, 2 МБ или 1 ГБ. В зависимости от структуры эта иерархия может сократиться до минимум двух уровней. В этом случае PDP будет функционировать как таблица страниц и хранить часть кадра размером 1 ГБ. См. рисунок 4-4, чтобы увидеть, как меняется формат записи в зависимости от размера страницы.

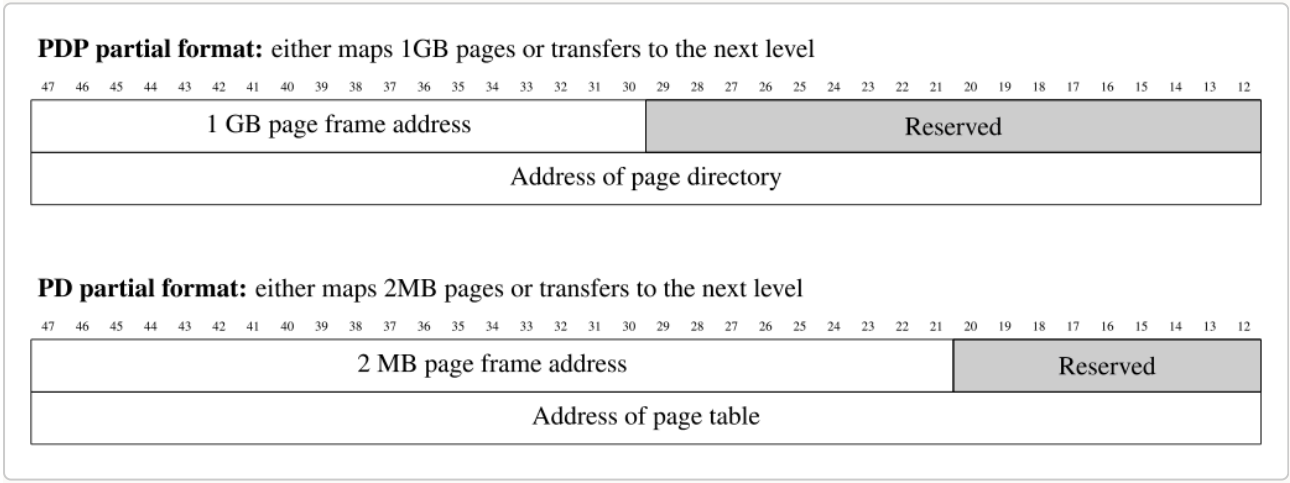


Рисунок 4-4. Формат записи таблицы Page Directory Pointer и таблицы Page Directory

Это контролируется 7-м битом в соответствующей записи PDP или PD. Если он установлен, соответствующая таблица отображает страницы; иначе она хранит адреса таблиц следующего уровня.

4.8 Отображение памяти

Отображение (mapping) означает «проекцию», установление соответствия между сущностями (файлами, устройствами, физической памятью) и областями виртуальной памяти. Когда загрузчик заполняет адресное пространство процесса, когда процесс запрашивает страницы у операционной системы, когда операционная система проецирует файлы с диска в адресные пространства процессов — всё это примеры отображения памяти. Для всех типов отображения памяти используется системный вызов `mmap`. Чтобы его выполнить, мы следуем тем же простым шагам, что описаны в Главе 2. Таблица 4-1 показывает его аргументы.

Таблица 4-1. Системный вызов `mmap`

РЕГИСТР	ЗНАЧЕНИЕ	СМЫСЛ
`rax`	9	Идентификатор системного вызова
`rdi`	addr	Операционная система пытается выполнить отображение в страницы, начиная с этого конкретного адреса. Этот адрес должен соответствовать началу страницы. Нулевой адрес означает, что операционная система вольна выбрать любое начало.
`rsi`	len	Размер региона
`rdx`	prot	Флаги защиты (чтение, запись, выполнение…)
`r10`	flags	Служебные флаги (общие или приватные, анонимные страницы и т. д.)
`r8`	fd	Необязательный дескриптор отображаемого файла. Поэтому файл должен быть открыт.
`r9`	offset	Смещение в файле.

После вызова `mmap` регистр `rax` будет содержать указатель на вновь выделенные страницы.

4.9 Пример: отображение файла в память

Нам понадобится ещё один системный вызов, а именно `open`. Он используется, чтобы открыть файл по имени и получить его *дескриптор*. Подробности см. в таблице 4-2.

Таблица 4-2. Системный вызов `open`

РЕГИСТР	ЗНАЧЕНИЕ	СМЫСЛ
`rax`	2	Идентификатор системного вызова
`rdi`	file name	Указатель на строку, завершающуюся нулём, содержащую имя файла
`rsi`	flags	Комбинация флагов прав доступа (только чтение, только запись или оба).
`rdx`	mode	Если `sys open` вызывается для создания файла, здесь будут храниться его права доступа в файловой системе.

Отображение файла в память выполняется в три простых шага:

- Открыть файл с помощью системного вызова `open`. `rax` будет содержать дескриптор файла.
- Вызвать `mmap` с соответствующими аргументами. Одним из них будет дескриптор файла, полученный на шаге 1.
- Использовать процедуру `print_string`, которую мы создали в Главе 2. Ради краткости мы опускаем закрытие файла и проверки ошибок.

4.9.1 Мнемонические имена для констант

Linux был написан на C, поэтому для упрощения взаимодействия с ним некоторые полезные константы предопределены в стиле C. Строка

```
#define NAME 42
```

задаёт подстановку, выполняемую во время компиляции. Каждый раз, когда программист пишет `NAME`, компилятор подставляет вместо неё 42. Это полезно для создания мнемонических имён для различных констант. NASM предоставляет схожую функциональность с помощью

```
%define directive
%define NAME  42
```

Подробнее о том, как выполняются такие подстановки, см. в разделе 5.1 «Препроцессор». Давайте посмотрим на страницу `man` для системного вызова `mmap`, описывающую его третий аргумент `prot`. Аргумент prot описывает желаемую защиту памяти для отображения (и не должен противоречить режиму открытия файла). Это либо `PROT_NONE`, либо побитовое ИЛИ одного или нескольких из следующих флагов:

```
     PROT_EXEC  Pages may be executed.
     PROT_READ  Pages may be read.
     PROT_WRITE Pages may be written.
     PROT_NONE  Pages may not be accessed.
```

`PROT_NONE` и его собратья — примеры таких мнемонических имён для целых чисел, используемых для управления поведением `mmap`. Помните, что и C, и NASM позволяют выполнять вычисления над константными значениями во время компиляции, включая побитовые операции И и ИЛИ. Ниже пример такого вычисления:

```
%define PROT_EXEC 0x4
%define PROT_READ 0x1
   mov rdx, PROT_READ | PROT_EXEC
```

Если вы не пишете на C или C++, вам придётся где-то проверять эти предопределённые значения и копировать их в свою программу. Вот как узнать конкретные значения этих констант для Linux:

1. Искать их в заголовочных файлах Linux API в `/usr/include`.
2. Использовать один из онлайн-сервисов Linux Cross Reference (lxr), например: `http://lxr.free-`

```
electrons.com.
```

Пока что мы рекомендуем второй способ, так как мы ещё не знаем C. Вы даже можете воспользоваться поисковой системой вроде Google и ввести `lxr PROT_READ` в качестве поискового запроса, чтобы сразу получить нужные результаты после перехода по первой ссылке. Например, вот что показывает LXR при запросе `PROT_READ`:

```
PROT_READ
Defined as a preprocessor macro in:
arch/mips/include/uapi/asm/mman.h, line 18
arch/xtensa/include/uapi/asm/mman.h, line 25
arch/alpha/include/uapi/asm/mman.h, line 4
arch/parisc/include/uapi/asm/mman.h, line 4
include/uapi/asm-generic/mman-common.h, line 9
```

Перейдя по одной из этих ссылок, вы увидите

```
18 #define PROT_READ      0x01          /* page can be read */
```

Итак, мы можем написать `%define PROT_READ 0x01` в начале файла ассемблера, чтобы использовать эту константу, не запоминая её значение.

4.9.2 Полный пример

Создайте файл `test.txt` с любым содержимым, а затем скомпилируйте и запустите файл из листинга 4-4 в том же каталоге. Вы увидите содержимое файла, выведенное в `stdout`.

Листинг 4-4. `mmap.asm`

```
; These macrodefinitions are copied from linux sources
; Linux is written in C, so the definitions looked a bit
; different there.
; We could have just looked up their values and use
; them directly in right places
; However it would have made the code much less legible
%define O_RDONLY 0
%define PROT_READ 0x1
%define MAP_PRIVATE 0x2
section .data
; This is the file name. You are free to change it.
fname: db 'test.txt', 0
section .text
global _start
; These functions are used to print a null terminated string
print_string:
    push rdi
    call string_length
    pop rsi
    mov rdx, rax
    mov rax, 1
    mov rdi, 1
    syscall
    ret
string_length:
    xor rax, rax
.loop:
    cmp byte [rdi+rax], 0
    je .end
    inc rax
    jmp .loop
.end:
    ret
_start:
; call open
mov rax, 2
mov rdi, fname
mov rsi, O_RDONLY     ; Open file read only
mov rdx, 0            ; We are not creating a file
                      ; so this argument has no meaning
syscall
; mmap
mov r8, rax           ; rax holds opened file descriptor
```

```
                        ; it is the fourth argument of mmap
mov rax, 9              ; mmap number
mov rdi, 0              ; operating system will choose mapping destination
mov rsi, 4096           ; page size
mov rdx, PROT_READ      ; new memory region will be marked read only
mov r10, MAP_PRIVATE    ; pages will not be shared
mov r9, 0               ; offset inside test.txt
syscall                 ; now rax will point to mapped location
mov rdi, rax
call print_string
mov rax, 60             ; use exit system call to shut down correctly
xor rdi, rdi
syscall
```

4.10 Итоги

В этой главе мы изучили концепцию и реализацию виртуальной памяти. Мы рассмотрели её как частный случай кэширования. Затем мы разобрали разные типы адресных пространств (физическое, виртуальное) и связь между ними через набор таблиц трансляции. Далее мы углубились в детали реализации виртуальной памяти. Наконец, мы привели минимальный рабочий пример отображения памяти с помощью системных вызовов Linux. Мы снова используем его в задании к Главе 13, где построим на его основе наш динамический аллокатор памяти. В следующей главе мы изучим процесс трансляции и компоновки, чтобы увидеть, как операционная система использует механизм виртуальной памяти для загрузки и выполнения программ.

> **Вопрос 50** Что такое область виртуальной памяти?

> **Вопрос 51** Что произойдёт, если попытаться изменить код выполняемой программы во время её выполнения?

> **Вопрос 52** Что такое запрещённые адреса?

> **Вопрос 53** Что такое канонический адрес?

> **Вопрос 54** Что такое таблицы трансляции?

Вопрос 55 Что такое страничный кадр?

Вопрос 56 Что такое область памяти?

Вопрос 57 Что такое виртуальное адресное пространство? Чем оно отличается от физического?

Вопрос 58 Что такое буфер ассоциативной трансляции?

Вопрос 59 Что обеспечивает производительность механизма виртуальной памяти?

Вопрос 60 Как переключается адресное пространство?

Вопрос 61 Какие механизмы защиты включает в себя виртуальная память?

Вопрос 62 Какова цель бита `EXB` ?

Вопрос 63 Какова структура виртуального адреса?

Вопрос 64 Есть ли что-то общее у виртуального и физического адреса?

Вопрос 65 Можем ли мы записать строку в секцию `.text` ? Что произойдёт, если её прочитать? А если перезаписать?

Вопрос 66 Напишите программу, которая вызовет системные вызовы `stat` , `open` и `mmap` (проверьте таблицу системных вызовов в Приложении C). Она должна вывести длину файла и его содержимое.

Вопрос 67 Напишите следующие программы, каждая из которых отображает в память текстовый файл `input.txt` , содержащий целое число x, с помощью системного вызова `mmap` , и выводит следующее:

1. $x!$ (факториал, $x! = 1 \cdot 2 \cdot \cdots \cdot (x - 1) \cdot x$). Гарантируется, что $x \geq 0$.
2. 0, если введённое число простое, иначе 1.
3. Сумму всех цифр числа.
4. x-е число Фибоначчи.
5. Проверяет, является ли x числом Фибоначчи.

Конвейер компиляции

Эта глава посвящена процессу компиляции. Мы разделяем его на три основных этапа: препроцессирование, трансляция и компоновка. Рисунок 5-1 показывает пример процесса компиляции. Есть два исходных файла: `first.asm` и `second.asm`. Каждый обрабатывается отдельно перед этапом компоновки.

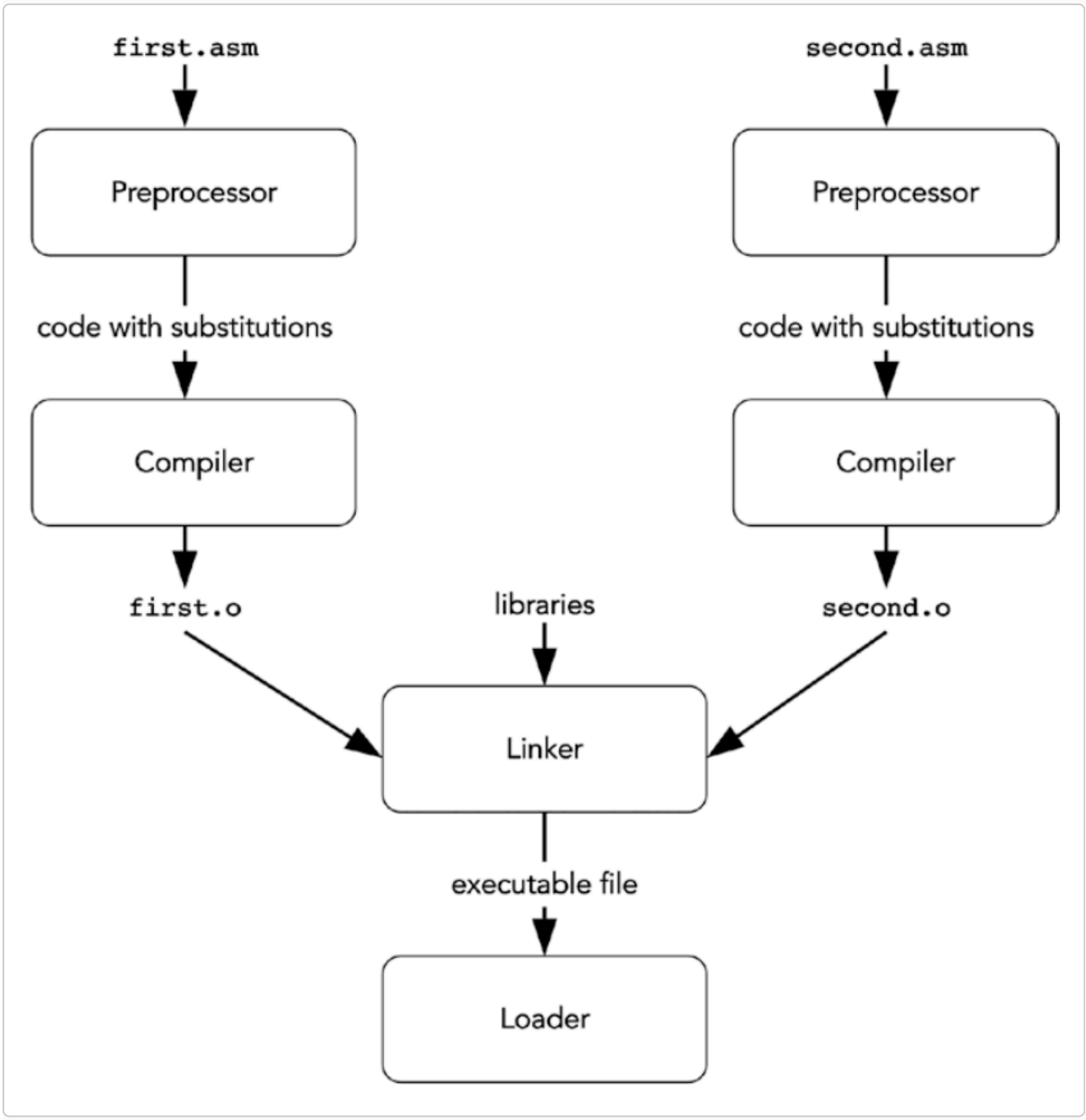


Рисунок 5-1. Конвейер компиляции

Препроцессор преобразует исходный текст программы, чтобы получить другую программу на том же языке. Преобразования обычно представляют собой подстановку одной строки вместо другой. *Компилятор* преобразует каждый исходный файл в файл с закодированными машинными инструкциями. Однако такой файл ещё не готов к выполне-

нию, потому что ему не хватает правильных связей с другими отдельно скомпилированными файлами. Мы говорим о случаях, когда инструкции обращаются к данным или инструкциям, объявленным в других файлах. *Компоновщик* устанавливает связи между файлами и создаёт исполняемый файл. После этого программа готова к запуску. Компоновщики работают с объектными файлами, типичные форматы которых — ELF (Executable and Linkable Format) и COFF (Common Object File Format). *Загрузчик* принимает исполняемый файл. Такие файлы обычно имеют структурированный вид с включёнными метаданными. Затем он заполняет свежее адресное пространство новорождённого процесса его инструкциями, стеком, глобально определёнными данными и кодом времени выполнения, предоставляемым операционной системой.

5.1 Препроцессор

Каждая программа создаётся как текст. Первый этап компиляции называется препроцессированием. На этом этапе специальная программа обрабатывает директивы препроцессора, найденные в исходном тексте программы. В соответствии с ними выполняются текстовые подстановки. В результате мы получаем изменённый исходный код без директив препроцессора, *написанный на том же языке программирования*. В этом разделе мы обсудим использование макропроцессора NASM.

5.1.1 Простые подстановки

Одна из базовых директив препроцессора называется `%define`. Она выполняет простую подстановку. Для кода, показанного в листинге 5-1, препроцессор будет заменять `cat_count` на 42 всякий раз, когда встречает такую подстроку в исходном тексте программы.

Листинг 5-1. `define_cat_count.asm`

```
%define cat_count 42
mov rax, cat_count
```

Чтобы увидеть результаты препроцессирования для входного файла `file.asm`, запустите `nasm -E file.asm`. Это часто очень полезно для отладки. Посмотрим на результат в листинге 5-2 для файла из листинга 5-1.

Листинг 5-2. `define_cat_count_preprocessed.asm`

```
%line 2+1 define_cat_count.asm
mov rax, 42
```

Команды для объявления подстановок называются *макросами*. В процессе, называемом *раскрытием макроса (macro expansion)*, их вхождения заменяются фрагментами текста. Получившиеся текстовые фрагменты называются *экземплярами макроса*. В листинге 5-2 число 42 в строке `mov rax,` `cat_count` — это экземпляр макроса. Такие имена, как `cat_count`, часто называют *символами препроцессора*.

> **Переопределение** NASM позволяет переопределять существующие символы препроцессора.

Важно, что препроцессор почти ничего не знает о *синтаксисе* языка программирования. Синтаксис определяет допустимые языковые конструкции. Например, код, показанный в листинге 5-3, корректен. Не важно, что ни `a`, ни `b` сами по себе не образуют допустимую конструкцию ассемблера; пока конечный результат подстановок синтаксически верен, компилятора это устраивает.

Листинг 5-3. `macro_asm_parts.asm`

```
%define a mov rax,
    %define b rbx
    a b
```

В другом примере, в языках более высокого уровня, оператор *if* имеет форму *if (<выражение>) then <оператор> else <оператор>*. Макросы могут оперировать частями этой конструкции, которые сами по себе синтаксически некорректны (например, отдельная конструкция *else <оператор>*). Пока результат синтаксически корректен, у компилятора не будет с ним проблем. В противоположность этому существуют и другие виды макросов, а именно синтаксические макросы, привязанные к структуре языка и оперирующие его конструкциями. Такие макросы изменяют их структурированным образом. Языки вроде LISP, OCaml и Scala используют синтаксические макросы. Зачем нам вообще нужны макросы? Помимо автоматизации, которую мы увидим позже, они дают мнемонические имена для фрагментов кода.[15] Для констант это позволяет нам отличать вхождения 42, используемые для подсчёта кошек, от тех, что используются для подсчёта собак или чего-то ещё. Иначе некоторые изменения программы были бы более болезненными и подверженными ошибкам, поскольку нам приходилось бы принимать больше решений исходя из того, что означает это конкретное число. Для наборов языковых конструкций они дают нам определённую автоматизацию, точно так же, как это делают подпрограммы. Макросы раскрываются во время компиляции, тогда как процедуры выполняются во время выполнения. Выбор за вами. Для ассемблера над программами не выполняется никаких оптимизаций. Однако в языках более высокого уровня для этого используют глобальные *константные* переменные. Хороший компилятор заменит их вхождения их значением. Плохой же не может знать об оптимиза-

циях, что может быть случаем при программировании микроконтроллеров или приложений для экзотических операционных систем. В таких случаях люди часто выполняют работу компилятора сами, используя макросы, как в языке ассемблера.

> **Стиль** Хорошая практика — давать имена всем константам в вашей программе.

В ассемблере и C люди обычно определяют глобальные константы с помощью определений макросов.

5.1.2 Подстановки с аргументами

Макросы даже лучше, чем кажется: у них могут быть аргументы. В листинге 5-4 показан простой макрос с тремя аргументами.

Листинг 5-4. `macro_simple_3arg.asm`

```
%macro test 3
dq %1
dq %2
dq %3
%endmacro
```

Его действие простое: для каждого аргумента он создаст запись данных типа «учетверённое слово». Как видите, на аргументы ссылаются по их индексам, начиная с 1. Когда этот макрос определён, строка `test 666, 555, 444` будет заменена на показанное в листинге 5-5

Листинг 5-5. `macro_simple_3arg_inst.asm`

```
dq 666
dq 555
dq 444
```

> **Вопрос 68** Найдите больше примеров использования `%define` и `%macro` в документации NASM.

5.1.3 Простая условная подстановка

Макросы в NASM поддерживают различные условия. Простейшее из них — `%if`. Листинг 5-6 показывает минимальный пример.

Листинг 5-6. `macroif.asm`

```
BITS 64
%define x 5
%if x == 10
mov rax, 100
%elif x == 15
mov rax, 115
%elif x == 200
mov rax, 0
%else
mov rax, rbx
%endif
```

Листинг 5-7 показывает инстанцированный макрос. Помните, что вы можете проверить результат препроцессирования с помощью `nasm -E`.

Листинг 5-7. `macroif_preprocessed.asm`

```
%line 1+1 if.asm
[bits 64]
%line 15+1 if.asm
mov rax, rbx
```

Условие — это выражение, похожее на то, что можно увидеть в языках высокого уровня: арифметика и логические связки (and, or, not).

5.1.4 Условие по определению

Можно решить во время компиляции, следует ли ассемблировать часть файла. Один из многочисленных аналогов `%if` — `%ifdef`. Он работает схожим образом, но условие выполняется, если определён определённый символ препроцессора. Пример, показанный в листинге 5-8, включает такую директиву.

Листинг 5-8. `defining_in_cla.asm`

```
%ifdef flag
hellostring: db "Hello",0
%endif
```

Как видите, символ `flag` здесь не определён с помощью директивы `%define`. Поэтому у нас есть строка, помеченная меткой `hellostring`. Стоит упомянуть, что символы препроцессора можно определять прямо при вызове NASM благодаря ключу `-d`. Например, условие макроса в листинге 5-8 будет выполнено, когда NASM вызывается с аргументом `-d myflag`.

> **Вопрос 69** Проверьте вывод препроцессора для файла, показанного в листинге 5-8.

В следующих разделах мы увидим больше директив препроцессора, похожих на `%if`.

5.1.5 Условие по совпадению текста

`%ifidn` используется для проверки равенства двух текстовых строк (различия в пробелах не учитываются). В зависимости от результата сравнения последующий код будет или не будет ассемблирован. Это позволяет нам создавать очень гибкие макросы, которые зависят, например, от имени аргумента. Для иллюстрации создадим макроинструкцию `pushr` (см. листинг 5-9). Она будет функционировать точно так же, как ассемблерная инструкция `push`, но также будет принимать регистры `rip` и `rflags`.

Листинг 5-9. pushr.asm

```
%macro pushr 1
%ifidn %1, rflags
pushf
%else
push %1
%endif
%endmacro
pushr rax
pushr rflags
```

Листинг 5-10 показывает, во что превращаются два макроса из листинга 5-9 после инстанцирования.

Листинг 5-10. `pushr_preprocessed.asm`

```
%line 8+1 pushr/pushr.asm
push rax
pushf
```

Как видите, макрос подстроил своё поведение на основе текстового представления аргумента. Заметьте, что конструкции `%else` допустимы точно так же, как и для обычного `%if`. Чтобы сделать сравнение регистронезависимым, используйте вместо этого директиву `%ifidni`.

5.1.6 Условие по типу аргумента

Препроцессор NASM немного осведомлён об элементах языка ассемблера (типах токенов). Он может отличать строки в кавычках от чисел и идентификаторов. Для этого существует тройка аналогов `%if`: `%ifid` для проверки, является ли аргумент идентификатором, `%ifstr` для проверки строки и `%ifnum` для проверки, является ли аргумент числом. Листинг 5-11 показывает пример макроса, который выводит либо число, либо строку (используя идентификатор). Он использует несколько процедур, разработанных в ходе первого задания, для вычисления длины строки, вывода строки и вывода целого числа.

Листинг 5-11. `macro_arg_types.asm`

```
%macro print 1
   %ifid %1
      mov rdi, %1
      call print_string
   %else
     %ifnum %1
        mov rdi, %1
        call print_uint
     %else
        %error "String literals are not supported yet"
     %endif
   %endif
%endmacro
myhello: db 'hello', 10, 0
_start:
   print myhello
   print 42
   mov rax, 60
   syscall
```

Отступы полностью необязательны и сделаны исключительно для читаемости. Если аргумент не является ни строкой, ни идентификатором, мы используем директиву `%error`, чтобы заставить NASM выбросить ошибку. Если бы мы использовали вместо неё `%fatal`, мы бы полностью остановили ассемблирование, и все последующие ошибки были

бы проигнорированы; простой же `%error` даст NASM шанс сообщить и о последующих ошибках тоже, прежде чем он остановит обработку входных файлов. Посмотрим на инстанцирования макроса в листинге 5-12

Листинг 5-12. `macro_arg_types_preprocessed.asm`

```
%line 73+1 macro_arg_types/macro_arg_types.asm
myhello: db 'hello', 10, 0
_start:
 mov rdi, myhello
%line 76+0 macro_arg_types/macro_arg_types.asm
 call print_string
%line 77+1 macro_arg_types/macro_arg_types.asm
%line 77+0 macro_arg_types/macro_arg_types.asm
 mov rdi, 42
 call print_uint
%line 78+1 macro_arg_types/macro_arg_types.asm
 mov rax, 60
 syscall
```

5.1.7 Порядок вычисления: Define, xdefine, Assign

У всех языков программирования есть понятие стратегии вычисления. Она описывает порядок вычисления в сложных выражениях. Как нам следует вычислять f ($g(1)$, $h(4)$)? Стоит ли сначала вычислить $g(1)$ и $h(4)$, а затем позволить f подействовать на результаты? Или же стоит подставить $g(1)$ и $h(4)$ прямо в тело f и отложить их собственные вычисления до момента, когда они действительно понадобятся? Макросы вычисляются макропроцессором NASM, и у них действительно сложная структура, поскольку любое инстанцирование макроса может включать инстанцирование других макросов. Возможна тонкая настройка порядка вычисления, потому что NASM предоставляет немного различающиеся версии директив определения макросов, а именно

- `%define` для отложенной подстановки. Если тело макроса содержит другие макросы, они будут раскрыты после подстановки.
- `%xdefine` выполняет подстановки в момент определения. Затем получившаяся строка будет использоваться в подстановках.
- `%assign` похож на `%xdefine`, но также принудительно вычисляет арифметические выражения и выбрасывает ошибку, если результат вычисления — не число.

Чтобы лучше понять тонкое различие между `%define` и `%xdefine`, взгляните на пример, показанный в листинге 5-13.

Листинг 5-13. `defines.asm`

```
%define i 1
%define d i * 3
%xdefine xd i * 3
%assign a i * 3
mov rax, d
mov rax, xd
mov rax, a
; let's redefine i
%define i 100
mov rax, d
mov rax, xd
mov rax, a
```

Листинг 5-14 показывает результат препроцессирования.

Листинг 5-14. `defines_preprocessed.asm`

```
%line 2+1 defines.asm
%line 6+1 defines.asm
mov rax, 1 * 3
mov rax, 1 * 3
mov rax, 3
mov rax, 100 * 3
mov rax, 1 * 3
mov rax, 3
```

Ключевые различия в том, что

- `%define` *может изменить своё значение между инстанцированиями, если его части переопределены.*
- `%xdefine` приклеивает к себе другие макросы, от которых напрямую зависит, сразу после определения.
- `%assign` принудительно вычисляет и подставляет значения. Там, где xdefine оставил бы вам

символ препроцессора, равный 4+2+3, `%assign` вычислит его и присвоит ему значение 9. Мы воспользуемся замечательными свойствами `%assign`, чтобы показать немного магии, после того как познакомимся с повторениями макросов.

5.1.8 Повторение

Директива `times` выполняется после того, как все определения макросов полностью раскрыты, и потому не может использоваться для повторения частей макросов. Но есть другой способ, которым NASM может создавать циклы макросов: размещая тело цикла между директивами `%rep` и `%endrep`. Циклы могут выполняться только фиксированное количество раз, заданное как аргумент `%rep`. Пример в листинге 5-15 показывает, как препроцессор вычисляет сумму целых чисел от 1 до 10, а затем использует это значение для инициализации глобальной переменной `result`.

Листинг 5-15. `rep.asm`

```
%assign x 1
%assign a 0
%rep 10
%assign a x + a
%assign x x + 1
%endrep
result: dq a
```

После препроцессирования значение `result` корректно инициализируется как 55 (см. листинг 5-16). Вы можете проверить это вручную.[16]

Листинг 5-16. `rep_preprocessed.asm`

```
%line 7+1 rep/rep.asm
result: dq 55
```

Мы можем использовать `%exitrep`, чтобы немедленно выйти из цикла. Это, таким образом, аналог инструкции `break` в языках высокого уровня.

5.1.9 Пример: вычисление простых чисел

Макрос, показанный в листинге 5-17, используется для построения решета простых чисел. Это означает, что он определяет статический массив байт, где каждый i-й байт равен 1 тогда и только тогда, когда i — простое число. *Простое число* — это натуральное число больше 1, у которого нет положительных делителей, кроме 1 и самого себя. Алгоритм прост:

- *0 и 1 не являются простыми.*
- 2 — простое число.

- Для каждого *current* до *limit* мы проверяем, что ни одно i от 2 до $n/2$ не является делителем n.

Листинг 5-17. `prime.asm`

```
%assign limit 15
is_prime: db 0, 0, 1
%assign n 3
%rep limit
    %assign current 1
    %assign i 1
        %rep n/2
           %assign i i+1
           %if n % i = 0
                %assign current 0
                %exitrep
```

( )

1 2 `%endif%endrep`

```
db current ; n
    %assign n n+1
%endrep
```

Обращаясь к n-му элементу массива `is_prime`, мы можем узнать, является ли n простым числом или нет. После препроцессирования следующего кода в листинге 5-18 будет сгенерировано:

Листинг 5-18. `prime_preprocessed.asm`

```
%line 2+1 prime/prime.asm
is_prime: db 0, 0, 1
%line 16+1 prime/prime.asm
db 1
%line 16+0 prime/prime.asm
db 0
db 1
db 0
db 1
db 0
db 0
db 0
db 1
db 0
db 1
db 0
db 0
db 0
db 1
```

Читая *i*-й байт, начиная с `is_prime`, мы получаем 1, если *i* простое; иначе 0.

> **Вопрос 70** Измените макрос так, чтобы он строил битовую таблицу, занимающую в восемь раз меньше памяти. Добавьте функцию, которая будет проверять число на простоту и возвращать 0 или 1 на основе этой предварительно вычисленной таблицы.

> **Подсказка** для этого макроса вам, вероятно, придётся много копировать и вставлять.

5.1.10 Метки внутри макросов

Без меток в ассемблере много не сделаешь. Использование фиксированных имён меток внутри макросов не очень распространено. Когда макрос инстанцируется много раз внутри одного файла, повторно определённые метки могут вызывать конфликты, останавливающие компиляцию. Есть возможность использовать локальные метки макроса — метку, к которой нельзя обратиться вне текущего инстанцирования макроса. Для этого можно добавить перед именем такой метки двойной знак процента, вот так: `%%labelname`. Каждая локальная метка макроса получит случайный префикс, который

будет меняться между экземплярами макроса, но останется одинаковым внутри одного экземпляра. Листинг 5-19 показывает пример. Листинг 5-20 содержит результаты пре-процессирования.

Листинг 5-19. `macro_local_labels.asm`

```
%macro mymacro 0
%%labelname:
%%labelname:
%endmacro
Mymacro
Mymacro
mymacro
```

Макрос `mymacro` инстанцируется три раза. Каждый раз локальная метка получает уникальное имя. К базовому имени (после двойного процента) добавляется спереди числовой префикс, разный в каждом экземпляре. Первый префикс — `..@0.`, второй — `..@1.`, и так далее.

Листинг 5-20. `macro_local_labels_inst.asm`

```
%line 5+1 macro_local_labels/macro_local_labels.asm
..@0.labelname:
%line 6+0 macro_local_labels/macro_local_labels.asm
..@0.labelname:
%line 7+1 macro_local_labels/macro_local_labels.asm
..@1.labelname:
%line 8+0 macro_local_labels/macro_local_labels.asm
..@1.labelname:
%line 9+1 macro_local_labels/macro_local_labels.asm
..@2.labelname:
%line 10+0 macro_local_labels/macro_local_labels.asm
..@2.labelname:
```

5.1.11 Заключение

О макросах можно думать как о метаязыке программирования, выполняемом во время компиляции. Он может выполнять довольно сложные вычисления и ограничен двумя способами:

- Эти вычисления не могут зависеть от пользовательского ввода (так что они могут оперировать только константами).

- Циклы могут выполняться не более фиксированного количества раз. Это означает, что конструкции вроде `while` закодировать невозможно.

5.2 Трансляция

Компилятор обычно транслирует исходный код с одного языка на другой. В случае трансляции с языков программирования высокого уровня в машинный код этот процесс включает несколько внутренних этапов. На этих этапах мы постепенно продвигаем код в виде *IR (Intermediate Representation, промежуточного представления)* в сторону целевого языка. Каждый шаг продвижения IR приближает его к целевому языку. Непосредственно перед созданием ассемблерного кода *IR* будет очень близок к ассемблеру, так что мы можем сбросить IR в читаемый листинг вместо кодирования инструкций. Трансляция не только сложный процесс — она ещё и теряет информацию о структуре исходного кода, поэтому восстановить читаемый код высокого уровня из файла ассемблера невозможно. Компилятор работает с атомарными единицами кода, называемыми *модулями*. Модуль обычно соответствует исходному файлу кода (но не заголовочному файлу или файлу для включения). Каждый модуль компилируется независимо от остальных модулей. Из каждого модуля создаётся *объектный файл*. Он содержит двоично закодированные инструкции, но обычно не может быть выполнен сразу же. Тому есть несколько причин. Например, объектный файл завершается отдельно от других файлов, но ссылается на внешний код и данные. Пока не ясно, будут ли этот код или данные находиться в памяти, равно как и позиция самого объектного файла. Трансляция языка ассемблера довольно прямолинейна, потому что соответствие между мнемониками ассемблера и машинными инструкциями почти однозначное. Помимо разрешения меток здесь почти нет нетривиальной работы. Поэтому сейчас мы сосредоточимся на следующем этапе компиляции, а именно на компоновке.

5.3 Компоновка

Вернёмся к нашим первым примерам программ на ассемблере. Чтобы превратить программу «Hello, world!» из исходного кода в исполняемый файл, мы использовали следующие две команды:

```
> nasm -f elf64 -o hello.o hello.asm
> ld -o hello hello.o
```

Сначала мы использовали NASM для создания *объектного файла*. Его формат, `elf64`, был указан ключом `-f`. Затем мы использовали другую программу, `ld` (компоновщик), чтобы создать файл, готовый к выполнению. Мы возьмём этот формат файла как пример, чтобы показать вам, что компоновщик делает на самом деле.

5.3.1 Executable and Linkable Format

ELF (Executable and Linkable Format) — формат объектных файлов, довольно типичный для *nix-систем. Мы ограничимся его 64-битной версией. ELF допускает три типа файлов.

1. *Перемещаемые объектные файлы (relocatable object files)* — это .o-файлы, создаваемые компилятором.

Перемещение (relocation) — это процесс присвоения окончательных адресов различным частям программы и изменения кода программы так, чтобы все связи были правильно приписаны. Мы говорим обо всех видах обращений к памяти по абсолютным адресам. Перемещение нужно, например, когда программа состоит из нескольких модулей, ссылающихся друг на друга. Порядок, в котором они будут размещены в памяти, ещё не зафиксирован, так что абсолютные адреса не определены. Компоновщики могут объединять эти файлы, чтобы создать следующий тип объектных файлов.

2. *Исполняемый объектный файл* можно загрузить в память и сразу же выполнить. По сути,

это структурированное хранилище кода, данных и служебной информации.

3. *Разделяемые объектные файлы* можно загружать по мере необходимости главной программой. Они компонуются с ней динамически. В Windows это хорошо известные `dll`-файлы; в *nix-системах их имена часто оканчиваются на `.so`.

Цель любого компоновщика — создать исполняемый (или разделяемый) объектный файл на основе набора перемещаемых. Для этого компоновщик должен выполнить следующие задачи:

- *Перемещение*
- *Разрешение символов*. Каждый раз, когда символ (функция, переменная) разыменовывается, компоновщик должен изменить объектный файл и заполнить часть инструкции, соответствующую адресу операнда, правильным значением.

5.3.1.1 Структура

ELF-файл начинается с главного заголовка, хранящего общую метаинформацию. Типичный заголовок ELF см. в листинге 5-21. Файл `hello` — результат компиляции программы «Hello, world!», показанной в листинге 2-4.

Листинг 5-21. hello_elfheader ELF Header:

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x4000b0
  Start of program headers:          64 (bytes into file)
  Start of section headers:          552 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         2
  Size of section headers:           64 (bytes)
  Number of section headers:         6
  Section header string table index: 3
```

ELF-файлы затем предоставляют информацию о программе, которую можно рассматривать с двух точек зрения:

- *Представление для компоновки (linking view)*, состоящее из секций. Оно описывается таблицей секций, доступной через `readelf -S`. Каждая секция, в свою очередь, может быть:
- Необработанными данными для загрузки в память.
- Форматированными метаданными о других секциях, используемыми загрузчиком (например, *.bss*), компоновщиком (например, *таблицы перемещений*) или отладчиком (например, *.line*).

Код и данные хранятся внутри секций.

- *Представление для выполнения (execution view)*, состоящее из сегментов.

Оно описывается *таблицей заголовков программы (Program Header Table)*, которую можно изучить с помощью `readelf -l`. Мы подробнее рассмотрим её в разделе 5.3.5.

Каждая запись может описывать

- Некоторую информацию, нужную системе для выполнения программы. ELF-сегмент, содержащий ноль или более секций. У них одинаковый набор прав доступа (чтение,

запись, выполнение), обеспечиваемый виртуальной памятью. У каждого сегмента есть начальный адрес, и он загружается в отдельную *область памяти*, состоящую из последовательных страниц.

Пересмотрев листинг 5-21, мы замечаем, что он точно описывает позицию и размеры заголовков программы и заголовков секций. Мы начнём с представления секций, поскольку компоновщик работает в основном с ними.

5.3.1.2 Секции в ELF-файлах

Язык ассемблера позволяет вручную управлять секциями. `section` в NASM соответствует секциям объектного файла. Вы уже видели пару таких секций, а именно `.text` и `.data`. Ниже приведён список наиболее часто используемых секций; полный список можно найти в [24]. *.text* хранит машинные инструкции. *.rodata* хранит данные только для чтения. *.data* хранит инициализированные глобальные переменные. *.bss* хранит доступные для чтения и записи глобальные переменные, инициализированные нулём. Нет нужды сбрасывать их содержимое в объектный файл, поскольку они всё равно все заполнены нулями. Вместо этого хранится общий размер секции. Операционная система может знать более быстрые способы инициализации такой памяти, чем обнуление вручную. В ассемблере вы можете размещать здесь данные, помещая директивы `resb`, `resw` и подобные после `section .bss`. *.rel.text* хранит *таблицу перемещений* для секции `.text`. Она используется, чтобы запомнить места, которые компоновщик должен изменить в `.text` после выбора адреса загрузки для этого конкретного объектного файла. *.rel.data* хранит таблицу перемещений для данных, на которые ссылается *модуль*. *.debug* хранит таблицу символов, используемую для отладки программы. Если программа была написана на C или C++, она будет хранить информацию не только о глобальных переменных (как это делает *.symtab*), но и о локальных переменных. *.line* определяет соответствие между фрагментами кода и номерами строк в исходном коде. Это нужно нам, потому что соответствие между строками исходного кода на языках высокого уровня и инструкциями ассемблера неочевидно. Эта информация позволяет отлаживать программу на языке высокого уровня построчно. *.strtab* хранит символьные строки. Это как массив строк. Другие секции, такие как *.symtab* и *.debug*, используют не сами строки, а их индексы в *.strtab*. *.symtab* хранит таблицу символов. Каждый раз, когда программист определяет метку, NASM создаёт для неё *символ*.[17] Эта таблица также хранит служебную информацию, которую мы рассмотрим позже. Теперь, когда у нас есть общее представление о представлении для компоновки ELF-файла, мы рассмотрим несколько примеров, показывающих особенности трёх разных типов ELF-файлов.

5.3.2 Перемещаемые объектные файлы

Давайте исследуем объектный файл, полученный компиляцией простой программы, показанной в листинге 5-22.

Листинг 5-22. `symbols.asm`

```
section .data
datavar1: dq 1488
datavar2: dq 42
section .bss
bssvar1: resq 4*1024*1024
bssvar2: resq 1
section .text
extern somewhere
global _start
    mov rax, datavar1
    mov rax, bssvar1
    mov rax, bssvar2
    mov rdx, datavar2
_start:
jmp _start
    ret
textlabel: dq 0
```

Эта программа использует директивы `extern` и `global`, чтобы по-разному отмечать *символы*. Эти две директивы управляют созданием *таблицы символов*. По умолчанию все символы локальны для текущего модуля. `extern` определяет символ, который определён в других модулях, но на который ссылаются в текущем. С другой стороны, `global` определяет глобально доступный символ, на который другие модули могут сослаться, объявив его у себя как `extern`.

> **Не путайте** Не путайте глобальные и локальные символы с глобальными и локальными метками!

GNU binutils — набор инструментов для работы с двоичными файлами, используемых для работы с объектными файлами. В него входит несколько утилит для изучения содержимого объектных файлов. Некоторые из них представляют для нас особый интерес.

- Если вам нужно только посмотреть таблицу символов, используйте `nm`.
- Используйте `objdump` как универсальный инструмент для отображения общей информации об объектном файле. Помимо ELF, он поддерживает и другие форматы объектных файлов.
- Если вы знаете, что файл в формате ELF, `readelf` часто оказывается лучшим и наиболее информативным выбором.

Давайте скормим эту программу `objdump`, чтобы получить результаты, показанные в листинге 5-23.

Листинг 5-23. Символы

```
> nasm -f elf64 main.asm && objdump -tf -m intel main.o
main.o:     file format elf64-x86-64
architecture: i386:x86-64, flags 0x00000011:
HAS_RELOC, HAS_SYMS
start address 0x0000000000000000
SYMBOL TABLE:
0000000000000000 l    df *ABS*   0000000000000000 main.asm
0000000000000000 l    d  .data   0000000000000000 .data
0000000000000000 l    d  .bss    0000000000000000 .bss
0000000000000000 l    d  .text   0000000000000000 .text
0000000000000000 l       .data   0000000000000000 datavar1
0000000000000008 l       .data   0000000000000000 datavar2
0000000000000000 l       .bss    0000000000000000 bssvar1
0000000002000000 l       .bss    0000000000000000 bssvar2
0000000000000029 l       .text   0000000000000000 textlabel
0000000000000000         *UND*   0000000000000000 somewhere
0000000000000028 g       .text   0000000000000000 _start
```

Нам показана таблица символов, где каждый символ снабжён полезной информацией. Что означают её столбцы?

1. Виртуальный адрес данного символа. Пока мы не знаем начальный адрес секции,

поэтому все виртуальные адреса даны *относительно начала секции*. Например, `datavar1` — первая переменная, хранящаяся в `.data`, её адрес равен 0, а размер — 8 байт. Вторая переменная, `datavar2`, находится в той же секции с большим смещением, равным 8, сразу после `datavar1`. Поскольку `somewhere` определён как `extern`, он, очевидно, находится в каком-то другом модуле, так что пока его адрес не имеет смысла и оставлен равным нулю.

2. Строка из семи букв и пробелов; каждая буква некоторым образом характеризует

символ. Некоторые из них представляют для нас интерес.

(a) `l, g,-` — локальный, глобальный или ни то ни другое.

(b) ...

(c) ...

(d) ...

(e) `I,-` — ссылка на другой символ или обычный символ.

(f) `d, D,-` — отладочный символ, динамический символ или обычный символ.

(g) `F, f, O,-` — имя функции, имя файла, имя объекта или обычный символ.

3. Какой секции соответствует эта метка? `*UND*` для неизвестной секции (символ,

на который ссылаются, но который здесь не определён), `*ABS*` означает отсутствие секции вообще.

4. Обычно это число показывает выравнивание (или его отсутствие).
5. Имя символа.

Например, давайте рассмотрим первый символ, показанный в листинге 5-23. Это

`f` — имя файла, `d` — нужен только для отладки, `l` — локален для этого модуля. Глобальная метка `_start` (которая также является *точкой входа*) отмечена буквой g во втором столбце.

> **Примечание** Имена символов регистрозависимы: `_start` и `_STaRT` — разные.

Поскольку адреса в таблице символов — это ещё не *настоящие* виртуальные адреса, а относительные к секциям, мы можем задаться вопросом: как это выглядит в машинном коде? NASM уже выполнил свою работу, и машинные инструкции должны быть собраны. Мы можем заглянуть внутрь интересующих нас секций объектных файлов, вызвав `objdump` с параметрами `-D` (дизассемблировать) и, опционально, `-M intel-mnemonic` (чтобы он показывал синтаксис в стиле Intel, а не AT&T). Листинг 5-24 показывает результаты.

> **Как читать дампы дизассемблирования** Левый столбец обычно показывает абсолютный адрес, по которому будут загружены данные. До компоновки это адрес относительно начала секции. Второй столбец показывает необработанные байты в виде шестнадцатеричных чисел. Третий столбец может содержать результаты дизассемблирования мнемоник ассемблерных команд.

Листинг 5-24. `objdump_d`

```
> objdump -D -M intel-mnemonic main.o
main.o:     file format elf64-x86-64
Disassembly of section .data:
0000000000000000 <datavar1>:        ...
0000000000000008 <datavar2>:        ...
Disassembly of section .bss:
0000000000000000 <bssvar1>:         ...
0000000002000000 <bssvar2>:         ...
Disassembly of section .text:
0000000000000000 <_start-0x28>:
   0:   48 b8 00 00 00 00 00      movabs rax,0x0
   7:   00 00 00
   a:   48 b8 00 00 00 00 00      movabs rax,0x0
   11:  00 00 00
   14:  48 b8 00 00 00 00 00      movabs rax,0x0
   1b:  00 00 00
   1e:  48 ba 00 00 00 00 00      movabs rdx,0x0
   25:  00 00 00
0000000000000028 <_start>:
   28:  c3                        ret
0000000000000029 <textlabel>:
```

Операнд `mov` в секции `.text` со смещениями 0 и 14 относительно начала секции должен быть адресом `datavar1`, но он равен нулю! То же самое произошло с `bssvar`. Это означает, что компоновщику нужно изменить скомпилированный машинный код, заполнив аргументы инструкций правильными абсолютными адресами. Чтобы этого добиться, для каждого символа все ссылки на него запоминаются в *таблице перемещений*. Как только компоновщик понимает, каким будет его настоящий виртуальный адрес, он проходит по списку вхождений символа и заполняет пробелы. Для каждой секции, которой это нужно, существует отдельная таблица перемещений. Чтобы посмотреть таблицы перемещений, используйте `readelf --relocs`. См. листинг 5-25.

Листинг 5-25. `readelf_relocs`

```
> readelf --relocs  main.o
Relocation section '.rela.text' at offset 0x440 contains 4 entries:
  Offset          Info           Type            Sym. Value    Name+Addend
000000000002  000200000001 R_X86_64_64       0000000000000000 .data + 0
00000000000c  000300000001 R_X86_64_64       0000000000000000 .bss + 0
000000000016  000300000001 R_X86_64_64       0000000000000000 .bss + 2000000
000000000020  000200000001 R_X86_64_64       0000000000000000 .data + 8
```

Альтернативный способ показать таблицу символов — использовать более лёгкую и минималистичную утилиту `nm`. Для каждого символа она показывает виртуальный адрес символа, тип и имя. Заметьте, что флаг типа представлен в другом формате по сравнению с `objdump`. Минимальный пример см. в листинге 5-26.

Листинг 5-26. `nm`

```
> nm main.o
0000000000000000 b bssvar
0000000000000000 d datavar
                 U somewhere
000000000000000a T _start
000000000000000b t textlabel
```

5.3.3 Исполняемые объектные файлы

Второй тип объектного файла можно выполнить сразу же. Он сохраняет свою структуру, но теперь адреса привязаны к точным значениям. Давайте рассмотрим ещё один пример, показанный в листинге 5-27. Он включает две глобальные переменные, `somewhere` и `private`, одна из которых доступна всем модулям (помечена `global`). Кроме того, символ `func` помечен как `global`.

Листинг 5-27. `executable_object.asm`

```
global somewhere
global func
section .data
somewhere: dq 999
private: dq 666
section .text
func:
    mov rax, somewhere
    ret
```

Мы скомпилируем его как обычно с помощью `nasm -f elf64`, а затем скомпонуем с помощью `ld` вместе с предыдущим объектным файлом, полученным при компиляции файла из листинга 5-22. Листинг 5-28 показывает изменения в выводе `objdump`.

Листинг 5-28. `objdump_tf`

```
> nasm -f elf64 symbols.asm
> nasm -f elf64  executable_object.asm
> ld symbols.o executable_object.o -o main
> objdump -tf main
main:     file format elf64-x86-64
architecture: i386:x86-64, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x0000000000000000
SYMBOL TABLE:
00000000004000b0 l    d  .code  0000000000000000 .code
00000000006000bc l    d  .data  0000000000000000 .data
0000000000000000 l    df *ABS*  0000000000000000 executable_object.asm
00000000006000c4 l       .data  0000000000000000 private
00000000006000bc g       .data  0000000000000000 somewhere
0000000000000000         *UND*  0000000000000000 _start
00000000006000cc g       .data  0000000000000000 __bss_start
00000000004000b0 g     F .code  0000000000000000 func
00000000006000cc g       .data  0000000000000000 _edata
00000000006000d0 g       .data  0000000000000000 _end
```

Флаги другие: теперь файл можно выполнять (`EXEC_P`); таблиц перемещений больше нет (флаг `HAS_RELOC` снят). Виртуальные адреса теперь неизменны, как и адреса в коде. Этот файл готов к загрузке и выполнению. Он сохраняет таблицу символов, и если вы хотите вырезать её, чтобы сделать исполняемый файл меньше, используйте утилиту `strip` .

> **Вопрос 71** Почему `ld` выдаёт предупреждение, если `_start` не помечена как global? Найдите адрес точки входа в этом случае с помощью `readelf` с подходящими аргументами.

> **Вопрос 72** Найдите опцию `ld` для автоматического удаления таблицы символов после компоновки.

5.3.4 Динамические библиотеки

Почти каждая программа использует код из библиотек. Есть два типа библиотек: статические и динамические. *Статические библиотеки* состоят из нескольких перемещаемых объектных файлов. Они компонуются с главной программой и объединяются с итоговым исполняемым файлом.

В мире Windows такие файлы имеют расширение `.lib`.

В мире Unix это либо файлы `.o`, либо архивы `.a`, хранящие внутри несколько файлов `.o`.

Динамические библиотеки также известны как *разделяемые объектные файлы* — третий из трёх ранее определённых нами типов объектных файлов.

Они компонуются с программой во время её выполнения.

В мире Windows это печально известные файлы `.dll`.

В мире Unix такие файлы имеют расширение `.so` (shared objects, разделяемые объекты). В то время как статические библиотеки — это просто недоготовленные исполняемые файлы без точек входа, у динамических библиотек есть некоторые отличия, которые мы сейчас рассмотрим. Динамические библиотеки загружаются по мере необходимости. Поскольку они сами по себе являются объектными файлами, у них есть всевозможная метаинформация о том, какой код они предоставляют для внешнего использования. Эту информацию использует загрузчик, чтобы определить точные адреса экспортируемых функций и данных. Динамические библиотеки можно поставлять отдельно и обновлять независимо. Это одновременно хорошо и плохо. Хотя производитель библиотеки может предоставлять исправления ошибок, он также может сломать обратную совместимость, например, изменив аргументы функций, фактически поставив мину замедленного действия. Программа может работать с любым количеством разделяемых библиотек. Такие библиотеки должны быть загружаемыми по любому адресу. Иначе они были бы привязаны к одному и тому же адресу, что ставит нас точно в такую же ситуацию, как при попытке выполнить несколько программ в одном и том же адресном пространстве физической памяти. Есть два способа этого добиться:

- Можно выполнить перемещение во время выполнения, когда библиотека загружается. Однако

это лишает нас одной очень привлекательной возможности: повторного использования кода библиотеки в физической памяти без его дублирования, когда его используют несколько процессов. Если каждый процесс выполняет перемещение библиотеки на свой собственный адрес, соответствующие страницы оказываются пропатчены разными значениями адресов и потому становятся разными для разных процессов.

По сути, секция *.data* всё равно была бы перемещена из-за своей изменяемой природы. Отказ от глобальных переменных позволяет нам избавиться и от самой секции, и от необходимости её перемещать.

Другая проблема в том, что секция *.text* должна оставаться доступной для записи, чтобы можно было выполнить её изменение в процессе перемещения. Это создаёт определённые риски безопасности, оставляя возможность её изменения вредоносным кодом. Более того, изменение *.text* каждого разделяемого объекта, когда для запуска исполняемого файла требуется несколько библиотек, может занимать очень много времени.

- Можно писать *PIC (Position Independent Code, позиционно-независимый код)*. Теперь возможно писать код,

который можно выполнить независимо от того, где он находится в памяти. Для этого нам нужно полностью избавиться от абсолютных адресов. Сегодня процессоры поддерживают адресацию относительно `rip`, например `mov rax, [rip + 13]`. Эта возможность облегчает создание PIC.

Эта техника позволяет разделять секцию *.text*. Сегодня программистам настоятельно рекомендуется использовать PIC вместо перемещений.

> **Примечание** Всякий раз, когда вы используете неконстантные глобальные переменные, вы лишаете свой код повторной входимости (reentrancy), то есть возможности выполняться внутри нескольких потоков одновременно без изменений. Как следствие, у вас возникнут трудности с повторным использованием его в разделяемой библиотеке. Это один из многих аргументов против глобального изменяемого состояния в программе.

Динамические библиотеки экономят место на диске и в памяти. Помните, что страницы могут быть помечены либо как приватные, либо как общие для нескольких процессов. Если библиотеку использует несколько процессов, большая часть её не дублируется в физической памяти. Сейчас мы покажем вам, как построить минимальный разделяемый объект. Однако объяснение таких вещей, как таблица глобальных смещений (Global Offset Table) и таблица связей процедур (Procedure Linkage Table), мы отложим до Главы 15. Листинг 5-29 показывает содержимое минимального разделяемого объекта. Обратите внимание на внешний символ `_GLOBAL_OFFSET_TABLE` и спецификацию `:function` для глобального символа `func`. Листинг 5-30 показывает минимальный запускающий файл, который вызывает функцию в разделяемом объектном файле и корректно завершается.

Листинг 5-29. `libso.asm`

```
Extern  _GLOBAL_OFFSET_TABLE_
global func:function
section .rodata
message: db "Shared object wrote this", 10, 0
section .text
func:
    mov     rax, 1
    mov     rdi, 1
    mov     rsi, message
    mov     rdx, 14
    syscall
ret
```

Листинг 5-30. `libso_main.asm`

```
global _start
extern func
section .text
_start:
    mov rdi, 10
    call func
    mov rdi, rax
    mov rax, 60
    syscall
```

Листинг 5-31 показывает команды сборки и два представления ELF-файла. Обратите внимание, что у динамической библиотеки есть более специфичные секции, такие как `.dynsym`. Секции `.hash`, `.dynsym` и `.dynstr` необходимы для перемещения. *.dynsym* хранит символы, видимые снаружи библиотеки. *.hash* — хеш-таблица, нужная для уменьшения времени поиска символов в *.dynsym*. *.dynstr* хранит строки, запрашиваемые по их индексам из *.dynsym*.

Листинг 5-31. `libso`

```
> nasm -f elf64 -o main.o main.asm
> nasm -f elf64 -o libso.o libso.asm
> ld -o main main.o -d libso.so
> ld -shared -o libso.so libso.o --dynamic-linker=/lib64/ld-linux-x86-64.so.2
> readelf -S libso.so
There are 13 section headers, starting at offset 0x5a0:
Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .hash             HASH             00000000000000e8  000000e8
       000000000000002c  0000000000000004   A       2     0      8
  [ 2] .dynsym           DYNSYM           0000000000000118  00000118
       0000000000000090  0000000000000018   A       3     2      8
  [ 3] .dynstr           STRTAB           00000000000001a8  000001a8
       000000000000001e  0000000000000000   A       0     0      1
  [ 4] .rela.dyn         RELA             00000000000001c8  000001c8
       0000000000000018  0000000000000018   A       2     0      8
  [ 5] .text             PROGBITS         00000000000001e0  000001e0
       000000000000001c  0000000000000000   AX      0     0      16
  [ 6] .rodata           PROGBITS         00000000000001fc  000001fc
       000000000000001a  0000000000000000   A       0     0      4
  [ 7] .eh_frame         PROGBITS         0000000000000218  00000218
       0000000000000000  0000000000000000   A       0     0      8
  [ 8] .dynamic          DYNAMIC          0000000000200218  00000218
       00000000000000f0  0000000000000010   WA      3     0      8
  [ 9] .got.plt          PROGBITS         0000000000200308  00000308
       0000000000000018  0000000000000008   WA      0     0      8
  [10] .shstrtab         STRTAB           0000000000000000  00000320
       0000000000000065  0000000000000000           0     0      1
  [11] .symtab           SYMTAB           0000000000000000  00000388
       00000000000001c8  0000000000000018          12    15      8
  [12] .strtab           STRTAB           0000000000000000  00000550
       000000000000004f  0000000000000000           0     0      1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), l (large)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
> readelf -S main
There are 14 section headers, starting at offset 0x650:
Section Headers:
  [Nr] Name              Type             Address           Offset
```

```
      Size              EntSize          Flags  Link  Info  Align
  [ 0]                  NULL             0000000000000000  00000000
      0000000000000000  0000000000000000           0     0     0
  [ 1] .interp          PROGBITS         0000000000400158  00000158
      000000000000000f  0000000000000000   A       0     0     1
  [ 2] .hash            HASH             0000000000400168  00000168
      0000000000000028  0000000000000004   A       3     0     8
  [ 3] .dynsym          DYNSYM           0000000000400190  00000190
      0000000000000078  0000000000000018   A       4     1     8
  [ 4] .dynstr          STRTAB           0000000000400208  00000208
      0000000000000027  0000000000000000   A       0     0     1
  [ 5] .rela.plt        RELA             0000000000400230  00000230
      0000000000000018  0000000000000018  AI       3     6     8
  [ 6] .plt             PROGBITS         0000000000400250  00000250
      0000000000000020  0000000000000010  AX       0     0     16
  [ 7] .text            PROGBITS         0000000000400270  00000270
      0000000000000014  0000000000000000  AX       0     0     16
  [ 8] .eh_frame        PROGBITS         0000000000400288  00000288
      0000000000000000  0000000000000000   A       0     0     8
  [ 9] .dynamic         DYNAMIC          0000000000600288  00000288
      0000000000000110  0000000000000010  WA       4     0     8
  [10] .got.plt         PROGBITS         0000000000600398  00000398
      0000000000000020  0000000000000008  WA       0     0     8
  [11] .shstrtab        STRTAB           0000000000000000  000003b8
      0000000000000065  0000000000000000           0     0     1
  [12] .symtab          SYMTAB           0000000000000000  00000420
      00000000000001e0  0000000000000018          13    15     8
  [13] .strtab          STRTAB           0000000000000000  00000600
      000000000000004d  0000000000000000           0     0     1
```

Вопрос 73 Изучите таблицы символов полученного разделяемого объекта с помощью `readelf --dyn-syms` и

```
objdump -ft.
```

Вопрос 74 В чём смысл переменной окружения `LD_LIBRARY_PATH`?

Вопрос 75 Разделите первое задание на два модуля. Первый модуль будет хранить все функции, определённые в `lib.inc`. Второй будет содержать точку входа и вызывать некоторые из этих функций.

> **Вопрос 76** Возьмите одну из стандартных утилит Linux (из coreutils). Изучите структуру её объектного файла с помощью

```
readelf and objdump.
```

То, что мы наблюдали в этом разделе, применимо в большинстве ситуаций. Однако есть более широкая картина разных *моделей кода*, влияющих на адресацию. Мы углубимся в эти детали в Главе 15, познакомившись поближе с ассемблером и C. Там мы также снова рассмотрим динамические библиотеки и введём понятия таблицы глобальных смещений и таблицы связей процедур.

5.3.5 Загрузчик

Загрузчик — часть операционной системы, которая готовит исполняемый файл к выполнению. Это включает отображение его соответствующих секций в память, инициализацию *.bss* и иногда отображение других файлов с диска. Заголовки программы для файла `symbols.asm`, показанного в листинге 5-22, показаны в листинге 5-32.

Листинг 5-32. `symbols_pht`

```
> nasm -f elf64 symbols.asm
> nasm -f elf64 executable_object.asm
> ld symbols.o executable_object.o -o main
> readelf -l main
Elf file type is EXEC (Executable file)
Entry point 0x4000d8
There are 2 program headers, starting at offset 64
Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x00000000000000e3 0x00000000000000e3  R E    200000
  LOAD           0x00000000000000e4 0x00000000006000e4 0x00000000006000e4
                 0x0000000000000010 0x000000000200001c  RW     200000
 Section to Segment mapping:
  Segment Sections...
   00     .text
   01     .data .bss
```

Таблица сообщает нам, что присутствуют два сегмента.

1. Сегмент `00`

- Загружается по адресу `0x400000`, выровненному по `0x200000`.
- Содержит секцию `.text`.
- Можно выполнять и можно читать. Нельзя записывать (так что вы не можете перезаписать код).

2. Сегмент `01`

- Загружается по адресу `0x6000e4`, выровненному по `0x200000`.
- Можно читать и записывать.

Выравнивание означает, что настоящий адрес будет ближайшим к началу, делящимся на `0x200000`. Благодаря виртуальной памяти вы можете загружать все программы по одному и тому же начальному адресу. Обычно это 0x400000. Стоит отметить несколько важных вещей:

- Секции ассемблера с похожими именами, определённые в разных файлах, объединяются.
- В чистом исполняемом файле таблица перемещений не нужна. Перемещения частично остаются

для разделяемых объектов. Давайте запустим получившийся файл и посмотрим на его файл `/proc/<pid>/maps`, как мы делали в Главе 4. Листинг 5-33 показывает пример его содержимого. Исполняемый файл написан так, чтобы бесконечно выполнять цикл.

Листинг 5-33. `symbols_maps`

```
00400000-00401000 r-xp 00000000 08:01 1176842
                           /home/sayon/repos/spbook/en/listings/chap5/main
00600000-00601000 rwxp 00000000 08:01 1176842
                           /home/sayon/repos/spbook/en/listings/chap5/main
00601000-02601000 rwxp 00000000 00:00 0
7ffe19cf2000-7ffe19d13000 rwxp 00000000 00:00 0
                           [stack]
7ffe19d3e000-7ffe19d40000 r-xp 00000000 00:00 0
                           [vdso]
7ffe19d40000-7ffe19d42000 r--p 00000000 00:00 0
                           [vvar]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0
                           [vsyscall]
```

Как видим, заголовок программы говорит нам правду о размещении секций.

> **Примечание** В некоторых случаях вам потребуется тонко настроить компоновщик. Адреса загрузки секций

> **Примечание.** и их относительное расположение можно настроить с помощью **скриптов компоновщика**, описывающих итоговый файл. Такие случаи обычно возникают при программировании операционной системы или прошивки микроконтроллера. Эта тема выходит за рамки данной книги, но мы рекомендуем вам заглянуть в [4], если у вас возникнет такая необходимость.

5.4 Задание: словарь

Это задание продвинет нас ещё на шаг ближе к рабочему интерпретатору Форта. Некоторые вещи в нём могут показаться натянутыми, как, например, конструкция макроса, но это заложит хороший фундамент для интерпретатора, который мы напишем позже. Наша задача — реализовать словарь. Он будет устанавливать соответствие между ключами и значениями. Каждая запись содержит адрес следующей записи, ключ и значение. Ключи и значения в нашем случае — строки, завершающиеся нулём. Записи словаря образуют структуру данных, называемую *связным списком*. Пустой список представлен нулевым указателем, равным нулю. Непустой список — это указатель на его первый элемент. Каждый элемент хранит некоторое значение и указатель на следующий элемент (или ноль, если это последний элемент). Листинг 5-34 показывает пример связного списка, хранящего элементы 100, 200 и 300. На него можно ссылаться указателем на его первый элемент, то есть `x1`.

Листинг 5-34. `linked_list_ex.asm`

```
section .data
x1:
dq x2
dq 100
x2:
dq x3
dq 200
x3:
dq 0
dq 300
```

Связные списки часто полезны в ситуациях с многочисленными вставками и удалениями в середине списка. Однако доступ к элементам по индексу затруднён, потому что он не сводится к простому сложению указателя. Взаимное расположение элементов

связного списка в плоской памяти обычно непредсказуемо. В этом задании словарь будет построен статически как список, и каждый вновь определяемый элемент будет добавляться в его начало. Вам нужно использовать макросы с локальными метками и переопределением символов, чтобы автоматизировать создание связного списка. Мы явно указываем вам сделать макрос `colon` с двумя аргументами, где первый будет хранить строку ключа словаря, а второй — имя внутреннего представления элемента. Это различие нужно, потому что строки ключей иногда могут содержать символы, которые не являются частью допустимых имён меток (пробел, знаки препинания, арифметические знаки и т. д.). Листинг 5-35 показывает пример такого словаря.

Листинг 5-35. `linked_list_ex_macro.asm`

```
section .data
colon "third word", third_word
db "third word explanation", 0
colon "second word", second_word
db "second word explanation", 0
colon "first word", first_word
db "first word explanation", 0
```

В задание войдут следующие файлы:

1. main.asm
2. lib.asm
3. dict.asm
4. colon.inc

Чтобы выполнить задание, следуйте этим шагам:

1. Создайте отдельный файл ассемблера, содержащий функции, которые вы уже написали

в первом задании. Мы назовём его `lib.o`.

Не забудьте пометить все нужные метки как `global`, иначе они не будут видны за пределами этого объектного файла!

2. Создайте файл `colon.inc` и определите там макрос `colon` для создания слов словаря.

Этот макрос будет принимать два аргумента:

- Ключ словаря (в кавычках).
- Имя метки ассемблера. Ключи могут содержать пробелы и другие символы, недопустимые в именах меток.

Каждая запись должна начинаться с указателя на следующую запись, затем содержать ключ в виде строки, завершающейся нулём. Содержимое затем напрямую описывается программистом — например, с помощью директив `db`, как в примере, показанном в листинге 5-35.

3. Создайте функцию `find_word` в новом файле `dict.asm`. Она принимает два аргумента:

(a) Указатель на строку ключа, завершающуюся нулём.

(b) Указатель на последнее слово в словаре. Имея указатель на последнее определённое слово, мы можем следовать по последовательным связям, чтобы перечислить все слова в словаре.

`find_word` пройдёт циклом по всему словарю, сравнивая заданный ключ с каждым ключом в словаре. Если запись не найдена, она возвращает ноль; иначе возвращает адрес записи.

4. Отдельный файл для включения `words.inc`, определяющий слова словаря с помощью макроса `colon`.

Включите его в `main.asm`.

5. Простая функция `_start`. Она должна выполнять следующие действия:

- Прочитать входную строку в буфер длиной максимум 255 символов.
- Попытаться найти этот ключ в словаре. Если найден, вывести соответствующее значение. Если нет, вывести сообщение об ошибке.

Не забудьте: все сообщения об ошибках должны записываться в `stderr`, а не в `stdout`! Мы предоставляем набор файлов-заготовок (см. раздел 2.1 «Настройка окружения»); вы вольны их использовать. Дополнительный `Makefile` описывает процесс сборки; наберите `make` в каталоге задания, чтобы собрать исполняемый файл `main`. Краткое руководство по системе GNU Make доступно в Приложении B. Как и в первом задании, есть файл `test.py` для автоматизированной проверки.

5.5 Итоги

В этой главе мы рассмотрели разные этапы компиляции. Мы подробно изучили макропроцессор NASM и разобрали условия и циклы. Затем мы поговорили о трёх типах объектных файлов: перемещаемых, исполняемых и разделяемых. Мы разобрали структуру ELF-файла и рассмотрели процесс перемещения, выполняемый компоновщиком. Мы затронули разделяемые объектные файлы и вернёмся к ним снова в Главе 15.

Вопрос 77 Что такое связный список?

Вопрос 78 Какие есть этапы компиляции?

Вопрос 79 Что такое препроцессирование?

Вопрос 80 Что такое инстанцирование макроса?

Вопрос 81 Что такое директива `%define` ?

Вопрос 82 Что такое директива `%macro` ?

Вопрос 83 В чём разница между `%define` , `%xdefine` и `%assign`?

Вопрос 84 Зачем нам нужен оператор `%%` внутри макроса?

Вопрос 85 Какие типы условий поддерживает макропроцессор NASM? Какие директивы для этого используются?

Вопрос 86 Какие есть три типа объектных файлов ELF?

Вопрос 87 Какие виды заголовков присутствуют в ELF-файле?

Вопрос 88 Что такое перемещение?

Вопрос 89 Какие секции могут присутствовать в ELF-файлах?

Вопрос 90 Что такое таблица символов? Какую информацию она хранит?

Вопрос 91 Есть ли связь между секциями и сегментами?

Вопрос 92 Есть ли связь между секциями ассемблера и секциями ELF?

Вопрос 93 Какой символ отмечает точку входа программы?

Вопрос 94 Какие есть два разных типа библиотек?

Вопрос 95 Есть ли разница между статической библиотекой и перемещаемым объектным файлом?

Прерывания и системные вызовы

В этой главе мы обсудим две темы. Во-первых, поскольку архитектуре фон Неймана не хватает интерактивности, для этого были введены прерывания. Хотя мы не будем углубляться в аппаратную часть прерываний, мы разберём в точности, как программист видит прерывания. Кроме того, мы поговорим о портах ввода и вывода, используемых для взаимодействия с внешними устройствами. Во-вторых, операционная система (ОС) обычно предоставляет интерфейс для взаимодействия с ресурсами, которые она контролирует: памятью, файлами, CPU (центральным процессором) и т. д. Это реализуется через механизм системных вызовов. Передача управления процедурам операционной системы требует хорошо определённого механизма повышения привилегий, и мы посмотрим, как он работает в архитектуре Intel 64.

6.1 Ввод и вывод

Когда мы расширяли архитектуру фон Неймана для работы с внешними устройствами, мы упоминали прерывания только как способ взаимодействия с ними. На самом деле есть вторая возможность, порты ввода/вывода (I/O), которая дополняет её и позволяет обмен данными между CPU и устройствами. Приложения могут обращаться к портам ввода/вывода двумя способами:

1. Через отдельное адресное пространство ввода/вывода. Есть 2^{16} 1-байтовых адресуемых портов ввода/вывода, от `0` до `FFFFH`. Команды `in` и `out` используются для обмена данными между портами и регистром `eax` (или его частями). Права на выполнение записи и чтения из портов контролируются проверкой:

- Поля IOPL (уровень привилегий ввода/вывода) регистра `rflags`
- Битовой карты разрешений ввода/вывода *сегмента состояния задачи*. Мы поговорим о ней в разделе 6.1.1.

2. Через ввод/вывод, отображаемый на память. Часть адресного пространства специально отображается, чтобы обеспечить взаимодействие с такими внешними устройствами, которые ведут себя как компоненты памяти. Соответственно, для выполнения ввода/вывода с этими устройствами можно использовать любые инструкции адресации памяти (`mov` , `movsb` и т. д.). К таким задачам ввода/вывода применяются стандартные механизмы защиты сегментации и страничной организации памяти.

Поле IOPL в регистре `rflags` работает следующим образом: если текущий уровень привилегий меньше или равен IOPL, разрешено выполнение следующих инструкций:

- `in` и `out` (обычный ввод/вывод).
- `ins` и `outs` (строковый ввод/вывод).

- `cli` и `sti` (сброс/установка флага прерываний).

Таким образом, установка IOPL для отдельного приложения позволяет нам запретить ему запись, даже если оно работает на более высоком уровне привилегий, чем пользовательские приложения. Кроме того, Intel 64 позволяет ещё более тонкий контроль прав доступа с помощью битовой карты разрешений ввода/вывода. Если проверка IOPL пройдена, процессор проверяет бит, соответствующий используемому порту. Операция выполняется, только если этот бит не установлен. Битовая карта разрешений ввода/вывода — часть *сегмента состояния задачи* (TSS), который был создан как сущность, уникальная для процесса. Однако, поскольку аппаратный механизм переключения задач считается устаревшим, в длинном режиме может существовать только один TSS (и одна битовая карта разрешений ввода/вывода).

6.1.1 Регистр TR и сегмент состояния задачи

Есть некоторые пережитки защищённого режима, которые всё ещё тем или иным образом используются в длинном режиме. Сегментация — один пример, теперь используемый в основном для реализации колец защиты. Другой — пара из регистра `tr` и управляющей структуры *сегмента состояния задачи*. Регистр `tr` хранит селектор сегмента дескриптора TSS. Последний находится в GDT (глобальной таблице дескрипторов) и имеет формат, схожий с дескрипторами сегментов. Как и для сегментных регистров, есть *теневой регистр*, который обновляется из *GDT* при обновлении `tr` инструкцией `ltr` (load task register, загрузить регистр задачи). TSS — это область памяти, используемая для хранения информации о задаче при наличии аппаратного механизма переключения задач. Поскольку ни одна популярная ОС не использовала его в защищённом режиме, этот механизм был убран из длинного режима. Однако TSS в длинном режиме всё ещё используется, хотя и с совершенно другой структурой и назначением. Сегодня операционная система использует только один TSS, структура которого описана на

Рисунке 6-1.

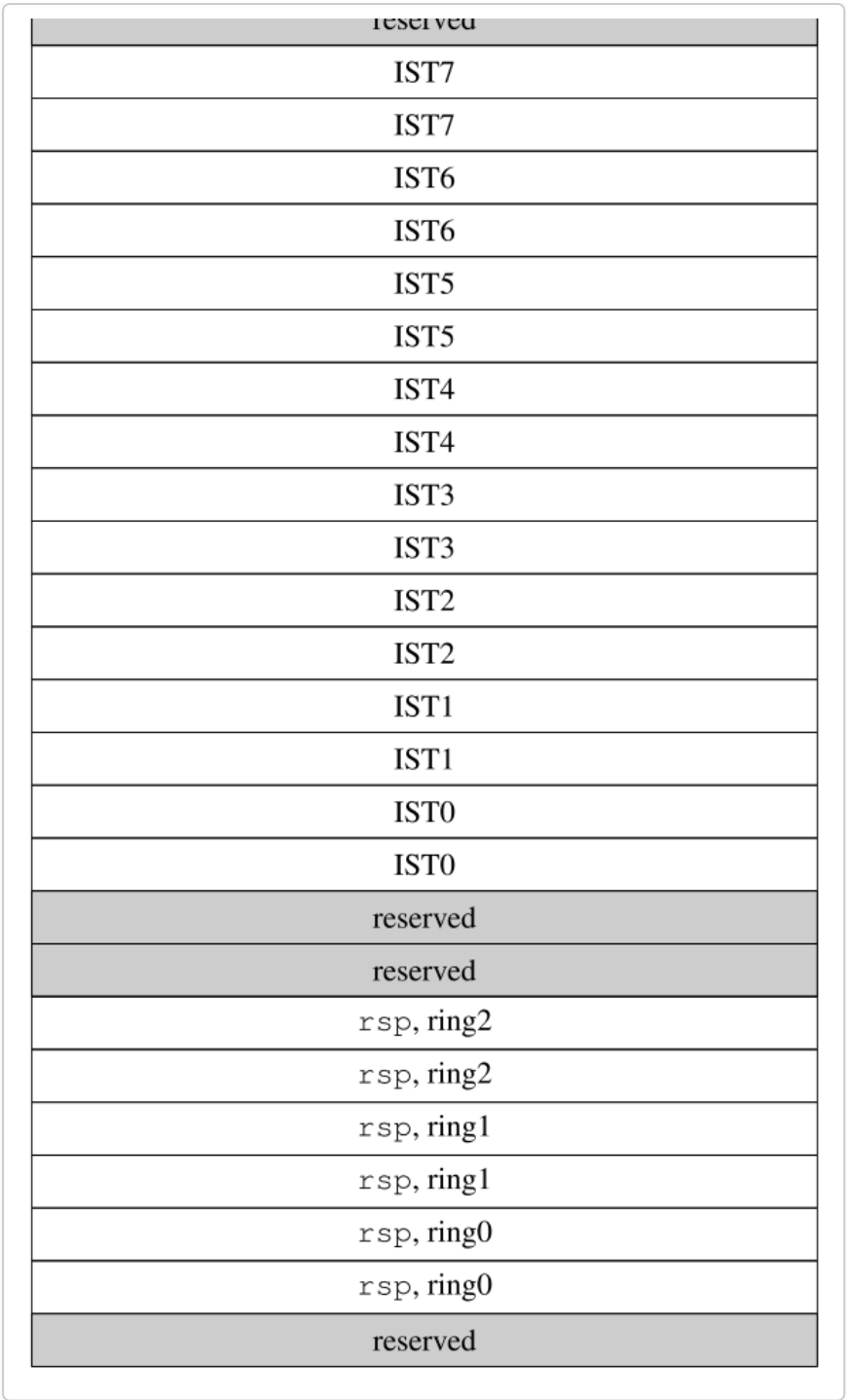


Рисунок 6-1. Сегмент состояния задачи в длинном режиме

Первые 16 бит хранят смещение до карты разрешений портов ввода/вывода, которую мы уже обсуждали в разделе 6.1. Затем TSS хранит восемь указателей на специальные *таблицы стеков прерываний* (IST) и указатели стека для разных колец. Каждый раз при смене уровня привилегий стек автоматически меняется соответствующим образом. Обычно новое значение `rsp` будет взято из поля TSS, соответствующего новому кольцу защиты. Назначение IST объясняется в разделе 6.2.

6.2 Прерывания

Прерывания позволяют нам изменить поток управления программы в произвольный момент времени. Пока программа выполняется, внешние события (устройству требуется внимание CPU) или внутренние события (деление на ноль, недостаточный уровень привилегий для выполнения инструкции, неканонический адрес) могут вызвать прерывание, в результате которого выполняется какой-то другой код. Этот код называется *обработчиком прерывания* и является частью операционной системы или программного обеспечения драйвера. В [15] Intel отделяет внешние асинхронные прерывания от внутренних синхронных исключений, но обрабатываются они одинаково. Каждое прерывание помечено фиксированным числом, служащим его идентификатором. Для нас не важно, как именно процессор получает номер прерывания от контроллера прерываний. Когда происходит *n*-е прерывание, CPU проверяет *таблицу дескрипторов прерываний* (IDT), находящуюся в памяти. По аналогии с *GDT*, её адрес и размер хранятся в `idtr`. Рисунок 6-2 описывает `idtr`.



Рисунок 6-2. Регистр `idtr`

Каждая запись в IDT занимает 16 байт, и *n*-я запись соответствует *n*-му прерыванию. Запись включает некоторую служебную информацию, а также адрес обработчика прерывания. Рисунок 6-3 описывает формат дескриптора прерывания.

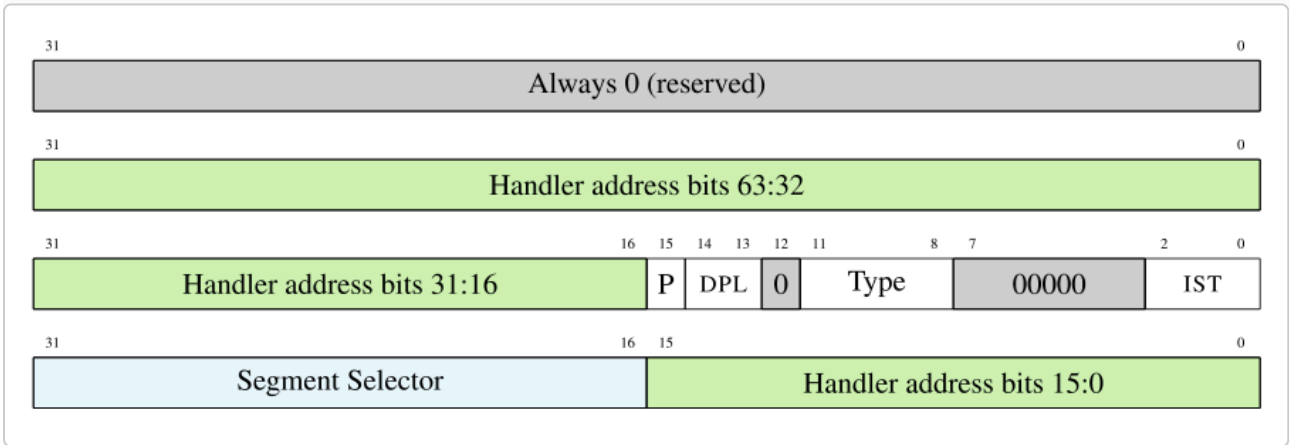


Рисунок 6-3. Дескриптор прерывания

DPL Descriptor Privilege Level (уровень привилегий дескриптора)

Текущий уровень привилегий должен быть меньше или равен DPL, чтобы вызвать этот обработчик с помощью инструкции `int`. В противном случае проверка не выполняется. Тип 1110 (interrupt gate, шлюз прерывания, `IF` автоматически сбрасывается в обработчи-

ке) или 1111 (trap gate, шлюз-ловушка, `IF` не сбрасывается). Первые 30 прерываний зарезервированы. Это означает, что вы можете предоставить для них обработчики прерываний, но CPU будет использовать их для своих внутренних событий, таких как некорректная кодировка инструкции. Остальные прерывания могут использоваться системным программистом. Когда флаг `IF` установлен, прерывания обрабатываются; иначе они игнорируются.

> **Вопрос 96** Что такое немаскируемые прерывания? Как они связаны с прерыванием с кодом 2 и флагом `IF` ?

Код приложения выполняется с низкими привилегиями (в кольце 3). Прямое управление устройством возможно только на более высоких уровнях привилегий. Когда устройству требуется внимание, и оно посылает прерывание CPU, обработчик должен выполняться в кольце с более высокими привилегиями, что требует изменения селектора сегмента. А как насчёт стека? Стек тоже нужно переключить. Здесь у нас есть несколько вариантов в зависимости от того, как настроено поле IST дескриптора прерывания.

- Если IST равно 0, используется стандартный механизм. Когда происходит прерывание, в `ss` загружается 0, а новое значение `rsp` загружается из TSS. Затем поле RPL регистра `ss` устанавливается на подходящий уровень привилегий. Затем *старые* `ss` и `rsp` сохраняются на этом новом стеке.
- Если IST установлено, используется одна из семи IST, определённых в TSS. Причина создания IST в том, что некоторые серьёзные сбои (немаскируемые прерывания, двойной сбой и т. д.) могут выиграть от выполнения на заведомо исправном стеке. Так что системный программист может создать несколько стеков даже для кольца 0 и использовать некоторые из них для обработки конкретных прерываний.

Есть специальная инструкция `int`, принимающая номер прерывания. Она вызывает обработчик прерывания вручную, учитывая содержимое его дескриптора. Она игнорирует флаг `IF`: установлен он или сброшен, обработчик всё равно будет вызван. Для контроля выполнения привилегированного кода с помощью инструкции `int` существует поле DPL. Перед тем как обработчик прерывания начнёт выполняться, некоторые регистры автоматически сохраняются в стек. Это `ss`, `rsp`, `rflags`, `cs` и `rip`. См. диаграмму стека на рисунке 6-4. Обратите внимание, как селекторы сегментов дополняются нулями до 64 бит.

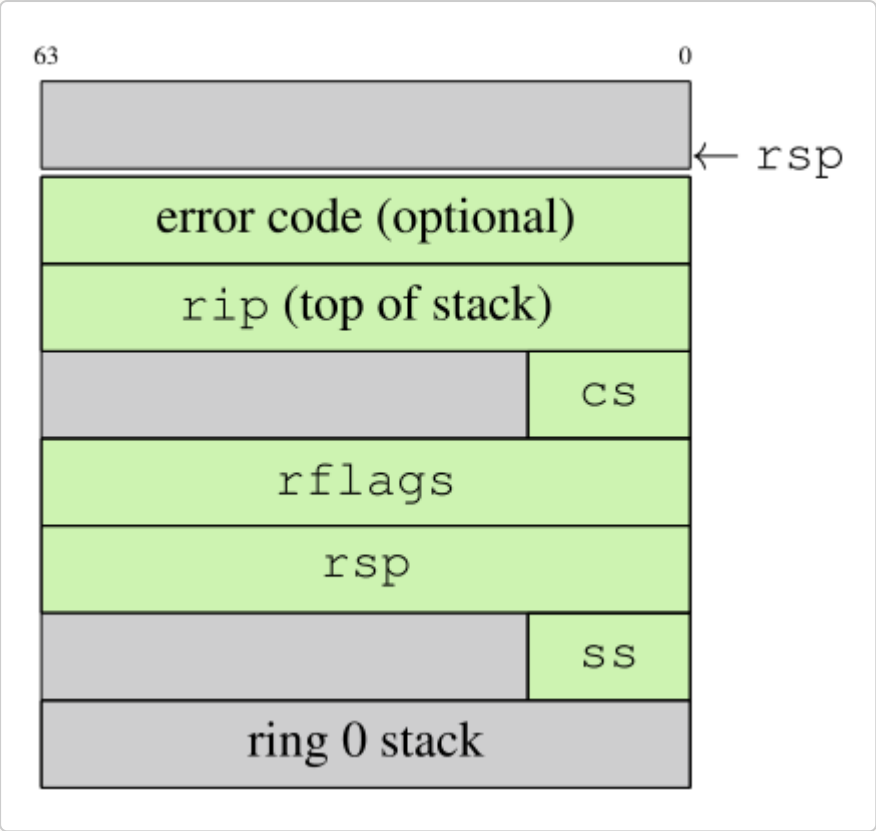


Рисунок 6-4. Стек в момент начала работы обработчика прерывания

Иногда обработчику прерывания нужна дополнительная информация о событии. Тогда в стек помещается *код ошибки прерывания*. Этот код содержит различную информацию, специфичную для этого типа прерывания. Многие прерывания описываются с помощью специальных мнемоник в документации Intel. Например, 13-е прерывание называется *#GP* (general protection, общая защита).[18] Краткое описание некоторых интересных прерываний вы найдёте в таблице 6-1.

Таблица 6-1. Некоторые важные прерывания

ВЕКТОР	МНЕМОНИКА	ОПИСАНИЕ
0	#DE	Ошибка деления
2		Немаскируемое внешнее прерывание
3	#BP	Точка останова
6	#UD	Недопустимый код операции инструкции
8	#DF	Сбой при обработке прерывания
13	#GP	Общая защита
14	#PF	Страничный отказ

Не весь двоичный код соответствует корректно закодированным машинным инструкциям. Когда `rip` указывает не на действительную инструкцию, CPU генерирует прерывание #UD. Прерывание #GP встречается очень часто. Оно генерируется при попытке разыменовать запрещённый адрес (не соответствующий ни одной выделенной странице), при попытке выполнить действие, требующее более высокого уровня привилегий, и так далее. Прерывание #PF генерируется при обращении к странице, у которой в соответствующей записи таблицы страниц снят флаг присутствия. Это прерывание используется для реализации механизма подкачки и отображения файлов в целом. Обработчик прерывания может загрузить отсутствующие страницы с диска. Отладчики сильно полагаются на прерывание #BP. Когда в `rflags` установлен `TF`, прерывание с этим кодом генерируется после выполнения *каждой* инструкции, что позволяет выполнять программу пошагово. Очевидно, это прерывание обрабатывается ОС. Таким образом, обязанность ОС — предоставить пользовательским приложениям интерфейс, позволяющий программистам писать собственные отладчики. Подводя итог, когда происходит n-е прерывание, с точки зрения программиста выполняются следующие действия:

1. Адрес *IDT* берётся из `idtr`.
2. Дескриптор прерывания находится начиная со 128 × n-го байта IDT.
3. Селектор сегмента и адрес обработчика загружаются из записи IDT в

`cs` и `rip`, возможно изменяя уровень привилегий. Старые `ss`, `rsp`, `rflags`, `cs` и `rip` сохраняются в стек, как показано на рисунке 6-4.

4. Для некоторых прерываний в стек обработчика помещается код ошибки. Он предоставляет

дополнительную информацию о причине прерывания.

5. Если поле `type` дескриптора определяет его как Interrupt Gate (шлюз прерывания), флаг прерываний `IF`

сбрасывается. Trap Gate (шлюз-ловушка), однако, не сбрасывает его автоматически, что позволяет вложенную обработку прерываний.

Если флаг прерываний не сбрасывается сразу после начала работы обработчика прерывания, у нас нет никаких гарантий, что мы выполним даже его первую инструкцию, прежде чем асинхронно возникнет другое прерывание, требующее нашего внимания.

> **Вопрос 97** Сбрасывается ли флаг `TF` автоматически при входе в обработчики прерываний? Обратитесь к [15].

Работа обработчика прерывания завершается инструкцией `iretq`, которая восстанавливает все регистры, сохранённые в стеке, как показано на рисунке 6-4, в отличие от простой инструкции `call`, которая восстанавливает только `rip`.

6.3 Системные вызовы

Системные вызовы, как вы уже знаете, — это функции, которые ОС предоставляет пользовательским приложениям. В этом разделе описывается механизм, обеспечивающий их безопасное выполнение с более высоким уровнем привилегий. Механизмы, используемые для реализации системных вызовов, различаются в разных архитектурах. В целом подойдёт любая инструкция, приводящая к прерыванию, например деление на ноль или любая некорректно закодированная инструкция. Будет вызван обработчик прерывания, а дальше CPU разберётся сам. В защищённом режиме на архитектуре Intel *nix-операционные системы использовали прерывание с кодом 0x80. Каждый раз, когда пользователь выполнял `int0x80`, обработчик прерывания проверял содержимое регистров на номер системного вызова и аргументы. Системные вызовы происходят довольно часто, и без них невозможно никакое взаимодействие с внешним миром. Однако прерывания могут быть медленными, особенно в Intel 64, поскольку они требуют обращений к памяти для *IDT*. Поэтому в Intel 64 есть новый механизм выполнения системных вызовов, реализующий их с помощью инструкций `syscall` и `sysret`. По сравнению с прерываниями, у этого механизма есть несколько ключевых отличий:

- Переход возможен только между кольцом 0 и кольцом 3. Поскольку практически никто не использует кольцо 1 и кольцо 2, это ограничение не считается важным.
- Обработчики прерываний различаются, а все системные вызовы обрабатываются одним и тем же кодом с единственной точкой входа.
- Некоторые регистры общего назначения теперь неявно используются во время системного вызова.
- `rcx` используется для хранения старого `rip`, `r11` используется для хранения старого `rflags`

6.3.1 Модельно-специфичные регистры

Иногда, когда появляется новый CPU, у него есть дополнительные регистры, которых нет у более старых. Довольно часто это так называемые *модельно-специфичные регистры*. Поскольку эти регистры изменяются редко, работа с ними выполняется с помощью двух команд: `rdmsr` для их чтения и `wrmsr` для их изменения. Эти две команды оперируют *идентифицирующим номером* регистра. `rdmsr` принимает номер MSR в `ecx`, возвращает значение регистра в `edx`:`eax`. `wrmsr` принимает номер MSR в `ecx` и сохраняет в него значение, взятое из `edx`:`eax`.

6.3.2 syscall и sysret

Инструкция `syscall` зависит от нескольких *MSR*.

- STAR (номер MSR 0xC0000081), хранящий две пары значений `cs` и `ss`: для

обработчика системного вызова и для инструкции `sysret`. Рисунок 6-5 показывает его структуру.



Рисунок 6-5. MSR STAR

- LSTAR (номер MSR 0xC0000082) хранит адрес обработчика системного вызова (новый `rip`).
- SFMASK (номер MSR 0xC0000084) показывает, какие биты в `rflags` должны быть сброшены в

обработчике системного вызова. `syscall` выполняет следующие действия:

- Загружает `cs` из `STAR`;
- Изменяет `rflags` в соответствии с `SFMASK`;
- Сохраняет `rip` в `rcx`; и
- Инициализирует `rip` значением `LSTAR` и берёт новые `cs` и `ss` из `STAR`.

Заметьте, теперь мы можем объяснить, почему системные вызовы и процедуры принимают аргументы в слегка разных наборах регистров. Процедуры принимают свой четвёртый аргумент в `rcx`, который, как мы знаем, используется для хранения старого значения `rip`. В отличие от прерываний, даже если уровень привилегий меняется, указатель стека должен быть изменён самим обработчиком. Обработка системного вызова заканчивается инструкцией `sysret`, которая загружает `cs` и `ss` из `STAR`, а `rip` из `rcx`. Как мы знаем, изменение селектора сегмента приводит к чтению из *GDT* для обновления парного ему *теневого регистра*. Однако при выполнении `syscall` эти теневые регистры загружаются фиксированными значениями, и никаких чтений из GDT не выполняется. Вот эти два фиксированных значения в расшифрованном виде:

- Теневой регистр сегмента кода:
- Base = 0
- Limit = `FFFFFH`
- Type = 11_2 (можно выполнять, было обращение)
- S = 1 (System)
- DPL = 0
- P = 1
- L = 1 (длинный режим)
- D = 0
- G = 1 (всегда так в длинном режиме)

Кроме того, CPL (текущий уровень привилегий) устанавливается в 0

- Теневой регистр сегмента стека: Base = 0
- Limit = `FFFFFH` Type = 11_2 (можно выполнять, было обращение)
- S = 1 (System) DPL = 0
- P = 1 L = 1 (длинный режим)
- D = 1 G = 1

Однако системный программист обязан выполнить одно требование: в GDT должны быть дескрипторы, соответствующие этим фиксированным значениям. Так что в GDT должны храниться два особых дескриптора для кода и данных специально для поддержки `syscall`.

6.4 Итоги

В этой главе мы дали обзор механизмов прерываний и системных вызовов. Мы изучили их реализацию вплоть до системных структур данных, находящихся в памяти. В следующей главе мы рассмотрим разные модели вычислений, включая стековые машины наподобие Форта и конечные автоматы, и наконец поработаем над интерпретатором и компилятором Форта на языке ассемблера.

> **Вопрос 98** Что такое прерывание?

> **Вопрос 99** Что такое IDT?

> **Вопрос 100** Что меняет установка IF?

> **Вопрос 101** В какой ситуации возникает ошибка #GP?

> **Вопрос 102** В каких ситуациях возникает ошибка #PF?

> **Вопрос 103** Как #PF связана с подкачкой? Как операционная система её использует?

> **Вопрос 104** Можем ли мы реализовать системные вызовы с помощью прерываний?

Вопрос 105 Зачем нужна отдельная инструкция для реализации системных вызовов?

Вопрос 106 Зачем обработчику прерывания нужно поле DPL?

Вопрос 107 Каково назначение таблиц стеков прерываний?

Вопрос 108 Есть ли у однопоточного приложения только один стек?

Вопрос 109 Какие механизмы ввода/вывода предоставляет Intel 64?

Вопрос 110 Что такое модельно-специфичный регистр?

Вопрос 111 Что такое теневые регистры?

Вопрос 112 Как модельно-специфичные регистры используются в механизме системных вызовов?

Вопрос 113 Какие регистры используются инструкцией `syscall` ?

Модели вычислений

В этой главе мы изучим две модели вычислений: конечные автоматы и стековые машины. *Модель вычислений* сродни языку, который вы используете для описания решения задачи. Как правило, задача, которую действительно трудно корректно решить в одной модели вычислений, может быть почти тривиальной в другой. Именно поэтому программисты, знакомые со многими разными моделями вычислений, могут быть более продуктивными. Они решают задачи в наиболее подходящей модели вычислений, а затем реализуют решение с помощью доступных им инструментов. Когда вы пытаетесь освоить новую модель вычислений, не думайте о ней со «старой» точки зрения, например, не пытайтесь думать о конечных автоматах в терминах переменных и присваиваний. Попробуйте начать с чистого листа и логически выстроить новую систему понятий. Мы уже многое знаем об Intel 64 и его модели вычислений, произошедшей от модели фон Неймана. В этой главе будут представлены конечные автоматы (используемые для реализации регулярных выражений) и стековые машины наподобие машины Форта.

7.1 Конечные автоматы

7.1.1 Определение

Детерминированный конечный автомат (deterministic finite automaton) — это абстрактная машина, действующая на входную строку по некоторым правилам. Мы будем использовать термины «конечный автомат» и «автомат состояний» как взаимозаменяемые. Чтобы определить конечный автомат, нужно задать следующие части:

1. Множество состояний.
2. Алфавит — набор символов, которые могут встречаться во входной строке.
3. Выбранное начальное состояние.
4. Одно или несколько выбранных конечных состояний
5. Правила перехода между состояниями. Каждое правило потребляет символ из входной строки. Его действие можно описать так: «если автомат находится в состоянии *S* и встречается входной символ *C*, следующим текущим состоянием будет *Z*».

Если для текущего состояния нет правила для текущего входного символа, мы считаем поведение автомата неопределённым. *Неопределённое поведение* — понятие, более знакомое математикам, чем инженерам. Ради краткости мы описываем только «хорошие» случаи. «Плохие» случаи нам не интересны, поэтому мы не определяем поведение машины в них. Однако при реализации таких машин мы будем считать все неопределённые случаи ошибочными и ведущими в специальное состояние ошибки.

Зачем вообще возиться с автоматами? Некоторые задачи особенно легко решаются при применении такой парадигмы мышления. К таким задачам относится управление встраиваемыми устройствами и поиск подстрок, соответствующих определённому шаблону. Например, мы проверяем, можно ли интерпретировать строку как целое число. Давайте нарисуем диаграмму, показанную на рисунке 7-1. Она определяет несколько состояний и показывает возможные переходы между ними.

- Алфавит состоит из букв, пробелов, цифр и знаков препинания.
- Множество состояний — *{A, B, C}*.
- Начальное состояние — *A*.
- Конечное состояние — *C*.

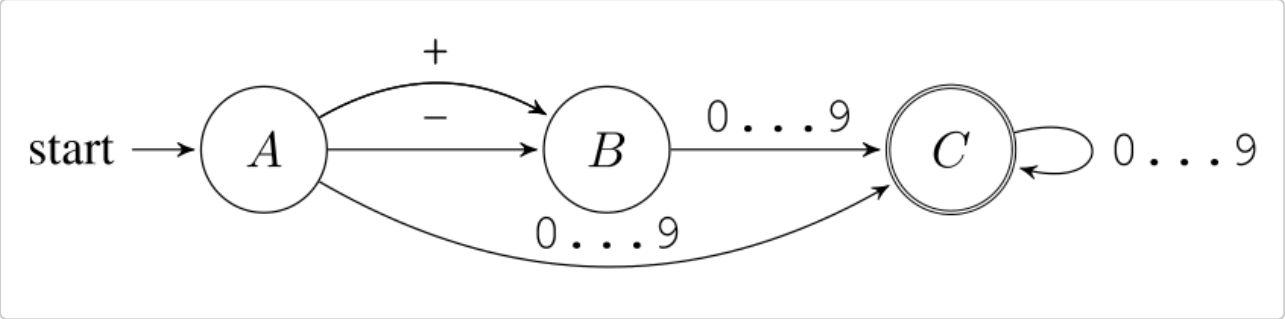


Рисунок 7-1. Распознавание чисел

Мы начинаем выполнение с состояния *A*. Каждый входной символ заставляет нас менять текущее состояние на основе доступных переходов.

> **Примечание** Стрелки, помеченные диапазонами символов вроде 0...9, на самом деле обозначают несколько правил. Каждое из этих правил описывает переход для одного входного символа.

Таблица 7-1 показывает, что произойдёт при выполнении этой машины с входной строкой `+34`. Это называется *трассировкой выполнения*.

Таблица 7-1. Трассировка конечного автомата, показанного на рисунке 7-1, вход: `+34`

СТАРОЕ СОСТОЯНИЕ	ПРАВИЛО	НОВОЕ СОСТОЯНИЕ
A	+	B
B	3	C
C	4	C

Машина пришла в конечное состояние *C*. Однако, получив на вход `idkfa`, мы не смогли бы прийти ни в какое состояние, потому что нет правил для реакции на такие входные символы. Именно здесь поведение автомата неопределено. Чтобы сделать его тотальным и всегда приходить либо в состояние *да*, либо в состояние *нет*, нам нужно добавить ещё одно конечное состояние и добавить правила во всех существующих состояниях. Эти правила должны направлять выполнение в новое состояние в случае, если ни одно старое правило не соответствует входному символу.

7.1.2 Пример: чётность битов

Нам дана строка из *нулей* и *единиц*. Мы хотим узнать, чётное или нечётное количество *единиц*. Рисунок 7-2 показывает решение в виде конечного автомата.

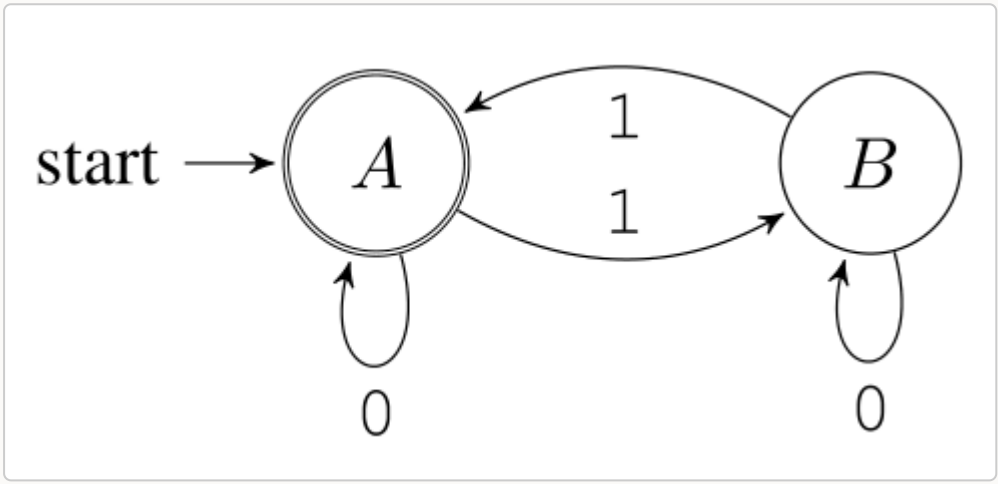


Рисунок 7-2. Чётно ли количество единиц во входной строке?

У пустой строки ноль *единиц*; ноль — чётное число. Из-за этого состояние *A* одновременно является и начальным, и конечным. Все *нули* игнорируются независимо от состояния. Однако каждая *единица*, встречающаяся во входных данных, меняет состояние на противоположное. Если, получив входную строку, мы приходим в конечное состояние *A*, то количество *единиц* чётное. Если мы приходим в конечное состояние *B*, то оно нечётное.

> **Путаница** В конечных автоматах нет памяти, нет присваиваний, нет конструкций if-then-else. Таким образом, это совершенно другая абстрактная машина по сравнению с машиной фон Неймана. Здесь действительно нет ничего, кроме состояний и переходов между ними. В модели фон Неймана состояние — это состояние памяти и значений регистров.

7.1.3 Реализация на языке ассемблера

После проектирования конечного автомата для решения конкретной задачи реализовать эту машину на императивном языке программирования, таком как ассемблер или C, тривиально. Ниже приведён прямолинейный способ реализации таких машин на ассемблере:

1. Сделайте спроектированный автомат тотальным: у каждого состояния должны быть правила перехода для *любого* возможного входного символа. Если это не так, добавьте отдельное состояние, чтобы обозначить ошибку или ответ «нет» на решаемую задачу. Для простоты мы будем называть его *правилом-иначе*.
2. Реализуйте процедуру получения входного символа. Помните, что символ не обязательно является буквой: это может быть сетевой пакет, действие пользователя и другие виды глобальных событий.
3. Для каждого состояния нужно

- Создать метку.
- Вызвать процедуру чтения ввода.
- Сравнить входной символ с описанными в правилах перехода и перейти к соответствующим состояниям, если они совпадают.
- Обработать все остальные символы с помощью *правила-иначе*.

Чтобы реализовать пример автомата на ассемблере, мы сначала сделаем его тотальным, как показано на рисунке 7-3

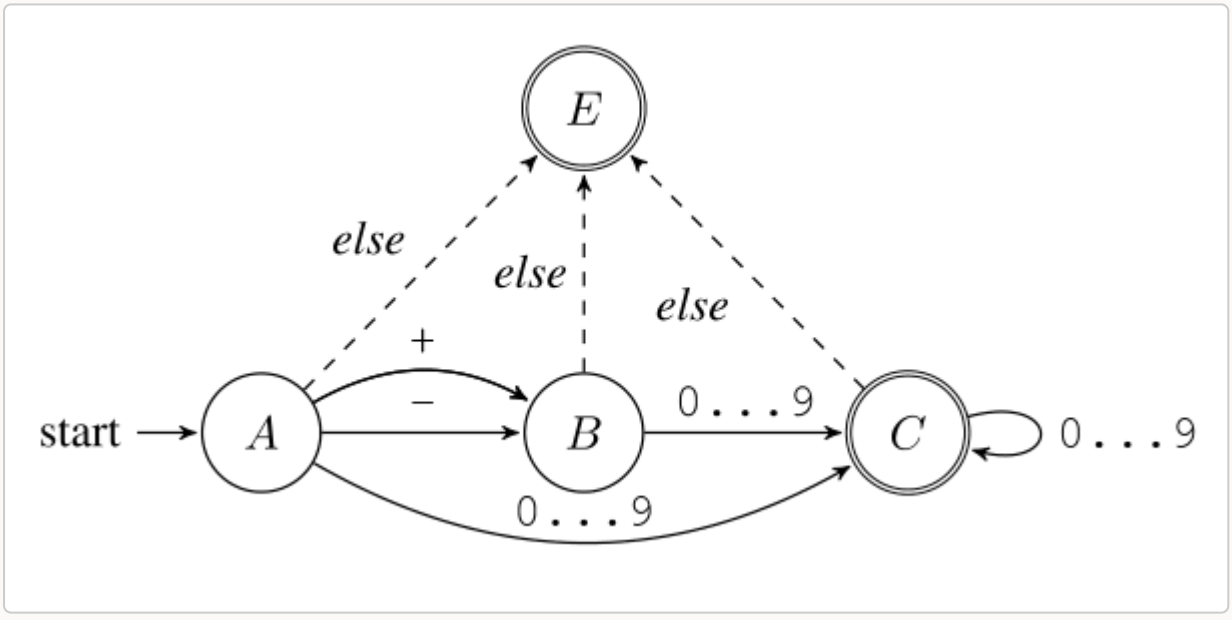


Рисунок 7-3. Проверка, является ли строка числом: тотальный автомат

Мы немного изменим этот автомат, чтобы заставить входную строку завершаться нулём, как показано на рисунке 7-4. Листинг 7-1 показывает пример реализации.

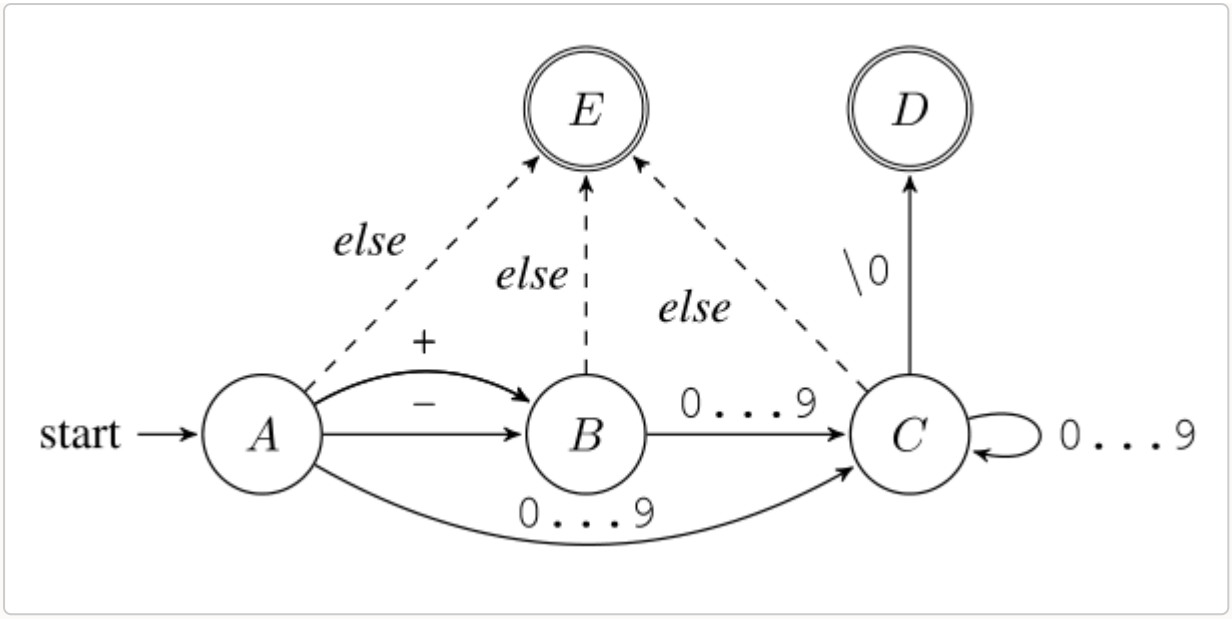


Рисунок 7-4. Проверка, является ли строка числом: тотальный автомат для строки, завершающейся нулём

Листинг 7-1. `automaton_example_bits.asm`

```
section .text
; getsymbol is a routine to
; read a symbol (e.g. from stdin)
; into al
_A:
    call getsymbol
    cmp al, '+'
    je _B
    cmp al, '-'
    je _B
; The indices of the digit characters in ASCII
; tables fill a range from '0' = 0x30 to '9' = 0x39
; This logic implements the transitions to labels
; _E and _C
    cmp al, '0'
    jb _E
    cmp al, '9'
    ja _E
    jmp _C
_B:
    call getsymbol
    cmp al, '0'
    jb _E
    cmp al, '9'
    ja _E
    jmp _C
_C:
    call getsymbol
    cmp al, '0'
    jb _E
    cmp al, '9'
    ja _E
    test al, al
    jz _D
    jmp _C
_D:
; code to notify about success
_E:
; code to notify about failure
```

Этот автомат приходит в состояние *D* или *E;* управление будет передано инструкциям на метке `_D` либо `_E`. Код можно изолировать внутри функции, возвращающей либо 1 (истина) в состоянии `_D`, либо 0 (ложь) в состоянии `_E`.

7.1.4 Практическая ценность

Прежде всего, есть важное ограничение: не все программы можно закодировать в виде конечных автоматов. Эта модель вычислений не является полной по Тьюрингу, она не может анализировать сложные рекурсивно построенные тексты, такие как XML-код. C и язык ассемблера полны по Тьюрингу, а значит, они более выразительны и могут использоваться для решения более широкого круга задач. Например, если длина строки не ограничена, мы не можем подсчитать её длину или количество слов в ней. Каждый результат был бы состоянием, а в конечных автоматах есть лишь ограниченное число состояний, тогда как количество слов может быть произвольно большим, как и сами строки.

> **Вопрос 114** Нарисуйте конечный автомат для подсчёта слов во входной строке. Длина входных данных не более восьми символов.

Конечные автоматы часто используются для описания встраиваемых систем, таких как кофемашины. Алфавит состоит из событий (нажатых кнопок); вход — последовательность действий пользователя. Сетевые протоколы тоже часто можно описать как конечные автоматы. Каждое правило можно снабдить необязательным выходным действием: «если прочитан символ *X*, изменить состояние на *Y* и вывести символ *Z*». Вход состоит из полученных пакетов и глобальных событий, таких как тайм-ауты; выход — последовательность отправленных пакетов. Есть также несколько техник верификации, таких как model checking (проверка модели), позволяющих доказать определённые свойства конечных автоматов — например, «если автомат достиг состояния *B*, он никогда не достигнет состояния *C*». Такие доказательства могут быть очень ценными при построении систем, требующих высокой надёжности.

> **Вопрос 115** Нарисуйте конечный автомат для проверки, чётное или нечётное число слов во входной строке.

> **Вопрос 116** Нарисуйте и реализуйте конечный автомат, отвечающий на вопрос, нужно ли обрезать строку слева, справа, с обеих сторон или не нужно вовсе. Строку нужно обрезать, если она начинается или заканчивается подряд идущими пробелами.

7.1.5 Регулярные выражения

Регулярные выражения — это способ закодировать конечные автоматы. Их часто используют для определения текстовых шаблонов для сопоставления. Их можно использовать для поиска вхождений определённого шаблона или для их замены. Ваш любимый текстовый редактор, вероятно, уже их реализует. Существует множество диалектов регулярных выражений. В качестве примера мы возьмём диалект, похожий на используемый в утилите `egrep`. Регулярное выражение `R` может быть:

1. Буквой.
2. Последовательностью двух регулярных выражений: `R Q`.
3. Метасимволами `^` и `$`, соответствующими началу и концу строки.
4. Парой группирующих скобок с регулярным выражением внутри: `(R)`.
5. Выражением ИЛИ: `R | Q`.
6. `R*` обозначает ноль или более повторений *R*.
7. `R+` обозначает одно или более повторений `R`.
8. `R?` обозначает ноль или одно повторение `R`.
9. Точка соответствует любому символу.
10. Квадратные скобки обозначают диапазон символов, например `[0-9]` эквивалентен

```
(0|1|2|3|4|5|6|7|8|9).
```

Вы можете протестировать регулярные выражения с помощью утилиты `egrep`. Она обрабатывает свой стандартный ввод и отфильтровывает только те строки, что соответствуют заданному шаблону. Чтобы предотвратить его обработку оболочкой, заключите его в одинарные кавычки вот так: `egrep 'expression'`. Ниже несколько примеров простых регулярных выражений:

- `hello .+` соответствует `hello Frank` или hello 12; не соответствует `hello`.
- `[0-9]+` соответствует беззнаковому целому числу, возможно начинающемуся с нулей.
- `-?[0-9]+` соответствует возможно отрицательному целому числу, возможно начинающемуся с нулей.
- `0|(-?[1-9][0-9]*)` соответствует любому целому числу, не начинающемуся с нуля (если только

это не сам ноль). Эти правила позволяют нам определять сложные шаблоны поиска. Движок регулярных выражений будет пытаться сопоставить шаблон, начиная с каждой позиции в тексте. Движки регулярных выражений обычно следуют одному из двух подходов:

- Прямолинейный подход, пытающийся сопоставить все описанные последовательности символов. Например, при сопоставлении строки `ab` с регулярным выражением `aa?a?b` может получиться такая последовательность событий: 1. Попытка сопоставить с `aaab` — неудача. 2. Попытка сопоставить с `aab` — неудача. 3. Попытка сопоставить с `ab` — успех. То есть мы перебираем разные ветви решений, пока не наткнёмся на успешную либо не убедимся окончательно, что все варианты ведут к неудаче. Этот подход обычно довольно быстрый и к тому же прост в реализации. Однако есть худший случай, в котором сложность начинает расти экспоненциально. Представьте сопоставление строки: `aaa...a` (повторить `a` *n* раз) с регулярным выражением: `a?a?a?...a?aaa...a` (повторить `a?` *n* раз, затем повторить a *n* раз) Данная строка точно будет соответствовать регулярному выражению. Однако при прямолинейном подходе движку придётся перебрать все возможные строки, которые действительно соответствуют этому регулярному выражению. Для этого он будет рассматривать два возможных варианта для каждого выражения `a?`, а именно те, что содержат `a`, и те, что его не содержат. Таких строк будет 2. Это столько же, сколько подмножеств у множества из *n* элементов. Вам не нужно больше символов, чем в этой строке текста, чтобы написать регулярное выражение, которое современный компьютер будет вычислять днями или даже годами. Даже для длины $n = 50$ число вариантов достигнет $2^{50} = 1125899906842624$ вариантов. Такие регулярные выражения называются «патологическими», потому что из-за природы алгоритма сопоставления они обрабатываются крайне медленно.
- Построение конечного автомата на основе регулярного выражения. Обычно это *НКА (недетерминированный конечный автомат, NFA)*. В отличие от ДКА (детерминированного конечного автомата, DFA), у них может быть несколько правил для одного и того же состояния и входного символа. Когда возникает такая ситуация, автомат выполняет оба перехода и теперь находится в нескольких состояниях одновременно. Другими словами, есть не одно состояние, а множество состояний, в которых находится автомат. Этот подход в целом немного медленнее, но не имеет худшего случая с экспоненциальным временем работы. Стандартные утилиты Unix, такие как `grep`, используют именно этот подход. Как построить НКА из регулярного выражения? Правила довольно прямолинейны:
- Символу соответствует автомат, принимающий строку из одного такого символа, как показано на рисунке 7-5.
- Мы можем расширить алфавит дополнительными символами, которые помещаем в начало и конец каждой строки.

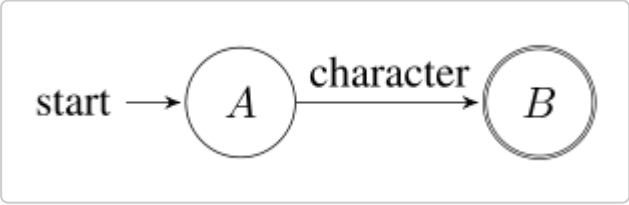


Рисунок 7-5. НКА для одного символа

- Таким образом мы обрабатываем `^` и `$` точно так же, как любой другой символ.
- Группирующие скобки позволяют применять правила к группам символов. Они используются только для корректного разбора регулярного выражения. Другими словами, они предоставляют структурную информацию, нужную для правильного построения автомата.
- ИЛИ соответствует объединению двух НКА путём слияния их начальных состояний. Рисунок 7-5 иллюстрирует эту идею.

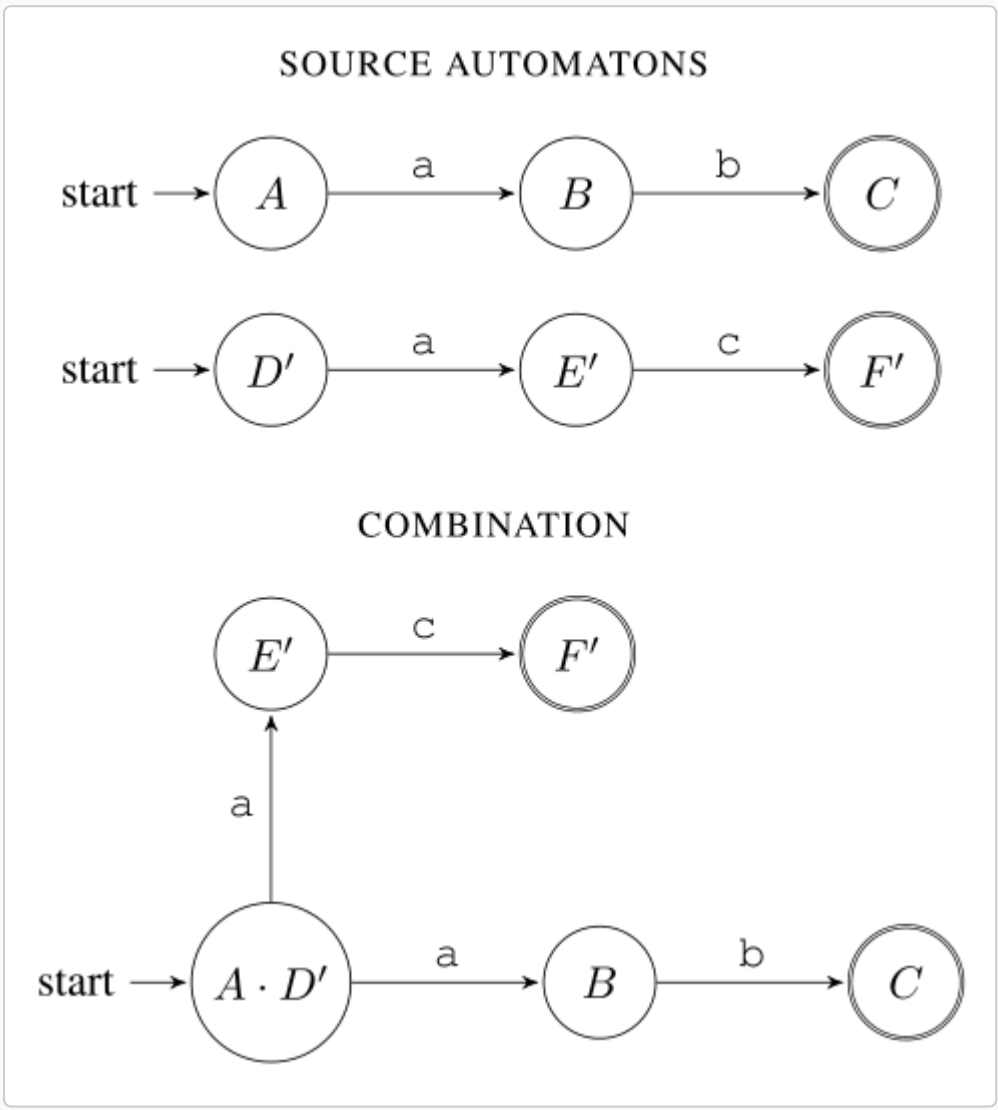


Рисунок 7-6. Объединение НКА через ИЛИ

- У звёздочки есть переход в саму себя и особая вещь, называемая ϵ-правилом. Это правило срабатывает всегда. Рисунок 7-7 показывает автомат для выражения `a*b`.

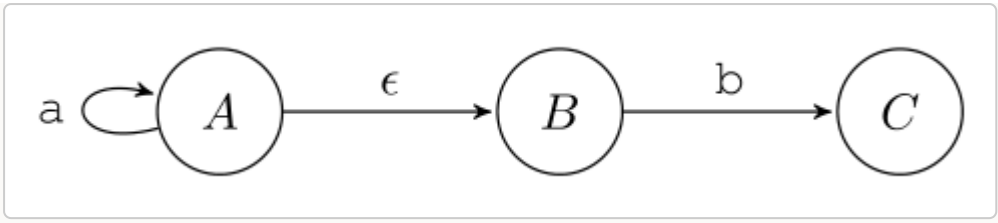


Рисунок 7-7. НКА: реализация звёздочки

- `?` реализуется похожим образом на `*`. `R+` кодируется как `RR*`.

> **Вопрос 117** На любом известном вам языке реализуйте аналог `grep` на основе построения **НКА**. Дополнительную информацию можно найти в [11].

> **Вопрос 118** Изучите это регулярное выражение: `^1?$|^(11+?)\1+$`. В чём может быть его смысл? Представьте, что на вход подаётся строка, состоящая исключительно из символов `1`. Как результат сопоставления этого регулярного выражения соотносится с длиной строки?

7.2 Машина Форта

Форт — язык, созданный Чарльзом Муром в 1971 году для 11-метрового радиотелескопа, эксплуатируемого Национальной радиоастрономической обсерваторией (NRAO) на пике Китт-Пик в Аризоне. Эта система работала на двух ранних мини-компьютерах, соединённых последовательной линией связи. Будучи одновременно мультипрограммной и мультипроцессорной системой (в том смысле, что оба компьютера разделяли ответственность за управление телескопом и его научными приборами), она управляла телескопом, собирала данные и поддерживала интерактивный графический терминал для взаимодействия с телескопом и анализа записанных данных. Сегодня Форт остаётся уникальным и интересным языком, одновременно увлекательным для изучения и отлично меняющим взгляд на программирование. Он всё ещё используется, в основном во встраиваемом ПО, благодаря удивительному уровню интерактивности. Форт также может быть весьма эффективным. Интерпретаторы Форта можно встретить в таких местах, как

- загрузчик FreeBSD.
- прошивки роботов.
- встраиваемое ПО (принтеры).
- ПО космических кораблей.

Так что Форт вполне можно назвать языком системного программирования. Реализовать интерпретатор Форта *и компилятор* для Intel 64 на языке ассемблера не так уж и сложно. Остаток этой главы объяснит детали. Диалектов Форта почти столько же, сколько и программистов на Форте; мы будем использовать собственный простой диалект.

7.2.1 Архитектура

Начнём с изучения абстрактной машины Форта. Она состоит из процессора, двух отдельных стеков — для данных и для адресов возврата — и линейной памяти, как показано на рисунке 7-8.

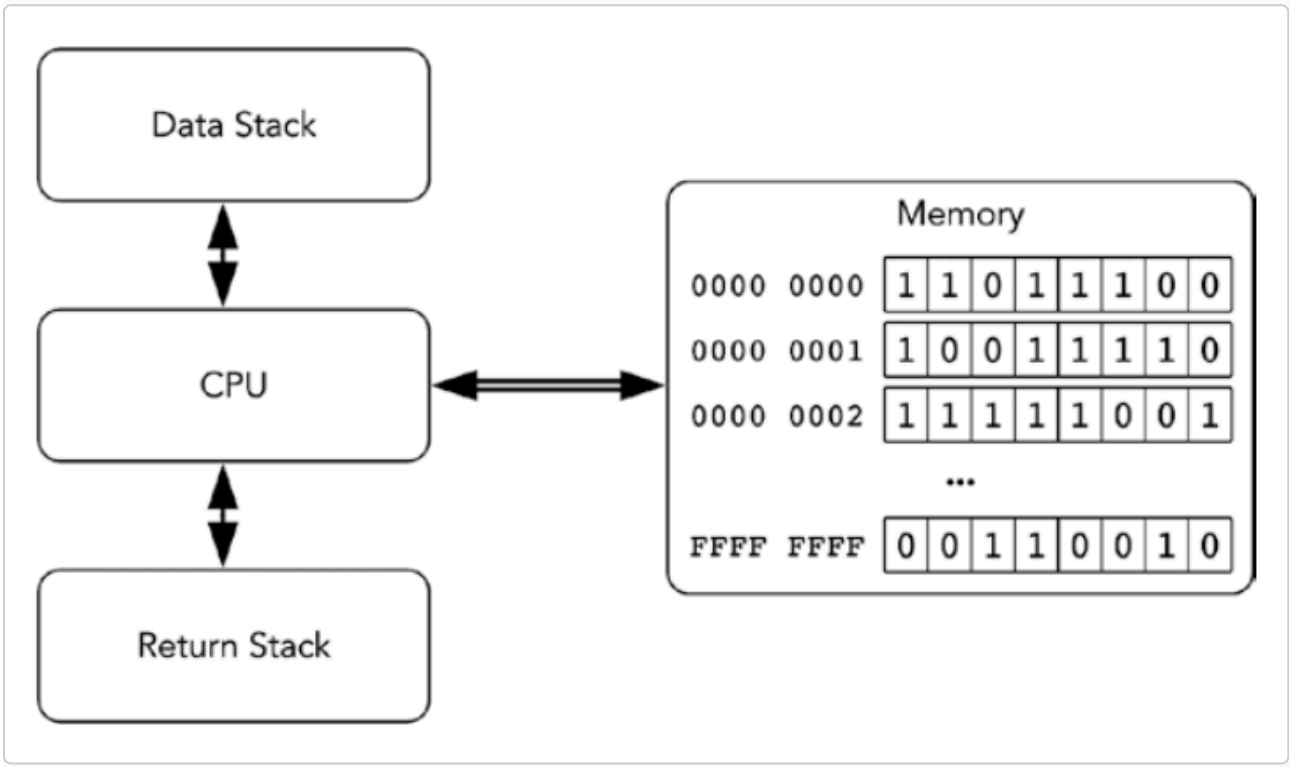


Рисунок 7-8. Машина Форта: архитектура

Стеки не обязательно должны быть частью одного и того же адресного пространства памяти. У машины Форта есть параметр, называемый размером ячейки. Обычно он равен размеру машинного слова целевой архитектуры. В нашем случае размер ячейки — 8 байт. Стек состоит из элементов одинакового размера. Программы состоят из *слов*, разделённых пробелами или переводами строк. Слова выполняются последовательно. Целочисленные слова обозначают помещение в стек данных. Например, чтобы поместить числа 42, 13 и 9 в стек данных, можно просто написать `42 13 9`. Есть три типа слов:

1. Целочисленные слова, описанные выше.
2. Нативные слова, написанные на ассемблере.

3. Слова-двоеточия, написанные на Форте как последовательность других слов Форта.

Стек возврата нужен, чтобы иметь возможность вернуться из слов-двоеточий, как мы увидим позже. Большинство слов оперируют стеком данных. Отныне, говоря о стеке в Форте, мы будем неявно подразумевать стек данных, если не указано иное. Слова берут свои аргументы из стека и помещают туда результат. Все инструкции, работающие со стеком, потребляют свои операнды. Например, слова `+`, `-`, `*` и `/` потребляют два операнда из стека, выполняют арифметическую операцию и помещают её результат обратно в стек. Программа `1 4 8 8 + * +` вычисляет выражение (8 + 8) `*` 4 + 1. Мы будем следовать соглашению, что второй операнд снимается со стека первым. Это означает, что программа `'1 2 -'` вычисляется в *−*1, а не в 1. Слово `:` используется для определения новых слов. За ним следует имя нового слова и список других слов, завершающийся словом `;`. И точка с запятой, и двоеточие сами по себе являются словами, и потому должны отделяться пробелами. Слово `sq`, которое берёт аргумент из стека и кладёт обратно его квадрат, будет выглядеть так:

```
: sq dup * ;
```

Каждый раз, когда мы используем `sq` в программе, будут выполнены два слова: `dup` (дублировать ячейку на вершине стека) и `*` (перемножить два слова на вершине стека). Чтобы описать действия слова, в Форте принято использовать *диаграммы стека*:

```
swap (a b -- b a)
```

В скобках вы видите состояние стека до и после выполнения слова. Ячейки стека именованы, чтобы подчеркнуть изменения в содержимом стека. Так, слово `swap` меняет местами два верхних элемента стека. *Верхний элемент — справа*, так что диаграмма `1 2` соответствует тому, что Форт сначала поместил 1, а затем 2 в результате выполнения каких-то слов. `rot` помещает наверх третье число из стека:

```
rot    (a b c -- b c a)
```

7.2.2 Трассировка примера программы на Форте

Листинг 7-2 показывает простую программу для вычисления дискриминанта квадратного уравнения $1x^2 + 2x + 3 = 0$.

Листинг 7-2. `forth_discr`

```
: sq dup * ;
: discr rot 4 * * swap sq swap - ;
1 2 3 discr
```

Теперь мы выполним `discr a b c` пошагово для некоторых чисел *a*, *b* и *c*. Состояние стека в конце каждого шага показано справа.

```
a    ( a )
b    ( a b )
c    ( a b c )
```

Затем выполняется слово `discr`. Мы заходим внутрь него.

```
rot  ( b c a )
4    ( b c a 4 )
*    ( b c (a*4) )
*    ( b (c*a*4) )
swap ( (c*a*4) b )
sq   ( (c*a*4) (b*b) )
swap ( (b*b) (c*a*4) )
-    ( (b*b - c*a*4) )
```

Теперь проделаем то же самое с начала, но для *a* = 1, *b* = 2 и *c* = 3.

```
1    ( 1 )
2    ( 1 2 )
3    ( 1 2 3 )
rot  ( 2 3 1 )
4    ( 2 3 1 4 )
*    ( 2 3 4 )
*    ( 2 12 )
swap ( 12 2 )
sq   ( 12 4 )
swap ( 4 12 )
-    ( -8 )
```

7.2.3 Словарь

Словарь — это часть машины Форта, хранящая определения слов. Каждое слово — это заголовок, за которым следует последовательность других слов. Заголовок хранит связь с предыдущим словом (как в связных списках), само имя слова в виде строки, завершающейся нулём, и некоторые флаги. Мы уже изучали похожую структуру данных в задании, описанном в разделе 5.4. Вы можете повторно использовать значительную часть этого кода, чтобы упростить определение новых слов Форта. См. рисунок 7-9 с заголовком слова, сгенерированным для слова `discr`, описанного в разделе 7.2.2

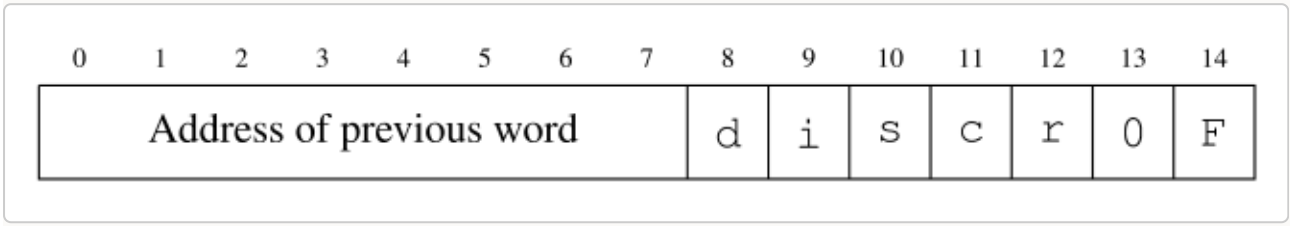


Рисунок 7-9. Заголовок слова для `discr`

7.2.4 Как реализуются слова

Есть три способа реализации слов.

- Косвенно-связный код (indirect threaded code)
- Прямо-связный код (direct threaded code)
- Код на основе подпрограмм (subroutine threaded code)

Мы используем классический способ косвенно-связного кода. Этому типу кода нужны две специальные ячейки (которые мы можем назвать регистрами Форта):

PC указывает на следующую команду Форта. Скоро мы увидим, что команда Форта — это адрес адреса кода реализации соответствующего слова на ассемблере. Другими словами, это указатель на исполняемый код ассемблера с двумя уровнями косвенности.

W используется в нативных словах не напрямую. Когда слово начинает выполняться, этот регистр указывает на его первое слово. Эти два регистра можно реализовать через использование настоящих регистров. Либо же их содержимое можно хранить в памяти. Рисунок 7-10 показывает, как устроены слова при использовании техники косвенно-связного кода. Он включает два слова: нативное слово `dup` и слово-двоеточие `square`.

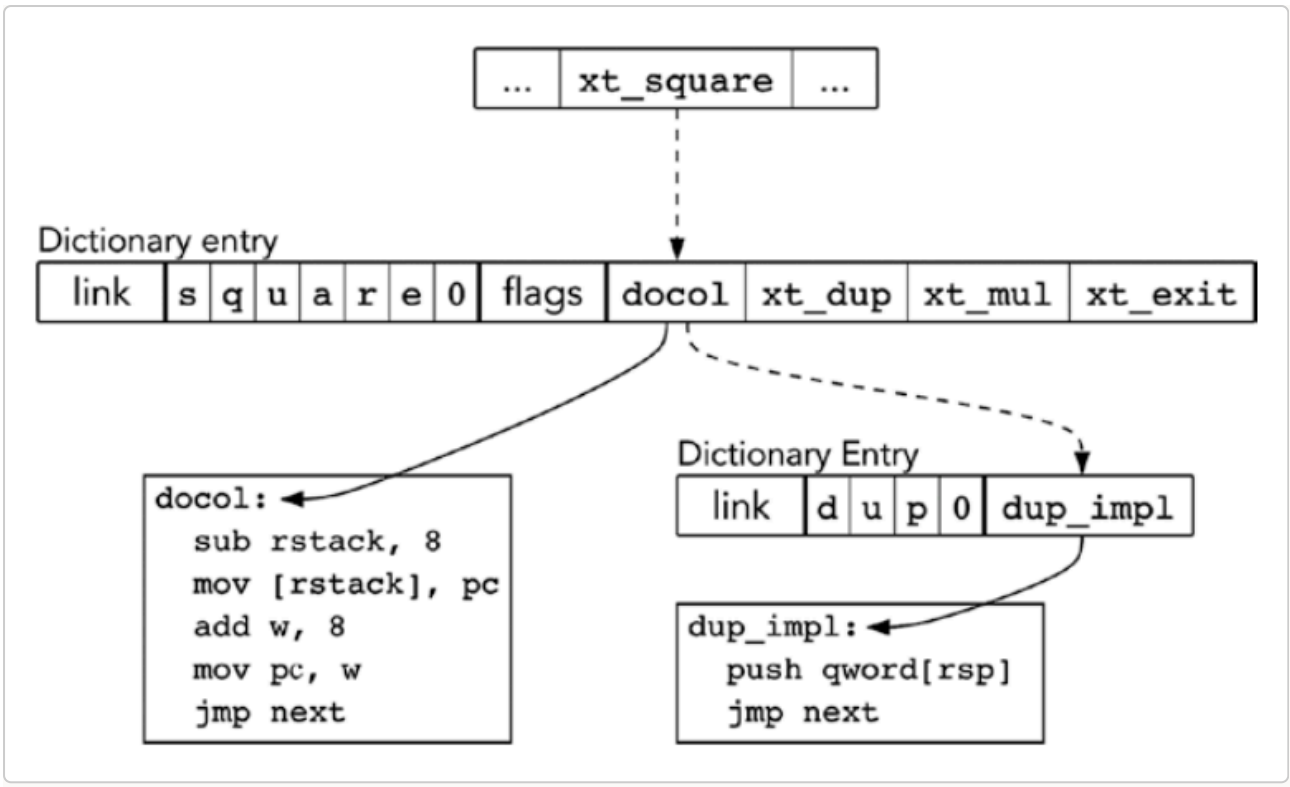


Рисунок 7-10. Косвенно-связный код

Каждое слово хранит адрес своей нативной реализации (код на ассемблере) сразу после заголовка. Для слов-двоеточий реализация всегда одна и та же: `docol`. Реализация вызывается с помощью инструкции `jmp`. *Токен выполнения (execution token)* — это адрес этой ячейки, указывающей на реализацию. Таким образом, токен выполнения — это адрес адреса реализации слова. Другими словами, зная адрес *A* записи слова в словаре, вы можете получить его токен выполнения, просто прибавив к *A* общий размер заголовка. Листинг 7-3 даёт нам пример словаря. Он содержит два нативных слова (начинающихся с `w_plus` и `w_dup`) и слово-двоеточие (`w_sq`).

Листинг 7-3. `forth_dict_sample.asm`

```
section .data
w_plus:
    dq 0        ; The first word's pointer to the previous word is zero
    db '+',0
    db 0        ; No flags
xt_plus:        ; Execution token for `plus`, equal to
                ; the address of its implementation
    dq plus_impl
w_dup:
    dq w_plus
    db 'dup', 0
    db 0
xt_dup:
    dq dup_impl
w_double:
    dq w_dup
    db 'double', 0
    db 0
    dq docol     ; The `docol` address -- one level of indirection
    dq xt_dup    ; The words consisting `dup` start here.
    dq xt_plus
    dq xt_exit
last_word: dq w_double
section .text
    plus_impl:
        pop rax
        add rax, [rsp]
        mov [rsp], rax
        jmp next
    dup_impl:
        push qword [rsp]
        jmp next
```

Ядро движка Форта — *внутренний интерпретатор*. Это простая процедура на ассемблере, извлекающая код из памяти. Она показана в листинге 7-4.

Листинг 7-4. `forth_next.asm`

```
next:
     mov w, pc
     add pc, 8 ; the cell size is 8 bytes
     mov w, [w]
     jmp [w]
```

Она делает две вещи:

1. Она читает память, начиная с *PC*, и настраивает *PC* на следующую инструкцию. Помните,

что PC указывает на ячейку памяти, хранящую токен выполнения слова.

2. Она настраивает *W* на значение токена выполнения. Другими словами, после выполнения `next`, *W*

хранит адрес указателя на ассемблерную реализацию слова.

3. Наконец, она переходит к коду реализации.

Каждая реализация нативного слова заканчивается инструкцией `jmp next`. Это гарантирует, что будет извлечена следующая инструкция. Чтобы реализовать слова-двоеточия, нам нужно использовать стек возврата, чтобы сохранять и восстанавливать PC до и после вызова. Хотя *W* не нужен при выполнении нативных слов, для слов-двоеточий он весьма важен. Давайте взглянем на `docol`, реализацию всех слов-двоеточий, показанную в листинге 7-5. Там же есть `exit`, ещё одно слово, предназначенное для завершения всех слов-двоеточий.

Листинг 7-5. `forth_docol.asm`

```
docol:
   sub rstack, 8
   mov [rstack], pc
   add w, 8       ;     8
   mov pc, w
   jmp next
exit:
   mov pc, [rstack]
   add rstack, 8
   jmp next
```

`docol` сохраняет *PC* в стеке возврата и устанавливает новый *PC* на первый токен выполнения, хранящийся внутри текущего слова. Возврат выполняется словом `exit`, которое восстанавливает *PC* из стека. Этот механизм похож на пару инструкций `call` / `ret`.

> **Вопрос 119** Прочитайте [32]. В чём разница между нашим подходом (косвенно-связный код) и прямо-связным кодом, а также кодом на основе подпрограмм? Какие преимущества и недостатки вы можете назвать?

Чтобы лучше уловить концепцию косвенно-связного кода и внутреннее устройство Форта, мы подготовили минимальный пример, показанный в листинге 7-6. Он использует процедуры, разработанные в первом задании из раздела 2.7. Не торопясь, запустите его (исходный код поставляется вместе с книгой) и проверьте, что он действительно читает слово из ввода и выводит его обратно.

Листинг 7-6. `itc.asm`

```
%include "lib.inc"
global _start
%define pc r15
%define w r14
%define rstack r13
section .bss
resq 1023
rstack_start: resq 1
input_buf: resb 1024
section .text
; this one cell is the program
main_stub: dq xt_main
; The dictionary starts here
; The first word is shown in full
; Then we omit flags and links between nodes for brevity
; Each word stores an address of its assembly implementation
; Drops the topmost element from the stack
dq 0 ; There is no previous node
db "drop", 0
db 0 ; Flags = 0
xt_drop: dq i_drop
i_drop:
    add rsp, 8
    jmp next
; Initializes registers
xt_init: dq i_init
i_init:
    mov rstack, rstack_start
    mov pc, main_stub
    jmp next
; Saves PC when the colon word starts
xt_docol: dq i_docol
i_docol:
    sub rstack, 8
    mov [rstack], pc
    add w, 8
    mov pc, w
    jmp next
; Returns from the colon word
xt_exit: dq i_exit
i_exit:
    mov pc, [rstack]
    add rstack, 8
```

```
    jmp next
; Takes a buffer pointer from stack
; Reads a word from input and stores it
; starting in the given buffer
xt_word: dq i_word
i_word:
    pop rdi
    call read_word
    push rdx
    jmp next
; Takes a pointer to a string from the stack
; and prints it
xt_prints: dq i_prints
i_prints:
    pop rdi
    call print_string
    jmp next
; Exits program
xt_bye: dq i_bye
i_bye:
    mov rax, 60
    xor rdi, rdi
    syscall
; Loads the predefined buffer address
xt_inbuf: dq i_inbuf
i_inbuf:
    push qword input_buf
    jmp next
; This is a colon word, it stores
; execution tokens. Each token
; corresponds to a Forth word to be
; executed
xt_main: dq i_docol
    dq xt_inbuf
    dq xt_word
    dq xt_drop
    dq xt_inbuf
    dq xt_prints
    dq xt_bye
; The inner interpreter. These three lines
; fetch the next instruction and start its
; execution
next:
    mov w, [pc]
    add pc, 8
    jmp [w]
```

```
; The program starts execution from the init word
_start: jmp i_init
```

7.2.5 Компилятор

Форт может работать либо в режиме интерпретатора, либо в режиме компилятора. Интерпретатор просто читает команды и выполняет их. При выполнении слова-двоеточия `:` Форт переключается в режим компилятора. Кроме того, двоеточие `:` читает следующее слово и использует его, чтобы создать новую запись в словаре с `docol` в качестве реализации. Затем Форт читает слова, находит их в словаре и добавляет их к текущему определяемому слову. Так что нам нужно добавить ещё одну переменную, `here`, хранящую адрес текущей позиции для записи слов в режиме компиляции. Каждая запись будет продвигать `here` на одну ячейку вперёд. Чтобы выйти из режима компилятора, нам нужны специальные *немедленные слова (immediate words)*. Они выполняются независимо от того, в каком режиме мы находимся. Без них мы бы никогда не смогли выйти из режима компилятора. Немедленные слова помечены *флагом немедленности (immediate flag)*. Интерпретатор кладёт числа в стек. Компилятор не может встраивать их в слова напрямую, потому что иначе они будут восприниматься как токены выполнения. Попытка запустить команду по токену выполнения `42`, вероятнее всего, приведёт к ошибке сегментации. Однако решение — использовать специальное слово `lit`, за которым следует само число. Назначение `lit` — прочитать следующее целое число, на которое указывает *PC*, и продвинуть *PC* на одну ячейку вперёд, чтобы *PC* никогда не указывал на встроенный операнд.

7.2.5.1 Условия в Форте

Мы выделим два слова в нашем диалекте Форта: `branch n` и `0branch n`. Они допустимы только в режиме компиляции! Они похожи на `lit n`, потому что смещение хранится сразу после их токена выполнения.

7.3 Задание: компилятор и интерпретатор Форта

В этом разделе описывается большое задание: написать собственный интерпретатор Форта. Прежде чем начать, убедитесь, что вы понимаете основы языка Форт. Если вы не уверены в этом, можете поиграть с любым свободным интерпретатором Форта, таким как gForth.

> **Вопрос 120** Посмотрите документацию по командам `sete`, `setl` и их аналогам.

> **Вопрос 121** Что делает инструкция `cqo`? Обратитесь к [15].

Удобно хранить *PC* и *W* в каких-нибудь регистрах общего назначения, особенно в тех, что гарантированно сохраняются неизменными при вызовах функций (*сохраняемые вызывающей функцией*): `r13`, `r14` или `r15`.

7.3.1 Статический словарь, интерпретатор

Начнём со статического словаря нативных слов. Примените знания, полученные в разделе 5.4. Отныне мы не можем определять новые слова во время выполнения. Для этого задания мы будем использовать следующие определения макросов:

- `native`, принимающий три аргумента:
- Имя слова;
- Часть идентификатора слова; и
- Флаги. Он создаёт и заполняет заголовок в `.data` и метку в `.text`. Эта метка будет обозначать код на ассемблере, следующий за экземпляром макроса. Поскольку большинство слов не будут использовать флаги, мы можем перегрузить `native`, чтобы он принимал либо два, либо три аргумента. Для этого мы создаём похожее определение макроса, принимающее два аргумента и запускающее `native` с тремя аргументами, где третий заменяется на ноль, а первые два передаются как есть, как показано в листинге 7-7.

Листинг 7-7. `native_overloading.asm`

```
%macro native 2
native %1, %2, 0
%endmacro
```

Сравните два способа определения словаря Форта: без макросов (показан в листинге 7-8) и с ними (показан в листинге 7-9).

Листинг 7-8. `forth_dict_example_nomacro.asm`

```
section .data
w_plus:
   dq w_mul ; previous
   db '+',0
   db 0
xt_plus:
   dq plus_impl
section .text
   plus_impl:
      pop rax
      add [rsp], rax
      jmp next
```

Листинг 7-9. forth_dict_example_macro.asm

```
native '+', plus
     pop rax
     add [rsp], rax
     jmp next
```

Затем определите макрос `colon`, аналогичный предыдущему. Листинг 7-10 показывает его использование.

Листинг 7-10. forth_colon_usage.asm

```
colon '>', greater
   dq xt_swap
   dq xt_less
   dq exit
```

Не забывайте про адрес `docol` в каждом слове-двоеточии! Затем создайте и протестируйте следующие процедуры на ассемблере:

- `find_word`, принимающую указатель на строку, завершающуюся нулём, и возвращающую адрес начала заголовка слова. Если слова с таким именем нет, возвращается ноль.
- `cfa` (code from address, код по адресу), принимающую начало заголовка слова и пропускающую весь заголовок, пока не дойдёт до значения *XT*.

Используя эти две функции и те, что вы уже написали в разделе 2.7, вы можете написать цикл интерпретатора. Интерпретатор будет либо помещать число в стек, либо заполнять специальную заглушку, состоящую из двух ячеек, показанную в листинге 7-11. Он должен записать только что найденный токен выполнения в `program_stub`. Затем ему нужно направить *PC* на начало заглушки и перейти к `next`. Это выполнит слово, которое мы только что разобрали, а затем вернёт управление интерпретатору. Помните, что токен выполнения — это просто адрес адреса ассемблерного кода. Именно поэтому вторая ячейка заглушки указывает на третью, а третья хранит адрес интерпретатора — мы просто скармливаем эти данные существующему механизму Форта.

Листинг 7-11. `forth_program_stub.asm`

```
program_stub: dq 0
xt_interpreter:  dq .interpreter
.interpreter: dq interpreter_loop
```

Рисунок 7-11 показывает псевдокод, иллюстрирующий логику интерпретатора.

```
1:  interpreter_loop:
2:  word ← word from stdin
3:  if word is empty then
4:      exit
5:  if word is present in dictionary, its address is addr then
6:      xt ← cfa(addr)
7:      [program_stub] ← xt
8:      PC ← program_stub
9:      goto next
10: else
11:     if word is a number n then
12:         push n
13:     else
14:         Error: unknown word
```

Рисунок 7-11. Интерпретатор Форта: псевдокод

Помните, что у машины Форта также есть память. Мы заранее выделим для неё 65536 ячеек Форта.

> **Вопрос 122** Стоит ли выделять эти ячейки в секции `.data`, или есть варианты получше?

Чтобы дать Форту знать, где находится память, мы создадим слово `mem`, которое будет просто помещать начальный адрес памяти на вершину стека.

7.3.1.1 Список слов

Сначала вам нужно сделать интерпретатор, поддерживающий следующие слова:

- `.S` — выводит всё содержимое стека; не изменяет его. Чтобы это реализовать, сохраните `rsp` перед

началом работы интерпретатора.

- Арифметика: `+ - * /, = <`. Операции сравнения помещают на вершину стека либо 1, либо 0.
- Логика: `and`, `not`. Все ненулевые значения считаются истиной; нулевое значение считается ложью.

В случае успеха эти инструкции помещают 1, иначе 0. Они также уничтожают свои операнды.

- Простые манипуляции со стеком:

```
rot (a b c -- b c a)
swap (a b -- b a)
dup (a -- a a)
drop (a -- )
```

- `. ( a -- )` снимает число со стека и выводит его.
- Ввод/вывод: `key ( -- c)` — читает один символ из `stdin`; верхняя ячейка стека хранит 8 байт — это код символа, расширенный нулями. `emit ( c -- )` — записывает один символ в `stdout.` `number ( -- n )` — читает знаковое целое число из `stdin` (гарантированно помещается в одну ячейку).
- mem — сохраняет начальный адрес пользовательской памяти на вершине стека.
- Работа с памятью: `! (address data -- )` — сохраняет данные из стека начиная с адреса. `c! (address char -- )` — сохраняет один байт по адресу. `@ (address -- value)` — читает одну ячейку начиная с адреса `c@ (address -- charvalue)` — читает один байт начиная с адреса Затем протестируйте получившийся интерпретатор.

Затем создайте область памяти для стека возврата и реализуйте `docol` и `exit`. Мы рекомендуем выделить регистр, указывающий на вершину стека возврата. Реализуйте слова-двоеточия `or` и `greater`, используя макрос `colon`, и протестируйте их.

7.3.2 Компиляция

Теперь реализуем компиляцию. Это легко!

1. Нам нужно выделить ещё 65536 ячеек Форта для расширяемой части словаря.
2. Добавьте переменную `state`, равную 1 в режиме компиляции, 0 в режиме интерпретации.
3. Добавьте переменную `here`, указывающую на первую свободную ячейку в предварительно выделенном пространстве словаря.
4. Добавьте переменную `last_word`, хранящую адрес последнего определённого слова.
5. Добавьте два новых слова-двоеточия, а именно `:` и `;`. Двоеточие: 1: *word* ← `stdin` 2: Заполнить заголовок нового слова начиная с `here`. Не забудьте его обновить! 3: Добавить адрес `docol` сразу по `here`; обновить `here`. 4: Обновить `last_word`. 5: *state* ← 1; 6: Перейти к `next`.

Точка с запятой должна быть помечена как *Immediate (немедленная)*!

1: `here` ← XT слова `exit`; обновить `here`.

2: *state* ← 0;

3: Перейти к `next`.

6. Вот как выглядит цикл компилятора. Вы можете реализовать его отдельно или смешать

с уже реализованным циклом интерпретатора.

1: *compilerloop:*

2: *word* ← слово из `stdin`

3: *if*word пусто *then*

4: exit

5: *if*word присутствует и имеет адрес *addrthen*

6: *xt* ← *cf* a(*addr*)

7: *if*word помечено Immediate *then*

8: интерпретировать *word*

9: *else*

10: `[here]` ← xt

11: `here` ← `here` + 8

12: *else*

13: *ifword* является числом *nthen*

14: *if* предыдущее слово было `branch` или `0branch` *then*

```
15: [here]← n
```

16: `here` ← `here` + 8

17: *else*

18: `[here]` ← xt lit

19: `here` ← `here` + 8

```
20: [here]← n
```

21: `here` ← `here` + 8

22: *else*

23: Ошибка: неизвестное слово Реализуйте `0branch` и `branch` и протестируйте их (полный список слов Форта с их значениями см. в разделе 7.3.3).

> **Вопрос 123** Зачем нам нужен отдельный случай для `branch` и `0branch` ?

7.3.3 Форт с bootstrap

Интерпретатор Форта можно разделить на две части. Самая необходимая называется *внутренним интерпретатором*; она написана на ассемблере. Её назначение — извлекать следующий *XT* из памяти. Это процедура `next` , показанная в

Листинге 7-4.

Другая часть — внешний интерпретатор, принимающий пользовательский ввод и либо компилирующий слово в текущее определение, либо выполняющий его сразу же. Интересно то, что этот интерпретатор можно определить как слово-двоеточие. Чтобы этого

добиться, нам нужно определить несколько дополнительных слов Форта. Мы создали Forthress, диалект Форта, описанный в этой главе. Интерпретатор и компилятор также поставляются вместе с этой книгой. Вот полный набор слов, известных Forthress.

- drop(a --)
- swap(a b -- b a)
- dup(a -- a a)
- rot(a b c -- b c a)
- Арифметика:

```
+  ( y  x  -- [ x  +  y  ] )
*  ( y  x  -- [ x  *  y  ] )
/ ( y  x  -- [ x  / y  ] )
%  ( y  x  -- [ x  mod  y  ] )
- ( y x -- [x - y] )
```

- Логика:
- `not( a -- a' )` a’ = 0, если a != 0 a’ = 1, если a == 0
- `=( a b -- c )` c = 1, если a == b c = 0, если a != b
- `count( str -- len )` Принимает строку, завершающуюся нулём, вычисляет её длину.
- `.` Снимает элемент со стека и отправляет его в *stdout*.
- `.S` Показывает содержимое стека. Не снимает элементы.
- `init` Сохраняет базу стека данных. Полезно для *.S*.
- `docol` Это реализация любого слова-двоеточия. Сам XT не используется, но реализация (известная как *docol*) используется.
- `exit` Выход из слова-двоеточия.
- `>r` Переложить из стека возврата в стек данных.
- `r>` Переложить из стека данных в стек возврата.
- `r@` Неразрушающее копирование с вершины стека возврата на вершину стека данных.
- `find( str -- header addr )` Принимает указатель на строку, возвращает указатель на заголовок слова в словаре.
- `cfa( word addr -- xt )` Преобразует начальный адрес заголовка слова в токен выполнения.
- `emit( c -- )` Выводит один символ в `stdout.`
- `word( addr -- len )` Читает слово из *stdin* и сохраняет его начиная с адреса `addr.`

Длина слова помещается в стек.

- `number ( str -- num len)` Разбирает целое число из строки.
- `prints ( addr -- )` Выводит строку, завершающуюся нулём.
- `bye` Выход из Forthress
- syscall (call num a1 a2 a3 a4 a5 a6 -- new rax) Выполняет системный вызов

Следующие регистры хранят аргументы (согласно ABI) `rdi`, `rsi`, `rdx`, `r10`, `r8` и `r9`.

- `branch <offset>` Переход к позиции. Позиция — это смещение относительно конца аргумента

Например:

```
|branch|    24 | <next command>
                ^ branch adds 24 to this address and stores it in PC
```

- `0branch<offset>` Branch — слово, допустимое только при компиляции. Переход к позиции, если TOS = 0.

Позиция вычисляется схожим образом. 0branch — слово, допустимое только при компиляции.

- `lit <value>` Помещает значение, следующее сразу за этим XT.
- `inbuf` Адрес входного буфера (используется интерпретатором/компилятором).
- `mem` Адрес пользовательской памяти.
- `last word` Адрес заголовка последнего слова.
- `state` Адрес ячейки state. Ячейка state хранит либо 1 (режим компиляции), либо 0

(режим интерпретации).

- `here` Указывает на последнюю ячейку слова, определяемого в данный момент.
- `execute ( xt -- )` Выполняет слово с этим токеном выполнения на TOS.
- `@ ( addr -- value )` Извлекает значение из памяти.
- `! ( addr val -- )` Сохраняет значение по адресу.
- `@c ( addr -- char )` Читает один байт начиная с addr.
- `, ( x -- )` Добавляет x к определяемому слову.
- `c, ( c -- )` Добавляет один байт к определяемому слову.
- `create ( flags name -- )` Создаёт запись в словаре с именем, равным новому

имени. На данный момент реализован только флаг immediate.

- `:` Читает слово из `stdin` и начинает его определение.
- `;` Завершает определение текущего слова

- `interpreter` Интерпретатор/компилятор Forthress.

Мы призываем вас попробовать построить собственный самозагружающийся (bootstrapped) Форт. Вы можете начать с рабочего цикла интерпретатора, написанного на Форте. Измените файл `itc.asm`, показанный в листинге 7-6, введя слово `interpreter` и написав его, используя только слова Форта.

7.4 Итоги

Эта глава познакомила нас с двумя новыми моделями вычислений: конечными автоматами (также известными как автоматы состояний) и стековыми машинами наподобие машины Форта. Мы увидели связь между конечными автоматами и регулярными выражениями, используемыми во многих текстовых редакторах и других утилитах обработки текста. Мы завершили первую часть нашего путешествия, построив интерпретатор и компилятор Форта, что мы считаем прекрасным итогом нашего знакомства с языком ассемблера. В следующей главе мы переключимся на язык C, чтобы писать код более высокого уровня. Ваши знания ассемблера послужат фундаментом для понимания C, поскольку его модель вычислений очень близка к классической модели вычислений фон Неймана.

> **Вопрос 124** Что такое модель вычислений?

> **Вопрос 125** Какие модели вычислений вы знаете?

> **Вопрос 126** Что такое конечный автомат?

> **Вопрос 127** Когда полезны конечные автоматы?

> **Вопрос 128** Что такое конечный автомат (finite automaton)?

> **Вопрос 129** Что такое регулярное выражение?

> **Вопрос 130** Как связаны регулярные выражения и конечные автоматы?

> **Вопрос 131** Какова структура абстрактной машины Форта?

Вопрос 132 Какова структура словаря в Форте?

Вопрос 133 Что такое токен выполнения?

Вопрос 134 В чём разница реализации между встроенными словами и словами-двоеточиями?

Вопрос 135 Почему в Форте используются два стека?

Вопрос 136 Какие есть два разных режима работы Форта?

Вопрос 137 Зачем нужен флаг немедленности?

Вопрос 138 Опишите слово-двоеточие и слово-точку-с-запятой.

Вопрос 139 Каково назначение регистров `PC` и `W`?

Вопрос 140 Каково назначение `next`?

Вопрос 141 Каково назначение `docol`?

Вопрос 142 Каково назначение `exit`?

Вопрос 143 Ведут ли себя одинаково интерпретатор и компилятор при встрече целочисленного литерала?

Вопрос 144 Добавьте встроенное слово для проверки остатка от деления двух чисел. Напишите слово для проверки, делится ли одно число на другое.

Вопрос 145 Добавьте встроенное слово для проверки остатка от деления двух чисел. Напишите слово для проверки числа на простоту.

Вопрос 146 Напишите слово Форта для вывода первых n чисел последовательности Фибоначчи.

Вопрос 147 Напишите слово Форта для выполнения системных вызовов (оно будет брать содержимое регистров из стека). Напишите слово, которое выведет «Hello, world!» в `stdout`.

Язык программирования С

Основы

В этой главе мы начнём изучать другой язык под названием C. Это низкоуровневый язык с довольно минимальными абстракциями над ассемблером. В то же время он достаточно выразителен, чтобы проиллюстрировать некоторые очень общие понятия и идеи, применимые ко всем языкам программирования (такие как система типов или полиморфизм). C почти не предоставляет абстракций над памятью, так что задача управления памятью лежит на программисте. В отличие от языков более высокого уровня, таких как C# или Java, программист должен сам выделять и освобождать зарезервированную память, а не полагаться на автоматизированную систему сборки мусора. C — переносимый язык, так что при правильном написании кода его часто можно выполнить на других архитектурах после простой перекомпиляции. Причина в том, что модель вычислений в C — практически всё та же старая модель фон Неймана, что делает её близкой к моделям программирования большинства процессоров. Изучая C, помните, что, несмотря на иллюзию языка более высокого уровня, он не прощает ошибок, и система не всегда будет достаточно любезна, чтобы уведомить вас о том, что в вашей программе что-то сломано. Ошибка может проявиться намного позже, на других входных данных, в совершенно не связанной с ней части программы.

> **Описание стандарта языка** Очень важный документ о языке — стандарт языка C. Вы можете бесплатно получить PDF-файл черновика стандарта онлайн [7]. Этот документ так же важен для нас, как и Intel Software Developer's Manual [15].

8.1 Введение

Прежде чем начать, нам нужно отметить несколько важных моментов.

- C *всегда* регистрозависим*.*
- C не заботится о пробелах, пока парсер способен отделять лексемы друг от друга. Программы, показанные в листинге 8-1 и листинге 8-2, эквивалентны.

Листинг 8-1. `spacing_1.c`

```
int main      (int argc ,   char * * argv)
{
    return 0;
}
```

Листинг 8-2. `spacing_2.c`

```
int main(int argc, char** argv)
{
    return 0;
}
```

- Существуют разные стандарты языка C. Мы не изучаем GNU C (версию,

обладающую различными расширениями), поддерживаемую в основном GCC. Вместо этого мы сосредоточимся на C89 (также известном как ANSI C или C90) и C99, которые поддерживаются многими различными компиляторами. Мы также упомянём несколько новых возможностей C11, некоторые из которых не обязательны для реализации в компиляторах.

К сожалению, C89 всё ещё остаётся наиболее распространённым стандартом, так что компиляторы, поддерживающие C89, есть практически для любой существующей платформы. Именно поэтому мы сначала сосредоточимся на этой конкретной редакции, а затем расширим её более новыми возможностями.

Чтобы заставить компилятор использовать только те возможности, что поддерживаются определённым стандартом, мы используем следующий набор флагов:

- `-std=c89` или `-std=c99` для выбора стандарта C89 или C99.
- `-pedantic-errors` для отключения нестандартных расширений языка.
- `-Wall` для показа всех предупреждений, независимо от их важности.
- `-Werror` для превращения предупреждений в ошибки, чтобы вы не могли скомпилировать код с предупреждениями.

> **Предупреждения — это ошибки** Очень плохая практика — поставлять код, который не компилируется без предупреждений. Предупреждения выдаются неспроста. Иногда бывают очень специфичные случаи, когда людям приходится делать нестандартные вещи, например вызывать функцию с большим числом аргументов, чем она принимает, но такие случаи крайне редки. В таких случаях гораздо лучше отключить один конкретный тип предупреждения для одного конкретного файла с помощью соответствующего ключа компилятора. Иногда директивы компилятора позволяют ему пропускать определённое предупреждение для выбранной области кода, что ещё лучше.

Например, чтобы скомпилировать исполняемый файл `main` из исходных файлов `file1.c` и `file2.c`, можно использовать следующую команду:

```
> gcc -o main -ansi -pedantic-errors -Wall -Werror file1.c file2.c
```

Эта команда выполнит полный проход компиляции, включая создание объектных файлов и компоновку.

8.2 Структура программы

Любая программа на C состоит из

- Определений типов данных (структур, новых типов и т. д.), основанных на других существующих

типах. Например, мы можем создать новое имя `new_int_type_name_t` для целочисленного типа `int`.

```
typedef int new_int_type_name_t;
```

- Глобальных переменных (объявленных вне функций). Например, мы можем создать глобальную переменную `i_am_global` типа `int`, инициализированную значением 42, вне всех областей видимости функций. Заметьте, что глобальные переменные могут быть инициализированы только константными значениями.

```
int i_am_global = 42;
```

- Функций. Например, функции с именем `square`, принимающей аргумент `x` типа `int` и возвращающей его квадрат.

```
int square( int x ) { return x * x; }
```

- Комментариев между `/*` и `*/`.

```
/* this is a rather complex comment
which span over multiple lines */
```

- Комментариев, начинающихся с `//` до конца строки (в C99 и более новых).

```
int x; // this is a one line comment, which ends at the end of the line
```

- Директив препроцессора и компилятора. Они часто начинаются с #.

```
#define CATS_COUNT 42
#define ADD(x, y) (x) + (y)
```

Внутри функций мы можем определять переменные или типы данных, локальные для этой функции, либо выполнять действия. Каждое действие — это *оператор (statement)*; они обычно разделяются точкой с запятой. Действия выполняются последовательно. Нельзя определять функции внутри других функций. Операторы будут объявлять переменные, выполнять вычисления и присваивания, а также выполнять разные ветви кода в зависимости от условий. Особый случай — *блок* между фигурными скобками {}, используемый для группировки операторов. Листинг 8-3 показывает пример программы на C. Она выводит `Hello, world! y=42 x=43`. Она определяет функцию `main`, объявляющую две переменные `x` и `y`, первая из которых равна 43, а вторая вычисляется как значение `x` минус один. Затем выполняется вызов функции `printf`. Функция `printf` используется для вывода строк в *stdout*. У строки есть некоторые части (так называемые *спецификаторы формата*), заменяемые следующими аргументами. Спецификатор формата, как следует из названия, предоставляет информацию о природе аргумента, что обычно включает его размер и наличие знака. Пока мы будем использовать очень немного спецификаторов формата.

- `%d` для аргументов `int`, как в примере*.*
- `%f` для аргументов `float`.

Объявления переменных, присваивание и вызов функции, завершённые точкой с запятой, — всё это операторы.

> **Предпочитайте puts форматному выводу printf** Когда это возможно, используйте `puts` вместо `printf`. Эта функция может выводить только одну строку (и завершает её переводом строки); спецификаторы формата не учитываются. Она не только быстрее, но и работает единообразно со всеми строками, и у неё нет изъянов безопасности, описанных в разделе 14.7.3.

Пока что мы всегда будем начинать наши программы со строки `#include <stdio.h>`. Она даёт нам доступ к части стандартной библиотеки C. Однако мы твёрдо заявляем, что это *не* импорт библиотеки в каком бы то ни было смысле и его *никогда* не следует так воспринимать.

Листинг 8-3. `hello.c`

```
/* This is a comment. The next line has a preprocessor directive */
#include <stdio.h>
/* `main` is the entry point for the program, like _start in assembly
 * Actually, the hidden function _start is calling `main`.
 * `main` returns the `return code` which is then given to the `exit` system
 * call.
 * The `void` keyword instead of argument list means that `main` accepts no
 * arguments */
int main(void) {
   /* A  variable local to `main`. Will be destructed as soon as `main` ends*/
   int x = 43;
   int y;
   y = x - 1;
   /* Calling a standard function `printf` with three arguments.
    * It will print 'Hello, world! y=42 x=43
    * All %d  will be replaced by the consecutive arguments */
   printf( "Hello, world! y=%d  x=%d\n", y, x);
   return 0;
}
```

Литерал — это последовательность символов в исходном коде, представляющая непосредственное значение. В C литералы существуют для

- Целых чисел, например, `42`.
- Чисел с плавающей точкой, например, `42.0`.
- ASCII-кода символов, записанного в одинарных кавычках, например, `'a'`.
- Указателей на строки, завершающиеся нулём, например, `"abcde"`.

Выполнение любой программы на C по сути представляет собой манипуляцию данными. *Абстрактная машина* C имеет архитектуру фон Неймана. Это сделано намеренно, потому что C — язык, который должен быть максимально близок к оборудованию. Переменные хранятся в линейной памяти, и у каждой из них есть начальный адрес. Можно думать о переменных как о метках в ассемблере.

8.2.1 Типы данных

Поскольку практически всё, что происходит, — манипуляция данными, природа этих самых данных представляет для нас особый интерес. У всех видов данных в C есть *тип*, что означает, что они попадают в одну из (обычно) различных категорий. Типизация в C слабая и статическая. *Статическая типизация* означает, что все типы известны во время компиляции. Не может быть абсолютно никакой неопределённости относи-

тельно типов данных. Используете ли вы переменную, литерал или более сложное выражение, вычисляющееся в некоторые данные, — его тип будет известен. *Слабая типизация* означает, что иногда элемент данных может быть неявно преобразован в другой тип, если это уместно. Например, при вычислении `1 + 3.0` очевидно, что эти два числа имеют разные типы. Одно из них целое; другое — вещественное число. Нельзя напрямую сложить одно с другим, потому что их двоичные представления различаются. Нужно преобразовать оба к одному типу (вероятно, к числу с плавающей точкой). Только тогда можно будет выполнить сложение. В строго типизированных языках, таких как OCaml, эта операция не допускается; вместо этого есть две отдельные операции сложения чисел: одна действует на целые числа (и записывается `+`), другая на вещественные числа (записывается `+.` в OCaml). Слабая типизация в C существует не просто так: в ассемблере совершенно нормально взять практически любые данные и интерпретировать их как данные другого типа (указатель как целое число, часть строки как целое число и т. д.) Посмотрим, что произойдёт, если мы попытаемся вывести значение с плавающей точкой как целое число (см. листинг 8-4). Результатом будет значение с плавающей точкой, *переинтерпретированное* как целое число, что не имеет особого смысла.

Листинг 8-4. `float_reinterpret.c`

```
#include <stdio.h>
int main(void) {
    printf("42.0 as an integer %d  \n", 42.0);
    return 0;
}
```

Вывод этой программы зависит от целевой архитектуры. В нашем случае вывод был

```
42.0 as an integer -266654968
```

Для этого краткого вводного раздела мы будем считать, что все типы в C попадают в одну из следующих категорий:

- Целые числа (`int`, `char`, ...).
- Числа с плавающей точкой (`double` и `float`).
- Типы указателей.
- Составные типы: структуры и объединения.
- Перечисления.

В Главе 9 мы подробнее рассмотрим систему типов. Если у вас есть опыт работы с языком более высокого уровня, вы можете заметить отсутствие некоторых широко известных элементов в этом списке. К сожалению, *в C89 нет строкового и булева типов*.

Целочисленное значение, равное нулю, считается ложью; любое ненулевое значение считается истиной.

8.3 Управляющие конструкции

Согласно принципам фон Неймана, выполнение программы последовательно. Каждый оператор выполняется один за другим. Есть несколько операторов для изменения потока управления.

8.3.1 if

Листинг 8-5 показывает оператор `if` с необязательной частью `else`. Если условие выполнено, выполняется первый блок. Если условие не выполнено, выполняется второй блок, но второй блок не обязателен.

Листинг 8-5. `if_example.c`

```
int x  =  100;
if (42) {
    puts("42 is not equal to zero and thus considered truth");
}
if (x > 3) {
    puts("X is greater than 3");
}
else
{
    puts("X is less than 3");
}
```

Фигурные скобки необязательны. Без скобок частью каждой ветви будет считаться только один оператор, как показано в листинге 8-6.

Листинг 8-6. `if_no_braces.c`

```
if (x == 0)
    puts("X is zero");
else
    puts("X is not zero");
```

Заметьте, что здесь есть синтаксическая ловушка, называемая *висячим else (dangling else)*. Посмотрите на листинг 8-7 и попробуйте однозначно определить, к какому `if` — первому или второму — относится ветвь `else`. Чтобы разрешить эту неоднозначность

в случае вложенных `if`, используйте фигурные скобки.

Листинг 8-7. `dangling_else.c`

```
if (x == 0)   if (y == 0) { puts("A"); }  else { puts("B"); }
/* You might have considered one of the following interpretations.
 * The compiler can issue a warning to prevent you */
if (x == 0) {
    if (y == 0) { printf("A"); }
    else { puts("B"); }
}
if (x == 0) {
    if (y == 0) { puts("A"); }
} else { puts("B"); }
```

8.3.2 while

Оператор `while` используется для создания циклов.

Листинг 8-8. while_example.c

```
int x = 10;
while ( x != 0 ) {
    puts("Hello");
    x = x - 1;
}
```

Если условие выполнено, тело выполняется. Затем условие проверяется снова, и если оно выполнено, тело выполняется снова, и так далее. Альтернативная форма `do ... while ( condition );` позволяет проверять условие после выполнения тела цикла, тем самым гарантируя хотя бы одну итерацию. Листинг 8-9 показывает пример. Заметьте, что тело может быть пустым, например: `while (x == 0);`. Точка с запятой после скобок завершает этот оператор.

Листинг 8-9. `do_while_example.c`

```
int x = 10;
do {
    printf("Hello\n");     x = x - 1;
}
while ( x != 0 );
```

8.3.3 for

Оператор `for` идеален для перебора конечных коллекций, таких как связные списки или массивы. У него следующая форма: `for ( initializer ; condition; step ) body`. Листинг 8-10 показывает пример.

Листинг 8-10. `for_example.c`

```
int a[] = {1, 2, 3, 4}; /* an array of 4 elements */
int i = 0;
for ( i = 0; i < 4; i++ ) {
    printf( "%d",  a[i])
}
```

Сначала выполняется инициализатор. Затем проверяется условие, и если оно истинно, выполняется тело цикла, а затем оператор шага. В данном случае оператором шага является оператор инкремента `++`, изменяющий переменную, увеличивая её значение на единицу. После этого цикл начинается снова с проверки условия, и так далее. Листинг 8-11 показывает два эквивалентных цикла.

Листинг 8-11. `while_for_equiv.c`

```
int i;
/* as a `while` loop */
i = 0;
while ( i < 10 ) {
    puts("Hello!");
    i = i + 1;
}
/* as a `for` loop */
for( i = 0; i < 10; i = i + 1 ) {
    puts("Hello!");
}
```

Оператор `break` используется для преждевременного завершения цикла и перехода к следующему оператору в коде. `continue` завершает текущую итерацию и сразу начинает следующую. Листинг 8-12 показывает пример.

Листинг 8-12. `loop_cont.c`

```
int n = 0;
for( n = 0; n < 20; n++ ) {
    if (n % 2) continue;
    printf("%d is odd", n );
}
```

Заметьте также, что в цикле `for` инициализатор, шаг или условие могут быть оставлены пустыми. Листинг 8-13 показывает пример.

Листинг 8-13. `infinite_for.c`

```
for( ; ; ) {
    /* this cycle will loop forever, unless `break` is issued in its body */
    break; /* `break` is here, so we stop iterating */
}
```

8.3.4 goto

Оператор `goto` позволяет выполнять переходы к метке внутри той же функции. Как и в ассемблере, метки могут отмечать любой оператор, и синтаксис такой же: `label: statement`. Это часто считается плохим стилем кода; однако он может быть весьма кстати при кодировании *конечных автоматов*. Чего не стоит делать — так это отказываться от продуманных условий и циклов в пользу goto-спагетти. Оператор `goto` иногда используется как способ выйти из нескольких вложенных циклов. Однако это часто симптом плохого проектирования, потому что внутренние циклы можно абстрагировать в отдельную функцию (благодаря оптимизациям компилятора, вероятно, вообще без затрат на производительность). Листинг 8-14 показывает, как использовать `goto`, чтобы выйти из всех внутренних циклов.

Листинг 8-14. `goto.c`

```
int i;
int j;
for (i = 0; i < 100; i++ )
for( j = 0; j < 100; j++ ) {
    if (i * j == 432)
        goto end;
    else
        printf("%d * %d != 432\n", i, j );
}
end:
```

Оператор `goto` вместе с императивным стилем затрудняет анализ поведения программы как для людей, *так* и для машин (компиляторов), так что хитрые оптимизации, на которые способны современные компиляторы, становятся менее вероятными, а код становится труднее поддерживать. Мы рекомендуем ограничивать использование `goto` фрагментами кода, которые *не выполняют присваиваний*, например реализациями конечных автоматов. Тогда вам не придётся отслеживать все возможные пути выполнения программы и то, как меняются значения определённых переменных при том или ином варианте выполнения программы.

8.3.5 switch

Оператор `switch` используется как несколько вложенных `if`, когда условие — это некоторая целочисленная переменная, равная тому или иному значению. Листинг 8-15 показывает пример.

Листинг 8-15. `case_example.c`

```
int i = 10;
switch ( i ) {
    case 1: /* if i is equal to 1...*/
        puts( "It is one" );
        break; /* Break is mandatory */
    case 2: /* if i is equal to 2...*/
        puts( "It is two" );
        break;
    default: /* otherwise... */
        puts( "It is not one nor two" );
        break;
}
```

Каждый case на самом деле является меткой. Case ничем не ограничены, кроме необязательного оператора `break` для выхода из блока `switch`. Это позволяет проделывать интересные трюки.[19] Однако забытый `break` обычно становится источником ошибок. Листинг 8-16 показывает эти два поведения: во-первых, несколько меток приписаны одному и тому же case, а значит, независимо от того, равен ли `x` 0, 1 или 10, выполнится один и тот же код. Во-вторых, поскольку `break` не завершает этот case, после выполнения первого `printf` управление перейдёт к следующей инструкции, помеченной `case 15`, — другому `printf`.

Листинг 8-16. `case_magic.c`

```
switch ( x ) {
    case 0:
    case 1:
    case 10:
        puts( "First case: x = 0, 1 or 10" );
```

```
        /* Notice the absence of `break`! */
    case 15:
        puts( "Second case: x = 0, 1, 10 or 15" );
        break;
}
```

8.3.6 Пример: делитель

Листинг 8-17 демонстрирует программу, ищущую первый делитель, который затем выводится в `stdout`. Функция `first_divisor` принимает аргумент n и ищет целое r от 1 не включительно до n включительно такое, что n кратно r. Если $r = n$, мы, очевидно, нашли *простое число*. Обратите внимание, что оператор после `for` не заключён в фигурные скобки, потому что это единственный оператор внутри цикла. То же самое произошло с телом `if`, состоящим из единственного `return i`. Разумеется, вы можете заключить его в скобки, и некоторые программисты действительно это поощряют.

Листинг 8-17. `divisor.c`

```
#include <stdio.h>
int first_divisor( int n ) {
    int i;
    if ( n == 1 ) return 1;
    for( i = 2; i <= n; i++ )
        if ( n %  i == 0 ) return i;
    return 0;
}
int main(void) {
    int i;
    for( i = 1; i < 11; i++ )
        printf( "%d \n", first_divisor( i ) );
    return 0;
}
```

8.3.7 Пример: является ли число числом Фибоначчи?

Листинг 8-18 показывает программу, проверяющую, является ли число числом Фибоначчи. Последовательность Фибоначчи определяется рекурсивно следующим образом:

```
f1= 1
f2= 1
fn= fn-1+ fn-2
```

У этой последовательности множество применений, в частности в комбинаторике. Последовательности Фибоначчи встречаются даже в биологическом контексте, например при ветвлении деревьев, расположении листьев на стебле и т. д. Первые числа Фибоначчи — 1, 1, 2, 3, 5, 8 и т. д. Как видите, каждое число — сумма двух предыдущих чисел. Чтобы проверить, содержится ли заданное число n в последовательности Фибоначчи, мы применяем прямолинейный (не обязательно оптимальный) подход — вычисление всех членов последовательности до n. Природа последовательности Фибоначчи подразумевает, что она возрастающая, так что если мы нашли член, больший n, и всё ещё не встретили n, мы делаем вывод, что n не входит в последовательность. Функция `is_fib` принимает целое число n и вычисляет все элементы, меньшие или равные n. Если последний элемент этой последовательности равен n, то n — число Фибоначчи, и функция возвращает 1; иначе она возвращает 0.

Листинг 8-18. `is_fib.c`

```
#include <stdio.h>
int is_fib( int n ) {
    int a = 1;
    int b = 1;
    if ( n == 1 ) return 1;
    while ( a <= n && b <= n ) {
        int t = b;
        if (n == a || n == b) return 1;
        b = a;
        a = t + a;
    }
    return 0;
}
void check(int n) { printf( "%d -> %d\n", n, is_fib( n ) ); }
int main(void) {
    int i;
    for( i = 1; i < 11; i = i + 1 ) {
        check( i );
    }
    return 0;
}
```

8.4 Операторы и выражения

Язык C основан на понятиях операторов и *выражений*. Выражения соответствуют сущностям данных. Все литералы и имена переменных — выражения. Кроме того, сложные выражения можно строить с помощью операций (`+`, `-` и других логических, арифметических и битовых операций) и вызовов функций (за исключением процедур, возвращающих `void`). Листинг 8-19 показывает несколько примеров выражений.

Листинг 8-19. `expr_example.c`

```
1
13  +  37
17 + 89 * square( 1 )
x
```

Выражения — это данные, так что их можно использовать справа от оператора присваивания `=`. Некоторые выражения также можно использовать слева от присваивания. Они должны соответствовать сущностям данных, имеющим адрес в памяти.[20] Такие выраже-

ния называются *lvalue*; все остальные выражения, не имеющие адреса, называются *rvalue*. Это различие на самом деле весьма интуитивно, если думать в терминах абстрактной машины. Выражения, подобные показанным в листинге 8-20, лишены смысла, потому что присваивание означает изменение памяти.

Листинг 8-20. `rvalue_example.c`

```
4 = 2;
"abc"="bcd";
square(3)  =  9;
```

8.4.1 Типы операторов

Операторы — это команды для абстрактной машины C. Каждая команда — это императив: сделай что-то! Отсюда и название «императивное программирование»: это последовательность команд. Есть три типа операторов:

1. Выражения, завершённые точкой с запятой.

```
1 + 3;
42;
square(3);
```

Назначение таких операторов — вычисление заданных выражений. Если они не вызывают присваиваний (напрямую как часть самого выражения или внутри одной из вызываемых функций) или операций ввода/вывода, их влияние на состояние программы не наблюдаемо.

2. Блок, ограниченный { и }. Он содержит произвольное число предложений. Блок

не должен сам завершаться точкой с запятой (хотя операторы внутри него, вероятно, должны). Листинг 8-21 показывает типичный блок.

Листинг 8-21. `block_example.c`

```
int y = 1 + 3;
{
    int x;
    x = square( 2 ) + y;
    printf( "%d\n", x );
}
```

3. Операторы управления потоком: `if`, `while`, `for`, `switch`. Они не требуют точки с запятой.

Мы уже говорили о присваиваниях; неприятная правда в том, что присваивания сами являются выражениями, а значит, их можно объединять в цепочки. Например, `a = b = c` означает

- Присвоить `c` переменной `b`;
- Присвоить новое значение `b` переменной `a`.

Таким образом, типичное присваивание — это оператор из первой категории: выражение, завершённое точкой с запятой. Присваивание — право-ассоциативная операция. Это значит, что при разборе компилятором (или вашим взглядом) скобки неявно расставляются справа налево, а самая правая часть оказывается наиболее глубоко вложенной. Листинг 8-22 приводит пример двух эквивалентных способов записи сложного присваивания.

Листинг 8-22. `assignment_assoc.c`

```
x = y = z;
(x = (y = z));
```

С другой стороны, лево-ассоциативные операции подразумевают обратный порядок вложенности, как показано в листинге 8-23

Листинг 8-23. `div_assoc.c`

```
40 / 2 / 4
((40 / 2) / 4)
```

8.4.2 Построение выражений

Выражение строится из других выражений, соединённых операторами и вызовами функций. Операторы можно классифицировать

- По арности (количеству операндов)
- Унарные (например, унарный минус: `- expr`)
- Бинарные (например, бинарное умножение: `expr1 * expr2`)
- Тернарные. Есть только один тернарный оператор: `cond ? expr1 : expr2`. Если условие истинно, значение равно `expr1`, иначе `expr2`
- По смыслу Арифметические операторы: `* / + - % ++ --` Операторы отношения: `== != > < >= <=` Логические операторы: `! && || << >>`

- Битовые операторы: ~ ^ & |

```
Assignment Operators = += -= *= /= %= <<= >>= &= ^= |=
```

- Прочие операторы: 1. `sizeof(var)` в значении «замени это на размер `var` в байтах» 2. `&` в значении «взять адрес операнда» 3. в значении «разыменовать этот указатель» 4. `?:` — тернарный оператор, о котором мы уже говорили ранее. 5. `->`, используемый для обращения к полю структурного типа или объединения.

У большинства операторов очевидный смысл. Мы упомянём некоторые из менее используемых и более неочевидных.

- Операторы инкремента и декремента можно использовать как в префиксной, так и в постфиксной форме: для переменной `i` это либо `i++`, либо `++i`. Оба выражения немедленно повлияют на `i`, увеличив его на 1. Однако значением `i++` будет «старое» `i`, тогда как значением `++i` — «новое», увеличенное `i`.
- Есть разница между логическими и битовыми операторами. Для логических операторов любое ненулевое число по сути равнозначно, тогда как битовые операции применяются к каждому биту отдельно. Например, `2 & 4` равно нулю, потому что ни один бит не установлен одновременно и в 2, и в 4. Однако `2 && 4` вернёт 1, потому что и `2`, и `4` — ненулевые числа (истинные значения).
- Логические операторы вычисляются лениво. Рассмотрим логический оператор И &&. При применении к двум выражениям сначала вычисляется первое выражение. Если его значение равно нулю, вычисление немедленно завершается из-за природы операции И. Если хотя бы один из операндов равен нулю, результат всей конъюнкции тоже будет равен нулю, так что дальше вычислять нет смысла. Это важно для нас, потому что такое поведение заметно. Листинг 8-24 показывает пример, в котором программа выведет `F` и никогда не выполнит функцию `g`.

Листинг 8-24. logic_lazy.c

```
#include <stdio.h>
int f(void) { puts( "F" ); return 0; }
int g(void) { puts( "G" ); return 1; }
int main(void) {
    f() && g();
    return  0;
}
```

- Тильда (~) — это битовое унарное отрицание, крышка (^) — битовое бинарное исключающее ИЛИ.

В следующих главах мы вернёмся к некоторым из них, таким как операнды работы с адресами и `sizeof`.

8.5 Функции

Можно провести границу между процедурами (не возвращающими значение) и функциями (возвращающими значение определённого типа). Вызов процедуры нельзя встроить в более сложное выражение, в отличие от вызова функции. Листинг 8-25 показывает пример процедуры. Её имя — `myproc`; она возвращает `void`, то есть не возвращает ничего. Она принимает два целочисленных параметра с именами `a` и `b`.

Листинг 8-25. `proc_example.c`

```
void myproc ( int a, int b )
{
    printf("%d",  a+b);
}
```

Листинг 8-26 показывает пример функции. Она принимает два аргумента и возвращает значение типа `int`. Вызов этой функции позже используется как часть более сложного выражения.

Листинг 8-26. `function_example.c`

```
int myfunc ( int a, int b )
{
    return a + b;
}
int other( int x ) {
    return 1 + myfunc( 4, 5 );
}
```

Выполнение каждой функции завершается оператором `return`; иначе то, какое значение она вернёт, не определено. У процедур ключевое слово `return` может быть опущено; его всё же можно использовать без операнда, чтобы немедленно выйти из процедуры. Когда аргументов нет, в объявлении функции следует использовать ключевое слово `void`, как показано в

Листинге 8-27.

Листинг 8-27. `no_arguments_ex.c`

```
int always_return_0( void ) { return 0; }
```

Тело функции — это блок-оператор, так что оно заключено в фигурные скобки и не завершается точкой с запятой. Каждый блок определяет лексическую область видимости для переменных. Все переменные должны быть объявлены в начале блока, до любых операторов. Это ограничение присутствует в C89, но не в C99. Мы будем его придерживаться, чтобы сделать код более переносимым. Кроме того, это вынуждает к определённой самодисциплине. Если у вас большое количество локальных переменных, объявленных в начале области видимости, это будет выглядеть загромождённо. В то же время это обычно признак плохой декомпозиции программы и/или неудачного выбора структур данных. Листинг 8-28 показывает примеры хороших и плохих объявлений переменных.

Листинг 8-28. `block_variables.c`

```
/* Good */
void f(void) {
    int x;
    ...
}
/* Bad: `x` is declared after `printf` call */
void f(void) {
    int y = 12;
    printf( "%d", y);
    int x = 10;
    ...
}
/* Bad: `i` can not be declared in `for` initializer */
for( int i = 0; i < 10; i++ ) {
    ...
}
/* Good: `i` is declared before `for` */
int f(void) {
    int i;
    for( i = 0; i < 10; i++ ) {
        ...
    }
}
/* Good: any block can have additional variables declared in its beginning */
/* `x` is local to one `for` iteration and is always reinitialized to 10 */
for( i = 0; i < 10; i++ ) {
    int x = 10;
}
```

Если переменная в определённой области видимости имеет то же имя, что и переменная, уже объявленная в более внешней области видимости, более новая переменная скрывает старую. Синтаксически нет способа обратиться к скрытой переменной (если не сохранить её адрес где-то и не использовать этот адрес). Локальные переменные в разных функциях, разумеется, могут иметь одинаковые имена.

> **Примечание** Переменные видимы до конца своего соответствующего блока. Так что часто используемое понятие «локальных» переменных на самом деле означает «локальных для блока», а не «локальных для функции». Практическое правило: делайте переменные настолько локальными, насколько это возможно (включая, например, переменные, локальные для тел циклов). Это значительно снижает сложность программы, особенно в крупных проектах.

8.6 Препроцессор

Препроцессор C действует подобно препроцессору NASM. Однако его возможности гораздо более ограничены. Наиболее важные *директивы препроцессора*, которые вы увидите, — это

- #define
- #include
- #ifndef
- #endif

Директива `#define` очень похожа на свой аналог `%define` в NASM. У неё три основных применения.

- Определение глобальных констант (пример см. в листинге 8-29).

Листинг 8-29. `define_example1.c`

```
#define MY_CONST_VALUE 42
```

- Определение параметризованных макроподстановок (как показано в листинге 8-30).

Листинг 8-30. `define_example2.c`

```
#define MACRO_SQUARE( x ) ((x) * (x))
```

- Определение флагов; в зависимости от них, некоторый дополнительный код может включаться в исходники или исключаться из них.

Важно заключать в скобки все вхождения аргументов внутри определений макросов. Причина в том, что макросы C не синтаксические, а значит, препроцессор не осведомлён о структуре кода. Иногда это приводит к неожиданному поведению, как показано в листинге 8-31. Листинг 8-32 показывает код после препроцессирования.

Листинг 8-31. `define_parentheses.c`

```
#define SQUARE( x ) (x * x)
int x = SQUARE( 4+1 )
```

Как видите, значением `x` будет не 25, а 4+(1*4)+1, потому что умножение имеет более высокий приоритет по сравнению со сложением.

Листинг 8-32. `define_parentheses_preprocessed.c`

```
int x = 4+1 * 4+1
```

Директива `#include` вставляет содержимое указанного файла на своё место. Имя файла заключается либо в кавычки (`#include "file.h"`), либо в угловые скобки (`#include <stdio.h>`).

- В случае угловых скобок файл ищется в наборе предопределённых каталогов.
  Для GCC это обычно:
- /usr/local/include
- <libdir>/gcc/target/version/include Здесь `<libdir>` обозначает каталог, хранящий библиотеки (настройка GCC), и обычно по умолчанию это `/usr/lib` или `/usr/local/lib`.
- /usr/target/include
- /usr/include С помощью ключа `-I` можно добавить каталоги в этот список.
  Вы можете создать специальный каталог `include/` в корне вашего проекта и добавить его в список поиска включений GCC*.*
- В случае кавычек файлы также ищутся в текущем каталоге.

Вывод препроцессора можно получить, обработав файл `filename.c` так же, как при работе с NASM: `gcc -E filename.c`. Это выполнит все директивы препроцессора и сбросит результаты в `stdout`, не делая ничего больше.

8.7 Итоги

В этой главе мы разобрали основы C. Все переменные — это метки в памяти абстрактной машины языка C, чья архитектура во многом напоминает архитектуру фон Неймана. Описав универсальную структуру программы (функции, типы данных, глобальные переменные...), мы определили две синтаксические категории: операторы и выражения. Мы увидели, что выражения бывают либо `lvalues`, либо `rvalues`, и научились управлять выполнением программы с помощью вызовов функций и управляющих операторов, таких как `if` и `while`. Мы уже способны писать простые программы, выполняющие вычисления над целыми числами. В следующей главе мы обсудим систему типов в C и типы в целом, чтобы получить более полную картину того, как типы используются в разных языках программирования. Благодаря понятию массивов наши возможные входные и выходные данные станут намного разнообразнее.

> **Вопрос 148** Что такое литерал?

Вопрос 149 Что такое `lvalue` и `rvalue`?

Вопрос 150 В чём разница между операторами и выражениями?

Вопрос 151 Что такое блок операторов?

Вопрос 152 Как определить символ препроцессора?

Вопрос 153 Зачем нужен `break` в конце каждого case в `switch`?

Вопрос 154 Как в C89 кодируются значения истины и лжи?

Вопрос 155 Что является первым аргументом функции `printf`?

Вопрос 156 Проверяет ли `printf` типы своих аргументов?

Вопрос 157 Где можно объявлять переменные в C89?

Система типов

Понятие типа — одно из ключевых. Тип, по сути, — это метка, присвоенная сущности данных. Каждое преобразование данных определено для конкретных типов данных, что обеспечивает их корректность (вы же не захотите складывать количество активных пользователей Reddit со средней температурой в полдень в Сахаре, потому что в этом нет смысла). В этой главе мы подробно изучим систему типов C.

9.1 Базовая система типов C

Все типы в C попадают в одну из следующих категорий:

- Предопределённые числовые типы (`int` , `char` , `float` и т. д.).
- Массивы — несколько элементов одного типа, занимающих последовательные ячейки памяти.
- Указатели — по сути, ячейки, хранящие адреса других ячеек. Тип указателя кодирует тип ячейки, на которую он указывает. Частный случай указателей — указатели на функции.
- Структуры — наборы данных разных типов. Например, структура может хранить целое число и число с плавающей точкой. У каждого элемента данных есть своё имя.
- Перечисления — по сути, целые числа, принимающие одно из явно определённых значений. У каждого из этих значений есть символическое имя для обращения к нему.
- Функциональные типы.
- Константные типы, построенные поверх какого-то другого типа и делающие данные неизменяемыми.
- Псевдонимы типов для других типов.

9.1.1 Числовые типы

Самые базовые типы C — числовые. У них разные размеры, и они бывают знаковыми или беззнаковыми. Из-за долгой и слабо контролируемой эволюции языка их описание порой может показаться загадочным и довольно часто весьма несистематичным. Ниже приведён список базовых типов:

1. char

- Может быть `signed` и `unsigned` . По умолчанию обычно является знаковым числом, но это не требуется стандартом языка.
- Его размер всегда 1 байт;

- Несмотря на то что название прямо отсылает к слову «символ» (character), это целочисленный тип, и его следует воспринимать именно так. Его часто используют для хранения ASCII-кода символа, но в нём можно хранить любое 1-байтовое число.
- Литерал `'x'` соответствует ASCII-коду символа «x». Его тип — `int`, но его безопасно присваивать переменной типа `char`.[21]

Листинг 9-1 показывает пример.

Листинг 9-1. `char_example.c`

```
char number = 5;
char symbol_code = 'x';
char null_terminator = '\0';
```

2. `int`

- Целое число.
- Может быть `signed` и `unsigned`. По умолчанию `signed`.
- Может обозначаться также как: `signed`, `signed int` (аналогично для `unsigned`).
- Может быть `short` (2 байта), `long` (4 байта на 32-битных архитектурах, 8 байт в Intel 64). Большинство компиляторов также поддерживают `long long`, но вплоть до C99 он не входил в стандарт.
- Другие обозначения: short, short int, signed short, signed short int.
- Размер `int` без модификаторов зависит от архитектуры. Он был задуман равным размеру машинного слова. В эпоху 16-бит размер `int` был, очевидно, 2 байта, на 32-битных машинах — 4 байта. К сожалению, это не помешало программистам полагаться на `int` размером 4 в эпоху 32-битных вычислений. Из-за огромного объёма программного обеспечения, которое сломалось бы при изменении размера `int`, его размер оставили нетронутым — он по-прежнему 4 байта.
- Важно отметить, что все целочисленные литералы по умолчанию имеют формат `int`. Если мы добавим суффиксы L или UL, мы явно укажем, что эти числа имеют тип `long int` или `unsigned int`. Иногда крайне важно не забывать про эти суффиксы. Рассмотрим выражение `1 << 48`. Его значение не 2^{48}, как вы могли подумать, а 0. Почему? Причина в том, что 1 — это литерал типа `int`, занимающий 4 байта и потому способный принимать значения от -2^{31} до $2^{31} - 1$. Сдвигая 1 влево 48 раз, мы выдвигаем единственный установленный бит за пределы формата целого числа. Поэтому результат — ноль. Однако если мы добавим правильный суффикс, ответ станет более очевидным. Выражение `1L << 48` вычисляется в 2^{48}, потому что теперь `1L` занимает 8 байт.

3. long long

- В архитектуре x64 это то же самое, что `long` (кроме Windows, где `long` — 4 байта).
- Его размер — 8 байт.
- Его диапазон: -2^{63} ... $2^{63}-1$ для знакового и $0...2^{64}-1$ для беззнакового.

4. float

- Число с плавающей точкой.
- Его размер — 4 байта.
- Его диапазон: $\pm 1,17549 \times 10 ... \pm 3,40282 \times 10^{38}$ (приблизительно шесть значащих цифр).

5. double

- Число с плавающей точкой.
- Его размер — 8 байт.
- Его диапазон: $\pm 2,22507 \times 10 ... \pm 1,79769 \times 10^{308}$ (приблизительно 15 значащих цифр).

6. long double

- Число с плавающей точкой.
- Его размер обычно 80 бит.
- Он был введён только в стандарте C99.

> **Примечание об арифметике с плавающей точкой** Прежде всего, помните, что типы с плавающей точкой — очень грубое приближение вещественных чисел. Например, они точнее вблизи 0 и менее точны для больших значений. Именно поэтому их диапазон настолько велик даже по сравнению с `longs`. Как следствие, выполнение арифметики с плавающей точкой над значениями ближе к нулю даёт более точные результаты. Наконец, в некоторых контекстах (например, программирование ядра) арифметика с плавающей точкой недоступна. Как практическое правило — избегайте её, когда она не нужна. Например, если ваши вычисления можно выполнить, оперируя частным и остатком, вычисленными с помощью операторов `/` и `%`, придерживайтесь их.

9.1.2 Приведение типов

Язык позволяет относительно свободно преобразовывать данные между типами. Для этого нужно написать имя нового типа в скобках перед выражением, которое вы хотите преобразовать. Листинг 9-2 показывает пример.

Листинг 9-2. `type_cast.c`

```
int a = 4;
double b = 10.5 * (double)a; /* now a is a double */
int b = 129;
char k = (char)b; //???
```

Разумеется, этот чудесный открытый мир возможностей лучше держать под контролем вашей благосклонной диктатуры, потому что такие неявные преобразования часто приводят к неочевидным ошибкам, когда выражение вычисляется не в то значение, в которое оно «должно» вычислиться. Например, поскольку `char` — это (обычно) знаковое число в диапазоне -128...127, число 129 слишком велико, чтобы поместиться в этот диапазон. Результат действия, показанного в листинге 9-2, не описан в стандарте языка, но, учитывая то, как функционируют типичные процессоры и компиляторы, результатом, вероятно, будет отрицательное число, состоящее из тех же битов, что и беззнаковое представление числа 129.

> **Вопрос 158** Каким будет значение k? Попробуйте скомпилировать и посмотреть на своём компьютере.

9.1.3 Булев тип

Мы уже говорили, что в C89 нет булевых значений. Однако C99 ввёл булевы значения как тип `_Bool`. Если вы подключите `stdbool.h`, у вас появится доступ к значениям `true` / `false` и типу `bool`, являющемуся псевдонимом `_Bool`. Логика здесь простая. Во многих существующих проектах уже определён собственный булев тип, обычно называемый `bool`. Чтобы предотвратить конфликты имён, имя типа для булевых значений в C99 — `_Bool`. Подключение файла `stdbool.h` означает, что в вашем коде нет никакого собственного определения `bool`, и вы выбираете то, что соответствует стандарту, но с более человечным именем. Мы рекомендуем вам по возможности использовать псевдоним `bool`. В будущем имя типа `_Bool`, вероятно, будет объявлено устаревшим, а через несколько версий стандарта перестанет использоваться вовсе.

9.1.4 Неявные преобразования

Будучи слабо типизированным языком, C иногда позволяет опускать приведения типов, даже когда используются данные не того типа, что предполагалось. Когда требуемый числовой тип не совпадает с фактическим типом, выполняется неявное преобразование, называемое *целочисленным повышением (integer promotion)*. Если тип меньше

`int`, он повышается до `signed int` или `unsigned int`, в зависимости от того, был ли он изначально знаковым или беззнаковым.[22] Затем, если типы всё ещё различаются, мы поднимаемся по лестнице, показанной на рисунке 9-1

```
int → unsigned int → long → unsigned long → long long →
    unsigned long long → float → double → long double
```

Рисунок 9-1. Целочисленные преобразования

> **Примечание** Помните, что `long long` и `long double` появились только в C99. Однако многие компиляторы, ещё не поддерживающие C99, поддерживают их как расширение языка.

Правило «сначала преобразовать в `int`» означает, что переполнения в меньших типах могут обрабатываться *иначе*, чем в самом типе `int`. Пример, показанный в листинге 9-3, предполагает, что `sizeof(int) == 4`.

Листинг 9-3. `int_promotion_pitfall.c`

```
/* The lesser types */
unsigned char  x = 100, y = 100, z = 100;
unsigned char r = x + y + z; /* will give you 300 % 256 = 44 */
```

```
unsigned int r_int = x + y + z; /* equals to 300, because the promotion to
                                   integers is performed first */
/* Now with the greater types */
unsigned int x = 1e9, y = 2e9, z = 3e9;
unsigned int r_int = x + y + z;   /* 1705032704 equals 6000000000 % (2^32) */
unsigned long r_long = x + y + z;   /*  the same result: 1705032704 */
```

В последней строке ни `x`, ни `y`, ни `z` не повышаются до `long`, потому что это не требуется стандартом. Арифметика будет выполнена в рамках типа `int`, а затем результат будет преобразован в `long`.

> **Будьте однозначны** Как практическое правило, при неуверенности всегда указывайте типы явно! Например, вы

```
can write long x = (long)a + (long)b + (long)c.
```

> **Примечание.** Хотя после этого код может показаться более многословным, он по крайней мере будет работать так, как задумано.

Посмотрим на пример, показанный в листинге 9-4. Выражение в третьей строке будет вычислено следующим образом:

1. Значение `i` будет преобразовано в `float` (разумеется, сама переменная не изменится);
2. Это значение прибавляется к значению `f`, результирующий тип снова `float`; и
3. Этот результат преобразуется в `double`, чтобы быть сохранённым в d.

Листинг 9-4. `int_float_conv.c`

```
int i;
float f;
double d = f + i;
```

Все эти операции не бесплатны и кодируются как ассемблерные инструкции. Это значит, что всякий раз, когда вы оперируете числами разных форматов, это, вероятно, имеет издержки во время выполнения. Старайтесь избегать этого, особенно в циклах.

9.1.5 Указатели

Имея тип `T`, всегда можно построить тип `T*`. Этот новый тип соответствует единицам данных, хранящим адрес другой сущности типа `T`. Поскольку все адреса имеют одинаковый размер, все типы указателей тоже имеют одинаковый размер. Он зависит от архитектуры и в нашем случае равен 8 байтам. С помощью операндов `&` и `*` можно взять адрес переменной или разыменовать указатель (заглянуть в память по адресу, который хранит этот указатель). Листинг 9-5 показывает пример. В разделе 2.5.4 мы обсуждали тонкую проблему: если указатель — это просто адрес, как нам узнать размер сущности данных, которую мы пытаемся прочитать начиная с этого адреса? В ассемблере это было прямолинейно: либо размер можно было вывести из того, что оба операнда `mov` должны быть одного размера, либо размер нужно было указать явно, например, `mov qword [rax]`, `0xABCDE`. Здесь об этом заботится система типов: если указатель имеет тип `int*`, мы точно знаем, что его разыменование даёт значение размером `sizeof(int)`.

Листинг 9-5. `ptr_deref.c`

```
int x = 10;
int* px = &x; /* Took address of `x` and assigned it to `px` */
*px = 42; /* We modified `x` here! */
printf( "*px = %d\n", *px ); /* outputs: '*px = 42' */
printf( "x = %d\n", x ); /* outputs: 'x = 42' */
```

Когда вы программируете на C, указатели — ваш хлеб насущный. Пока вы не создаёте указатель на несуществующие данные, указатели будут служить вам верой и правдой. Особое значение указателя — `0`. Используемый в контексте указателя (в частности, при сравнении с `0`), `0` означает «особое значение указателя в никуда». Вместо 0 можно также написать `NULL`, и мы советуем именно так и делать. Обычная практика — присваивать `NULL` указателям, ещё не инициализированным действительным адресом объекта, или возвращать `NULL` из функций, возвращающих адрес чего-либо, чтобы дать вызывающей стороне знать об ошибке.

> **Ноль ли ноль?** Есть два контекста, в которых можно использовать выражение `0` в C. Первый контекст ожидает просто обычное целое число. Второй — контекст указателя, когда вы присваиваете указателю 0 или сравниваете его с 0. Во втором контексте `0` не всегда означает целочисленное значение со всеми сброшенными битами, но всегда будет равен этому значению «недействительного указателя». На некоторых архитектурах это может быть, например, значение со всеми установленными битами. Но этот код будет работать независимо от архитектуры благодаря следующему правилу:

```
int* px = ... ;
if ( px ) /* if `px` is not NULL */
if ( px == 0 ) /* same thing as the following: */
if (!px ) /* if `px` is NULL */
```

Есть особый вид типа указателя: `void*`. Это указатель на данные любого рода. C позволяет присваивать указатель любого типа переменной типа `void*`; однако эту переменную нельзя разыменовать. Прежде чем это сделать, нужно взять её значение и преобразовать к действительному типу указателя (например, `int*)`. Для этого используется простое приведение типа (см. раздел 9.1.2). Листинг 9-6 показывает пример.

Листинг 9-6. `void_deref.c`

```
int a = 10;
void* pa = &a;
printf("%d\n", *( (int*) pa) );
```

Вы также можете передать указатель типа `void*` в любую функцию, принимающую указатель на какой-то другой тип. У указателей много назначений, и мы перечислим несколько из них.

- Изменение переменной, созданной вне функции.
- Создание и обход сложных структур данных (например, связных списков).
- Вызов функций по указателям означает, что, меняя указатель, мы переключаемся между

разными вызываемыми функциями. Это позволяет создавать довольно элегантные архитектурные решения. Указатели тесно связаны с массивами, которые обсуждаются в следующем разделе.

9.1.6 Массивы

В C массив — это структура данных, хранящая фиксированное количество данных одного типа. Так что для работы с массивом нам нужно знать его начало, размер одного элемента и количество элементов, которое он может хранить. Обратитесь к листингу 9-7, чтобы увидеть несколько вариантов объявления массива.

Листинг 9-7. `array_decl.c`

```
/* This array's size is computed by compiler */
int arr[] = {1,2,3,4,5};
/* This array is initialized with zeros, its size is 256 bytes */
long array[32] = {0};
```

Поскольку количество элементов должно быть фиксированным, его нельзя прочитать из переменной.[23] Для выделения памяти под массивы, размеры которых заранее неизвестны, используются аллокаторы памяти (которые даже не всегда доступны, например, при программировании ядра). Мы научимся использовать стандартный аллокатор памяти C (`malloc / free`) и даже напишем собственный. Вы можете обращаться к элементам по индексу. Индексы начинаются с 0. Истоки такого решения — в природе адресного пространства. Нулевой элемент расположен по начальному адресу массива плюс 0, умноженный на размер элемента. Листинг 9-8 показывает объявление массива, два чтения и одну запись.

Листинг 9-8. `array_example_rw.c`

```
int myarray[1024];
int y = myarray[64];
int first = myarray[0];
myarray[10] = 42;
```

Если немного подумать об абстрактной машине C, массивы — это просто непрерывные области памяти, хранящие данные одного типа. Никакой информации о самом типе или о длине массива нет. Полностью на программисте лежит обязанность никогда не обращаться к элементу за пределами выделенного массива. Всякий раз, когда вы пишете имя выделенного массива, вы на самом деле обращаетесь к его адресу. Можно думать об этом как о константном *значении* указателя. Именно здесь аналогия между метками ассемблера и переменными наиболее сильна. Так, в листинге 9-8 выражение `myarray` на самом деле имеет тип `int*`, потому что это указатель на первый элемент массива! Это также означает, что выражение `*myarray` вычислится в его первый элемент, точно так же как `myarray[0]`.

9.1.7 Массивы как аргументы функций

Поговорим о функциях, принимающих массивы в качестве аргументов. Листинг 9-9 показывает функцию, возвращающую первый элемент массива (или -1, если массив пуст).

Листинг 9-9. `fun_array1.c`

```
int first (int array[], size_t sz ) {
    if ( sz == 0 ) return -1;
    return array[0];
}
```

Неудивительно, что ту же функцию можно переписать, сохранив то же поведение, как показано в листинге 9-10.

Листинг 9-10. `fun_array2.c`

```
int first (int* array, size_t sz ) {
    if ( sz == 0 ) return -1;
    return *array;
}
```

Но это ещё не всё. На самом деле их можно смешивать и использовать индексную запись с указателями, как показано в

Листинге 9-11.

Листинг 9-11. `fun_array3.c`

```
int first (int* array, size_t sz ) {
    if ( sz == 0 ) return -1;
    return array[0];
}
```

Компилятор сразу понижает конструкции вроде `int array[]` в списке аргументов до указателя `int* array`, а затем работает с ним именно так. Синтаксически, впрочем, вы всё ещё можете указать длину массива, как показано в листинге 9-12. Это число указывает, что данный массив должен содержать как минимум столько элементов. Однако компилятор *воспринимает это как комментарий и не выполняет никаких проверок ни во время выполнения, ни во время компиляции.*

Листинг 9-12. `array_param_size.c`

```
int first( int array[10], size_t sz ) { ... }
```

C99 ввёл специальный синтаксис, по сути соответствующий вашему обещанию компилятору, что соответствующий массив будет содержать как минимум столько элементов. Это позволяет компилятору выполнять некоторые специфичные оптимизации, исходя из этого предположения. Листинг 9-13 показывает пример.

Листинг 9-13. `array_param_size_static.c`

```
int fun(int array[static 10] ) {...}
```

9.1.8 Инициализация массивов по указанным позициям

C99 вводит интересный способ инициализации массивов. Можно неявно инициализировать массив значениями по умолчанию, кроме нескольких указанных позиций, для которых заданы другие значения. Например, чтобы инициализировать массив из восьми элементов `int` нулями, кроме индексов 1 и 5, которые будут хранить значения 15 и 29 соответственно, можно использовать следующий код:

```
int a[8] = { [5] = 29, [1] = 15 };
```

Порядок инициализации значения не имеет. Часто бывает полезно использовать значения `enum` или символьные значения в качестве индексов. Листинг 9-14 показывает пример.

Листинг 9-14. `designated_initializers_arrays.c`

```
int whitespace[256] = {
    [' ' ] = 1,
    ['\t'] = 1,
    ['\f'] = 1,
    ['\n'] = 1,
    ['\r'] = 1 };
enum colors {
    RED,
    GREEN,
    BLUE,
    MAGENTA,
    YELLOW
};
int good[5] = { [ RED ] = 1, [ MAGENTA ] = 1 };
```

9.1.9 Псевдонимы типов

Вы можете определять собственные типы на основе существующих с помощью ключевого слова `typedef`. Код, показанный в листинге 9-15, создаёт новый тип `mytype_t`. Он абсолютно эквивалентен `unsignedshort int`, за исключением своего имени. Эти два типа становятся полностью взаимозаменяемыми (если только позже кто-то не изменит `typedef`).

Листинг 9-15. `typedef_example.c`

```
typedef unsigned short int mytype_t;
```

Суффикс `_t` в именах типов встречается довольно часто. Все имена, заканчивающиеся на `_t`, зарезервированы стандартом POSIX.[24] Так более новые стандарты смогут вводить новые типы, не опасаясь коллизий с типами в существующих проектах. Поэтому использовать такие имена типов не рекомендуется. Мы поговорим о практических соглашениях об именовании позже. Для чего же нужны эти новые типы?

1. Иногда они упрощают чтение кода.
2. Они могут повышать переносимость, потому что для изменения формата всех переменных вашего пользовательского типа достаточно изменить только `typedef`.
3. Типы, по сути, — ещё один способ документирования программы.
4. Псевдонимы типов чрезвычайно полезны при работе с типами указателей на функции из-за их громоздкого синтаксиса.

Очень важный пример псевдонима типа — `size_t`. Это тип, определённый в стандарте языка (для его использования требуется подключить один из заголовков стандартной библиотеки, например, `#include <stddef.h>`). Его назначение — хранить длины массивов и индексы массивов. Обычно это псевдоним для `unsigned long`; таким образом, в Intel 64 это, как правило, беззнаковое 8-байтовое целое число.

> **Никогда не используйте `int` для индексов массивов** Если только вы не имеете дело с плохо спроектированной библиотекой, вынуждающей вас использовать `int` в качестве индекса, всегда предпочитайте `size_t`.

Всегда используйте типы уместно. Большинство функций стандартной библиотеки, работающих с размерами, возвращают значение типа `size_t` (даже оператор `sizeof()` возвращает `size_t`!). Взглянем на пример, показанный в листинге 9-16. Выражение `s` типа `size_t` могло быть получено из одного из библиотечных вызовов, например, `strlen`. Использование `int` порождает несколько проблем:

- `int` занимает 4 байта и является знаковым, так что его максимальное значение — 2^{31} – 1. Что если `i` используется как

индекс массива? На современных системах вполне возможно создать массив побольше, так что не все элементы могут быть проиндексированы. Стандарт гласит, что размер массивов ограничен количеством элементов, кодируемым переменной типа `size_t` (беззнаковое 64-битное целое число).

- Каждая итерация выполняется, только если текущее значение `i` меньше `s`. Таким образом,

нужно сравнение, но у этих двух переменных разный формат! Из-за этого на каждой итерации будет выполняться специальный код преобразования чисел, что может быть весьма значимо для маленьких циклов с большим числом итераций.

- При работе с битовыми массивами (не такая уж редкость) программист, скорее всего, будет вычислять `i/8`

для смещения байта в байтовом массиве и `i%8`, чтобы узнать, о каком именно бите идёт речь. Эти операции можно оптимизировать в сдвиги вместо настоящего деления, но только для беззнаковых целых чисел. Разница в производительности между сдвигами и «честным» делением радикальна.

Листинг 9-16. `size_int_difference.c`

```
size_t s;
int i;
...
for( i = 0; i < s; i++ ) {
    ...
}
```

9.1.10 Функция main ещё раз

Мы уже привыкли писать функцию `main`, служащую точкой входа, как функцию без параметров. Однако на самом деле она должна принимать два параметра: количество аргументов командной строки и массив самих аргументов. Что такое аргументы командной строки? Что ж, каждый раз, когда вы запускаете программу (например, `ls`), вы можете указать дополнительные аргументы, например, `ls -l -a`. Приложение `ls` будет запущено, и у него будет доступ к этим аргументам в его функции `main`. В этом случае

- `argv` будет содержать три указателя на последовательности `char`:

```
INDEX STRING
  0   "ls"
  1   "-l"
  2   "-a"
```

Оболочка разобьёт всю строку вызова на части по пробелам, табуляциям и символам новой строки, а загрузчик и стандартная библиотека C позаботятся о том, чтобы `main` получила эту информацию.

- `argc` будет равен `3`, поскольку это количество элементов в `argv`.

Листинг 9-17 показывает пример. Эта программа выводит все переданные аргументы, каждый на отдельной строке.

Листинг 9-17. `main_revisited.c`

```
#include <stdio.h>
int main( int argc, char* argv[] ) {
    int i;
    for( i = 0; i < argc; i++ )
        puts( argv[i] );
    return 0;
}
```

9.1.11 Оператор sizeof

Мы уже упоминали оператор `sizeof` в разделе 8.4.2. Он возвращает значение типа `size_t`, хранящее размер операнда в байтах. Например, `sizeof(long)` вернёт 8 на компьютерах x64. `sizeof` — *не* функция, потому что он должен вычисляться во время компиляции. У `sizeof` есть интересное применение: можно вычислить общий размер массива, *но только* если аргумент — сам этот массив. Листинг 9-18 показывает пример.

Листинг 9-18. `sizeof_array.c`

```
#include <stdio.h>
long array[] = { 1, 2, 3 };
int main(void) {
    printf( "%zu \n", sizeof( array    ) ); /* output: 24 */
    printf( "%zu \n", sizeof( array[0] ) ); /* output: 8 */
    return 0;
}
```

Заметьте, что нельзя использовать `sizeof`, чтобы получить размер массива, принятого функцией в качестве аргумента. Листинг 9-19 показывает пример. Эта программа выведет 8 на нашей архитектуре

Листинг 9-19. `sizeof_array_fun.c`

```
#include <stdio.h>
const int arr[] = {1, 2, 3, 4};
void f(int const arr[]) {
    printf("%zu\n", sizeof( arr ) );
}
int main( void ) {
    f(arr);
    return 0;
}
```

> **Какой спецификатор формата?** Начиная с C99, вы можете использовать спецификатор формата `%zu` для `size_t`. В более ранних версиях следует использовать `%lu`, что означает `unsigned long`.

> **Вопрос 159** Создайте примеры программ, чтобы изучить значения этих выражений:

- sizeof(void)
- sizeof(0)
- sizeof('x')
- sizeof("hello")

> **Вопрос 160** Каким будет значение `x`?

```
int x = 10;
size_t t = sizeof(x=90);
```

> **Вопрос 161** Как вычислить, сколько элементов хранит массив, с помощью `sizeof`?

9.1.12 Константные типы

Для любого типа `T` мы также можем использовать тип `T const` (или, эквивалентно, `const T`). Переменные такого типа нельзя изменить напрямую, так что они *неизменяемы*. Это означает, что такие данные должны быть инициализированы одновременно с объявлением. Листинг 9-20 показывает пример инициализации константных переменных и работы с ними.

Листинг 9-20. `const_def.c`

```
int a;
a = 42 ;      /* ok */
...
const int a; /* compilation error */
...
const int a = 42; /* ok */
a = 99;  /* compilation error, should not change constant value */
int const a = 42;  /* ok */
const int b = 99;  /* ok, const int === int const */
```

Интересно отметить, как модификатор `const` взаимодействует с модификатором-звёздочкой `*`. Тип читается справа налево, и в этом же порядке применяются модификаторы `const` и звёздочка. Вот варианты:

- `int const* x` означает «изменяемый указатель на неизменяемый `int`». Таким образом, `*x = 10` не допускается, но изменять сам `x` можно. Альтернативный синтаксис — `const int* x`.
- `int* const x = &y;` означает «неизменяемый указатель на изменяемый `int y`». Другими словами, `x` никогда не будет указывать ни на что, кроме y.
- Суперпозиция этих двух случаев: `int const* const x = &y;` — это «неизменяемый указатель на неизменяемый `int y`».

> **Простое правило** Модификатор `const` слева от звёздочки защищает данные, на которые мы указываем; модификатор `const` справа защищает сам указатель.

Сделать переменную константной — не панацея. Всё ещё есть способ её изменить. Продемонстрируем это для переменной `const int x` (см. листинг 9-21).

- Взять на неё указатель. Он будет иметь тип `const int*`.
- Привести этот указатель к `int*`.
- Разыменовать этот новый указатель. Теперь можно присвоить `x` новое значение.

Листинг 9-21. `const_cast.c`

```
#include <stdio.h>
int main(void) {
    const int x = 10;
    *( (int*)&x ) = 30;
    printf( "%d\n", x );
    return 0;
}
```

Эта техника настоятельно не рекомендуется, но она может понадобиться при работе с плохо спроектированным устаревшим кодом. Модификаторы `const` существуют неспроста, и если ваш код без них не компилируется, это никоим образом не оправдывает подобные трюки. Заметьте, что нельзя присвоить указатель `int const*` переменной `int*` (это верно для всех типов). Первый указатель гарантирует, что его содержимое никогда не изменится, тогда как второй — нет. Листинг 9-22 показывает пример.

Листинг 9-22. `const_discard.c`

```
int x;
int y;
int const* px = &x;
int * py = &y;
py = px; /* Error, const qualifier is discarded */
px = py; /* OK  */
```

> **Стоит ли вообще использовать `const`? Это громоздко.** Безусловно. В крупных проектах это может сэкономить вам уйму времени на отладку. Я сам помню несколько очень тонких ошибок, пойманных компилятором и приведших к ошибке компиляции. Без защиты переменных с помощью `const` компилятор принял бы программу, что привело бы к неправильному поведению. Кроме того, компилятор может использовать эту информацию для выполнения полезных оптимизаций.

9.1.13 Строки

В C строки завершаются нулём. Одиночный символ представлен своим ASCII-кодом типа `char`. Строка определяется указателем на своё начало, а значит, эквивалентом типа строки был бы `char*`. О строках также можно думать как о массивах символов, чей последний элемент всегда равен нулю. Тип строковых литералов — `char*`. Их изменение, впрочем, хотя и возможно синтаксически (например, `"hello"[1] = 32`), даёт неопределённый результат. Это один из случаев *неопределённого поведения* в C. Обычно

это приводит к ошибке во время выполнения, которую мы объясним в следующей главе. Когда два строковых литерала написаны один за другим, они конкатенируются (даже если разделены переводами строк). Листинг 9-23 показывает пример.

Листинг 9-23. `string_literal_breaks.c`

```
char const* hello = "Hel" "lo"
"world!";
```

> **Примечание** Язык C++ (в отличие от C) принудительно задаёт типу строковых литералов `char const*`, так что если вы хотите, чтобы ваш код был переносимым, учтите это. Кроме того, он принудительно обеспечивает неизменяемость строк (что часто желательно) на уровне синтаксиса. Так что, когда это возможно, присваивайте строковые литералы переменным типа `const char*`.

9.1.14 Функциональные типы

Довольно малоизвестная часть C — функциональные типы. В отличие от большинства типов, их нельзя инстанцировать как переменные, но в некотором смысле сами функции являются литералами этих типов. Однако вы можете объявлять аргументы функций функциональных типов, которые автоматически будут преобразованы в указатели на функции. Листинг 9-24 показывает пример аргумента функции `f` функционального типа.

Листинг 9-24. `fun_type_example.c`

```
#include <stdio.h>
double g( int number ) { return 0.5 + number; }
double apply( double (f)(int), int x ) {
    return f( x ) ;
}
int main( void ) {
    printf( "%f\n",  apply( g, 10 ) );
    return 0;
}
```

Синтаксис, как видите, довольно своеобразный. Объявление типа смешано с самим именем аргумента, так что общий шаблон такой:

```
return_type (pointer_name) ( arg1, arg2, ... )
```

Эквивалентную программу вы видите в листинге 9-25.

Листинг 9-25. `fun_type_example_alt.c`

```
#include <stdio.h>
double g( int number ) { return 0.5 + number; }
double apply( double (*f)(int), int x ) {
    return f( x ) ;
}
int main( void ) {
    printf( "%f\n",  apply( g, 10 ) );
    return 0;
}
```

Для чего полезны эти типы? Поскольку типы указателей на функции довольно трудно писать и читать, их часто прячут в `typedef`. Плохая (но очень распространённая) практика — добавлять звёздочку *внутри* объявления псевдонима типа. Листинг 9-26 показывает пример, где создаётся тип для процедуры, ничего не возвращающей.

Листинг 9-26. `typedef_bad_fun_ptr.c`

```
Typedef  void(*proc)(void);
```

В этом случае можно написать прямо `proc my_pointer = &some_proc`. Однако это скрывает информацию о том, что `proc` — указатель: это можно вывести, но не видно сразу, а это плохо. Природа языка C, конечно, в том, чтобы абстрагировать вещи насколько возможно, но указатели — настолько фундаментальное понятие и настолько распространены в C, что их не стоит абстрагировать, особенно при наличии *слабой типизации*. Так что лучшим решением было бы написать то, что показано в листинге 9-27.

Листинг 9-27. `typedef_good_fun_ptr.c`

```
typedef void(proc)(void);
...
proc*  my_ptr  =  &some_proc;
```

Кроме того, эти типы можно использовать для записи объявлений функций. Листинг 9-28 показывает пример.

Листинг 9-28. `fun_types_decl.c`

```
typedef double (proc)(int);
/* declaration */
proc myproc;
/* ... */
/* definition */
double myproc( int x ) { return 42.0 + x; }
```

9.1.15 Хороший код

9.1.15.1 Общие соображения

В этой книге мы дадим несколько заданий, которые нужно будет написать на C. Но сначала мы хотим сформулировать несколько правил, которым вам следует следовать не только здесь и сейчас, но практически каждый раз, когда вы пишете программу.

1. Всегда отделяйте логику программы от операций ввода и вывода. Это

позволит лучше повторно использовать код. Если функция одновременно выполняет действия над данными и выводит сообщения, вы не сможете повторно использовать её логику в другой ситуации (например, она могла бы выводить сообщения в приложение с графическим интерфейсом, а в другом случае вы захотите использовать её на удалённом сервере).

2. Всегда комментируйте свой код на понятном языке.
3. Называйте переменные исходя из их смысла для программы. Очень трудно

понять, что означают переменные с бессмысленными именами вроде `aaa`.

4. Не забывайте ставить `const`, где только можете.
5. Используйте подходящие типы для индексации.

9.1.15.2 Пример: суммирование массива

Этот раздел *обязателен к прочтению*, если вы новичок в C, и тем более, если вы самоучка. Мы напишем простую программу в «начинающем стиле», посмотрим, что с ней не так, и подходящим образом изменим её, чтобы сделать лучше. Вот задача: реализовать функциональность суммирования массива. Какой бы простой она ни была, есть огромная разница между решением, написанным новичком, и решением более опытного программиста. Новичок придумает программу, похожую на показанную в листинге 9-29.

Листинг 9-29. `beg1.c`

```
#include <stdio.h>
int array[] = {1,2,3,4,5};
int main( int argc, char** argv ) {
    int i;
    int sum;
    for( i = 0; i < 5; i++ )
        sum = sum + array[i];
    printf("The sum is: %d\n", sum );
    return 0;
}
```

Прежде чем начать полировать код, мы можем сразу заметить ошибку: начальное значение `sum` не определено и может быть случайным. Локальные переменные в C по умолчанию не инициализируются, так что нужно делать это вручную. Смотрите листинг 9-30.

Листинг 9-30. `beg2.c`

```
#include <stdio.h>
int array[] = {1,2,3,4,5};
int main( int argc, char** argv ) {
    int i;
    int sum = 0;
    for( i = 0; i < 5; i++ )
        sum = sum + array[i];
    printf("The sum is: %d\n", sum );
    return 0;
}
```

Прежде всего, этот код совершенно не пригоден для повторного использования. Давайте вынесем часть логики в процедуру `array_sum`, показанную в листинге 9-31.

Листинг 9-31. `beg3.c`

```
#include <stdio.h>
int array[] = {1,2,3,4,5};
void array_sum( void ) {
    int i;
    int sum = 0;
    for( i = 0; i < 5; i++ )
        sum = sum + array[i];
    printf("The sum is: %d\n", sum );
}
int main( int argc, char** argv ) {
    array_sum();
    return 0;
}
```

Что это за магическое число 5? Каждый раз, когда мы меняем массив, нам приходится менять и это число, так что, вероятно, мы хотим вычислять его динамически, как показано в листинге 9-32.

Листинг 9-32. `beg4.c`

```
#include <stdio.h>
int array[] = {1,2,3,4,5};
void array_sum( void ) {
    int i;
    int sum = 0;
    for( i = 0; i < sizeof(array) / 4; i++ )
        sum = sum + array[i];
    printf("The sum is: %d\n", sum );
}
int main( int argc, char** argv ) {
    array_sum();
    return 0;
}
```

Но зачем мы делим размер массива на 4? Размер `int` зависит от архитектуры, так что нам нужно вычислять и его (во время компиляции), как показано в листинге 9-33.

Листинг 9-33. `beg5.c`

```
#include <stdio.h>
int array[] = {1,2,3,4,5};
void array_sum( void ) {
    int i;
    int sum = 0;
    for( i = 0; i < sizeof(array) / sizeof(int); i++ )
        sum = sum + array[i];
    printf("The sum is: %d\n", sum );
}
int main( int argc, char** argv ) {
    array_sum();
    return 0;
}
```

Мы сразу же сталкиваемся с проблемой: `sizeof` возвращает число типа `size_t`, а не `int`. Так что нам нужно изменить тип `i`, и мы делаем это не просто так (см. раздел 9.1.9). Листинг 9-34 показывает результат.

Листинг 9-34. `beg6.c`

```
#include <stdio.h>
int array[] = {1,2,3,4,5};
void array_sum( void ) {
    size_t i;
    int sum = 0;
    for( i = 0; i < sizeof(array) / sizeof(int); i++ )
        sum = sum + array[i];
    printf("The sum is: %d\n", sum );
}
int main( int argc, char** argv ) {
    array_sum();
    return 0;
}
```

Прямо сейчас `array_sum` работает только со статически определёнными массивами, потому что только для них размер можно вычислить с помощью `sizeof`. Далее мы хотим добавить в `array_sum` достаточно параметров, чтобы она могла суммировать любой массив. Нельзя добавить только указатель на массив, потому что размер массива по умолчанию неизвестен, так что вы даёте ей два параметра: сам массив и количество элементов в массиве, как показано в листинге 9-35.

Листинг 9-35. `beg7.c`

```
#include <stdio.h>
int array[] = {1,2,3,4,5};
void array_sum( int* array, size_t count ) {
    size_t i;
    int sum = 0;
    for( i = 0; i < count; i++ )
        sum = sum + array[i];
    printf("The sum is: %d\n", sum );
}
int main( int argc, char** argv ) {
    array_sum(array, sizeof(array) / sizeof(int));
    return 0;
}
```

Этот код намного лучше, но он всё ещё нарушает правило не смешивать ввод/вывод и логику. Вы не можете использовать `array_sum` где-либо в графических программах, вы также ничего не можете сделать с её результатом. Мы избавимся от вывода в функции суммирования и заставим её возвращать результат. Смотрите листинг 9-36.

Листинг 9-36. `beg8.c`

```
#include <stdio.h>
int g_array[] = {1,2,3,4,5};
int array_sum( int* array, size_t count ) {
    size_t i;
    int sum = 0;
    for( i = 0; i < count; i++ )
        sum = sum + array[i];
    return sum;
}
int main( int argc, char** argv ) {
    printf(
            "The sum is: %d\n",
            array_sum(g_array, sizeof(g_array) / sizeof(int))
         );
    return 0;
}
```

Для удобства мы переименовали глобальную переменную `array` в `g_array`, но это не обязательно. Наконец, нам нужно подумать о добавлении квалификаторов `const`. Самое важное место — аргументы функций типа указателя. Мы действительно

хотим заявить, что `array_sum` никогда не изменит массив, на который указывает её аргумент. Нам также может понравиться идея защитить саму глобальную переменную `array` от изменения, добавив квалификатор `const`. Помните, что если мы сделаем константной саму `g_array`, но не пометим так же `array` в списке аргументов, мы не сможем передать `g_array` в `array_sum`, потому что нет гарантий, что `array_sum` не изменит данные, на которые указывает её аргумент. Листинг 9-37 показывает итоговый результат.

Листинг 9-37. `beg9.c`

```
#include <stdio.h>
const int g_array[] = {1,2,3,4,5};
int array_sum( const int* array, size_t count ) {
    size_t i;
    int sum = 0;
    for( i = 0; i < count; i++ )
        sum = sum + array[i];
    return sum;
}
int main( int argc, char** argv ) {
    printf(
            "The sum is: %d\n",
            array_sum(g_array, sizeof(g_array) / sizeof(int))
         );
    return 0;
}
```

Когда вы пишете решение для задания из этой книги, помните все перечисленные выше пункты и проверяйте, соответствует ли им ваша программа, а если нет — как её можно улучшить. Можно ли улучшить эту программу ещё больше? Конечно, и мы дадим вам несколько подсказок, как именно.

- Может ли указатель array быть `NULL?` Если да, как нам сигнализировать об этом, не разыменовывая

указатель `NULL`, что, вероятно, приведёт к аварийному завершению?

- Может ли `sum` переполниться?

9.1.16 Задание: скалярное произведение

Скалярное произведение двух векторов (a_1, a_2, ..., a_n) и (b_1, b_2, ..., b_n) — это сумма:

$$\mathbf{a \cdot b = a_1b_1 + a_2b_2 + ... + a_nb_n}$$

Например, скалярное произведение векторов (1, 2, 3) и (4, 5, 6) равно

1 . 4 + 2 . 5 + 3 . 6 = 4 + 10 + 18 = 32 Решение должно состоять из

- Двух глобальных массивов `int` одинакового размера.
- Функции для вычисления скалярного произведения двух заданных массивов.
- Функции `main`, вызывающей вычисление произведения и выводящей его результат.

9.1.17 Задание: проверка числа на простоту

Вам нужно написать функцию для проверки числа на простоту. Интересно то, что число будет иметь тип `unsigned long` и будет читаться из `stdin`.

- Вам нужно написать функцию `int is_prime( unsigned long n )`, проверяющую, является ли `n` простым числом. Если да, функция вернёт 1; иначе 0.
- Функция `main` прочитает число `unsigned long` и вызовет для него функцию `is_prime`. Затем, в зависимости от результата, она выведет либо `yes`, либо `no`.

Прочитайте `man scanf` и используйте функцию `scanf` со спецификатором формата `%lu`. Помните, `is_prime` принимает `unsigned long`, что не то же самое, что `unsigned int!`

9.2 Помеченные типы

В C есть три «помеченных» вида типов: структуры, объединения и перечисления. Мы называем их так, потому что их имена состоят из ключевого слова `struct`, `union` или `enum`, за которым следует мнемоническая метка, например, `struct`

```
pair or union pixel.
```

9.2.1 Структуры

Абстракция абсолютно ключевая для всего программирования. Она заменяет более низкоуровневые, более многословные понятия теми, что ближе к нашему мышлению: более высокоуровневыми, менее многословными. Когда вы думаете о походе в любимую пиццерию и планируете оптимальный маршрут, вы не думаете о «перемещении правой ноги на *X* сантиметров вперёд», а скорее о «переходе через дорогу» или «повороте направо». В то время как для логики программы механизм абстракции реализуется с помощью функций, абстракция данных реализуется с помощью составных типов данных. Структура — это тип данных, упаковывающий несколько полей. Каждое поле — переменная своего собственного типа. Математика, вероятно, была бы рада называть структуры «кортежами с именованными полями». Чтобы создать переменную структурного типа, можно обратиться к примеру, показанному в листинге 9-38. Там мы определяем пере-

менную `d`, у которой два поля: `a` и `b` типов `int` и `char` соответственно. Тогда `d.a` и `d.b` становятся допустимыми выражениями, которые можно использовать точно так же, как имена переменных.

Листинг 9-38. `struct_anon.c`

```
struct { int a; char b; } d;
d.a = 0;
d.b = 'k';
```

Однако таким образом вы создаёте лишь одноразовую структуру. По сути, вы описываете тип `d`, но не создаёте новый именованный структурный тип. Последнее можно сделать с помощью синтаксиса, показанного в листинге 9-39.

Листинг 9-39. `struct_named.c`

```
struct pair {
    int a;
    int b;
};
...
struct pair d;
d.a = 0;
d.b = 1;
```

Имейте в виду, что имя типа — это *не* `pair`, а `struct pair`, и *нельзя* опустить ключевое слово `struct`, не запутав компилятор. У языка C есть понятие *пространств имён*, довольно отличное от пространств имён в других языках (включая C++). Есть глобальное пространство имён типов, а также пространство имён меток, общее для типов данных `struct`, `union` и `enum`. Имя, следующее за ключевым словом `struct`, — это метка. Можно определить структурный тип, чьё имя совпадает с именем другого типа, и компилятор будет различать их по наличию ключевого слова `struct`. Пример, показанный в листинге 9-40, демонстрирует две переменные типов `struct type` и `type`, которые прекрасно принимаются компилятором.

Листинг 9-40. `struct_namespace.c`

```
typedef unsigned int type;
struct type {
    char c;
};
int main( int argc, char** argv ) {
    struct type st;
    type t;
    return 0;
}
```

Это, впрочем, не значит, что вам действительно стоит создавать типы с похожими именами. Однако, поскольку `struct type` — вполне корректное имя типа, для него можно создать псевдоним `type` с помощью ключевого слова `typedef`, как показано в листинге 9-41. Тогда имена `type` и `struct type` станут полностью взаимозаменяемыми.

Листинг 9-41. `typedef_struct_simple.c`

```
typedef struct type type;
```

> **Пожалуйста, не делайте так** Создавать псевдонимы структурных типов с помощью `typedef` — плохая практика, потому что это скрывает информацию о природе типа.

Структуры можно инициализировать аналогично массивам (см. листинг 9-42).

Листинг 9-42. `struct_init.c`

```
struct S {char const* name; int value; };
...
struct S new_s = { "myname", 4 };
```

Можно также присвоить 0 всем полям структуры, как показано в листинге 9-43.

Листинг 9-43. `struct_zero.c`

```
struct pair { int a; int b; };
...
struct pair p = { 0 };
```

В C99 есть лучший синтаксис для инициализации структур, позволяющий именовать поля для инициализации. Неупомянутые поля будут инициализированы нулями. Листинг 9-44 показывает пример.

Листинг 9-44. `struct_c99_init.c`

```
struct pair {
    char a;
    char b;
};
struct pair st = { .a = 'a',.b = 'b' };
```

Поля структур гарантированно не перекрываются; однако, в отличие от массивов, структуры не непрерывны в том смысле, что между их полями может быть свободное пространство. Таким образом, `sizeof` структурного типа может быть больше суммы размеров элементов из-за таких промежутков. Мы поговорим об этом в Главе 12.

9.2.2 Объединения

Объединения очень похожи на структуры, но их поля всегда перекрываются. Другими словами, все поля объединения начинаются с одного и того же адреса. Объединения делят своё пространство имён со структурами и перечислениями. Листинг 9-45 показывает пример.

Листинг 9-45. `union_example.c`

```
union dword {
    int integer;
    short shorts[2];
};
...
dword test;
test.integer = 0xAABBCCDD;
```

Мы только что определили объединение, хранящее число размером 4 байта (на архитектурах x86 или x64). В то же время оно хранит массив из двух чисел, каждое шириной 2 байта. Эти два поля (4-байтовое число и пара 2-байтовых чисел) перекрываются. Изменяя поле `.integer`, мы также изменяем массив `.shorts`. Если мы присвоим `.integer = 0xAABBCCDD`, а затем попробуем вывести `shorts[0]` и `shorts[1]`, мы увидим `ccdd aabb`.

> **Вопрос 162** Почему эти shorts кажутся перевёрнутыми? Будет ли так всегда, или это зависит от архитектуры?

Смешивая структуры и объединения, можно достичь интересных результатов. Пример, показанный в листинге 13-17, демонстрирует, как можно обращаться к частям 3-байтовой структуры с помощью индексов.[25]

Листинг 9-46. `pixel.c`

```
union pixel {
    struct {
        char a,b,c;
    };
    char at[3];
};
```

Помните, что если вы присвоили значение полю объединения, стандарт не гарантирует вам ничего о значениях других полей. Исключение сделано для структур, имеющих одинаковую начальную последовательность полей. Листинг 9-47 показывает пример.

Листинг 9-47. `union_guarantee.c`

```
struct sa {
    int x;
    char y;
    char z;
};
struct sb {
    int x;
    char y;
    int notz;
};
union test {
    struct sa as_sa;
    struct sb as_sb;
};
```

9.2.3 Анонимные структуры и объединения

Начиная с C11, объединения и структуры могут быть анонимными, если находятся внутри других структур или объединений. Это позволяет использовать менее многословный синтаксис при обращении к внутренним полям. В примере, показанном

в листинге 9-48, чтобы обратиться к полю `x` из `vec`, нужно написать `vec.named.x`. Нельзя опустить `named`.

Листинг 9-48. `anon_no.c`

```
union vec3d {
    struct {
        double x;
        double y;
        double z;
    } named ;
    double raw[3];
};
union vec3d vec;
```

Теперь, в следующем примере, показанном в листинге 9-49, мы избавились от имени первого поля (`named`). Это анонимная структура, и теперь мы можем обращаться к её полям так, будто это поля самой `vec`: `vec.x`.

Листинг 9-49. `anon_struct.c`

```
union vec3d {
    struct {
        double x;
        double y;
        double z;
    };
    double raw[3];
};
union vec3d vec;
```

9.2.4 Перечисления

Перечисления — это простой тип данных, основанный на типе `int`. Он фиксирует определённые значения и даёт им имена, похоже на то, как работает `DEFINE`. Например, светофор может находиться в одном из следующих состояний (в зависимости от того, какие огни включены):

- Красный.
- Красный и жёлтый.
- Жёлтый.
- Зелёный.

- Нет огней.

Это можно закодировать в C, как показано в листинге 9-50.

Листинг 9-50. `enum_example.c`

```
enum light {
    RED,
    RED_AND_YELLOW,
    YELLOW,
    GREEN,
    NOTHING
};
...
enum light l = nothing;
...
```

Когда это полезно? Это часто используется для кодирования состояния сущности, например, как часть конечного автомата; это может служить набором кодов ошибок или мнемоник кодов. Константное значение 0 получило имя `RED`, `RED_AND_YELLOW` означает 1 и т. д.

9.3 Типы данных в языках программирования

Мы дали обзор типов данных в C; теперь давайте отступим на шаг назад от C и посмотрим на более широкую картину и виды систем типов в языках программирования. Во многих областях компьютерных наук и программирования эволюция шла от нетипизированной вселенной к типизации. Например, следующие сущности нетипизированы:

1. Лямбда-термы в нетипизированном лямбда-исчислении;
2. Множества во многих теориях множеств, например, ZF;
3. S-выражения в языке LISP; и
4. Битовые строки.

Сейчас нас в первую очередь интересуют битовые строки. Для компьютера всё — это битовая строка некоторого фиксированного размера. Их можно интерпретировать как числа (целые или вещественные), последовательности кодов символов или что-то ещё. Можно сказать, что ассемблер — нетипизированный язык. Однако, когда мы начинаем работать в нетипизированной среде, мы стараемся разделить объекты на несколько категорий. Мы работаем с объектами одной категории схожим образом. Так что мы устанавливаем соглашение: эти битовые строки — целые числа, те — числа с плавающей точкой и т. д. Это уже типизация? Ещё не совсем. Мы всё ещё не ограниче-

ны в своих возможностях и можем прибавить число с плавающей точкой к указателю на строку, потому что язык программирования не обеспечивает никакого контроля типов. Эта проверка типов может выполняться во время компиляции (*статическая типизация*) или во время выполнения (*динамическая типизация*). Так что мы не только разделяем все виды возможных объектов на категории, но и объявляем, какие операции можно выполнять над каждым типом. Данные разных типов также часто кодируются по-разному.

9.3.1 Виды типизации

Помимо статической и динамической типизации, есть и другие, ортогональные классификации. *Сильная типизация* означает, что все операции требуют ровно тех аргументов, которые им нужны. Никакие неявные преобразования из других типов в нужные не допускаются. *Слабая типизация* означает, что существуют неявные преобразования между типами, делающие возможными операции над данными не совсем требуемого типа (но преобразование к требуемому типу существует). Это разделение не строго бинарно; в реальном мире языки скорее тяготеют к одному из этих двух полюсов. Есть довольно крайние случаи, такие как Ada для сильной типизации и JavaScript для слабой. Иногда мы также делим языки по многословности. При *явной типизации* мы всегда сопровождаем данные аннотациями типов. При *неявной типизации* мы позволяем компилятору выводить тип, когда это возможно. Теперь мы приведём реальные примеры всех комбинаций статической/динамической и сильной/слабой типизации.

9.3.1.1 Статическая сильная типизация

Типы проверяются во время компиляции, и компилятор к ним педантичен. В языке OCaml есть два разных оператора сложения: `+` для целых чисел и `+.` для вещественных. Так что этот код вызовет ошибку во время компиляции:

```
4 +. 1.0
```

Мы использовали данные типа `int`, когда компилятор ожидал `float`, и, в отличие от C, где произошло бы преобразование, он выбросил ошибку. В этом суть очень сильной типизации.

9.3.1.2 Статическая слабая типизация

У языка C именно такая типизация. Все типы известны во время компиляции, но неявные преобразования происходят довольно часто. Почти идентичная строка `double x = 4 + 3.0;` не вызывает ошибок компилятора, потому что 4 автоматически повышается до `double`, а затем складывается с `3.0`. Слабость выражается в том, что программист не указывает операции преобразования явно.

9.3.1.3 Сильная динамическая типизация

Такой тип типизации используется в Python. Python не допускает неявных преобразований между типами так же охотно, как JavaScript. Однако ошибки типов не будут сообщены, пока вы не запустите программу и действительно не попытаетесь выполнить ошибочный оператор. У Python есть интерпретатор, где можно вводить выражения и операторы и сразу же их выполнять. Если вы попробуете вычислить выражение `"3" + 2` и увидеть его результат в интерактивном интерпретаторе Python, вы получите ошибку, потому что первый объект — строка, а второй — число. Хотя эта строка содержит число (так что преобразование можно было бы написать), сложение не допускается. Листинг 9-51 показывает дамп.

Листинг 9-51. Python Typing Error

```
>>> "3" + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
```

Теперь попробуем вычислить выражение `1 if True else "3" + 2`. Это выражение вычисляется в 1, если `True` истинно (что, очевидно, верно); иначе его значение — результат той же недопустимой операции `"3" + 2`. Однако, поскольку мы никогда не попадаем в ветвь `else`, ошибка не будет выброшена даже во время выполнения. Листинг 9-52 показывает дамп терминала. При применении к двум строкам плюс действует как оператор конкатенации.

Листинг 9-52. Python Typing: No Error Because the Statement Is Not Executed

```
>>> 1 if True else "3" + 2
1
>>> "1" + "2"
'12'
```

9.3.1.4 Слабая динамическая типизация

Пожалуй, самый используемый язык с такой типизацией — JavaScript. В примере, который мы привели для Python, мы попытались прибавить число к строке. Несмотря на то что строка содержала корректное десятичное число, была выдана ошибка, потому что строка есть строка, что бы она ни содержала. Её тип не изменится автоматически. Однако JavaScript намного менее строг в том, что вам разрешено делать. Мы воспользуемся интерактивной консолью JavaScript (доступной практически в любом современном веб-браузере) и введём несколько выражений. Листинг 9-53 показывает результат.

Листинг 9-53. JavaScript Implicit Conversions

```
>>> 3 == '3'
true
>>> 3 == '4'
false
>>> "7.0" == 7
true
```

Изучив только этот пример, можно сделать вывод, что при сравнении числа и строки обе стороны, судя по всему, преобразуются в число, а затем сравниваются. Неясно, целые это числа или вещественные, но количество задействованных здесь неявных операций весьма впечатляет.

9.3.2 Полиморфизм

Теперь, когда у нас есть общее понимание типизации, перейдём к одному из важнейших понятий, связанных с системами типов, а именно к полиморфизму. *Полиморфизм* (от греч. polys — «много» и morphe — «форма, вид») — это возможность единообразно вызывать разные действия для разных типов. Можно думать об этом и иначе: сущности данных могут принимать разные типы. Есть четыре разных вида полиморфизма [8], которые также можно разделить на две категории:

1. Универсальный полиморфизм, когда функция принимает аргумент бесконечного

числа типов (включая, возможно, даже ещё не определённые) и ведёт себя схожим образом для каждого из них.

- Параметрический полиморфизм, когда функция принимает дополнительный аргумент, определяющий тип другого аргумента. В таких языках, как Java или C#, обобщённые функции — пример параметрического полиморфизма времени компиляции.
- Включение (inclusion), когда некоторые типы являются подтипами других типов. Так что при получении аргумента дочернего типа функция будет вести себя так же, как при передаче родительского типа.

2. Ad hoc, когда функции принимают параметр из фиксированного набора типов, и эти

функции могут по-разному оперировать на каждом типе.

- Перегрузка, когда существует несколько функций с одинаковым именем, и одна из них вызывается в зависимости от типа аргумента.
- Приведение (coercion), когда существует преобразование из типа *X* в тип *Y*, и функция, принимающая аргумент типа *Y*, вызывается с аргументом типа *X*.

Популярная парадигма объектно-ориентированного программирования популяризировала понятие полиморфизма, но в весьма специфичном смысле. Объектно-ориентированное программирование обычно подразумевает только один вид полиморфизма, а именно *подтипизацию (subtyping)*, которая, по сути, то же самое, что и включение, поскольку объекты дочернего типа образуют подмножество объектов родительского типа. Иногда трудно сказать, какой вид полиморфизма используется в конкретном месте. Рассмотрим следующие четыре строки:

```
3 + 4
3 + 4.0
3.0 + 4
3.0 + 4.0
```

Операция «плюс» здесь, очевидно, полиморфна, поскольку она используется одинаковым образом со всеми видами операндов `int` и `double`. Но как она на самом деле реализована? Можно представить разные варианты, например,

- У этого оператора четыре перегрузки для всех комбинаций.
- У этого оператора две перегрузки для случаев `int` + `int` и `double` + `double`. Кроме того, определено приведение из `int` в `double`.
- Этот оператор может складывать только два вещественных числа, а все `int` приводятся к `double`.

9.4 Полиморфизм в C

Язык C допускает разные виды полиморфизма, и некоторые можно эмулировать небольшими трюками.

9.4.1 Параметрический полиморфизм

Можем ли мы сделать функцию, которая будет вести себя по-разному для разных типов аргументов на основе явно заданного типа? В некоторой степени можем, даже в C89. Однако для гладкого результата нам понадобится довольно тяжёлая макро-машинерия. Сначала нужно узнать, что делает этот причудливый символ `#` в контексте макроса. Внутри макроса `# symbol` заключает содержимое `symbol` в кавычки. Листинг 9-54 показывает пример.

Листинг 9-54. `macro_str.c`

```
#define mystr hello
#define res #mystr
puts( res );  /* will be replaced with `puts("hello")` 
```

Оператор ## ещё интереснее. Он позволяет нам динамически формировать имена символов. Листинг 9-55 показывает пример.

Листинг 9-55. `macro_concat.c`

```
#define x1 "Hello"
#define x2 " World"
#define str(i) x##i
puts( str(1) );  /* str(1) -> x1 -> "Hello" */
puts( str(2) );  /* str(2) -> x2 -> " World" */
```

Некоторые возможности языков более высокого уровня можно свести к логике компилятора, анализирующего программу и вызывающего ту или иную функцию, использующего ту или иную структуру данных и т. д. В C мы можем имитировать это, полагаясь на препроцессор. Листинг 9-56 показывает пример.

Листинг 9-56. `c_parametric_polymorphism.c`

```
#include <stdio.h>
#include <stdbool.h>
#define pair(T) pair_##T
#define DEFINE_PAIR(T) struct pair(T) {\
    T fst;\
    T snd;\
};\
bool pair_##T##_any(struct pair(T) pair, bool (*predicate)(T)) {\
    return predicate(pair.fst) || predicate(pair.snd); \
}
#define any(T) pair_##T##_any
DEFINE_PAIR(int)
bool is_positive( int x ) { return x > 0; }
int main( int argc, char** argv ) {
    struct pair(int) obj;
    obj.fst = 1;
    obj.snd = -1;
    printf("%d\n", any(int)(obj, is_positive) );
    return 0;
}
```

Сначала мы подключили файл `stdbool.h`, чтобы получить доступ к типу `bool`, как мы говорили в разделе 9.1.3.

- `pair(T)`, при вызове вот так: `pair(int)`, будет заменён строкой `pair_int.`
- `DEFINE_PAIR` — макрос, который при вызове вот так: `DEFINE_PAIR(int)` будет

заменён кодом, показанным в листинге 9-57.

Обратите внимание на обратные слэши в конце каждой строки: они используются для экранирования символа новой строки, тем самым позволяя этому макросу занимать несколько строк. Последняя строка макроса не заканчивается обратным слэшем.

Этот код определяет новый структурный тип `struct pair_int`, по сути содержащий два целых числа в качестве полей. Если бы мы инстанцировали этот макрос с параметром, отличным от `T`, у нас получилась бы пара элементов другого типа.

Затем определяется функция, у которой для каждого инстанцирования макроса будет своё особое имя, поскольку имя параметра `T` закодировано в её имени. В нашем случае это `pair_int_any`, чьё назначение — проверить, удовлетворяет ли хотя бы один из двух элементов пары условию. Она принимает саму пару в качестве первого аргумента

и условие в качестве второго. Условие — это, по сути, указатель на функцию, принимающую `T` и возвращающую `bool`, — предикат, как и следует из его названия. `pair_int_any` запускает функцию `condition` на первом элементе, а затем на втором.

При использовании `DEFINE_PAIR` определяет структуру, хранящую два элемента заданного типа, и функции для работы с ней. У нас может быть только одна копия этих функций и определения структуры для каждого типа, но они нам нужны, так что мы хотим инстанцировать `DEFINE_PAIR` один раз для каждого типа, с которым хотим работать.

Листинг 9-57. `macro_define_pair.c`

```
struct pair_int {
    int fst;
    int snd;
};
bool pair_int_any(struct pair_int pair, bool (*predicate)(int)) {
    return predicate(pair.fst) || predicate(pair.snd);
}
```

- Затем определяется макрос `#define any(T) pair_##T##_any`. Заметьте, что его единственное назначение — по сути, просто сформировать корректное имя функции в зависимости от типа. Он позволяет нам вызывать `pair_##T##_any` довольно элегантным образом: `any(int)`, как будто это функция, возвращающая указатель на функцию.

Так что синтаксически мы очень близко подошли к понятию параметрического полиморфизма: мы предоставляем дополнительный аргумент (`int`), служащий для определения типа другого аргумента (`struct pair_int`). Конечно, это не так хорошо, как типовые аргументы в функциональных языках или даже обобщённые параметры типов в C# или Scala, но всё же что-то.

9.4.2 Включение

Включение довольно легко достижимо в C для типов указателей. Идея в том, что адрес любой структуры совпадает с адресом её первого члена. Взгляните на пример, показанный в листинге 9-58.

Листинг 9-58. `c_inclusion.c`

```
#include <stdio.h>
struct parent {
    const char* field_parent;
};
struct child {
    struct parent base;
    const char* field_child;
};
void parent_print( struct parent* this ) {
    printf( "%s\n", this->field_parent );
}
int main( int argc, char** argv ) {
    struct child c;
    c.base.field_parent = "parent";
    c.field_child = "child";
    parent_print( (struct parent*) &c );
    return 0;
}
```

Функция `parent_print` принимает аргумент типа `parent*`. Как следует из определения `child`, её первое поле имеет тип `parent`. Так что каждый раз, когда у нас есть действительный указатель `child*`, существует указатель на экземпляр `parent`, равный первому. Таким образом, безопасно передавать указатель на child там, где ожидается указатель на parent. Однако система типов об этом не знает; поэтому вам приходится преобразовывать указатель `child*` в `parent*`, как видно в вызове `parent_print( (struct parent*) &c );`. В этом случае мы могли бы заменить тип `struct parent*` на `void*`, потому что любой тип указателя можно преобразовать в `void*` (см. раздел 9.1.5).

9.4.3 Перегрузка

Автоматическая перегрузка была невозможна в C вплоть до C11. До недавнего времени люди включали имена типов аргументов в имена функций, чтобы обеспечить разные «перегрузки» при данном базовом имени. Теперь более новый стандарт включает специальный макрос, раскрывающийся в зависимости от типа аргумента: `_Generic`. У него широкий спектр применений. Макрос `_Generic` принимает выражение *E*, а затем множество ассоциативных выражений, разделённых запятой. Каждое выражение имеет форму `type name: string`. При инстанцировании тип *E* сравнивается со всеми типами в списке ассоциаций, и результатом инстанцирования становится соответствующая строка справа от двоеточия. В примере, показанном в листинге 9-59, мы определим макрос `print_fmt`, выбирающий подходящий спецификатор формата `printf` на основе типа аргумента, и макрос `print`, формирующий корректный вызов `printf`, а затем выводящий перевод

строки. `print_fmt` сопоставляет тип выражения `x` с одним из двух типов: `int` и `double`. Если тип `x` не входит в этот список, выполняется случай `default`, предоставляющий довольно общий спецификатор `%x`. Однако в отсутствие случая `default` программа не скомпилируется, если вы передадите `print_fmt` выражение типа, скажем, `long double`. Так что в этом случае, вероятно, было бы разумно просто опустить случай `default`, заставляя компиляцию прерваться, когда мы действительно не знаем, что делать.

Листинг 9-59. `c_overload_11.c`

```
#include <stdio.h>
#define print_fmt(x) (_Generic( (x), \
            int: "%d",\
            double: "%f",\
            default: "%x"))
#define print(x) printf( print_fmt(x), x ); puts("");
int main(void) {
    int x = 101;
    double y = 42.42;
    print(x);
    print(y);
    return 0;
}
```

Мы можем использовать `_Generic`, чтобы написать макрос, оборачивающий вызов функции и выбирающий одну из *по-разному названных* функций на основе типа аргумента.

9.4.4 Приведения

В сам язык C встроено несколько приведений. Мы говорим, по сути, о преобразованиях указателей в `void*` и обратно и о целочисленных преобразованиях, описанных в разделе 9.1.4. Насколько нам известно, нет способа добавить пользовательские приведения или что-либо, хотя бы отдалённо похожее на неявные функции Scala или неявные преобразования C++. Как видите, в некоторой форме C допускает все четыре вида полиморфизма.

9.5 Итоги

В этой главе мы подробно изучили систему типов C: массивы, указатели, константные типы. Мы научились создавать простые указатели на функции, увидели подводные камни `sizeof`, пересмотрели строки и начали привыкать к лучшим практикам написания кода. Затем мы изучили структуры, объединения и перечисления. В конце

мы кратко поговорили о системах типов в основных языках программирования и полиморфизме и привели несколько продвинутых примеров кода, демонстрирующих, как добиться схожих результатов на чистом C. В следующей главе мы подробнее рассмотрим способы организации кода в проект и свойства языка, важные в этом контексте.

Вопрос 163 Каково назначение операторов `&` и `*`?

Вопрос 164 Как нам прочитать целое число по адресу `0x12345`?

Вопрос 165 Какой тип имеет литерал `42`?

Вопрос 166 Как создать литерал типов `unsigned long`, `long` и `long long`?

Вопрос 167 Зачем нам нужен тип `size_t`?

Вопрос 168 Как преобразовать значения из одного типа в другой?

Вопрос 169 Есть ли в C89 булев тип?

Вопрос 170 Что такое тип указателя?

Вопрос 171 Что такое `NULL`?

Вопрос 172 Каково назначение типа `void*`?

Вопрос 173 Что такое массив?

Вопрос 174 Можно ли интерпретировать любые последовательные ячейки памяти как массив?

Вопрос 175 Что происходит при попытке обратиться к элементу за границами массива?

Вопрос 176 Какая связь между массивами и указателями?

Вопрос 177 Можно ли объявить указатель на функцию?

Вопрос 178 Как создать псевдоним для определённого типа?

Вопрос 179 Как аргументы передаются функции main?

Вопрос 180 Каково назначение оператора `sizeof` ?

Вопрос 181 Вычисляется ли `sizeof` во время выполнения программы?

Вопрос 182 Почему важно ключевое слово `const` ?

Вопрос 183 Что такое структурные типы и зачем они нужны?

Вопрос 184 Что такое типы-объединения? Чем они отличаются от структурных типов?

Вопрос 185 Что такое типы-перечисления? Чем они отличаются от структурных типов?

Вопрос 186 Какие виды типизации существуют?

Вопрос 187 Какие виды полиморфизма существуют и в чём разница между ними?

Структура кода

В этой главе мы изучим, как лучше разбивать код на несколько файлов и какие для этого существуют соответствующие возможности языка. Иметь один файл с мешаниной из функций и определений типов далеко не удобно для крупных проектов. Большинство программ разбиваются на несколько модулей. Мы изучим, какие преимущества это даёт и как выглядит каждый модуль перед компоновкой.

10.1 Объявления и определения

Компиляторы C исторически писались как однопроходные программы. Это значит, что им приходилось проходить файл один раз и сразу же его транслировать. Однако для нас это кое-что значит. Когда вызывается функция, которая ещё не определена, компилятор отвергнет такую программу, потому что не знает, что означает это имя. Хотя мы осознаём своё намерение вызвать функцию в этом месте, для компилятора это просто неопределённый идентификатор, и из-за однопроходной трансляции компилятор не может заглянуть вперёд и попытаться найти определение. В простых случаях линейной зависимости мы можем просто определить все функции до их использования. Однако бывают случаи циклических зависимостей, когда эта схема не работает, а именно взаимно рекурсивные определения, будь то структуры или функции. В случае функций это две функции, вызывающие друг друга. Очевидно, что в каком бы порядке мы их ни определяли, мы не можем определить обе до того, как компилятор увидит вызов одной из них. Листинг 10-1 показывает пример.

Листинг 10-1. `fun_mutual_recursive_bad.c`

```
void f(void) {
    g();   /* What is `g`, asks mr. Compiler? */
}
void g(void) {
    f();
}
```

В случае структур речь идёт о двух структурных типах. У каждого из них есть поле типа указатель, указывающее на экземпляр другой структуры. Листинг 10-2 показывает пример.

Листинг 10-2. `struct_mutual_recursive_bad.c`

```
struct a {
    struct b* foo;
};
struct b {
    struct a* bar;
};
```

Решение — использовать раздельные объявления и определения. Когда объявление предшествует определению, это называется *опережающим объявлением (forward declaration)*.

10.1.1 Объявления функций

Для функций объявление выглядит как определение без тела, завершённое точкой с запятой. Листинг 10-3 показывает пример.

Листинг 10-3. `fun_decl_def.c`

```
/* This is declaration */
void f( int x );
/* This is definition */
void f( int x )    {
    puts( "Hello!" );
}
```

Такие объявления иногда называют *прототипами функций*. Каждый раз, когда вы используете функцию, чьё тело ещё не определено ИЛИ определено в другом файле, вам следует сначала написать её прототип. В прототипе функции имена аргументов можно опустить, как показано в листинге 10-4.

Листинг 10-4. `fun_proto_omit_arguments.c`

```
int square( int x );
/* same as */
int square( int );
```

Подводя итог, для функций корректны два сценария.

1. Сначала функция определяется, затем вызывается (см. листинг 10-5).

Листинг 10-5. `fun_sc_1.c`

```
int square( int x ) { return x * x; }
...
int z = square(5);
```

2. Сначала прототип, затем вызов, затем определение функции (см. листинг 10-6).

Листинг 10-6. `fun_sc_2.c`

```
int square( int x );
...
int z = square(5);
...
int square( int x ) { return x * x; }
```

Листинг 10-7 показывает типичную ошибочную ситуацию, когда тело функции объявлено после вызова, но перед вызовом нет объявления.

Листинг 10-7. `fun_sc_3.c`

```
int z = square( 5 );
...
int square( int x ) { return x * x; }
```

10.1.2 Объявления структур

Довольно распространено определять рекурсивную структуру данных, например, *связный список*. Каждый элемент хранит значение и связь со следующим элементом. Последний элемент хранит `NULL` вместо действительного указателя, чтобы отметить конец списка. Листинг 10-8 показывает определение связного списка.

Листинг 10-8. `list_definition.c`

```
struct list {
    int value;
    struct list* next;
};
```

Однако в случае двух взаимно рекурсивных структур вам нужно добавить опережающее объявление хотя бы для одной из них. Листинг 10-9 показывает пример.

Листинг 10-9. `mutually_recursive_structures.c`

```
struct b; /* forward declaration */
struct a {
    int value;
    struct b* next;
};
/* no need to forward declare struct a because it is already defined */
struct b {
    struct a* other;
};
```

Если у помеченного типа нет определения, а есть только объявление, это называется *неполным типом*. В этом случае мы можем свободно работать с указателями на него, но никогда не можем создать переменную такого типа, разыменовать его или работать с массивами такого типа. Функции не должны возвращать экземпляр такого типа, но, аналогично, они могут возвращать указатель. Листинг 10-10 показывает пример.

Листинг 10-10. `incomplete_type_example.c`

```
struct llist_t;
struct llist_t* f() { ... }   /* ok  */
struct llist_t g();           /* ok  */
struct llist_t g()  { ... }   /* bad */
```

У таких типов есть весьма специфичный случай применения, который мы разберём в Главе 13.

10.2 Доступ к коду из других файлов

10.2.1 Функции из других файлов

Разумеется, можно вызывать функции или ссылаться на глобальные переменные из других файлов. Чтобы выполнить вызов, нужно добавить прототип вызываемой функции в текущий файл. Например, у вас есть два файла: `square.c`, содержащий функцию `square`, и `main_square.c`, содержащий функцию main. Листинг 10-11 и листинг 10-12 показывают эти файлы.

Листинг 10-11. `square.c`

```
int square( int x ) { return x * x; }
```

Листинг 10-12. `main_square.c`

```
#include <stdio.h>
int square( int x );
int main(void) {
    printf( "%d\n", square( 5 ) );
    return 0;
}
```

Каждый файл кода — отдельный модуль и потому компилируется независимо, точно так же, как в ассемблере. Файл `.c` транслируется в объектный файл. Поскольку в наших учебных целях мы придерживаемся файлов ELF (Executable and Linkable Format), давайте вскроем получившиеся объектные файлы и посмотрим, что внутри. Обратитесь к листингу 10-13, чтобы увидеть таблицу символов внутри объектного файла `main_square.o`, и к листингу 10-14 для файла `square.o`. Объяснение формата таблицы символов см. в разделе 5.3.2.

Листинг 10-13. `main_square`

```
> gcc -c -std=c89 -pedantic -Wall main_square.c
> objdump -t main_square.o
main.o:     file format elf64-x86-64
SYMBOL TABLE:
0000000000000000 l    df *ABS*  0000000000000000 main.c
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l    d  .note.GNU-stack
0000000000000000 .note.GNU-stack
0000000000000000 l    d  .eh_frame
0000000000000000 .eh_frame
0000000000000000 l    d  .comment
0000000000000000 .comment
0000000000000000 g     F .text  000000000000001c main
0000000000000000         *UND*  0000000000000000 square
```

Листинг 10-14. `square`

```
> gcc  -c -std=c89 -pedantic -Wall square.c
> objdump -t square.o
square.o:     file format elf64-x86-64
SYMBOL TABLE:
0000000000000000 l    df *ABS*  0000000000000000 square.c
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l    d  .note.GNU-stack
0000000000000000 .note.GNU-stack
0000000000000000 l    d  .eh_frame
0000000000000000 .eh_frame
0000000000000000 l    d  .comment
0000000000000000 .comment
0000000000000000 g     F .text  0000000000000010 square
```

Как видите, все функции (а именно `square` и `main`) стали глобальными символами, о чём говорит буква `g` во втором столбце, несмотря на то что они никак особо не помечены. Это значит, что все функции подобны меткам, помеченным ключевым словом `global` в ассемблере — другими словами, видимым другим модулям. Прототип функции `square`, находящийся в `main_square.c`, приписан к неопределённой секции.

```
0000000000000000    *UND*    0000000000000000   square
```

GCC предоставляет вам доступ ко всей цепочке инструментов компилятора, а значит, он не только транслирует файлы, но и вызывает компоновщик с подходящими аргументами. Он также компонует файлы со стандартной библиотекой C. После компоновки таблица символов становится более насыщенной за счёт символов стандартной библиотеки и служебных символов,

```
such as .gnu.version.
```

> **Вопрос 188** Скомпилируйте файл `main` с помощью команды `gcc -o main main_square.o square.o`. Изучите его таблицу объектов с помощью `objdump -t main`. Что вы можете сказать о функциях `main` и `square`?

10.2.2 Данные из других файлов

Если есть глобальная переменная, определённая в другом `.c`-файле, к которой мы хотим обратиться, её следует, желательно, но не обязательно, объявить с ключевым словом `extern`. Не следует инициализировать переменные `extern`; иначе компилятор выдаст предупреждение. Листинг 10-15 и листинг 10-16 показывают первый пример использования глобальной переменной из другого файла.

Листинг 10-15. `square_ext.c`

```
extern int z;
int square( int x ) { return x * x + z; }
```

Листинг 10-16. `main_ext.c`

```
int z = 0;
int square( int x );
int main(void) {
    printf( "%d\n", square( 5 ) );
    return 0;
}
```

Стандарт C помечает ключевое слово `extern` как необязательное. Мы рекомендуем никогда не опускать ключевое слово `extern`, чтобы вы могли легко различать, в каком именно файле вы хотите создать переменную. Однако если вы всё же опускаете ключевое слово `extern`, как компилятор различает определение и объявление переменной, если инициализация не предоставлена? Это особенно интересно, учитывая, что файлы компилируются отдельно. Чтобы изучить этот вопрос, мы посмотрим на таблицы символов объектных файлов с помощью утилиты `nm`. Мы напишем файлы `main.c` и `other.c`, а затем скомпилируем их в `.o`-файлы с помощью флага `-c`, а затем скомпонуем их. Листинг 10-17 показывает последовательность команд.

Листинг 10-17. `glob_build`

```
> gcc -c -std=c89 -pedantic -Wall -o main.o main.c
> gcc -c -std=c89 -pedantic -Wall -o other.o other.c
> gcc -o main main.o other.o
```

Есть одна глобальная переменная по имени `x`. Ей не присваивается значение в `main.c`, но она инициализируется в `other.c`. С помощью `nm` мы можем быстро просмотреть таблицу символов, как показано в листинге 10-18. Мы намеренно сократили

таблицу для исполняемого файла `main`, чтобы не загромождать листинг служебными символами.

Листинг 10-18. `glob_nm`

```
> nm main.o
0000000000000000 T main
                 U printf
0000000000000004 C x
> nm other.o
0000000000000000 D x
> nm main
0000000000400526 T main
                 U printf@@GLIBC_2.2.5
0000000000601038 D x
```

Как мы видим, в `main.o` символ `x`, соответствующий переменной `int x`, помечен флагом `C` (глобальный общий), тогда как в другом объектном файле, `main.o`, он помечен `D` (глобальные данные). Таких похожих глобальных общих *символов* может быть сколько угодно, и в итоговом исполняемом файле все они будут схлопнуты в один. Однако нельзя иметь несколько объявлений одного и того же символа в одном исходном файле; вы ограничены максимум одним объявлением и одним определением.

10.2.3 Заголовочные файлы

Итак, теперь мы знаем, как разбивать код на несколько файлов. У каждого файла, использующего внешнее определение, должно быть написано его объявление до фактического использования. Однако при росте числа файлов поддерживать согласованность становится трудно. Обычная практика — использовать заголовочные файлы, чтобы упростить сопровождение. Допустим, есть два файла: `main_printer.c` и `printer.c`. Листинги 10-19 и 10-20 показывают их.

Листинг 10-19. `main_printer.c`

```
void print_one(void);
void print_two(void);
int main(void) {
    print_one();
    print_two();
    return 0;
}
```

Листинг 10-20. `printer.c`

```
#include <stdio.h>
void print_one(void) {
    puts( "One" );
}
void print_two(void) {
    puts( "Two" );
}
```

Вот реальный сценарий. Чтобы использовать функцию из файла `printer.c` в каком-то файле `other.c`, вам нужно написать прототипы функций, определённых в `printer.c`, где-то в начале `other.c`. Чтобы использовать их в третьем файле, вам придётся написать их прототипы и в третьем файле тоже. Так зачем делать это вручную, если можно создать отдельный файл, который будет содержать только объявления функций и глобальных переменных, но не определения, а затем подключить его с помощью препроцессора? Мы изменим этот пример, введя новый *заголовочный файл* `printer.h`, содержащий все объявления из `printer.c`. Листинг 10-21 показывает заголовочный файл.

Листинг 10-21. `printer.h`

```
void print_one( void );
void print_two( void );
```

Теперь каждый раз, когда вы хотите использовать функции, определённые в `printer.c`, вам достаточно поместить следующую строку в начало текущего файла кода:

```
#include "printer.h"
```

Препроцессор заменит эту строку содержимым `printer.h`. Листинг 10-22 показывает новый главный файл.

Листинг 10-22. `main_printer_new.c`

```
#include "printer.h"
int main(void) {
    print_one();
    print_two();
    return 0;
}
```

> **Примечание** Заголовочные файлы сами по себе не компилируются. Компилятор видит их только как части `.c`-файлов.

Этот механизм, который выглядит похожим на импорт модулей или библиотек в таких языках, как Java или C#, по своей природе очень отличается от него. Так что утверждение, что строка `#include "some.h"` означает «импортировать библиотеку под названием `some`», в корне неверно. Подключение текстового файла — это не импорт библиотеки! Статические библиотеки, как мы знаем, по сути те же объектные файлы, что и получаемые при компиляции `.c`-файлов. Так что картина для примерного файла `f.c` выглядит следующим образом:

- Начинается компиляция `f.c`.
- Препроцессор встречает директивы `#include` и подключает соответствующие `.h`

файлы «как есть».

- Каждый `.h`-файл содержит прототипы функций, которые после трансляции кода станут записями в таблице

символов.

- Для каждой такой похожей на импорт записи компоновщик будет искать среди всех входных объектных файлов определённый символ (в секции *.data, .bss* или *.text*). В одном месте он найдёт такой символ и свяжет с ним похожую на импорт запись. Этот символ может находиться в стандартной библиотеке C. Но погодите, разве мы передаём компоновщику стандартную библиотеку на вход? Мы обсудим это в следующем разделе.

10.3 Стандартная библиотека

Мы уже использовали заголовки, соответствующие частям стандартной библиотеки, такие как `stdio.h`. Они содержат не сами стандартные функции, а их прототипы. Вам не обязательно верить нам на слово, потому что вы можете проверить это сами. Для этого создайте файл `p.c`, содержащий только одну строку: `#include <stdio.h>`. Затем запустите на нём GCC с флагом `-E`, чтобы остановиться после препроцессирования и вывести результаты в `stdout`. Используйте утилиту `grep`, чтобы найти вхождение `printf`, и вы найдёте его прототип, как показано в листинге 10-23.

Листинг 10-23. `printf_check_header`

```
>  cat p.c
#include <stdio.h>
> gcc -E -pedantic -ansi p.c | grep " printf"
extern int printf (const char *__restrict__format, ...);
```

Пока мы не будем говорить о ключевом слове `restrict`, так что представим, что его здесь нет. Файл `stdio.h`, подключённый в нашем тестовом файле `p.c`, очевидно, содержит прототип функции `printf` (обратите внимание на точку с запятой в конце строки!), у которого нет тела. Три точки на месте последнего аргумента означают произвольное количество аргументов. Эта возможность будет обсуждаться в Главе 14. Тот же эксперимент можно провести для любой функции, доступ к которой вы получаете, подключив `stdio.h`. GCC — своего рода универсальный интерфейс: вы можете использовать его для отдельной компиляции одиночных файлов без компоновки (флаг `-c`), можете выполнить весь цикл компиляции, включая компоновку, для нескольких файлов, но также можете вызвать компоновщик косвенно, передав GCC `.o`-файлы на вход:

```
gcc -o executable_file obj1.o obj2.o ...
```

Выполняя компоновку, GCC не просто вслепую вызывает `ld`. Он также предоставляет ему правильную версию библиотеки C, или библиотек. Дополнительные библиотеки можно указать с помощью флага `-l`. В самом распространённом сценарии библиотека C состоит из двух частей:

- Статическая часть (обычно называемая `crt0` — C RunTime, ноль означает «самое начало») содержит процедуру `_start`, которая выполняет инициализацию стандартных служебных структур, требуемых конкретно этой реализацией библиотеки. Затем она вызывает функцию `main`. В Intel 64 аргументы командной строки передаются через стек. Это значит, что `_start` должна скопировать `argc` и `argv` со стека в `rdi` и `rsi`, чтобы соблюсти соглашение о вызове функций. Если вы скомпонуете один файл и проверите его таблицу символов до и после компоновки, вы увидите довольно много новых символов, происходящих из `crt0`, например, знакомую `_start`, которая и является настоящей точкой входа.
- Динамическая часть, содержащая сами функции и глобальные переменные. Поскольку их использует подавляющее большинство запущенных приложений, разумно не копировать её, а разделять между ними ради меньшего общего потребления памяти и лучшей локальности. Мы докажем её существование, применив утилиту `ldd` к скомпилированному примеру файла `main_ldd.c`, показанному в листинге 10-24.

Это поможет нам найти стандартную библиотеку C. Листинг 10-25 показывает вывод `ldd`.

Листинг 10-24. `main_ldd.c`

```
#include <stdio.h>
int main( void )
{
    printf("Hello World!\n");
    return 0;
}
```

Листинг 10-25. `ldd_locating_libc`

```
>  gcc main.c -o main
    > ldd main
    linux-vdso.so.1 (0x00007fff4e7fc000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f2b7f6bf000)
/lib64/ld-linux-x86-64.so.2 (0x00007f2b7fa76000)
```

Этот файл скомпонован с тремя динамическими библиотеками.

1. `ld-linux` — это сам загрузчик динамических библиотек, который ищет и загружает

все динамические библиотеки, требуемые исполняемым файлом.

2. `vdso`, что означает «virtual dynamic shared object» (виртуальный динамический разделяемый объект), — небольшая служебная библиотека,

используемая стандартной библиотекой C для ускорения взаимодействия с ядром в некоторых ситуациях.

3. Наконец, сама `libc` содержит исполняемый код стандартных функций.

Затем, поскольку стандартная библиотека — это просто ещё один ELF-файл, мы запустим `readelf`, чтобы вывести её таблицу символов и увидеть запись `printf` собственными глазами. Листинг 10-26 показывает результат. Первая запись действительно тот `printf`, который мы используем; метка после `@@` обозначает версию символа и используется для предоставления разных версий одной и той же функции. Старое ПО, использующее более старые версии функций, продолжит их использовать, тогда как новое ПО может переключиться на более удачно написанный, более свежий вариант, не нарушая совместимость.

Листинг 10-26. `printf_lib_entry`

```
> readelf -s /lib/x86_64-linux-gnu/libc.so.6  | grep " printf"
596: 0000000000050d50   161 FUNC    GLOBAL DEFAULT   12
printf@@GLIBC_2.2.5
1482: 0000000000050ca0    31 FUNC    GLOBAL DEFAULT   12
printf_size_info@@GLIBC_2.2.5
1890: 0000000000050480  2070 FUNC    GLOBAL DEFAULT   12
printf_size@@GLIBC_2.2.5
```

> **Вопрос 189** Попробуйте найти те же символы с помощью утилиты `nm` вместо `readelf`.

10.4 Препроцессор

Помимо определения глобальных констант с помощью `#define`, препроцессор также используется как обходной путь для решения проблемы многократного включения. Сначала мы кратко рассмотрим соответствующие возможности препроцессора. Директива `#define` используется в следующих типичных формах:

- `#define FLAG` означает, что символ препроцессора `FLAG` определён, но его значение —

пустая строка (или, можно сказать, у него нет значения). Этот символ по большей части бесполезен в подстановках, но мы можем проверить, существует ли определение вообще, и включить некоторый код на основе этого.

- `#define MY_CONST 42` — знакомый способ определения глобальных констант. Каждый раз, когда

`MY_CONST` встречается в тексте программы, он заменяется на 42.

- `#define MAX(a, b) ((a)>(b))?(a):(b)` — макроподстановка с параметрами.

Строка `int x = MAX(4+3, 9)` затем будет заменена на: `int x = ((4+3)>(9))?(4+3):(9)`.

> **Параметры макроса в скобках** Заметьте, что все параметры в теле макроса должны быть заключены в скобки. Это гарантирует, что сложные выражения, переданные макросу в качестве параметров, будут разобраны корректно. Представьте простой макрос `SQ`.

```
#define SQ(x) x*x
```

> **Примечание.** Строка `int z = SQ(4+3)` затем будет заменена на

```
int z = 4 + 3 * 4 + 3
```

> **Примечание.** что из-за более высокого приоритета умножения по сравнению со сложением будет разобрано как `4 + (3*4) + 3`, что не совсем то выражение, которое мы намеревались сформировать.

Если вы хотите определить дополнительные символы препроцессора, вы также можете задать их при запуске GCC с помощью ключа `-D`. Например, вместо того чтобы писать `#define SYM VALUE`, вы можете запустить `gcc -DSYM=VALUE`, или просто `gcc -DSYM` для простого `#define` SYM. Наконец, нам нужно условие макроса: `#ifdef`. Эта директива позволяет включать или исключать некоторый текстовый фрагмент из препроцессированного файла в зависимости от того, определён символ или нет. Вы можете включить строки между `#ifdef` SYMBOL и `#endif`, если SYMBOL определён, как показано в

Листинге 10-27.

Листинг 10-27. `ifdef_ex.c`

```
#ifdef SYMBOL
/*code*/
#endif
```

Вы можете включить строки между `#ifdef SYMBOL` и `#endif`, если `SYMBOL` определён, ИНАЧЕ включить другой код, как показано в листинге 10-28.

Листинг 10-28. `ifdef_else_ex.c`

```
#ifdef SYMBOL
/*code*/
#else
/*other code*/
#endif
```

Вы также можете указать, что некоторый код будет включён только если определённый символ не определён, как показано в

Листинге 10-29.

Листинг 10-29. `ifndef_ex.c`

```
#ifndef MYFLAG
/*code*/
#else
/*other code*/
#endif
```

10.4.1 Защита от повторного включения

Один файл может содержать максимум одно объявление и одно определение для любого заданного символа. Хотя вы не будете писать дублирующиеся объявления сами, вы, скорее всего, будете использовать заголовочные файлы, которые могут подключать другие заголовочные файлы, и так далее. Знать, какие объявления будут присутствовать в текущем файле, непросто: вам нужно пройтись по каждому заголовочному файлу и по каждому заголовочному файлу, который они подключают, и так далее. Например, есть три файла: `a.h`, `b.h` и `main.c`, показанные в листинге 10-30.

Листинг 10-30. `inc_guard_motivation.c`

```
/* a.h */
void a(void);
/* b.h */
#include "a.h"
void b(void);
/* main.c */
#include "a.h"
#include "b.h"
```

Как будет выглядеть препроцессированный файл `main.c`? Мы запустим `gcc -E main.c`. Листинг 10-31 показывает результат.

Листинг 10-31. `multiple_inner_includes.c`

```
# 1 "main.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2
# 1 "main.c"
# 1 "a.h" 1
void a(void);
# 2 "main.c" 2
# 1 "b.h" 1
# 1 "a.h" 1
void a(void);
# 2 "b.h" 2
void b(void);
# 2 "main.c" 2
```

Теперь `main.c` содержит дублирующееся объявление функции `void a(void)`, что приводит к ошибке компиляции. Первое объявление приходит напрямую из файла `a.h`; второе — из файла `b.h`, который сам подключает `a.h`. Есть две распространённые техники, чтобы это предотвратить.

- Использование директивы `#pragma once` в начале заголовка. Это нестандартный способ

запретить многократное включение заголовочного файла. Многие компиляторы его поддерживают, но, поскольку он не входит в стандарт C, его использование не рекомендуется.

- Использование так называемых *защитных конструкций включения (include guards)*.

Листинг 10-32 показывает защитную конструкцию включения для некоторого файла `file.h`.

Листинг 10-32. `file.h`

```
#ifndef _FILE_H_
#define _FILE_H_
void a(void);
#endif
```

Текст между директивами `#ifndef _FILE_H_` и `#endif` будет включён, только если символ `X` не определён. Как мы видим, самая первая строка в этом тексте: `#define _FILE_H_`. Это значит, что при следующем включении всего этого текста в результате выполнения директивы `#include` та же директива `#ifndef _FILE_H_` предотвратит повторное включение содержимого файла. Обычно люди называют такие символы препроцессора на основе имени файла, одно такое соглашение было показано и состоит из

- Написания имени файла заглавными буквами.
- Замены точек на подчёркивания.
- Добавления одного или нескольких подчёркиваний в начало и конец.

Мы подготовили типичный файл для включения, чтобы вы могли изучить его структуру. Листинг 10-33 показывает этот пример.

Листинг 10-33. `pair.h`

```
#ifndef _PAIR_H_
#define _PAIR_H_
#include <stdio.h>
struct pair {
    int x;
    int y;
};
void pair_apply( struct pair* pair, void (*f)(struct pair) );
void pair_tofile( struct pair* pair, FILE* file );
#endif
```

Защитная конструкция включения — первое, что мы видим в этом файле. Затем идут другие включения. Зачем нужно подключать файлы в заголовочных файлах? Иногда ваши функции или структуры зависят от внешних типов, определённых где-то ещё. В этом примере функция `pair_tofile` принимает аргумент типа `FILE*`, определённого в стандартном заголовочном файле `stdio.h` (или в одном из заголовков, которые он сам подключает). После этого идёт определение типа, а затем прототипы функций.

10.4.2 Почему препроцессор — это зло?

Активное использование препроцессора считается плохим по ряду причин:

- Он часто делает код меньше, но и намного менее читаемым.
- Он вносит ненужные абстракции.
- В большинстве случаев он усложняет отладку.
- Макросы часто сбивают с толку IDE (интегрированные среды разработки) и их

движки автодополнения, а также различные статические анализаторы. Не стоит смотреть на них свысока, потому что в крупных проектах они очень помогают. Препроцессор ничего не знает о структуре языка, так что каждая конструкция препроцессора в отдельности может быть недопустимым оператором языка. Например, макрос `#define OR else {` может стать частью корректного оператора после всех подстановок, но сам по себе он не является корректным оператором. Когда макросы смешиваются, а границы операторов нечётко определены, понять такой код трудно. Некоторые задачи из-за препроцессора могут быть почти неразрешимыми. Он ограничивает объём интеллекта, который можно заложить в среду программирования или инструменты статического анализа. Рассмотрим несколько подводных камней:

1. Насколько умным должен быть статический анализатор кода, чтобы понять, что возвращает `foo` (см.

листинг 10-34)?

Листинг 10-34. `ifdef_pitfall_sig.c`

```
#ifdef SOMEFLAG
int foo() {
#else
    void  foo()  {
#endif
/* ... */
}
```

2. Вам нужно найти все вхождения макроса `min`, определённого как

```
#define min(x,y) ((x) < (y) ? (x) : (y)).
```

Как вы видели в предыдущем примере, чтобы разобрать программу, сначала нужно выполнить проходы препроцессирования, иначе инструмент может даже не понять границы функций. После выполнения препроцессирования все макросы `min` подставляются и потому становятся неотслеживаемыми и неотличимыми от таких строк, как

```
int z = ((10) < (y) ? (5) : (3)).
```

3. Статический анализ (и даже ваше собственное понимание программы) пострадает

из-за использования макросов. Синтаксически инстанцирования макросов с параметрами неотличимы от вызовов функций. Однако если аргументы функции вычисляются *до* выполнения вызова функции, то аргументы макроса подставляются, а затем выполня-

ются получившиеся строки кода.

Например, возьмём тот же макрос `#define min(x,y) ((x) < (y) ? (x) : (y))`. Инстанцирование с аргументами `a` и `b--` будет выглядеть так: `((a) < (b--) ? (a) :(b--))`. Как видите, если `a >= b`, переменная `b` будет декрементирована *дважды*. Если бы `min` была функцией, `b--` выполнилось бы только один раз.

10.5 Пример: сумма динамического массива

10.5.1 Краткий обзор динамического выделения памяти

Чтобы выполнить следующее задание, вам нужно научиться использовать функции `malloc` и `free`. Мы обсудим их подробнее позже, а пока дадим краткое введение. Локальные, как и глобальные переменные, позволяют выделить фиксированное количество байт. Однако когда размер выделяемой памяти зависит от входных данных, можно либо выделить столько памяти, сколько, по вашему мнению, хватит во всех случаях, либо использовать функцию `malloc`, выделяющую столько памяти, сколько вы у неё запросите. `void* malloc(size_t sz)` возвращает начало выделенного буфера памяти размером `sz` (в байтах) или `NULL` в случае неудачи. Изначально этот буфер содержит случайные значения. Поскольку функция возвращает `void*`, этот указатель можно присвоить указателю любого другого типа. Все такие выделенные области памяти нужно освобождать, когда они больше не нужны, вызывая для них `free`. Чтобы использовать эти две функции, нужно подключить `malloc.h`. Листинг 10-35 показывает минимальный пример использования `malloc` и `free`.

Листинг 10-35. simple_malloc.c

```
#include <malloc.h>
int main( void ) {
    int* array;
    /* malloc returns the allocated memory starting address
     * Notice that its argument is the byte size, elements count multiplied
     * by element size */
array = malloc( 10 * sizeof( int ));
    /* actions on array are performed here */
    free( array ); /* now the related memory region is deallocated */
    return 0;
}
```

10.5.2 Пример

Листинг 10-36 показывает пример. Он содержит три интересующие нас функции:

- `array_read` для чтения массива из `stdin`. Именно здесь происходит выделение памяти.

Заметьте использование функции `scanf` для чтения из `stdin`. Не забывайте, что она принимает не значения переменных, а их адреса, чтобы иметь возможность действительно записать в них.

- `array_print` для вывода заданного массива в `stdout`.
- `array_sum` для суммирования всех элементов массива.

Заметьте, что массив, выделенный где-то с помощью `malloc`, существует до момента вызова `free` для его начального адреса. Освобождение уже освобождённого массива — ошибка.

Листинг 10-36. sum_malloc.c

```
#include <stdio.h>
#include <malloc.h>
int* array_read( size_t* out_count ) {
    int* array;
    size_t i;
    size_t cnt;
    scanf( "%zu", &cnt );
    array = malloc( cnt * sizeof( int ) );
    for( i = 0; i < cnt; i++ )
        scanf( "%d", & array[i] );
    *out_count = cnt;
    return array;
}
void array_print( int const* array, size_t count ) {
    size_t i;
    for( i = 0; i < count; i++ )
        printf( "%d  ", array[i] );
    puts("");
}
int array_sum( int const* array, size_t count ) {
    size_t i;
    int sum = 0;
    for( i = 0; i < count; i++ )
        sum = sum + array[i];
    return sum;
}
int main( void ) {
    int* array;
    size_t count;
    array = array_read( &count );
    array_print( array, count );
    printf( "Sum is: %d\n", array_sum( array, count ) );
    free( array );
    return 0;
}
```

10.6 Задание: связный список

10.6.1 Задание

Программа принимает произвольное количество целых чисел через `stdin`. Вам нужно

1. Сохранить их все в *связном списке в обратном порядке*.
2. Написать функцию для вычисления суммы элементов связного списка.
3. Использовать эту функцию для вычисления суммы элементов сохранённого списка.
4. Написать функцию для вывода *n*-го элемента списка. Если список слишком короткий, сигнализировать об этом.
5. Освободить память, выделенную под связный список.

Вам нужно научиться использовать

- Структурные типы для кодирования самого связного списка.
- Константу `EOF`. Прочитайте раздел «Return value» в `man scanf`.

Вы можете быть уверены, что

- Ввод не содержит ничего, кроме целых чисел, разделённых пробельными символами.
- Все входные числа помещаются в переменные `int`.

Ниже приведён рекомендуемый список функций для реализации:

- `list_create` – принимает число, возвращает указатель на новый узел связного списка.
- `list_add_front` – принимает число и указатель на указатель на связный список. Добавляет новый узел с числом в начало списка.

Например: список (1,2,3), число 5, и новый список — (5,1,2,3).

- `list_add_back` добавляет элемент в конец списка. Сигнатура такая же, как у

```
list_add_front.
```

- `list_get` получает элемент по индексу или возвращает 0, если индекс выходит за границы списка.
- `list_free` освобождает память, выделенную под все элементы списка.
- `list_length` принимает список и вычисляет его длину.
- `list_node_at` принимает список и индекс, возвращает указатель на `struct list`, соответствующий узлу по этому индексу. Если индекс слишком велик, возвращает `NULL`.
- `list_sum` принимает список, возвращает сумму элементов.

Вот некоторые дополнительные требования:

- Все части логики, используемые более одного раза (или те, что можно концептуально изолировать), следует абстрагировать в функции и использовать повторно.

- Исключение из предыдущего требования — случай, когда падение производительности становится критичным, потому что повторное использование кода радикально неэффективно меняет алгоритм. Например, можно использовать функцию `list_at`, чтобы получить *n*-й элемент списка в цикле для вычисления суммы всех элементов. Однако для этого нужно проходить весь список, чтобы добраться до элемента. По мере увеличения *n* вы будете проходить одни и те же элементы снова и снова.

На самом деле, для списка длины *N* мы можем вычислить, сколько раз будут запрошены элементы для вычисления суммы:

1 + 2 + 3 + ... + N = N(N + 1) / 2

2 Мы начинаем с суммы, равной 0. Затем прибавляем первый элемент, для этого нам нужно обратиться к нему одному (1). Затем прибавляем второй элемент, обращаясь к первому и второму (2). Затем прибавляем третий элемент, обращаясь к первому, второму и третьему, поскольку мы просматриваем список с его начала. В итоге получаем что-то вроде $O(N^2)$ для тех, кто знаком с *O*-нотацией. По сути это означает, что при увеличении размера списка на 1 время суммирования такого списка увеличивается на *N*. В таком случае действительно разумнее просто пройти список один раз, прибавляя текущий элемент к аккумулятору.

- Писать маленькие функции — почти всегда очень хорошая идея.
- Подумайте о том, чтобы написать отдельные функции для: добавления элемента в начало, добавления в конец,

создания нового узла связного списка.

- Не забывайте активно использовать `const`, особенно в функциях, принимающих указатели в качестве

аргументов!

10.7 Ключевое слово static

В C у ключевого слова `static` есть несколько значений в зависимости от контекста.

1. Применяя `static` к глобальным переменным или функциям, мы делаем их доступными только в

текущем модуле (`.c`-файле). Чтобы это проиллюстрировать, мы скомпилируем простую программу, показанную в листинге 10-37, и запустим `nm`, чтобы заглянуть в таблицу символов. Помните, что `nm` помечает глобальные символы заглавными буквами.

Листинг 10-37. static_example.c

```
int global_int;
static int module_int;
static int module_function() {
    static  int  static_local_var;
    int  local_var;
    return 0;
}
int main( int argc, char** argv ) {
    return 0;
}
```

Мы видим, что все имена символов помечены как глобальные, *кроме* тех, что помечены `static` в C. На уровне ассемблера это означает, что большинство меток помечены `global`, и, чтобы это предотвратить, нам нужно явно использовать ключевое слово `static`.

```
>  gcc  -c  --ansi  --pedantic  -o  static_example.o  static_example.c
>   nm   static_example.o
0000000000000004  C  global_int
000000000000000b T main
0000000000000000 t module_function
0000000000000000  b  module_int
0000000000000004  b  static_local_var.1464
```

2. Применяя `static` к локальной переменной, мы делаем её похожей на глобальную, но никакая другая функция не может обратиться к ней напрямую. Другими словами, она сохраняется между вызовами функции после однократной инициализации. В следующий раз, когда та же функция будет вызвана, значение локальной статической переменной будет таким же, каким оно было при последнем завершении этой функции.

Листинг 10-38 показывает пример.

Листинг 10-38. static_loc_var_example.c

```
int demo (void)
{
    static int a = 42;
    printf("%d\n", a++);
}
...
demo();  //outputs  42
demo();  //outputs  43
demo();  //outputs  44
```

10.8 Компоновка

Понятие связывания (linkage) определено в стандарте C и систематизирует то, что мы изучили в этой главе до сих пор. Согласно ему, «идентификатор, объявленный в разных областях видимости или несколько раз в одной и той же области видимости, можно заставить ссылаться на один и тот же объект или функцию с помощью процесса, называемого связыванием» [7]. Таким образом, у каждого идентификатора (имени переменной или функции) есть атрибут, называемый связыванием. Есть три типа связывания:

- Отсутствие связывания, соответствующее переменным, локальным для *блока*.
- Внешнее связывание, делающее идентификатор доступным всем модулям, которые могут захотеть к нему обратиться. Так обстоит дело с глобальными переменными и любыми функциями.
- Все экземпляры определённого имени с внешним связыванием ссылаются на один и тот же объект программы.
- У всех объектов с внешним связыванием должно быть одно и только одно определение. Однако количество объявлений в разных файлах не ограничено.
- Внутреннее связывание, ограничивающее видимость идентификатора файлом `.c`, в котором он был определён.

Нам легко сопоставить известные нам виды языковых сущностей с типами связывания:

- Обычные функции и глобальные переменные — внешнее связывание.
- Статические функции и глобальные переменные — внутреннее связывание.
- Локальные переменные (статические или нет) — внутреннее связывание.

Хотя это понятие важно усвоить, чтобы свободно читать стандарт, оно редко встречается в повседневной практике программирования.

10.9 Итоги

В этой главе мы научились разбивать код на отдельные файлы. Мы рассмотрели понятие заголовочных файлов, изучили защитные конструкции включения и научились изолировать функции и переменные внутри файла. Мы также увидели, как выглядят таблицы символов для базовых программ на C, и эффекты, которые ключевое слово `static` оказывает на объектные файлы. Мы выполнили задание и реализовали связные списки (одну из самых фундаментальных структур данных). В следующей главе мы подробнее изучим память с точки зрения C.

Вопрос 190 В чём разница между объявлением и определением?

Вопрос 191 Что такое опережающее объявление?

Вопрос 192 Когда нужны объявления функций?

Вопрос 193 Когда нужны объявления структур?

Вопрос 194 Как можно вызывать функции, определённые в других файлах?

Вопрос 195 Какой эффект оказывает объявление функции на таблицу символов?

Вопрос 196 Как нам обращаться к данным, определённым в других файлах?

Вопрос 197 Что такое понятие заголовочных файлов? Для чего они обычно используются?

Вопрос 198 Из каких частей состоит стандартная библиотека C?

Вопрос 199 Как программа принимает аргументы командной строки?

Вопрос 200 Напишите программу на ассемблере, которая выведет все аргументы командной строки, каждый на отдельной строке.

Вопрос 201 Как мы можем использовать функции из стандартной библиотеки C?

Вопрос 202 Опишите механизм, позволяющий программисту использовать внешние функции, подключая соответствующие заголовки.

Вопрос 203 Почитайте про `ld-linux` .

Вопрос 204 Какие основные директивы препроцессора C?

Вопрос 205 Для чего используется защитная конструкция включения и как её написать?

Вопрос 206 Какой эффект оказывают глобальные переменные и функции `static` на таблицу символов?

Вопрос 207 Что такое статические локальные переменные?

Вопрос 208 Где создаются статические локальные переменные?

Вопрос 209 Что такое связывание? Какие типы связывания существуют?

Память

Память — ключевая часть модели вычислений, используемой в C. Она хранит все виды переменных, а также функции. Эта глава подробно изучит модель памяти C и связанные с ней возможности языка.

11.1 Указатели снова

> **Б. Керниган и Д. Ритчи об указателях** «Указателей ставят в один ряд с оператором `goto` как чудесный способ создавать программы, которые невозможно понять. Это, безусловно, верно, когда их используют неосторожно, и легко создать указатели, указывающие куда-то в неожиданное место. Однако при должной дисциплине указатели можно использовать и для достижения ясности и простоты». [18]

11.1.1 Зачем нам нужны указатели?

Поскольку язык C имеет модель вычислений фон Неймана, выполнение программы по сути представляет собой последовательность команд манипуляции данными. Данные находятся в адресуемой памяти, и адресуемость данных — то свойство, что позволяет более тонко и эффективно манипулировать данными. Многим языкам более высокого уровня не хватает этого свойства, потому что прямые манипуляции с адресами запрещены. Однако это преимущество имеет свою цену: становится проще создавать неочевидные и обычно неисправимые ошибки в коде. Именно необходимость хранить и манипулировать адресами объясняет, зачем нам нужны указатели. Проведя типичный разбор для листинга 11-1, мы замечаем, что в терминах абстрактной машины C:

- `a` – это имя ячейки данных абстрактной машины, содержащей число 4 типа `int`.
- `p_a` – это имя ячейки данных абстрактной машины, содержащей адрес переменной типа `int`.
- `p_a` хранит адрес `a`.
- `*p_a` — то же самое, что и `a`;
- `&a` равно `p_a`, но эти две сущности не одно и то же. В то время как `p_a` — имя для некоторых последовательных ячеек данных, `&a` — содержимое `p_a`, битовая строка, представляющая адрес.

Листинг 11-1. pointers_ex.c

```
int a = 4;
int* p_a = &a;
*p_a = 10; /* a = 10*/
```

> **Примечание** Оператор `&` можно применить лишь один раз, потому что для любого `x` выражение `&x` уже не будет **lvalue**.

11.1.2 Арифметика указателей

Ниже перечислены *единственные* действия, которые можно выполнять над указателями:

- Прибавлять или вычитать целые числа (в том числе отрицательные);

Итак, у нас есть указатели, и они содержат адреса. Для компьютера нет разницы между адресом целого числа и адресом строки. В языке ассемблера, как мы видели, все адреса одного типа. Зачем нам вообще хранить информацию о типе о том, на что указывает указатель? В чём разница между `int*` и `char*`?

Размер элемента, на который мы указываем, имеет значение. Прибавляя или вычитая целочисленное значение *X* из указателя типа *T* *, мы на самом деле изменяем его на *X `* sizeof( T )`. Посмотрим на пример, показанный в листинге 11-2.

Листинг 11-2. ptr_change_ex.c

```
int a = 42;             /* Assume this integer's address is 1000 */
int* p_a = &a;
p_a += 42;              /* 1000 + 42 * sizeof( int ) */
p_a = p_a + 1;          /* 1168 + 1 * sizeof( int ) */
p_a --;                 /* 1172 - 1 * sizeof( int ) */
```

- Взять свой собственный адрес. Если указатель — переменная, он тоже расположен

где-то в памяти. Так что у него тоже есть собственный адрес! Используйте оператор `&`, чтобы его взять.

- Разыменовать — базовая операция, которую мы тоже уже видели. Мы забираем

запись данных из памяти, начиная с адреса, хранящегося в данном указателе. Это делает оператор `*`. Листинг 11-3 показывает пример.

Листинг 11-3. deref_ex.c

```
int catsAreCool = 0;
int* ptr = &catsAreCool;
*ptr = 1; /* catsAreCool = 1 */
```

- Сравнить (с <, >, == и подобными).

Можно сравнивать два указателя. Результат определён, только если оба указывают на один и тот же блок памяти (например, на разные элементы одного и того же массива). Иначе результат случаен и не определён стандартом языка.

- Вычесть другой указатель. Если и только если у нас есть два указателя, определённо указывающих на смежный блок памяти, то, вычитая указатель с меньшим значением из указателя с большим значением, мы получаем количество элементов между ними. Для указателей x и y мы говорим о диапазоне элементов от *x включительно до *y не включительно (так что x − x = 0). Начиная с C99, тип выражения `ptr2 - ptr1` — специальный тип `ptrdiff_t`. Это *знаковый* тип того же размера, что и `size_t`. Заметьте, что результат отличается от количества байт между *x и *y! Наивно вычисленная разница была бы количеством байт, тогда как результат вычитания — это количество байт, делённое на размер элемента. Листинг 11-4 показывает пример.

Листинг 11-4. ptr_diff_calc.c

```
int arr[128];
int* ptr1 = &arr[50]; /* `array` address + 50 int sizes */
int* ptr2 = &arr[90]; /* `array` address + 90 int sizes */
ptrdiff_t d = ptr2 - ptr1; /* exactly 40 */
```

Во всех остальных случаях (вычитание большего указателя из меньшего, вычитание указателей, указывающих в разные области, и т. д.) результат может быть абсолютно случайным. Сложение, умножение и деление двух указателей синтаксически некорректны; таким образом, они немедленно вызывают ошибку компиляции.

11.1.3 Тип void*

Помимо обычных типов указателей, существует тип `void*`, который в некотором роде особый. Он забывает всю информацию о сущности, на которую указывает, кроме её адреса. Арифметика указателей запрещена для указателей `void*`, потому что размер сущности, на которую мы указываем, неизвестен, и потому его нельзя прибавить или вычесть. Прежде чем работать с таким указателем, нужно явно привести его к другому типу. В качестве альтернативы C позволяет присвоить этот указатель любому дру-

гому указателю (и присвоить `void*` указатель любого типа) без каких-либо предупреждений. Другими словами, если присваивание `short*` переменной `long` — явная ошибка, то присваивания трактуют `void*` как равнозначный любому типу указателя. Листинг 11-5 показывает пример.

Листинг 11-5. void_ptr_ex.c

```
void* a = (void*)4;
short* b = (short*) a;
b ++; /* correct, b = 6 */
b = a; /* correct */
a = b; /* correct */
```

11.1.4 NULL

С определяет специальную константу препроцессора `NULL`, равную 0. Она означает указатель, «указывающий в никуда», недействительный указатель. Записав это значение в указатель, мы можем быть уверены, что он ещё не инициализирован действительным адресом. Иначе мы не смогли бы отличить инициализированные указатели. На большинстве архитектур для недействительных указателей резервируют специальное значение, предполагая, что ни одна программа на самом деле не будет хранить полезное значение по этому адресу. Как мы уже знаем, 0 в контексте указателя не всегда означает двоичное число со всеми сброшенными битами. Указатель-0 может быть равен 0, но это не гарантируется стандартом. История знает архитектуры, где нулевой указатель выбирался довольно экзотическим способом. Например, некоторые компьютеры серии Prime 50 использовали сегмент 07777, смещение 0, для нулевого указателя; некоторые мейнфреймы Honeywell-Bull используют битовый шаблон 06000 для своего рода нулевых указателей. Листинг 11-6 показывает правильные способы проверить, является ли указатель `NULL` или нет.

Листинг 11-6. `null_check.c`

```
if( x ) { ... }
if( NULL  != x ) { ... }
if( 0 != x ) { ... }
if( x != NULL  ) { ... }
if( x != 0 ) { ... }
```

11.1.5 Пара слов о ptrdiff_t

Взгляните на пример, показанный в листинге 11-7. Можете найти ошибку?

Листинг 11-7. `ptrdiff_bug.c`

```
int* max;
int* cur;
int f( unsigned int e )
{
    if ( max - cur > e )
        return 1;
    else
        return 0;
}
```

Что произойдёт, если `cur > max`? Это означает, что разница между `cur` и `max` отрицательна. Её тип — `ptrdiff_t`. Сравнение её со значением типа `unsigned int` — интересный случай для изучения. У `ptrdiff_t` столько же бит, сколько у адреса на целевой архитектуре. Рассмотрим два случая:

- 32-битная система, где `sizeof( unsigned int ) == 4` и `sizeof( ptrdiff_t ) == 4`.

В этом случае типы в нашем сравнении пройдут через следующие преобразования.

```
int < unsigned int
(unsigned  int)int <  unsigned  int
```

Компилятор выдаст предупреждение, потому что приведение из `int` в `unsigned int` не всегда сохраняет значения. Нельзя свободно отобразить значения из диапазона -2^{31} . . . $2^{31} - 1$ в диапазон 0 . . . $2^{32} - 1$.

Например, если левая часть была равна -1, после преобразования в тип `unsigned int` она станет максимальным значением, представимым в типе `unsignedint` (2^{32}*−* 1). Очевидно, результат этого сравнения почти всегда будет равен 0, что неверно, потому что -1 меньше *любого* беззнакового целого числа.

- 64-битная система, где `sizeof( unsigned int ) == 4` и `sizeof( ptrdiff_t ) == 8`. В этой ситуации `ptrdiff_t`, вероятно, будет псевдонимом для `signed long`.

```
signed long < unsigned int
long < (signed long)unsigned int
```

Здесь будет приведена правая часть. Это приведение сохраняет информацию, так что компилятор не выдаст предупреждения.

Как видите, поведение этого кода зависит от целевой архитектуры, а это большой недостаток. Чтобы этого избежать, `ptrdiff_t` всегда должен идти в паре с `size_t`, потому что только тогда их размеры гарантированно совпадают.

11.1.6 Указатели на функции

Модель вычислений фон Неймана подразумевает, что код и данные находятся в одной и той же адресуемой памяти. Так что у функций тоже есть собственные адреса. Мы можем взять начальные адреса функций, передать их другим функциям, вызывать функции по указателям, хранить их в переменных или массивах и т. д. Однако зачем нам всё это делать? Это позволяет добиться лучших абстракций. Мы можем написать функцию, запускающую другую функцию и измеряющую время её работы, или преобразующую массив, применяя функцию ко всем его элементам. Эта техника позволяет повторно использовать код на совершенно новом уровне. Указатель на функцию хранит информацию о типе функции точно так же, как указатели на данные. Тип функции включает типы аргументов и тип возвращаемого значения. Для объявления указателя на функцию используется синтаксис, имитирующий объявление функции:

```
<return_value_type> (*name) (arg1, arg2, ...);
```

Листинг 11-8 показывает пример.

Листинг 11-8. fun_ptr_example.c

```
double doubler (int a) { return a * 2.5; }
...
double (*fptr)( int );
double a;
fptr = &doubler;
a = fptr(10); /* a = 25.0 */
```

Мы описали указатель `fptr` типа «указатель на функцию, принимающую `int` и возвращающую `double`». Затем мы присвоили ему адрес функции `doubler` и выполнили вызов по этому указателю с аргументом 10, сохранив возвращённое значение в переменной `a`. `typedef` работает и иногда очень помогает. Предыдущий пример можно переписать, как показано в

Листинге 11-9.

Листинг 11-9. fun_ptr_example_typedef.c

```
double doubler (int a) { return a * 2.5; }
typedef double (megapointer_type)( int );
...
double a;
megapointer_type*  variable  =  &doubler;
a  =  variable(10);  /*  a  =  25.0  */
```

Теперь с помощью `typedef` мы создали тип функции, который нельзя инстанцировать напрямую. Однако мы можем создавать переменные указанного типа указателя. Мы не можем напрямую создавать переменные функциональных типов, поэтому добавляем звёздочку. *Объекты первого класса* в языках программирования — это сущности, которые можно передать в качестве параметра, вернуть из функции или присвоить переменной. Как видим, функции не являются объектами первого класса в C. Иногда их называют «объектами второго класса», потому что указатели на них — объекты первого класса.

11.2 Модель памяти

Память абстрактной машины C, будучи единообразной, имеет несколько областей. С практической точки зрения каждая такая область отображается на отдельную *область памяти*, состоящую из последовательных страниц. Рисунок 11-1 показывает эту модель.

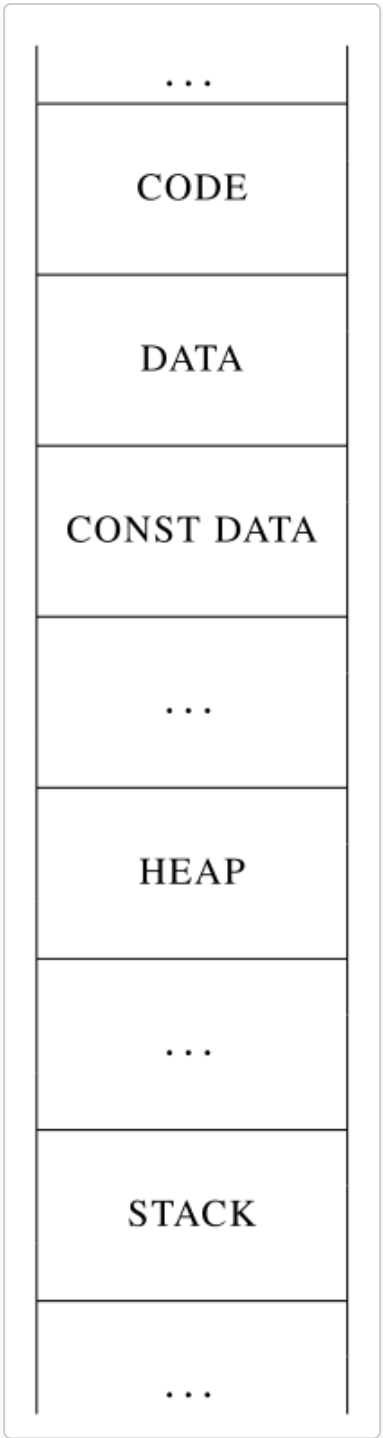


Рисунок 11-1. Модель памяти C

Области, которые есть почти в каждой программе на C:

- Код, хранящий машинные инструкции.
- Данные, хранящие обычные глобальные переменные.
- Константные данные, хранящие все неизменяемые данные, такие как строковые литералы и глобальные переменные, помеченные `const`. Операционная система обычно защищает соответствующие страницы с помощью механизма виртуальной памяти, разрешая или запрещая чтение/запись.

- Куча, хранящая динамически выделенные данные (с помощью `malloc`, как мы покажем в разделе 11.2.1).
- Стек, хранящий все локальные переменные, адреса возврата и другую служебную информацию. Если программа выполняется в нескольких потоках, каждый получает свой собственный стек.

11.2.1 Выделение памяти

Прежде чем использовать ячейки памяти, нужно выделить память. В C есть три типа выделения памяти.

- *Автоматическое выделение памяти* происходит при входе в процедуру. Когда мы входим в функцию, часть стека выделяется под её локальные переменные. Когда мы покидаем функцию, вся информация об этих переменных теряется. Время жизни этих данных ограничено временем жизни экземпляра функции. Как только функция завершается, память становится недоступной.
- На уровне ассемблера мы уже делали это в самом первом задании. Функции, выполнявшие вывод целых чисел, выделяли буфер на стеке для хранения итоговой строки. Это достигалось простым уменьшением `rsp` на размер буфера.

> **Примечание** Никогда не возвращайте из функций указатели на локальные переменные! Они указывают на данные, которых больше не существует.

- *Статическое выделение памяти* происходит во время компиляции в области данных или константных данных. Эти переменные существуют до завершения программы. По умолчанию переменные инициализируются нулями и потому оказываются в секции *.bss*. Константные данные выделяются в *.rodata*; изменяемые данные выделяются в *.data*.
- *Динамическое выделение памяти* нужно, когда мы не знаем размер памяти, которую нужно выделить, пока не произойдут некоторые внешние события. Этот тип выделения памяти опирается на реализацию в стандартной библиотеке C. Это значит, что когда стандартная библиотека C недоступна (например, программирование на голом железе), этот тип выделения памяти тоже недоступен. Этот тип выделения памяти использует кучу. Часть стандартной библиотеки отслеживает зарезервированные и доступные адреса памяти. Интерфейс этой части состоит из следующих функций, чьи прототипы находятся в заголовочном файле `malloc.h`. `void* malloc(size_t size)` выделяет `size` байт в куче и возвращает адрес первого из них. Возвращает `NULL` в случае неудачи.

Эта память не инициализирована и потому содержит случайные значения.

- void* calloc(size_t size, size_t count) выделяет size * count байт в куче и инициализирует их нулём. Возвращает адрес первого из них или `NULL` в случае неудачи. `void free(void* p)` освобождает память, выделенную в куче.
- `void* realloc(void* ptr, size_t newsize)` меняет размер блока памяти, начинающегося по адресу `ptr`, на `newsize` байт. Добавленная память не будет инициализирована. Содержимое копируется в новый блок, а старый блок освобождается. Возвращает указатель на новый блок памяти или `NULL` в случае неудачи.

Когда нам больше не нужен блок памяти, мы должны его освободить, иначе он навсегда останется в «зарезервированном» состоянии, никогда не будучи использованным повторно. Такая ситуация называется *утечкой памяти*. Когда вы используете тяжёлое программное обеспечение, содержащее ошибки, связанные с управлением памятью, его объём занимаемой памяти может значительно расти со временем, хотя программе на самом деле столько памяти не нужно. Обычно операционная система заранее предоставляет программе некоторое количество страниц. Эти страницы используются, пока программе не понадобится больше динамической памяти для выделения. Когда это происходит, вызов `malloc` может внутренне вызвать системный вызов (например, `mmap`), чтобы запросить больше страниц. Поскольку тип указателя `void*` можно присвоить указателю любого типа, следующий код не вызовет предупреждения (см. листинг 11-10) при компиляции как код на C.

Листинг 11-10. malloc_no_cast.c

```
#include <malloc.h>
...
int* a =  malloc(200);
a[4]  =  2;
```

Однако в C++, популярном языке, изначально произошедшем от C (и старающемся сохранять обратную совместимость), указатель void* нужно явно приводить к типу указателя, которому вы его присваиваете. Листинг 11-11 показывает эту разницу.

Листинг 11-11. malloc_cast_explicit.c

```
int* arr = (int*)malloc( sizeof(int) * 42 );
```

Почему некоторые программисты рекомендуют опускать приведение В более старых стандартах C было правило «неявного int» для объявлений функций. При отсутствии действительного объявления функции её первое использование считалось объявлением. Если имя, ранее не объявленное, встречается в выражении и за ним следует открывающая скобка, оно объявляет имя функции. Также предполагается, что эта функция возвращает значение `int`. Компилятор может даже создать для неё заглушку, возвращающую 0 (если не находит реализации). Если вы не подключите действительный заголовочный файл, содержащий объявление `malloc`, эта строка вызовет ошибку, потому что указателю присваивается целочисленное значение, возвращённое `malloc`:

```
int* x = malloc( 40 );
```

Примечание. Однако явное приведение скроет эту ошибку, потому что в C можно привести что угодно к любому нужному типу.

```
int* x  =  (int*)malloc( 40  );
```

Примечание. Современные версии стандарта C (начиная с C99) отменяют это правило, и объявления становятся обязательными, так что это рассуждение перестаёт быть верным. Преимущество явного приведения — лучшая совместимость с C++.

11.3 Массивы и указатели

Массивы в C особенные, потому что любую группу значений, расположенных последовательно в памяти, можно рассматривать как массив. Абстрактная машина считает, что имя массива — это адрес первого элемента, а значит, значение указателя! *i*-й элемент массива можно получить одной из следующих эквивалентных конструкций:

```
a[i] = 2;
*(a+i)  =  2
```

Адрес *i*-го элемента можно получить одной из следующих конструкций:

```
&a[i];
a+i;
```

Как видим, любую операцию с указателями можно переписать с использованием синтаксиса массива! И это ещё не всё. На самом деле синтаксис со скобками `a[i]` немедленно транслируется в `a + i`, что то же самое, что `i+a`. Из-за этого возможны и экзотические конструкции вроде `4[a]` (потому что `4+a` допустимо). Массивы можно инициализировать нулями с помощью следующего синтаксиса:

```
int a[10] = {0};
```

У массивов фиксированный размер. Однако есть два заметных исключения из этого правила, действительных в C99 и более новых версиях.

- Массивы, выделенные на стеке, могут иметь размер, определяемый во время выполнения. Они называются *массивами переменной длины*. Очевидно, что их нельзя пометить `static`, потому что последнее подразумевает выделение в секции *.data*.
- Начиная с C99, можно добавить *гибкий член-массив (flexible array member)* в качестве *последнего* члена структуры, как показано в листинге 11-12.

Листинг 11-12. `flex_array_def.c`

```
struct char_array {
    size_t length; char  data[];
};
```

В этом случае оператор `sizeof`, применённый к экземпляру структуры, вернёт размер структуры без массива. Массив будет ссылаться на память, непосредственно следующую за экземпляром структуры. Так, в примере из листинга 11-12 `sizeof(struct char_array) == sizeof(size_t)`. Предполагая, что это равно 8, `data[0]` ссылается на 8-й байт (считая с 0) от начального адреса экземпляра структуры.

Листинг 11-13 показывает пример.

Листинг 11-13. `flex_array.c`

```
#include <string.h>
#include <malloc.h>
struct int_array {
    size_t size;
    int  array[];
};
struct int_array* array_create( size_t size ) {
    struct int_array* array = malloc(
              sizeof( *array )
            + sizeof( int ) * size );
    array-> size = size;
    memset( array->array, 0, size );
    return array;
}
```

11.3.1 Детали синтаксиса

С позволяет определять несколько переменных подряд.

```
int a,b = 4, c;
```

Чтобы объявить несколько указателей, однако, нужно добавить звёздочку перед каждым указателем. Листинг 11-14 показывает пример: `a` и `b` — указатели, но тип `c` — `int`.

Листинг 11-14. `ptr_mult_decl.c`

```
int* a, *b, c;
```

Это правило можно обойти, создав псевдоним типа для `int*` с помощью `typedef`, скрывающий звёздочку. Определение нескольких переменных подряд обычно не рекомендуется, поскольку в большинстве случаев это затрудняет чтение кода. Можно создавать довольно сложные определения типов, смешивая указатели на функции, массивы, указатели и т. д. Чтобы их расшифровать, можно использовать следующий алгоритм:

1. Найдите идентификатор и начните с него.
2. Идите вправо до первой закрывающей скобки. Найдите её пару слева. Интерпретируйте

выражение между этими скобками.

3. Поднимитесь «на уровень выше» относительно выражения, разобранного на предыдущем

шаге. Найдите внешние скобки и повторите шаг 2. Мы проиллюстрируем этот алгоритм на примере, показанном в листинге 11-15. Таблица 11-1 описывает процесс разбора.

Листинг 11-15. complex_decl_1.c

```
int* (* (*fp) (int) ) [10];
```

Таблица 11-1. Разбор сложного определения

Выражение Интерпретация

Первый идентификатор.

```
fp
```

`(*fp)` Является указателем.

Функция, принимающая `int` и возвращающая указатель...

```
(* (*fp) (int))
```

... на массив из десяти указателей на int

```
int* (* (*fp) (int)) [10]
```

Как видите, процесс расшифровки сложных объявлений — не самое приятное занятие. Его можно упростить, используя `typedef` для частей объявлений.

11.4 Строковые литералы

Любую последовательность элементов `char`, завершённую нуль-терминатором, можно рассматривать как строку в C. Здесь, однако, мы хотим поговорить именно о непосредственно закодированных строках, то есть о строковых литералах. Большинство строковых литералов хранятся в *.rodata*, если они достаточно велики. Листинг 11-16 показывает пример строкового литерала.

Листинг 11-16. str_lit_example.c

```
char* str = "when the music is over, turn out the lights";
```

`str` — это просто указатель на первый символ строки. Согласно стандарту языка, строковые литералы (или указатели на строки, созданные таким образом) нельзя изменять.[26] Листинг 11-17 показывает пример.

Листинг 11-17. `string_literal_mut.c`

```
char* str = "hello world abcdefghijkl";
/* the following line produces a runtime error */
str[15] = '\'';
```

В C++ строковые литералы по умолчанию имеют тип `char const*`, что отражает их неизменяемую природу. Рассмотрите возможность использовать переменные типа `char const*`, когда это возможно, если строки, с которыми вы работаете, не предполагается изменять. Конструкции, показанные в листинге 11-18, тоже корректны, хотя вы, вероятнее всего, никогда не будете использовать вторую.

Листинг 11-18. `str_lit_ptr_ex.c`

```
char will_be_o = "hello, world!"[4];  /* is 'o' */
char const* tail = "abcde"+3 ; /* is "de", skipping 3 symbols */
```

При манипуляции строками есть несколько распространённых сценариев в зависимости от того, где строка выделена.

1. Можно создать строку среди глобальных переменных. Она будет изменяемой, и ни при каких

обстоятельствах не будет продублирована в области константных данных. Листинг 11-19 показывает пример.

Листинг 11-19. `str_glob.c`

```
char str[] = "something_global";
void f (void) { ... }
```

Другими словами, это просто глобальный массив, инициализированный на месте кодами символов.

2. Можно создать строку в стеке, в локальной переменной. Листинг 11-20

показывает пример.

Листинг 11-20. `str_loc.c`

```
void func(void) {
    char str[] = "something_local";
}
```

Однако саму строку `"something_local"` нужно где-то хранить, потому что локальные переменные инициализируются каждый раз при запуске функции, и нам нужно знать значения, которыми их следует инициализировать.

В случае относительно коротких строк компилятор попробует встроить их прямо в поток инструкций. Судя по всему, для более коротких строк разумнее просто разбить их на 8-байтовые фрагменты и выполнить инструкции `mov` с каждым фрагментом в качестве непосредственного операнда.

Длинные строки, однако, лучше хранить в *.rodata*. Оператор, показанный в листинге 11-20, выделит достаточно байт в стеке, а затем выполнит копирование из данных только для чтения в этот локальный буфер на стеке.

3. Можно выделить строку динамически через `malloc`. Заголовочный файл `string.h`

содержит несколько очень полезных функций, таких как `memcpy`, используемая для быстрого копирования.

Листинг 11-21 показывает пример.

Листинг 11-21. `str_malloc.c`

```
#include <malloc.h>
#include <string.h>
int main( int argc, char** argv )
{
    char* str = (char*)malloc( 25 );
    strcpy( str, "wow, such a nice string!" );
    free( str );
}
```

> **Вопрос 210** Почему мы выделили 25 байт для строки из 24 символов?

> **Вопрос 211** Прочитайте `man` для функций: `memcpy`, `memset`, `strcpy`.

11.4.1 Интернирование строк

«Интернирование строк» — термин, более привычный программистам на Java или C#. Однако на самом деле нечто подобное происходит и в C (но только во время компиляции). Компилятор старается избегать дублирования строк в области данных только для чтения. Это означает, что обычно всем трём переменным в коде, показанном в листинге 11-22, будут присвоены равные адреса.

Листинг 11-22. `str_intern.c`

```
char* best_guitar_solo  = "Firth of fifth";
char* good_genesis_song = "Firth of fifth";
char* best_1973_live = "Firth of fifth";
```

Интернирование строк было бы невозможно, если бы строковые литералы не были защищены от перезаписи. Иначе, изменяя такие строки в одном месте программы, мы вносили бы непредсказуемое изменение в данные, используемые в другом месте, поскольку оба места разделяют одну и ту же копию строки.

11.5 Модели данных

Мы говорили о размерах разных целочисленных типов. Стандарт языка навязывает набор правил вроде «размер `long` не меньше размера `short`» или «размер `signed short` должен быть таким, чтобы вмещать значения в диапазоне -2^{16} . . . 2^{16}*–* 1». Последнее правило, однако, не даёт нам фиксированный размер, потому что short мог бы быть шириной 8 байт и всё ещё удовлетворять этому ограничению. Так что эти требования далеки от того, чтобы высечь точные размеры в камне. Чтобы систематизировать разные наборы размеров, были созданы соглашения, называемые *моделями данных*. Каждое из них определяет размеры базовых типов. Рисунок 11-2 показывает некоторые примечательные модели данных, которые могут представлять для нас интерес.

short	int	long	ptr	long long	Name	Examples
-	16	-	16	-	IP16	PDP-11 Unix
16	16	32	32	-	I16LP32	Apple Macintosh 68K; early Microsoft Windows
16	32	32	32	-	ILP32	IBM 370; VAX Unix
16	32	32	32	64	ILP32LL	Amdahl; Microsoft Win32
16	32	32	64	64	LLP64 or IL32LLP64	Microsoft Win64
16	32	64	64	64	LP64	Most Unix systems (Linux, Solaris, HP UX 11, Mac OS...)
64	64	64	64	64	SILP64	UNICOS (Unix for Cray supercomputers)

Рисунок 11-2. Модели данных

Поскольку мы выбрали для изучения 64-битную систему GNU/Linux, наша модель данных — LP64. Когда вы разрабатываете для 64-битной системы Windows, размер `long` будет отличаться. Каждый хочет писать переносимый код, который можно повторно использовать на разных платформах, и, к счастью, есть соответствующий стандарту способ никогда не сталкиваться с изменениями модели данных. До C99 было обычной практикой создавать набор псевдонимов типов вида `int32` или `uint64` и использовать их исключительно по всей программе вместо вечно меняющихся `int` или `long`. Если целевая архитектура менялась, псевдонимы типов было легко исправить. Однако это порождало хаос, потому что каждый создавал собственный набор типов. C99 ввёл *платформонезависимые типы*. Чтобы их использовать, достаточно подключить заголовок `stdint.h`. Он даёт доступ к разным целочисленным типам фиксированного размера. Каждый из них имеет форму:

- `u`, если тип беззнаковый;
- `int`;
- Размер в битах: 8, 16, 32 или 64; и
- `_t`.

```
For example, uint8_t, int64_t, int16_t.
```

Функция `printf` (и схожие функции форматированного ввода/вывода) получили похожую обработку — были введены специальные макросы для выбора правильных спецификаторов формата. Они определены в файле `inttypes.h`.

В типичных случаях вам нужно читать или писать целые числа или указатели. Тогда имя макроса формируется следующим образом:

- `PRI` для вывода (`printf`, `fprintf` и т. д.) или `SCN` для ввода (`scanf`, `fscanf` и т. д.).

- Спецификатор формата: `d` для десятичного форматирования.
- `x` для шестнадцатеричного форматирования. `o` для восьмеричного форматирования.
- `u` для форматирования unsigned int. `i` для форматирования integer.
- Дополнительная информация включает одно из следующего: `N` для *N*-битных целых чисел.
- `PTR` для указателей. `MAX` для максимального поддерживаемого размера в битах.
- `FAST` определяется реализацией.

Нам нужно воспользоваться тем фактом, что несколько строковых литералов, разделённых пробелами, автоматически конкатенируются. Макрос создаст строку, содержащую правильный спецификатор формата, которая будет сконкатенирована с тем, что вокруг неё. Листинг 11-23 показывает пример.

Листинг 11-23. inttypes.c

```
#include <inttypes.h>
#include <stdio.h>
void f( void ) {
    int64_t i64 = -10;
    uint64_t u64 = 100;
    printf( "Signed 64-bit integer:   %" PRIi64 "\n", i64 );
    printf( "Unsigned 64-bit integer: %" PRIu64 "\n", u64 );
}
```

Полный список таких макросов см. в разделе 7.8.1 [7].

11.6 Потоки данных

Стандартная библиотека C предоставляет нам способ работать с файлами платформонезависимым образом. Она абстрагирует файлы как *потоки данных*, из которых можно читать и в которые можно писать. Мы видели, как файлы обрабатываются в Linux на уровне системных вызовов: системный вызов `open` открывает файл и возвращает его *дескриптор*, целое число, системные вызовы `write` и `read` используются для выполнения записи и чтения соответственно, а системный вызов `close` гарантирует корректное закрытие файла. Поскольку язык C создавался параллельно с операционной системой Unix, они разделяют один и тот же подход к взаимодействию с файлами. Библиотечные аналоги этих функций называются `fopen`, `fwrite`, `fread` и `fclose`. В Unix-подобных системах они действуют как адаптер для системных вызовов, предоставляя схожую функциональность, за исключением того, что они также работают на других платформах точно так же. Основные различия следующие:

1. Вместо дескрипторов файлов мы используем специальный тип `FILE`, хранящий всю информацию об определённом потоке. Его реализация скрыта, и вы никогда не должны изменять его внутреннее состояние вручную. Так что вместо работы с числовыми дескрипторами файлов (что зависит от платформы) мы используем `FILE` как чёрный ящик. Экземпляр `FILE` выделяется в куче внутренне самой библиотекой C, так что мы всегда будем работать с указателем на него, а не с самим экземпляром напрямую.
2. Хотя файловые операции в Unix более или менее единообразны, в C есть два типа потоков данных.

- Бинарные потоки состоят из необработанных байт, обрабатываемых «как есть».
- Текстовые потоки содержат символы, сгруппированные в строки; каждая строка завершается символом конца строки (зависящим от реализации).

Текстовые потоки ограничены рядом способов *на некоторых системах*.

- Длина строки может быть ограничена.
- Они могут работать только с печатаемыми символами, переводами строк, пробелами и табуляциями.
- Пробелы перед переводом строки могут исчезать. На некоторых операционных системах текстовые и бинарные потоки используют разные форматы файлов, и потому для работы с текстовым файлом способом, совместимым между всеми его программами, использование текстовых потоков обязательно. В то время как библиотека GNU C, обычно связываемая с GCC, не делает различий между бинарными и текстовыми потоками, на других платформах дело обстоит иначе, так что различать их критически важно.

Например, я видел ситуацию, когда чтение большого блока из файла изображения на Windows (компилятором был MSVC) завершалось преждевременно, потому что изображение, очевидно, было бинарным, тогда как связанный поток был создан в текстовом режиме. Стандартная библиотека предоставляет средства для создания потоков и работы с ними. Некоторые определённые в ней функции следует использовать только с текстовыми потоками (например, `fscanf`). Соответствующий заголовочный файл называется `stdio.h`. Разберём пример, показанный в листинге 11-24.

Листинг 11-24. `file_example.c`

```
int smth[]={1,2,3,4,5};
FILE* f = fopen( "hello.img", "rwb" );
fread( smth, sizeof(int), 1, f);
/* This line is optional. By means of `fseek` function we can navigate the file */
fseek( f, 0, SEEK_SET );
fwrite(smth, 5 * sizeof( int ), 1, f);
fclose( f );
```

- Экземпляр `FILE` создаётся вызовом функции `fopen`. Она принимает путь

к файлу и набор флагов, сжатых в строку. Здесь перечислены важные флаги `fopen`.

- `b` – открыть файл в бинарном режиме. Именно это создаёт настоящее различие между текстовыми и бинарными потоками. По умолчанию файлы открываются в текстовом режиме.
- `w` – открыть поток с возможностью записи в него.
- `r` – открыть поток с возможностью чтения из него.
- `+` – если вы просто напишете `w`, файл будет перезаписан. Когда присутствует +, записи будут дописываться в конец файла.

Если файл не существует, он будет создан.

Файл `hello.img` открывается в бинарном режиме и для чтения, и для записи. Содержимое файла будет перезаписано.

- После создания `FILE` хранит своего рода указатель на позицию внутри файла,

своего рода курсор. Чтения и записи двигают этот курсор дальше.

- Функция `fseek` используется для перемещения курсора без выполнения чтений или записей.

Она позволяет перемещать курсор относительно либо его текущей позиции, либо начала файла.

- Функции `fwrite` и `fread` используются для записи и чтения данных из открытого экземпляра

`FILE`. Возьмём, например, `fread`: она принимает буфер памяти, из которого читать. Два целочисленных параметра — это размер отдельного блока и количество читаемых блоков. Возвращаемое значение — количество блоков, успешно прочитанных из файла.

Чтение каждого блока атомарно: либо он читается полностью, либо не читается вовсе. В этом примере размер блока равен `sizeof(int)`, а количество блоков — один. Использование `fwrite` симметрично.

- `fclose` следует вызывать по завершении работы с файлом.

Существует специальная константа `EOF`. Когда её возвращает функция, работающая с файлом, это значит, что достигнут конец файла. Другая константа, `BUFSIZ`, хранит размер буфера, наиболее подходящий в текущем окружении для операций ввода и вывода. Потоки могут использовать буферизацию. Это значит, что у них есть внутренний буфер, через который проходят все чтения и записи. Это позволяет реже выполнять системные вызовы (которые дороги с точки зрения производительности из-за переключения контекста). Иногда, когда буфер заполняется, запись на самом деле вызывает системный вызов `write`. Буфер можно вручную `flushed` с помощью команды `fflush`. Все отложенные записи будут выполнены, а буфер сброшен. Когда программа запускается, создаются три экземпляра `FILE*`, привязанные к потокам с дескрипторами 0, 1 и 2. На них можно ссылаться как на `stdin`, `stdout` и `stderr`. Все три обычно используют буфер, но `stderr` автоматически сбрасывает буфер после каждой записи. Это необходимо, чтобы не задерживать и не терять сообщения об ошибках.

> **Примечание** Ещё раз: дескрипторы — это целые числа, а экземпляры `FILE` — нет. Функция `int fileno( FILE* stream )` используется, чтобы получить лежащий в основе дескриптор для файла `stream`.

```
■Question 212Read man for functions: fread, fread, fwrite, fprintf, fscanf, fopen, fclose, fflu
```

> **Вопрос 213** Проведите исследование и выясните, что произойдёт, если функцию `fflush` применить к двунаправленному потоку (открытому и для чтения, и для записи), когда последним действием над потоком перед этим было чтение.

11.7 Задание: функции высшего порядка и списки

11.7.1 Распространённые функции высшего порядка

В этом задании мы реализуем несколько функций высшего порядка над связными списками, которые должны быть знакомы тем, кто привык к парадигме функционального программирования. Эти функции известны под именами `foreach`, `map`, `map_mut` и `foldl`.

- `foreach` принимает указатель на начало списка и функцию (возвращающую void и принимающую `int`). Она запускает функцию на каждом элементе списка.
- `map` принимает функцию *f* и список. Она возвращает новый список, содержащий результаты применения *f* ко всем элементам исходного списка. Исходный список не затрагивается.

Например, $f(x) = x + 1$ отобразит список (1*,* 2*,* 3) в (2*,* 3*,* 4).

- `map_mut` делает то же самое, но изменяет исходный список.
- `foldl` немного сложнее. Она принимает:
- Начальное значение аккумулятора.
- Функцию $f(x, a)$.
- Список элементов.

Она возвращает значение того же типа, что и аккумулятор, вычисленное следующим образом:

1. Мы запускаем *f* на аккумуляторе и первом элементе списка. Результат — новое значение аккумулятора a'.
2. Мы запускаем *f* на a' и втором элементе списка. Результат — снова новое значение аккумулятора a''.
3. Мы повторяем процесс, пока список не будет исчерпан. В конце итоговое значение

аккумулятора и есть итоговый результат. Например, возьмём $f(x, a) = x * a$. Запустив `foldl` со значением аккумулятора 1 и этой функцией, мы вычислим произведение всех элементов списка.

- `iterate` принимает начальное значение *s*, длину списка *n* и функцию *f*. Затем она генерирует список

длины *n* следующим образом:

[s, f(s), f(f(s)), f(f(f(s))), ...]

Описанные выше функции называются *функциями высшего порядка*, потому что они принимают другие функции в качестве аргументов. Другой пример такой функции — функция сортировки массива `qsort`.

```
void qsort( void *base,
            size_t nmemb,
            size_t size,
            int (*compar)(const void *, const void *));
```

Она принимает начальный адрес массива base, количество элементов `nmemb`, размер отдельного элемента `size` и функцию-компаратор `compar`. Эта функция и есть тот, кто решает, какой из заданных элементов должен быть ближе к началу массива.

> **Вопрос 214** Прочитайте `man qsort`.

11.7.2 Задание

Вход содержит произвольное количество целых чисел.

1. Сохраните эти целые числа в связном списке.
2. Перенесите все функции, написанные в предыдущем задании, в отдельные файлы `.h` и `c`.

Не забудьте добавить защитную конструкцию включения!

3. Реализуйте `foreach`; используя её, выведите исходный список в `stdout` дважды:
   в первый раз

разделяя элементы пробелами, во второй раз выводя каждый элемент на новой строке.

4. Реализуйте `map`; используя её, выведите квадраты и кубы чисел из списка.
5. Реализуйте `foldl`; используя её, выведите сумму, а также минимальный и максимальный элемент

списка.

6. Реализуйте `map_mut`; используя её, выведите модули входных чисел.
7. Реализуйте `iterate;` используя её, создайте и выведите список степеней двойки
   (первые

10 значений: 1, 2, 4, 8, ...).

8. Реализуйте функцию `bool save(struct list* lst, const char* filename);`,

которая запишет все элементы списка в текстовый файл `filename`. Она должна возвращать `true`, если запись успешна, иначе `false`.

9. Реализуйте функцию `bool load(struct list** lst, const char* filename);`,

которая прочитает все целые числа из текстового файла filename и запишет сохранённый список в `*lst`. Она должна возвращать `true`, если запись успешна, иначе `false`.

10. Сохраните список в текстовый файл и загрузите его обратно с помощью двух функций выше. Убедитесь, что сохранение и загрузка корректны.

11. Реализуйте функцию `bool serialize(struct list* lst, const char* filename);`, которая запишет все элементы списка в *бинарный* файл filename. Она должна возвращать `true`, если запись успешна, иначе `false`.
12. Реализуйте функцию `bool deserialize(struct list** lst, const char* filename);`, которая прочитает все целые числа из *бинарного* файла filename и запишет сохранённый список в *lst. Она должна возвращать `true`, если запись успешна, иначе `false`.
13. Сериализуйте список в бинарный файл и загрузите его обратно с помощью двух функций выше. Убедитесь, что сериализация и десериализация корректны.
14. Освободите всю выделенную память.

Вам нужно будет научиться использовать

- Указатели на функции.
- `limits.h` и константы из него. Например, чтобы найти минимальный элемент в массиве, нужно использовать `foldl` с максимально возможным значением `int` в качестве аккумулятора и функцией, возвращающей минимум из двух элементов.
- Ключевое слово `static` для функций, которые вы хотите использовать только в одном модуле.

Вам гарантируется, что

- Входной поток содержит только целые числа, разделённые пробельными символами.
- Все числа на входе помещаются в `int`.

Вероятно, разумно написать отдельную функцию для чтения списка из `FILE`. Решение занимает около 150 строк кода, не считая функций, определённых в предыдущем задании.

> **Вопрос 215** В таких языках, как C#, возможен следующий код:

```
var count = 0;
mylist.Foreach(  x  =>  count  +=  1  );
```

> **Примечание.** Здесь мы запускаем анонимную функцию (то есть функцию, у которой нет имени, но чьим адресом можно манипулировать, например, передавать в другую функцию) для каждого элемента списка. Функция записана как `x  =>  count  +=  1` и эквивалентна

```
void no_name( int x ) { count += 1; }
```

> **Примечание.** Интересно здесь то, что эта функция осведомлена о некоторых локальных переменных вызывающей стороны и потому может их изменять. Можете ли вы переписать функцию `forall` так, чтобы она принимала указатель на своего рода «контекст», способный хранить произвольное количество адресов переменных, а затем передавала этот контекст функции, вызываемой для каждого элемента?

11.8 Итоги

В этой главе мы изучили модель памяти. Мы лучше разобрались в размерах типов и моделях данных, изучили арифметику указателей и научились расшифровывать сложные объявления типов. Кроме того, мы увидели, как использовать функции стандартной библиотеки для выполнения ввода и вывода. Мы попрактиковались в этом, реализовав несколько функций высшего порядка и выполнив небольшой ввод/вывод файлов. Мы ещё глубже разберёмся в раскладке памяти в следующей главе, где рассмотрим разницу между тремя «гранями» языка (синтаксисом, семантикой и прагматикой), изучим понятия неопределённого и неуточнённого поведения и покажем, почему важно выравнивание данных.

> **Вопрос 216** Какие арифметические операции можно выполнять над указателями и при каких условиях?

> **Вопрос 217** Каково назначение `void*`?

> **Вопрос 218** Каково назначение `NULL`?

> **Вопрос 219** В чём разница между 0 в контексте указателя и 0 как целочисленным значением?

> **Вопрос 220** Что такое `ptrdiff_t` и как он используется?

> **Вопрос 221** В чём разница между `size_t` и `ptrdiff_t`?

> **Вопрос 222** Что такое объекты первого класса?

Вопрос 223 Являются ли функции объектами первого класса в C?

Вопрос 224 Какие области данных содержит абстрактная машина C?

Вопрос 225 Обычно ли область константных данных защищена от записи оборудованием?

Вопрос 226 Какая связь между указателями и массивами?

Вопрос 227 Что такое динамическое выделение памяти?

Вопрос 228 Что такое оператор `sizeof` ? Когда он вычисляется?

Вопрос 229 Когда строковые литералы хранятся в **.rodata**?

Вопрос 230 Что такое интернирование строк?

Вопрос 231 Какую модель данных мы используем?

Вопрос 232 Какой заголовок содержит платформонезависимые типы?

Вопрос 233 Как конкатенировать строковые литералы во время компиляции?

Вопрос 234 Что такое поток данных?

Вопрос 235 Есть ли разница между потоком данных и дескриптором?

Вопрос 236 Как получить дескриптор из потока?

Вопрос 237 Открыты ли какие-нибудь потоки при запуске программы?

Вопрос 238 В чём разница между бинарными и текстовыми потоками?

Вопрос 239 Как открыть бинарный поток? Текстовый поток?

Синтаксис, семантика и прагматика

В этой главе мы пересмотрим саму суть того, что такое язык программирования. Эти основы позволят нам лучше понять структуру языка, поведение программы и детали трансляции, о которых вам следует знать.

12.1 Что такое язык программирования?

Язык программирования — формальный компьютерный язык, предназначенный для описания алгоритмов способом, понятным машине. Каждая программа — это последовательность символов. Но как отличить программы от всех остальных строк? Нужно как-то определить язык. Грубый способ — сказать, что сам компилятор и есть определение языка, поскольку он разбирает программы и транслирует их в исполняемый код. Такой подход плох по ряду причин. Что делать с ошибками компилятора? Действительно ли это ошибки, или они влияют на определение языка? Как нам писать другие компиляторы? Зачем смешивать определение языка с деталями реализации? Другой способ — предоставить более чистое и независимое от реализации описание языка. Довольно распространено рассматривать три грани одного языка.

- Правила построения операторов. Описание корректно структурированных программ часто делается с помощью формальных грамматик. Эти правила образуют *синтаксис* языка.
- Эффекты каждой языковой конструкции на абстрактную машину. Это *семантика* языка.
- В любом языке есть также третий аспект, называемый *прагматикой*. Он описывает влияние реальной реализации на поведение программы.
- В некоторых ситуациях стандарт языка не предоставляет достаточно информации о поведении программы. Тогда решение о том, как транслировать эту программу, полностью остаётся за компилятором, так что он часто приписывает таким программам некоторое конкретное поведение. Например, в вызове `f(g(x), h(x))` порядок вычисления `g(x)` и `h(x)` не определён стандартом. Мы можем вычислить сначала `g(x)`, а затем `h(x)`, или наоборот. Но компилятор выберет определённый порядок и сгенерирует инструкции, выполняющие вызовы именно в этом порядке.
- Иногда существуют разные способы транслировать языковые конструкции в целевой код. Например, хотим ли мы запретить компилятору встраивать (inline) определённые функции, или же придерживаемся стратегии невмешательства?

В этой главе мы рассмотрим эти три грани языков и применим их к C.

12.2 Синтаксис и формальные грамматики

Прежде всего, язык — это подмножество всех возможных строк, которые можно построить из определённого алфавита. Например, у языка арифметических выражений есть алфавит Σ = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, −, ×, /, .}, при условии что используются только эти четыре арифметические операции, а точка отделяет целую часть. Не все комбинации этих символов образуют допустимую строку — например, +++-+ не является допустимым предложением этого языка. Формальные грамматики были впервые формализованы Ноамом Хомским. Они были созданы в попытке формализовать естественные языки, такие как английский. Согласно им, предложения имеют древовидную структуру, где листья — своего рода «базовые блоки», а более сложные части строятся из них (и других сложных частей) по определённым правилам. Все эти примитивные и составные части обычно называются *символами*. Атомарные символы называются *терминалами*, а сложные — *нетерминалами*. Этот подход был взят на вооружение для построения синтетических языков с очень простыми (по сравнению с естественными языками) грамматиками. Формально грамматика состоит из

- Конечного множества терминальных символов.
- Конечного множества нетерминальных символов.
- Конечного множества правил вывода (продукций), хранящих информацию о структуре языка.
- Начального символа, нетерминала, которому будет соответствовать любой корректно построенный

оператор языка. Это отправная точка для разбора любого оператора. Класс грамматик, который нас интересует, обладает очень особой формой правил вывода. Каждое из них выглядит как

```
<nonterminal> ::= sequence of terminals and nonterminals
```

Как видим, это в точности описание сложной структуры нетерминала. Можно написать несколько возможных правил для одного и того же нетерминала, и будет применяться подходящее. Чтобы сделать это менее многословным, мы будем использовать обозначение с символом | для «или», точно так же, как в регулярных выражениях. Такой способ описания правил грамматики называется *БНФ (форма Бэкуса — Наура, BNF)*: терминалы обозначаются строками в кавычках, правила вывода записываются с помощью символов ::=, а имена нетерминалов записываются в угловых скобках. Иногда также довольно удобно ввести терминал ϵ, который при разборе будет сопоставлен с пустой (под)строкой. Итак, грамматики — это способ описать структуру языка. Они позволяют выполнять следующие виды задач:

- Проверить оператор языка на синтаксическую корректность.

- Сгенерировать корректные операторы языка.
- Разобрать операторы языка в иерархические структуры, где, например, условие

`if` отделяется от окружающего его кода и разворачивается в древовидную структуру, готовую к вычислению.

12.2.1 Пример: натуральные числа

Язык натуральных чисел можно представить с помощью грамматики. В качестве алфавита возьмём такой набор символов: Σ = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}. Однако нам нужно более приличное представление, чем просто все возможные строки, построенные из символов Σ, потому что числа с ведущими нулями (000124) выглядят некрасиво. Определим несколько нетерминальных символов: сначала `<notzero>` для любой цифры, кроме нуля, `<digit>` для любой цифры и `<raw>` для любой последовательности `<digit>s`. Как мы знаем, для одного нетерминала возможно несколько правил. Так что, чтобы определить `<notzero>`, можно написать столько правил, сколько есть разных вариантов:

```
<notzero> ::= '1'
<notzero> ::= '2'
<notzero> ::= '3'
<notzero> ::= '4'
<notzero> ::= '5'
<notzero> ::= '6'
<notzero> ::= '7'
<notzero> ::= '8'
<notzero> ::= '9'
```

Однако, поскольку это очень громоздко и не так легко читается, мы используем другую нотацию для описания в точности тех же правил:

```
<notzero> ::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

Эта нотация — часть канонической *БНФ*. Добавив ноль, мы получаем правило для нетерминала `<digit>`, кодирующего любую цифру.

```
<digit> ::= '0' | <notzero>
```

Затем мы определяем нетерминал `<raw>` для кодирования всех последовательностей цифр. Последовательность цифр определяется рекурсивно как либо одна цифра, либо цифра, за которой следует другая последовательность цифр.

```
<raw> ::= <digit> | <digit> <raw>
```

`<number>` послужит нам начальным символом. Либо мы имеем дело с однозначным числом, на которое не накладывается никаких ограничений, либо у нас несколько цифр, и тогда первая не должна быть нулём (иначе это ведущий ноль, который мы не хотим видеть); остальные могут быть произвольными. Листинг 12-1 показывает итоговый результат.

Листинг 12-1. grammar_naturals

```
<notzero> ::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
<digit> ::= '0' | <notzero>
<raw> ::= <digit> | <digit> <raw>
<number> ::=    <digit> | <notzero> <raw>
```

12.2.2 Пример: простая арифметика

Добавим пару простых бинарных операций. Для начала мы ограничимся сложением и умножением. Мы построим это на основе примера, показанного в листинге 12-1. Добавим нетерминал `<expr>`, который послужит новым начальным символом. Выражение — это либо число, либо число, за которым следует символ бинарной операции и ещё одно выражение (так что выражение тоже определяется рекурсивно). Листинг 12-2 показывает пример.

Листинг 12-2. grammar_nat_pm

```
<notzero> ::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
<digit> ::= '0' | <notzero>
<raw> ::= <digit> | <digit> <raw>
<number> ::=    <digit> | <notzero> <raw>
<expr> ::= <number> | <number> '+' <expr> | <number> '-' <expr>
```

Грамматика позволяет нам построить древовидную структуру поверх текста, где каждый лист — терминал, а каждый остальной узел — нетерминал. Например, применим текущий набор правил к строке 1+42 и посмотрим, как она раскладывается. Рисунок 12-1 показывает результат.

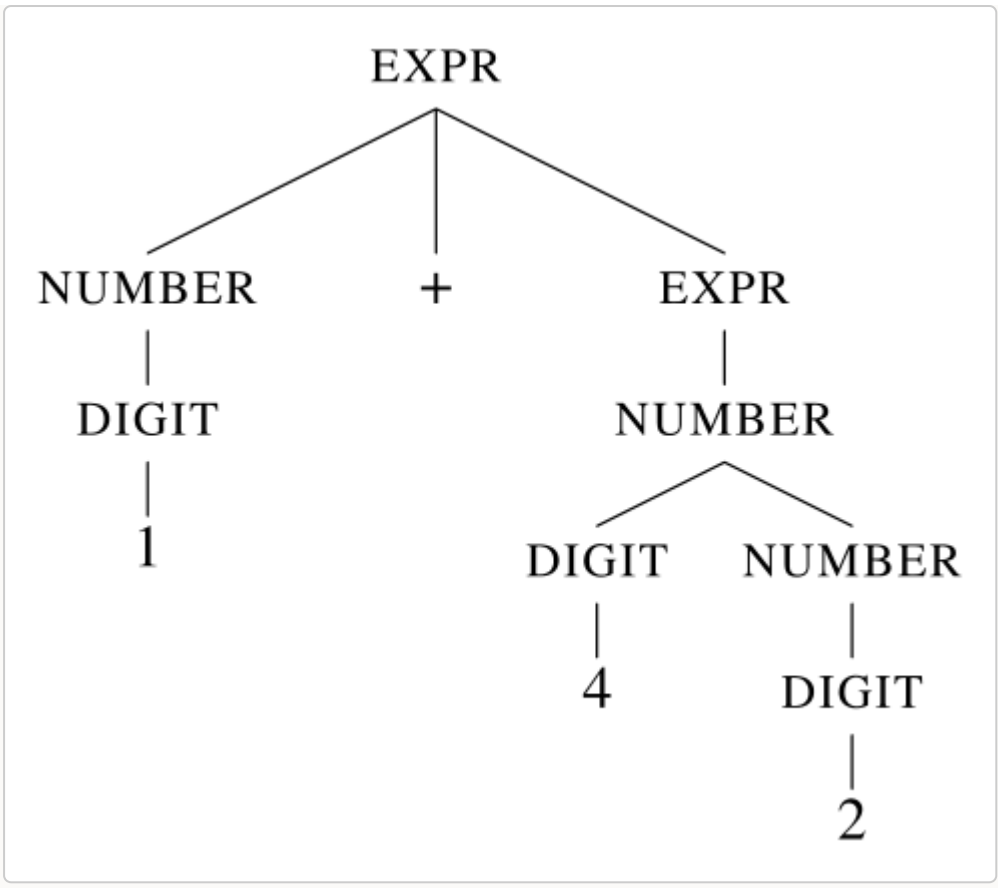


Рисунок 12-1. Дерево разбора для выражения 1+42

Первое раскрытие выполняется согласно правилу `<expr> ::= number '+' <expr>`. Последнее выражение — это просто число, которое, в свою очередь, представляет собой последовательность цифры и числа.

12.2.3 Рекурсивный спуск

Писать парсеры вручную не сложно. Чтобы это проиллюстрировать, мы покажем парсер, применяющий наши новые знания о грамматиках для буквального перевода описания грамматики в код разбора. Возьмём грамматику для натуральных чисел, которую мы уже описали в разделе 12.2.1, и добавим к ней ещё одно правило. Новым начальным символом будет `str`, соответствующий «числу, завершённому нуль-терминатором». Листинг 12-3 показывает пересмотренное определение грамматики.

Листинг 12-3. grammar_naturals_nullterm

```
<notzero> ::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
<digit> ::= '0' | <notzero>
<raw> ::= <digit> | <digit> <raw>
<number> ::= <digit> | <notzero> <raw>
<str> ::= <number> '\0'
```

При разборе по правилам грамматики обычно оперируют понятием потока. Поток — это последовательность того, что считается символами. Его интерфейс состоит из двух функций:

- `bool expect(symbol)` принимает один терминал и возвращает true, если поток содержит именно этот терминал в текущей позиции.
- `bool accept(symbol)` делает то же самое, а затем в случае успеха продвигает позицию потока на один символ.

До сих пор мы оперировали такими абстракциями, как символы и потоки. Мы можем отобразить все абстрактные понятия на конкретные экземпляры. В нашем случае символ будет соответствовать одному `char`.[27] Листинг 12-4 показывает пример текстового обработчика, построенного на основе определений правил грамматики. Это синтаксический проверятель, определяющий, содержит ли строка натуральное число без ведущих нулей и ничего больше (например, пробелов вокруг числа).

Листинг 12-4. rec_desc_nat.c

```
#include <stdio.h>
#include  <stdbool.h>
char const* stream = NULL ;
bool accept(char c) {
    if (*stream == c) {
        stream++;
        return  true;
    }
    else return false;
}
bool notzero( void ) {
    return accept( '1' ) || accept( '2' ) || accept( '3' )
    || accept( '4' )     || accept( '5' ) || accept( '6' )
    || accept( '7' )     || accept( '8' ) || accept( '9' );
}
bool digit( void ) {
    return accept('0') || notzero();
}
```

```
bool raw( void ) {
    if ( digit() ) { raw(); return true; }
    return false;
}
bool number( void ) {
    if ( notzero() ) {
         raw();
         return true;
    }  else return accept('0');
}
bool str( void ) {
    return number() && accept( 0 );
}
void check( const char* string ) {
    stream = string;
    printf("%s -> %d\n", string, str() );
}
int main(void) {
    check("12345");
    check("hello12");
    check("0002");
    check("10dbd");
    check("0");
    return 0;
}
```

Этот пример показывает, как каждый нетерминал сопоставляется с одноимённой функцией, пытающейся применить соответствующие правила грамматики. Разбор происходит сверху вниз: мы начинаем с самого общего начального символа и пытаемся разбить его на части и разобрать их. Когда правила начинают быть похожими, мы их *факторизуем*, сначала применяя общую часть, а затем пытаясь потребить остаток, как в функции `number`. Обе ветви начинаются с перекрывающихся нетерминалов: `<digit>` и `<notzero>`. Каждый из них содержит диапазон `1...9`, единственное различие в том, что `<digit>'s` диапазон включает ноль. Так что если мы нашли терминал в диапазоне `1...9`, мы пытаемся потребить после этого как можно больше цифр, и это в любом случае удаётся. Если нет, мы проверяем, является ли первая цифра 0, и останавливаемся, если это так, не потребляя больше терминалов. Функция `<notzero>` завершается успехом, если найден хотя бы один символ в диапазоне 1-9. Из-за ленивого применения || не все вызовы `accept` будут выполнены. Первый из них, что завершится успехом, закончит вычисление выра-

жения, так что произойдёт только одно продвижение по потоку. Функция `<digit>` завершается успехом, если найден ноль или если `<notzero>` завершился успехом, что является буквальным переводом правила:

```
<digit> ::= '0' | <notzero>
```

Остальные функции работают точно так же. Если бы мы не ограничивались нуль-терминатором, разбор ответил бы нам на вопрос: «начинается ли эта последовательность символов с допустимого предложения языка?» В листинге 12-4 мы намеренно использовали глобальную переменную, чтобы облегчить понимание. Мы всё же настоятельно не рекомендуем использовать их в реальных программах. Парсеры для реальных языков программирования обычно довольно сложны. Чтобы их писать, программисты используют специальный набор инструментов, способный генерировать парсеры из декларативного описания, близкого к БНФ. Если вам нужно написать парсер для сложного языка, мы рекомендуем вам взглянуть на генераторы парсеров ANTLR или yacc. Другая популярная техника ручного написания парсеров называется *комбинаторами парсеров (parser combinators)*. Она поощряет создание парсеров для самых базовых обобщённых текстовых элементов (одиночного символа, числа, имени переменной и т. д.). Затем эти маленькие парсеры комбинируются (ИЛИ, И, последовательность...) и преобразуются (одно или несколько вхождений, ноль или более вхождений...), чтобы получить более сложные парсеры. Эту технику, однако, легко применять, когда язык поддерживает функциональный стиль программирования, потому что она часто опирается на функции высшего порядка.

> **О рекурсии в грамматиках** Правила грамматики могут быть рекурсивными, как мы видим. Однако, в зависимости от техники разбора, использование определённых типов рекурсии может быть неразумным. Например, правило `expr ::= expr '+' expr`, хотя и допустимое, не позволит нам легко построить парсер. Чтобы хорошо написать грамматику в этом смысле, следует избегать левой рекурсии в правилах, подобных перечисленному выше, потому что при наивном кодировании это приведёт только к бесконечной рекурсии, когда функция `expr()` начнёт своё выполнение с очередного вызова `expr()`. Правила, уточняющие первый нетерминал в правой части продукции, избегают этой проблемы.

> **Вопрос 240** Напишите парсер рекурсивного спуска для арифметики с плавающей точкой с умножением, вычитанием и сложением. Для этого задания будем считать, что отрицательных литералов не существует (так что вместо того чтобы писать -1.20, мы будем писать 0-1.20.

12.2.4 Пример: арифметика с приоритетами

Интересная часть выражений в том, что у разных операций разные приоритеты. Например, у операции сложения приоритет ниже, чем у операции умножения, так что все умножения выполняются раньше сложения. Посмотрим на наивную грамматику для натуральных чисел со сложением и умножением в листинге 12-5.

Листинг 12-5. grammar_nat_pm_mult

```
<notzero> ::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
<digit> ::= '0' | <notzero>
<raw> ::= <digit> | <digit> <raw>
<number> ::= <digit> | <notzero> <raw>
<expr> ::= <number> | <number> '+' <expr>
       | <number> '-' <expr> | <number> '*' <expr>
```

Без учёта приоритета умножения дерево разбора для выражения `1*2+3` будет выглядеть так, как показано на рисунке 12-2.

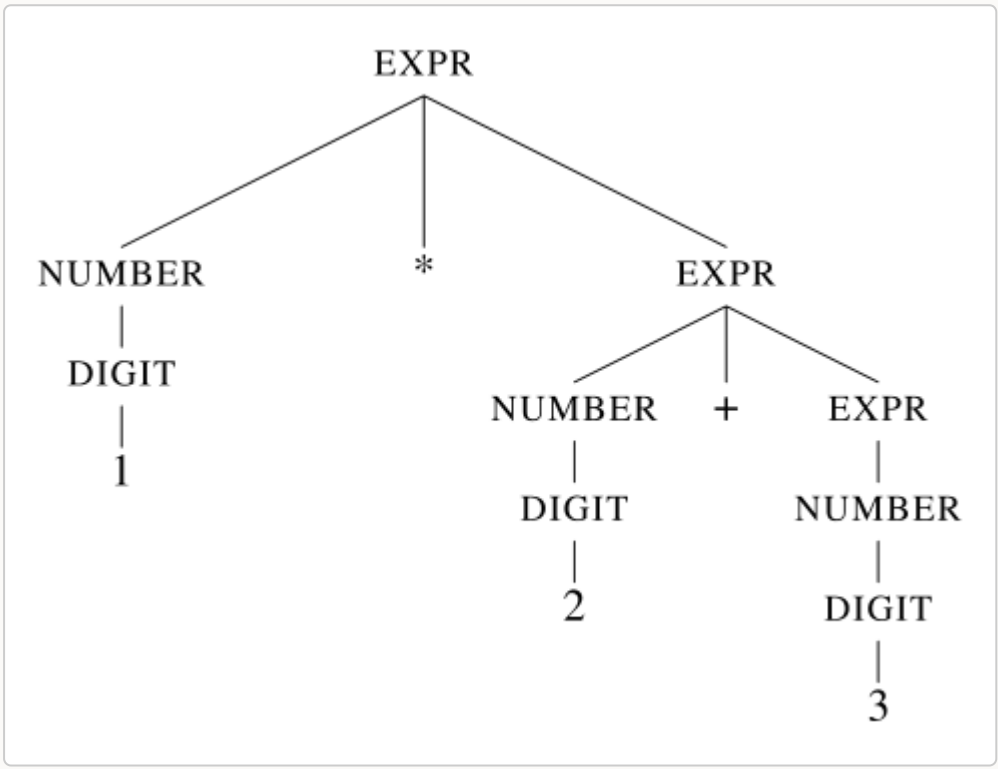


Рисунок 12-2. Деревья разбора без приоритетов для выражения `1*2+3`

Однако, как мы замечаем, умножение и сложение здесь равноправны: они раскрываются в порядке появления. Из-за этого выражение `1*2+3` разбирается как `1*(2+3)`, нарушая привычный порядок вычисления, привязанный к структуре дерева. С точки зрения парсера приоритет означает, что в дереве разбора узлы «сложить» должны быть ближе к корню, чем узлы «умножить», поскольку сложение выполняется над более крупными

частями выражения. Вычисление арифметических выражений неформально выполняется начиная с листьев и заканчивая корнем. Как нам расставить приоритеты одних операций над другими? Это достигается разбиением одной синтаксической категории `<expr>` на несколько классов. Каждый класс — своего рода уточнение предыдущего класса. Листинг 12-6 показывает пример.

Листинг 12-6. grammar_priorities

```
<expr> ::= <expr0> "<" <expr> | <expr0> "<=" <expr>
| <expr0> "==" <expr> | <expr0> ">" <expr> | <expr0> ">=" <expr> | <expr0>
<expr0> = <expr1> "+" <expr> | <expr1> "-" <expr> | <expr1>
<expr1> ::= <atom> "*" <expr1> | <atom> "/" <expr1> | <atom>
<atom> ::= "(" <expr> ")" | <NUMBER>
```

Этот пример можно понимать следующим образом:

- `<expr>` — это действительно любое выражение.
- `<expr0>` — это выражение без <, >, == и других терминалов, присутствующих в

первом правиле.

- `<expr1>` также свободно от сложения и вычитания.

12.2.5 Пример: простой императивный язык

Чтобы проиллюстрировать, что эти знания можно применить к языкам программирования, приведём пример синтаксиса одного из них. Это описание синтаксиса даёт определения операторов, составляющих типичные императивные конструкции: `if`, `while`, `print` и присваивания. Ключевые слова можно рассматривать как атомарные терминалы. Листинг 12-7 показывает грамматику.

Листинг 12-7. imp

```
<statements> ::= <statement> | <statement> ";" <statements>
<statement> ::= "{" <statements> "}" | <assignment> | <if> | <while> | <print>
<print> ::= "print" "(" <expr> ")"
<assignment>  ::=  IDENT  "="  <expr>
<if> ::= "<if>" "(" <expr> ")" <statement> "<else>" <statement>
<while> ::= "<while>" "(" <expr> ")" <statement>
<expr> ::= <expr0> "<" <expr> | <expr0> "<=" <expr>
| <expr0> "==" <expr> | <expr0> ">" <expr> | <expr0> ">=" <expr> | <expr0>
<expr0> = <expr1> "+" <expr> | <expr1> "-" <expr> | <expr1>
<expr1> ::= <atom> "*" <expr1> | <atom> "/" <expr1> | <atom>
<atom> ::= "(" <expr> ")" | NUMBER
```

12.2.6 Иерархия Хомского

Формальные грамматики, которые мы изучили, на самом деле лишь подкласс формальных грамматик в понимании Хомского. Этот класс называется *контекстно-свободными грамматиками* по причинам, которые скоро станут очевидны. Иерархия состоит из четырёх уровней, от 3 до 0, где более низкие уровни более выразительны и мощны.

3. Регулярные грамматики на удивление описываются нашими старыми знакомыми — *регулярными выражениями*. Конечные автоматы — самый слабый тип парсеров, потому что они не могут обрабатывать фрактальные структуры вроде арифметических выражений. Даже в простейшем случае, `<expr> ::= number '+' <expr>`, часть выражения справа от ‘+’ похожа на всё выражение целиком. Это правило можно применять рекурсивно произвольное количество раз.
4. Контекстно-свободные грамматики, которые мы уже изучили, имеют правила формы

```
nonterminal ::=
      <sequence  of  terminal  and  nonterminal  symbols>
```

Любое регулярное выражение также можно описать в терминах контекстно-свободных грамматик.

1. У *контекстно-зависимых грамматик* правила имеют форму:

```
a A  b ::= a y b
```

`a` и `b` обозначают произвольную (возможно, пустую) последовательность терминалов и/или нетерминалов, `y` обозначает *непустую* последовательность терминалов и/или нетерминалов, а A — раскрываемый нетерминал. Разница между уровнями 2 и 1 в том, что нетерминал слева заменяется на `y` только тогда, когда он встречается между `a` и `b` (которые остаются нетронутыми). Помните, и `a`, и `b` могут быть довольно сложными.

0. У *неограниченных грамматик* правила имеют форму:

```
sequence of terminal and nonterminal symbols ::=
    sequence of terminal and nonterminal symbols
```

Поскольку на левую и правую части правил не накладывается вообще никаких ограничений, эти грамматики самые мощные. Можно показать, что такие типы грамматик можно использовать для кодирования любой компьютерной программы, так что эти грамматики полны по Тьюрингу. Настоящие языки программирования почти никогда не являются по-настоящему контекстно-свободными. Например, использование ранее объявленной переменной, очевидно, контекстно-зависимая конструкция, потому что она допустима только после соответствующего объявления переменной. Однако для простоты их часто аппроксимируют контекстно-свободными грамматиками, а затем выполняют дополнительные проходы по преобразованию дерева разбора, чтобы проверить выполнение таких контекстно-зависимых условий.

12.2.7 Абстрактное синтаксическое дерево

Существует понятие *абстрактного синтаксиса*. Он описывает деревья, построенные из исходного кода. *Конкретный синтаксис* описывает точное соответствие между ключевыми словами и типами узлов дерева, на которые они отображаются. Например, представим, что мы переписали компилятор C так, что ключевое слово `while` заменено на `_while_`. Затем представим, что мы переписали все программы так, что вместо `while` используется это новое ключевое слово. Конкретный синтаксис действительно изменился, но абстрактный синтаксис остался тем же, потому что языковые конструкции не изменились. Напротив, если мы добавим к `if` конструкцию `finally`, включающую оператор, который выполняется независимо от значения условия, мы изменим и абстрактный синтаксис тоже. Абстрактное синтаксическое дерево обычно также намного минималистичнее по сравнению с деревьями разбора. Дерево разбора хранило бы информацию, важную только для разбора (см. рисунок 12-3).

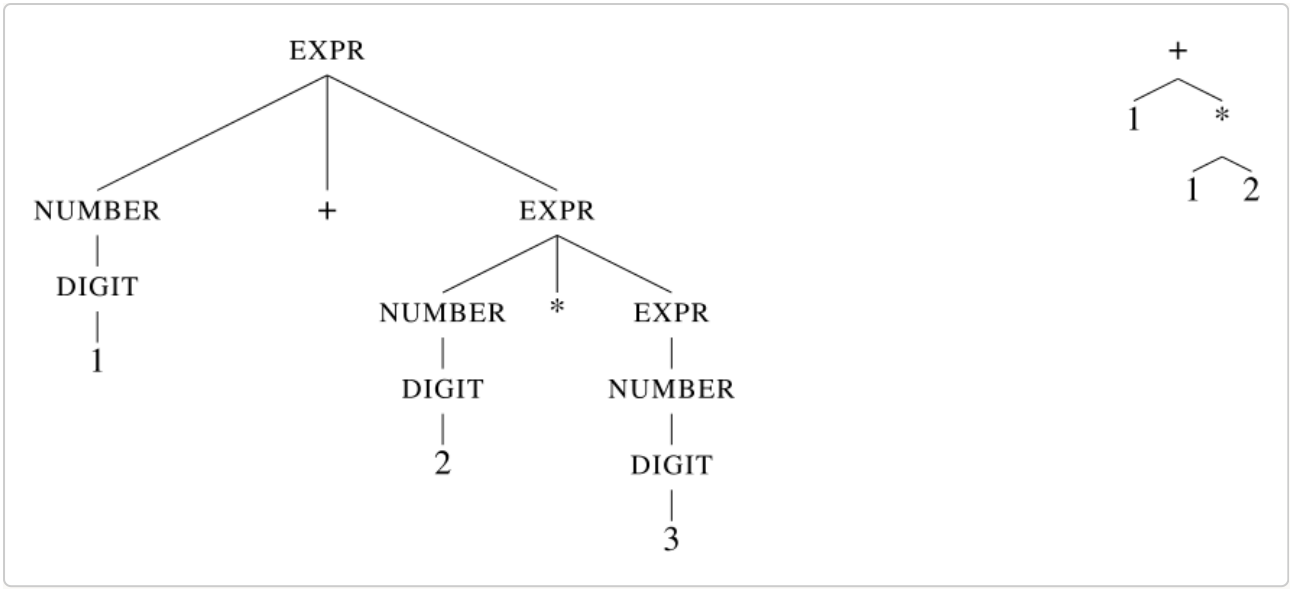


Рисунок 12-3. Дерево разбора и абстрактное синтаксическое дерево выражения `1 + 2*3`

Как видим, дерево справа намного лаконичнее и по существу. Это дерево можно напрямую вычислить интерпретатором или каким-то исполняемым кодом, чтобы посчитать то, что могло бы быть сгенерировано.

12.2.8 Лексический анализ

На практике применять правила грамматики напрямую к отдельным символам — излишество. Удобно добавить предварительный проход, называемый *лексическим анализом*. Исходный текст сначала преобразуется в последовательность лексем (также называемых *токенами*). Каждый токен описывается регулярным выражением и извлекается из потока символов. Например, число можно описать регулярным выражением `[0-9]+`, а идентификатор — `[a-zA-Z_][0-9a-zA-Z_]*`. После такой обработки текст больше не будет плоской последовательностью символов, а скорее связным списком токенов. Каждый токен будет помечен своим типом, и для парсера типы токенов будут отображены на терминалы. На этом этапе легко игнорировать все детали форматирования (такие как переводы строк и другие пробельные символы).

12.2.9 Итоги по разбору

Компилятор разбирает исходный код в несколько этапов. Два важных этапа — лексический и синтаксический анализ. Во время лексического анализа текст программы разбивается на лексемы, такие как целочисленные литералы или ключевые слова. После этого этапа форматирование текста уже не важно. Каждый тип лексемы лучше всего описывать регулярным выражением. Во время синтаксического анализа поверх потока токенов строится древовидная структура. Эта структура называется абстрактным синтаксическим деревом. Каждый узел соответствует языковой конструкции.

12.3 Семантика

Семантика языка — это соответствие между предложениями как синтаксическими конструкциями и их смыслом. Каждое предложение обычно описывается как тип узла в *абстрактном синтаксическом дереве* программы. Это описание выполняется одним из следующих способов:

- *Аксиоматически.* Текущее состояние программы можно описать набором логических формул. Тогда каждый шаг абстрактной машины будет преобразовывать эти формулы определённым образом.
- *Денотационально.* Каждое предложение языка отображается на математический объект определённой теории (например, теории доменов). Тогда эффекты программы можно описать в терминах этой теории. Это представляет особый интерес при рассуждениях о поведении разных программ, написанных на разных языках.
- *Операционально.* Каждое предложение вызывает определённое изменение состояния абстрактной машины, подлежащее описанию. Описания в стандарте C неформальны, но больше напоминают операционное семантическое описание, чем два других.

Стандарт языка — это описание языка в человекочитаемой форме. Однако, будучи более понятным для неподготовленного читателя, оно многословнее и порой менее однозначно. Чтобы писать сжатые описания, обычно используют язык математической логики и лямбда-исчисления. Мы не будем углубляться в детали в этой книге, потому что эта тема требует отдельного педантичного подхода. Мы отсылаем вас к книгам [29] и [35] для безупречного изучения теории типов и семантики языков.

12.3.1 Неопределённое поведение

Полнота описания семантики не гарантируется. Это значит, что некоторые языковые конструкции определены только для подмножества всех возможных ситуаций. Например, разыменование указателя `*x` гарантированно ведёт себя согласованно только тогда, когда `x` указывает на «действительную» ячейку памяти. Когда `x` равен `NULL` или указывает на освобождённую память, возникает *неопределённое поведение*. Однако такое выражение абсолютно корректно синтаксически. Стандарт намеренно вводит случаи неопределённого поведения. Зачем? Прежде всего, компиляторы, генерирующие код с меньшим числом гарантий, писать проще. Во-вторых, всё определённое поведение должно быть реализовано. Если мы хотим, чтобы разыменование нулевого указателя вызывало ошибку, компилятору придётся делать две вещи каждый раз, когда разыменовывается *любой* указатель:

- Попытаться вывести, что в этом конкретном месте указатель никогда не может иметь значение `NULL`.
- Если компилятор не может вывести, что этот указатель никогда не `NULL`, он генерирует ассемблерный код,

чтобы это проверить. Если указатель `NULL`, этот код выполнит для него обработчик. Иначе он продолжит разыменование указателя. Листинг 12-8 показывает пример.

Листинг 12-8. ptr_analysis1.c

```
int x = 0;
int* p = &x;
...
/* there are no writes to `p` in these lines */
...
*p =  10; /* this pointer can not be NULL  */
```

Однако это гораздо хитрее, чем может показаться. В примере из листинга 12-8 мы могли бы предположить, что, поскольку в переменную `p` не выполняется никаких записей, она всегда хранит адрес `x`. Однако это не всегда так, что иллюстрирует пример, показанный в листинге 12-9.

Листинг 12-9. ptr_analysis2.c

```
int x = 0;
int* p = &x;
...
/* there are no writes to `p` in these lines */
int**  z  =  &p;
*z = NULL; /* Still not a direct write to `p` */
...
*p =  10; /* this pointer can not be NULL -- not true anymore */
```

Так что решение этой проблемы на самом деле требует *очень* сложного анализа при наличии арифметики указателей. Как только берётся адрес переменной, или, что ещё хуже, её адрес передаётся в функцию, приходится анализировать всю последовательность вызовов функций, учитывать указатели на функции, указатели на указатели и т. д. Анализ не всегда даёт корректные результаты (в самом общем случае эта задача даже теоретически неразрешима), и производительность может из-за этого страдать. Так что, в соответствии с духом невмешательства C, корректность разыменования указателей оставлена на ответственность самого программиста. В управляемых языках, таких как Java или C#, определённого поведения при разыменовании указателей добиться гораздо легче. Во-первых, они обычно выполняются внутри каркаса, предоставляющего код для генерации и обработки исключений. Во-вторых, анализ на nullability гораздо проще в отсутствие арифметики адресов. Наконец, они обычно компилируются just-in-time, а значит, у компилятора есть доступ к информации времени выполнения, и он может использовать её для выполнения некоторых оптимизаций, недоступных

компилятору с опережающей компиляцией (ahead-of-time). Например, после запуска программы и получения пользовательского ввода компилятор вывел, что указатель `x` никогда не бывает `NULL`, если выполняется определённое условие `P`. Тогда он может сгенерировать две версии функции `f`, содержащей это разыменование: одну с проверкой и другую без проверки. Тогда каждый раз, когда вызывается `f`, вызывается только одна из двух версий. Если компилятор может доказать, что `P` выполняется в ситуации вызова, вызывается непроверяемая версия; иначе вызывается проверяемая. Неопределённое поведение может быть опасным (и обычно является таковым). Оно приводит к неочевидным ошибкам, потому что оно *не* гарантирует ошибку во время компиляции или выполнения. Программа может столкнуться с ситуацией неопределённого поведения и молча продолжить выполнение; однако её поведение случайным образом изменится после выполнения определённого количества инструкций. Типичная ситуация — повреждение кучи. Куча на самом деле структурирована; каждый блок ограничен служебной информацией, используемой стандартной библиотекой. Запись за границы блока (но близко к ним) с большой вероятностью повредит эту информацию, что приведёт к сбою во время одного из *будущих* вызовов `malloc` или `free`, превращая эту ошибку в бомбу замедленного действия. Вот несколько случаев неопределённого поведения, явно указанных в стандарте C99. Мы не приводим полный список, потому что таких случаев не менее 190.

- Переполнение знакового целого числа.
- Разыменование недействительного указателя.
- Сравнение указателей на элементы двух разных блоков памяти.
- Вызов функции с аргументами, не соответствующими её исходной сигнатуре (возможно при взятии указателя на неё и приведении к другому типу функции).
- Чтение из неинициализированной локальной переменной.
- Деление на 0.
- Обращение к элементу массива за его границами.
- Попытка изменить строковый литерал.
- Возвращаемое значение функции, у которой не выполнен оператор `return`.

12.3.2 Неуточнённое поведение

Важно различать неопределённое и неуточнённое поведение. Неуточнённое поведение определяет набор возможных вариантов поведения, но не указывает, какой именно будет выбран. Выбор будет зависеть от компилятора. Например,

- Порядок вычисления аргументов функции не уточнён. Это значит, что при вычислении `f(g(), h())` у нас нет гарантий, что `g()` будет вычислена первой, а `h()` — второй. Однако гарантируется, что и `g()`, и `h()` будут вычислены до `f()`.
- Порядок вычисления подвыражений в целом, `f(x) + g(x)`, не требует, чтобы `f` выполнялась раньше `g`. Неуточнённое поведение описывает случаи недетерминизма

в абстрактной машине C.

12.3.3 Поведение, определяемое реализацией

Стандарт также определяет поведение, определяемое реализацией, например, размер `int` (который, как мы уже сказали, зависит от архитектуры). Такие выборы можно рассматривать как параметры абстрактной машины: прежде чем её запустить, нам нужно выбрать эти параметры. Другой пример такого поведения — операция взятия остатка `x % y`. Результат в случае отрицательного `y` определяется реализацией. В чём разница между поведением, определяемым реализацией, и неуточнённым поведением? Ответ в том, что реализация (компилятор) обязана явно документировать выборы, которые она делает, тогда как в случаях неуточнённого поведения может произойти что угодно из набора возможных вариантов поведения.

12.3.4 Точки следования

Точки следования (sequence points) — это места в программе, где состояние абстрактной машины согласовано с состоянием целевой машины. Можно думать об этом так: когда мы отлаживаем программу, мы можем выполнять её пошагово, где каждый шаг примерно соответствует оператору C. Обычно мы останавливаемся на точке с запятой, вызовах функций, операторе ||, и т. д. Однако мы можем переключиться на представление ассемблера, где каждый оператор будет закодирован, возможно, многими инструкциями, и выполнять эти инструкции таким же образом. Это позволяет нам выполнить только часть оператора, приостановившись на полпути. В этот момент состояние абстрактной машины C не определено чётко. Как только мы заканчиваем выполнение инструкций, реализующих один-единственный оператор, состояния машин «синхронизируются», позволяя нам исследовать не только состояние на уровне ассемблера, но и состояние самой программы на C. Это и есть точка следования. Второе, эквивалентное определение точки следования — это место в программе, где побочные эффекты предыдущих выражений уже применены, а побочные эффекты следующих ещё не применены. Точки следования — это

- Точка с запятой.
- Запятая (которая в C может действовать так же, как точка с запятой, но также группирует операторы.

Её использование не рекомендуется.).

- Логическое И/ИЛИ (не битовые версии!).
- Момент, когда аргументы функции вычислены, но функция ещё не начала своё

выполнение.

- Вопросительный знак в тернарном операторе.

Многие реальные случаи неопределённого поведения связаны с понятием точек следования. Листинг 12-10 показывает пример.

Листинг 12-10. seq_points.c

```
int i = 0;
i = i ++ * 10;
```

Чему равен i? К сожалению, лучший ответ, который мы можем дать, следующий: в этом коде есть неопределённое поведение. Судя по всему, мы не знаем, будет ли `i` увеличена до присваивания `i*10` переменной `i` или после этого. Есть две записи в одну и ту же ячейку памяти до точки следования, и не определено, в каком порядке они произойдут. Причина в том, что, как мы видели в разделе 12.3.2, порядок вычисления подвыражений не зафиксирован. Поскольку подвыражения могут влиять на состояние памяти (вспомните вызовы функций или операторы пре- или постинкремента), и нет обязательного порядка, в котором происходят эти эффекты, даже результат одного подвыражения может зависеть от эффектов другого.

12.4 Прагматика

12.4.1 Выравнивание

С точки зрения абстрактной машины мы работаем с байтами памяти. У каждого байта есть свой адрес. Однако аппаратные протоколы, используемые на кристалле, довольно сильно различаются. Довольно распространено, что процессор может читать только пачки, скажем, по 16 байт, начинающиеся с адреса, кратного 16. Другими словами, он может прочитать либо первый 16-байтовый блок из памяти, либо второй, но не блок, начинающийся с произвольного адреса. Мы говорим, что данные *выровнены* по границе в *N* байт, если они начинаются с адреса, кратного *N*. Очевидно, что если данные выровнены по границе в *kn* байт, они автоматически выровнены и по границе в *n* байт. Например, если переменная выровнена по границе в 16 байт, она одновременно выровнена и по границе в 8 байт.

```
Aligned data (8-byte boundary):
0x00 00 00 00 00 00 00 00 :    11 22 33 44 55 66 77 88
Unaligned data (8-byte boundary):
0x00 00 00 00 00 00 00 00 :    .. .. .. 11 22 33 44 55
0x00 00 00 00 00 00 00 07 :    66 77 88 .. .. .. .. ..
```

Что происходит, когда программист запрашивает чтение многобайтового значения, охватывающего два таких блока (например, 8-байтового значения, чьи первые три байта лежат в одном блоке, а остальные — в другом)? Разные архитектуры дают разные ответы на этот вопрос. Некоторые аппаратные архитектуры запрещают невыровненный доступ к памяти. Это значит, что попытка прочитать любое значение, не выровненное, например, по границе в 8 байт, приводит к прерыванию. Пример такой архитектуры — SPARC. Операционные системы могут эмулировать невыровненный доступ, перехватывая сгенерированное прерывание и помещая сложную логику доступа в обработчик. Такие операции, как вы могли догадаться, чрезвычайно затратны, потому что обработка прерываний относительно медленная. Intel 64 использует менее строгий подход. Невыровненный доступ разрешён, но с накладными расходами. Например, если мы хотим прочитать 8 байт начиная с адреса 6, а можем читать только блоки по 8 байт, CPU (центральный процессор) выполнит два чтения вместо одного, а затем составит запрошенное значение из частей двух учетверённых слов. Так что выровненный доступ дешевле, потому что требует меньше чтений. Потребление памяти программиста часто заботит меньше, чем производительность; поэтому компиляторы автоматически подстраивают выравнивание переменных в памяти, даже если это создаёт промежутки из неиспользуемых байт. Это обычно называют *выравнивающими заполнителями структуры данных (data structure padding)*. Выравнивание — параметр генерации кода и выполнения программы, так что его обычно рассматривают как часть *прагматики* языка.

12.4.2 Выравнивающие заполнители структуры данных

Для структур выравнивание существует в двух разных смыслах:

- Выравнивание самого экземпляра структуры. Оно влияет на адрес, с которого начинается структура.
- Выравнивание полей структуры. Компилятор может намеренно вводить промежутки между полями структуры, чтобы ускорить доступ к ним. Именно к этому относятся выравнивающие заполнители структуры данных.

Например, мы создали структуру, показанную в листинге 12-11.

Листинг 12-11. align_str_ex1

```
struct mystr {
    uint16_t a;
    uint64_t b;
};
```

Предполагая выравнивание по границе в 8 байт, размер такой структуры, возвращаемый `sizeof`, будет 16 байт. Поле a начинается с адреса, кратного 8, а затем шесть байт тратятся впустую, чтобы выровнять b по границе в 8 байт. Есть несколько случаев, когда

стоит иметь это в виду:

- Возможно, вы захотите сместить баланс между потреблением памяти и

производительностью в сторону меньшего потребления памяти. Представьте, что вы создаёте миллион копий структур, и каждая структура тратит впустую 30% своего размера из-за промежутков выравнивания. Заставив компилятор уменьшить эти промежутки, вы получите выигрыш в использовании памяти на 30%, а это не пустяк. Это также даёт преимущества лучшей локальности, что может быть намного полезнее, чем выравнивание отдельных полей.

- При чтении заголовков файлов или получении сетевых данных в структуры нужно

учитывать возможные промежутки между полями структуры. Например, заголовок файла содержит поле в 2 байта, а затем поле в 8 байт. Между ними нет промежутков. Теперь мы пытаемся прочитать этот заголовок в структуру, как показано в листинге 12-12.

Листинг 12-12. align_str_read.c

```
struct str {
     uint16_t a; /* a gap of 4 bytes */
     uint64_t b;
};
struct str mystr;
fread( &mystr, sizeof( str ), 1, f );
```

Проблема в том, что в раскладке структуры есть промежутки, тогда как файл хранит поля непрерывно. Предполагая, что значения в файле — `a=0x1111` и `b=0x 22 22 22 22 22 22 22,` рисунок 12-4 показывает состояние памяти после чтения.

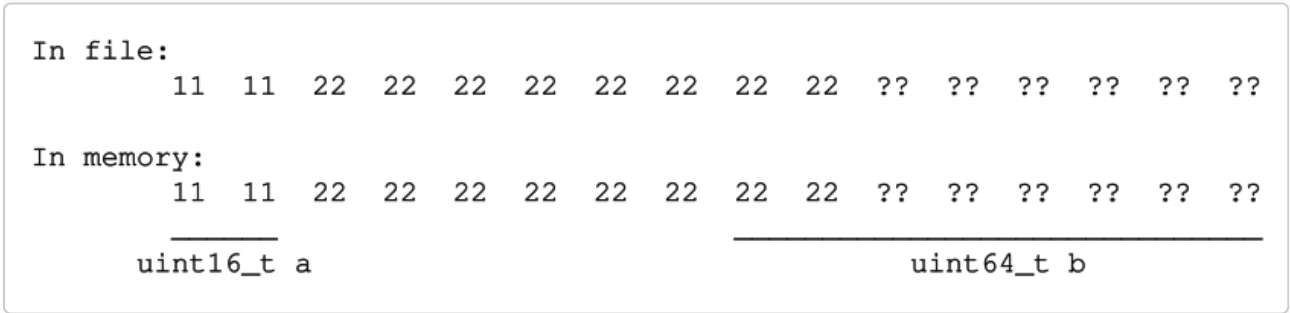


Рисунок 12-4. Раскладка структуры в памяти и данные, прочитанные из файла

Есть способы управлять выравниванием; вплоть до C11 они специфичны для компилятора. Сначала мы изучим их. Ключевое слово `#pragma` позволяет передать компилятору одну из прагматических команд. Оно поддерживается в MSVC, компиляторе C

от Microsoft, а также распознаётся GCC ради совместимости. Листинг 12-13 показывает, как использовать его для локального изменения стратегии выбора выравнивания с помощью прагмы `pack`.

Листинг 12-13. pragma_pack.c

```
#pragma pack(push, 2)
struct mystr {
    short a;
    long b;
};
#pragma pack(pop)
```

Второй аргумент pack — предполагаемый размер блока, который машина способна прочитать из памяти на аппаратном уровне. Первый аргумент `pack` — либо push, либо `pop`. Во время трансляции компилятор отслеживает текущее значение заполнения, проверяя вершину специального внутреннего стека. Мы можем временно переопределить текущее значение заполнения, поместив новое значение в этот стек, а затем восстановить старое значение, когда закончим. Изменить значение заполнения глобально можно с помощью следующей формы этой прагмы:

```
#pragma pack(2)
```

Однако это очень опасно, потому что приводит к непредсказуемым неочевидным изменениям в других частях программы, которые очень трудно отследить. Посмотрим, как значение выравнивания влияет на выравнивание отдельных полей, разобрав пример, показанный в листинге 12-14.

Листинг 12-14. pack_2.c

```
#pragma pack(push, 2)
struct mystr {
    uint16_t a;
    int64_t b;
};
#pragma pack(pop)
```

Значение заполнения говорит нам, сколько байт гипотетический целевой компьютер может извлечь из памяти за одно чтение. Компилятор старается минимизировать количество чтений для каждого поля. Здесь нет причин пропускать байты между `a` и `b`,

потому что это не даёт никакой выгоды с учётом значения заполнения. Предполагая, что `a=0x1111`, а `b=0x22 22 22 22 22 22 22 22`, раскладка памяти будет выглядеть так:

```
11 11 22 22 22 22 22 22 22 22
```

Листинг 12-15 показывает другой пример со значением заполнения, равным 4.

Листинг 12-15. pack_4.c

```
#pragma pack(push, 4)
struct mystr {
    uint16_t a;
    int64_t b;
};
#pragma pack(pop)
```

Что, если применить ту же раскладку памяти без промежутков? Поскольку мы можем читать только 4 байта за раз, это неоптимально. Мы отметили границы блоков памяти, читаемых атомарно.

```
Pack: 2
11 11 | 22 22 | 22 22 | 22 22 | 22 22 | ?? ??
Pack: 4, same memory layout
11 11   22 22 | 22 22   22 22 | 22 22   ?? ??
Pack: 4, memory layout really used
11 11   ?? ?? | 22 22   22 22 | 22 22   22 22
```

Как видим, когда заполнение установлено в 4, применение беспромежуточной раскладки памяти заставляет CPU выполнить три чтения для доступа к `b`. Так что, по сути, идея в том, чтобы минимизировать количество чтений, размещая члены структуры как можно ближе друг к другу. Специфичный для GCC способ сделать примерно то же самое — спецификация `packed` директивы `__attribute__`. В общем случае `__attribute__` описывает дополнительную спецификацию сущности кода, такой как тип или функция. Это ключевое слово `packed` означает, что поля структуры хранятся в памяти последовательно, вообще без промежутков. Листинг 12-16 показывает пример.

Листинг 12-16. str_attribute_packed.c

```
Struct__attribute__(( packed )) mystr {
    uint8_t first;
    float delta;
    float position;
};
```

Помните, что упакованные структуры не являются частью языка и не поддерживаются на некоторых архитектурах (таких как SPARC) *даже на аппаратном уровне*, что означает не только удар по производительности, но и сбои программы или чтение недействительных значений.

12.5 Выравнивание в C11

C11 ввёл стандартизированный способ управления выравниванием. Он состоит из

- Двух ключевых слов:
- _Alignas
- _Alignof
- Заголовочного файла `stdalign.h`, определяющего псевдонимы препроцессора для `_Alignas` и `_Alignof`

```
as alignas and alignof
```

- Функции aligned_alloc.

Выравнивание возможно только на степени двойки: 1, 2, 4, 8 и т. д. `alignof` используется, чтобы узнать выравнивание определённой переменной или типа. Оно вычисляется во время компиляции, точно так же, как `sizeof`. Листинг 12-17 показывает пример его использования. Обратите внимание на спецификатор формата `"%zu"`, используемый для вывода или чтения значений типа `size_t`.

Листинг 12-17. alignof_ex.c

```
#include <stdio.h>
#include <stdalign.h>
int  main(void) {
    short x;
    printf("%zu\n", alignof(x));
    return 0;
}
```

На самом деле `alignof(x)` возвращает наибольшую степень 2, на которую выровнен `x`, поскольку выравнивание чего-либо, например, на 8 подразумевает и выравнивание на 4, 2 и 1 (все его делители). Предпочитайте использовать `alignof` вместо `_Alignof` и `alignas` вместо `_Alignas`. `alignas` принимает константное выражение и используется, чтобы принудительно задать выравнивание определённой переменной или массива. Листинг 12-18 показывает пример. При запуске он выводит 8.

Листинг 12-18. alignas_ex.c

```
#include <stdio.h>
#include <stdalign.h>
int main( void ) {
    alignas( 8 ) short x;
    printf( "%zu\n", alignof( x ) );
    return 0;
}
```

Комбинируя `alignof` и `alignas`, можно выравнивать переменные по той же границе, что и другие переменные. Нельзя выровнять переменные на значение меньше их размера, и `alignas` нельзя использовать, чтобы получить

```
same effect as __attribute__((packed)).
```

12.6 Итоги

В этой главе мы систематизировали и расширили наши знания о том, что такое язык программирования. Мы увидели основы написания парсеров и изучили понятия неопределённого и неуточнённого поведения и то, почему они важны. Затем мы ввели понятие прагматики и разобрали одну из самых важных вещей. Мы откладываем задание к этой

главе до следующей, где разберём самые важные практики написания хорошего кода. Предполагая, что наши читатели ещё не очень хорошо знакомы с C, мы хотим, чтобы они приобрели хорошие привычки как можно раньше в своём пути изучения C.

Вопрос 241 Что такое синтаксис языка?

Вопрос 242 Для чего используются грамматики?

Вопрос 243 Из чего состоит грамматика?

Вопрос 244 Что такое БНФ?

Вопрос 245 Как написать парсер рекурсивного спуска, имея описание грамматики в БНФ?

Вопрос 246 Как включить приоритеты в описание грамматики?

Вопрос 247 Какие есть уровни иерархии Хомского?

Вопрос 248 Почему регулярные языки менее выразительны, чем контекстно-свободные грамматики?

Вопрос 249 Что такое лексический анализ?

Вопрос 250 Что такое семантика языка?

Вопрос 251 Что такое неопределённое поведение?

Вопрос 252 Что такое неуточнённое поведение и чем оно отличается от неопределённого?

Вопрос 253 Какие есть случаи неопределённого поведения в C?

Вопрос 254 Какие есть случаи неуточнённого поведения в C?

Вопрос 255 Что такое точки следования?

Вопрос 256 Что такое прагматика?

Вопрос 257 Что такое выравнивающие заполнители структуры данных? Переносимы ли они?

Вопрос 258 Что такое выравнивание? Как им можно управлять в C11?

Практики написания хорошего кода

В этой главе мы хотим сосредоточиться на стиле кода. Во время написания кода разработчик постоянно сталкивается с необходимостью принимать решения. Какие структуры данных ему следует использовать? Как их следует называть? Где и когда их следует выделять? Опытные программисты принимают эти решения иначе, чем новички, и мы считаем крайне важным поговорить об этом процессе принятия решений.

13.1 Принятие решений

Решения часто требуют баланса между двумя взаимоисключающими полюсами. Классический пример — нельзя поставить качественный продукт одновременно дёшево и быстро. Тонкая настройка производительности кода часто делает его труднее для чтения и отладки. Так что некоторым характеристикам кода стоит отдавать приоритет перед другими, исходя из здравого смысла и самой задачи. Именно поэтому такие рекомендации по коду — хорошая отправная точка, но следовать им слепо — не выход. Наши советы по написанию кода основаны на следующих предпосылках:

1. Мы хотим, чтобы код был максимально пригоден для повторного использования. Это часто требует тщательного планирования и координации между разработчиками, что не позволяет писать код по-настоящему *быстро*, но очень скоро окупается, потому что экономит время на отладку и на самом деле позволяет писать сложное программное обеспечение. Отладка программ обычно считается сложнее их написания. Так что меньше кода часто означает меньше времени на отладку и более надёжные функции. Это особенно важно для таких языков, как C, которые

- Небезопасны в широком смысле (допускают арифметику указателей, не выполняют проверку границ и т. д.)
- Лишены выразительной системы типов, какая есть в таких языках, как Scala, Haskell или OCaml. Такие типы накладывают на программу ряд ограничений, которые должны быть выполнены, иначе компилятор её отвергнет.

У этого правила есть заметное исключение. Если повторное использование функций приводит к резкому падению производительности, у алгоритма неоправданно большая *O*-сложность. Например, мы делали задание со связными списками в Главе 10. Там была функция для вычисления суммы всех целых чисел в определённом списке. Один из способов её создания примерно показан в листинге 13-1.

Листинг 13-1. `list_sum_bad.c`

```
int list_sum( const struct list* l ) {
    size_t i;
    int sum = 0;
    /* We do not want to launch the full computation
     * of size at each cycle iteration */
    size_t sz = list_size( l ) ;
    for( i = 0; i < sz; l = l-> next )
        sum = sum + l->value;
    return sum;
}
```

В этом примере для каждого i в диапазоне от 0 включительно до длины списка не включительно мы на самом деле начинаем проходить список с самого первого элемента. Это приводит к резкому падению производительности по сравнению с однократным суммирующим проходом по списку. В последнем случае добавление ещё одного элемента в список приводит к одному дополнительному обращению к списку, тогда как в программе, показанной в листинге 13-1, это приводит к `list_length(l)` дополнительным обращениям к списку!

2. Программу должно быть легко изменять. Этот пункт взаимосвязан с

предыдущим. Функции меньшего размера часто более пригодны для повторного использования, а значит, изменения становятся проще, потому что больше кода можно оставить нетронутым по сравнению с предыдущей версией.

3. Код должен быть максимально лёгким для чтения. Ключевые факторы здесь —

- Разумные имена. Даже если английский вам не родной, не стоит писать имена переменных, имена функций или комментарии на вашем родном языке.
- Согласованность. Используйте одни и те же соглашения об именовании и единообразные способы выполнения похожих операций.
- Короткие и лаконичные функции. Если описание логики чрезмерно многословно, это часто признак недостатка разумной декомпозиции или того, что вам нужен слой абстракции. Это также хорошо сказывается на сопровождаемости.

4. Код должен быть легко тестируемым. Тестирование убеждает нас, что хотя бы в некоторых

проработанных случаях код ведёт себя как задумано. Иногда задача требует обратного. Например, если мы пишем код для контроллера в отсутствие хорошего оптимизирующего компилятора и с очень ограниченными ресурсами, нас может вынудить отказаться

от красивой структуры кода то, что компилятор не может корректно встраивать функции; таким образом, каждый вызов будет сказываться на производительности, часто неприемлемым образом.

13.2 Элементы кода

13.2.1 Общее именование

Конкретное соглашение об именовании часто навязывается самим языком. В случаях, когда проект основан на существующей кодовой базе, разумно не отклоняться от неё ради согласованности. В этой книге мы используем следующие соглашения об именовании:

- Все имена пишутся строчными буквами.
- Части имени разделяются подчёркиванием, например: `list_count`.

Остальная часть этого раздела сосредоточена на разных возможностях языка и связанных с ними соглашениях об именовании и использовании.

13.2.2 Структура файлов

У файлов для включения должна быть защитная конструкция включения. Они должны быть самодостаточными — это значит, что для каждого заголовочного файла `thisfile.h` должен компилироваться `.c`-файл, содержащий только строку `#include "thisfile.h"`. Порядок включений часто выбирают следующим:

- Связанный заголовок.
- Библиотека C.
- `.h`-файлы других библиотек.
- `.h`-файлы вашего проекта.

Затем придерживайтесь согласованного порядка объявления макросов, типов, функций, переменных и т. д. Это значительно упрощает навигацию по проекту. Типичный порядок:

- для заголовков: 1. Файлы для включения. 2. Макросы. 3. Типы. 4. Переменные (глобальные). 5. Функции.
- для `.c`-файлов 1. Файлы для включения. 2. Макросы. 3. Типы. 4. Переменные (глобальные). 5. Статические переменные. 6. Функции. 7. Статические функции.

13.2.3 Типы

- Когда возможно (C99 или новее), предпочитайте типы, определённые в `stdint.h`, такие как `uint64_t`

```
or uint8_t.
```

- Если вы хотите соответствовать POSIX, не определяйте собственные типы с суффиксом `_t`. Он зарезервирован для стандартных типов, чтобы новые типы, которые могут быть введены в будущих редакциях стандарта, не конфликтовали с пользовательскими типами, определёнными в некоторых программах.
- Типы часто называют с префиксом, общим для проекта. Например, если вы пишете калькулятор, метки типов будут иметь префикс `calc_`.
- При определении структур, если вы можете выбирать порядок полей, определяйте

их в следующем порядке:

- Сначала постарайтесь минимизировать потери памяти из-за *выравнивающих заполнителей структуры данных*.
- Затем упорядочите поля по размеру. Наконец, отсортируйте их по алфавиту.
- Иногда у структур есть поля, которые пользователь не должен изменять напрямую. Например, библиотека определяет структуру, показанную в листинге 13-2.

Листинг 13-2. `struct_private_ex.c`

```
struct mypair {
    int x;
    int y;
    int _refcount;
};
```

Поля такой структуры можно изменить напрямую с помощью синтаксиса с точкой или стрелкой. Однако наше соглашение подразумевает, что только определённые функции библиотеки должны изменять поле `_refcount`, и пользователь библиотеки никогда не должен делать это вручную.

В C нет понятия приватных полей структуры, так что это максимально близко к нему, чего можно добиться без более или менее грязных хаков.

- Члены перечисления следует писать заглавными буквами, как константы.
  Для членов одного перечисления рекомендуется общий префикс. Пример показан в листинге 13-3.

Листинг 13-3. `enum_ex.c`

```
Enum exit_code   {
    EX_SUCCESS,
    EX_FAILURE,
    EX_INVALID_ARGUMENTS
};
```

13.2.4 Переменные

Выбор правильных имён для переменных и функций критически важен.

- Используйте существительные для имён.
- У булевых переменных тоже должны быть осмысленные имена. Рекомендуется добавлять к ним префикс `is_`,

а затем дописывать точное проверяемое свойство. `is_good` в большинстве случаев, вероятно, слишком общее имя, в отличие от `is_prime` или `is_before_last`.

Предпочитайте положительные имена отрицательным, так как человеческий мозг легче их разбирает — например, `is_even`, а не `is_not_odd`.

- Не рекомендуется использовать имена, лишённые смысла, вроде `a`, `b` или `x4`. Заметное

исключение — код, иллюстрирующий статью или работу, описывающую алгоритм в псевдокоде с использованием таких имён. В этом случае любое изменение именования, скорее всего, только запутает читателей, а не добавит ясности. Индексы традиционно называют `i` и `j`, и вас поймут, если вы будете их придерживаться.

- Включение единиц измерения в имя может быть хорошей идеей — например,

```
uint32_t delay_msecs.
```

- Другие суффиксы тоже полезны, такие как `cnt`, `max` и т. д. Например, `attempts_max` (максимально допустимое число попыток), `attempts_cnt` (сделанные попытки).
- Глобальные константы называют заглавными буквами. У глобальных *изменяемых* переменных префикс `g_`.
- По традиции глобальные константы принято определять с помощью директивы `#define`. Однако современный подход — использовать глобальные переменные `const static` или просто `const`. В отличие от `#define`, они типизированы, а также лучше

видны при отладке. Если у вас есть доступ к качественному компилятору, он всё равно встроит их (если решит, что так будет быстрее).

- Используйте модификатор `const`, где это уместно. C99 позволяет создавать переменные в произвольных местах внутри функций, а не только в начале блока. Используйте это, чтобы хранить промежуточные результаты в именованных константах.
- Не определяйте глобальные переменные в заголовочных файлах! Определяйте их в `.c`-файлах и объявляйте в `.h`-файле как `extern`.

13.2.5 О глобальных переменных

По возможности не используйте изменяемые глобальные переменные. Мы не можем подчеркнуть это достаточно. Вот самые важные проблемы, которые они приносят:

- В проектах среднего масштаба и, тем более, в крупных, с колоссальным числом строк, всю информацию об эффектах функции лучше локализовать в её сигнатуре. Функция `f` может вызвать другую функцию `g`, и так далее, и где-то в этой цепочке изменится глобальная переменная. Мы не можем увидеть, что это изменение *может* произойти, просто взглянув на `f`; нам приходится изучать все функции, которые она вызывает, и функции, которые вызывают они, и так далее.
- Они делают функции не *реентерабельными (reenterable)*. Это значит, что функцию `f` нельзя вызвать, если она уже выполняется. Последнее может произойти в двух случаях:
- Функция `f` вызывает другие функции, которые после нескольких внутренних вызовов могут вызвать `f` снова, когда первый экземпляр `f` ещё не завершился.

Листинг 13-4 показывает пример функции `f`, которая не реентерабельна.

Листинг 13-4. `reenterability.c`

```
bool flag = true;
int var = 0;
void g(void) {
    f();
    flag = false;
}
void f(void) {
    if (flag) g();
}
```

- Программа распараллелена, и функция используется в нескольких потоках (что часто бывает на современных компьютерах).

В случае сложной иерархии вызовов, чтобы узнать, реентерабельна ли функция, требуется дополнительный анализ.

- Они создают риски безопасности, потому что обычно их значения нужно проверять перед

изменением или использованием. Программисты склонны забывать эти проверки. Если что-то может пойти не так, оно пойдёт не так.

- Они усложняют тестирование функций из-за зависимости по данным, которую они вносят.

Однако писать код без тестов всегда следует избегать. Глобальные статические изменяемые переменные тоже зло, но, по крайней мере, они не засоряют глобальное пространство имён в других файлах. Глобальные статические неизменяемые переменные (`const static`), однако, вполне допустимы и их часто может встроить компилятор.

13.2.6 Функции

- Используйте глаголы для имён функций — например, `packet_checksum_calc`.
- Префикс `is_` также довольно распространён для функций, проверяющих условия — например,

```
int is_prime( long num ).
```

- Функции, оперирующие структурой с определённой меткой, часто получают префикс с

именем соответствующей метки — например, `bool list_is_empty(struct list* lst );`. Поскольку C не позволяет тонко контролировать пространства имён, это, похоже, простейшая форма контроля хаоса, возникающего, когда большинство функций доступны отовсюду.

- Используйте модификатор `static` для всех функций, кроме тех, что должны быть доступны всем.
- Пожалуй, самое важное место для использования `const` — аргументы функций типа

«указатель на неизменяемые данные». Это гарантирует, что функция случайно их не изменит из-за ошибки программиста.

13.3 Файлы и документация

По мере роста проекта количество файлов увеличивается, и по ним становится труднее ориентироваться. Чтобы справляться с объёмными проектами, их нужно структурировать с самого начала. Ниже приведён распространённый шаблон для корневого каталога проекта.

`SRC/`	ИСХОДНЫЕ ФАЙЛЫ
`doc/`	Документация
`res/`	Файлы ресурсов (например, изображения).
`lib/`	Статические библиотеки, которые будут скомпонованы с исполняемым файлом.
`build/`	Артефакты: исполняемый файл и другие сгенерированные файлы.
`include/`	Файлы для включения. Этот каталог добавляется в путь поиска включений компилятора флагом `-I`.
`obj/`	Сгенерированные объектные файлы. Они собираются в исполняемые файлы и библиотеки компоновщиком и не нужны после завершения компиляции.
`configure`	Начальный скрипт настройки, который следует запускать перед сборкой. Он может настраивать разные целевые архитектуры или включать и выключать возможности.
`Makefile`	Содержит инструкции для системы автоматической сборки. Имя файла и формат зависят от используемой системы.

Существует много систем сборки; некоторые из самых популярных для C — `make`, `cmake` и `automake`. У разных языков разные экосистемы, и у них часто есть выделенные инструменты сборки (например, Gradle или OCamlBuild).

- Мы рекомендуем вам изучить эти проекты, которые, насколько нам известно, хорошо организованы

```
www.gnu.org/software/gsl/
```

- www.gnu.org/software/gsl/design/gsl-design.html
- www.kylheku.com/kaz/kazlib.html

Doxygen — де-факто стандарт для создания документации к программам на C и C++. Он позволяет сгенерировать полностью структурированный набор HTML- или LATEX-страниц из исходного кода программы. Описания функций и переменных берутся

из специально отформатированных комментариев. Листинг 13-5 показывает пример исходного файла, принимаемого Doxygen.

Листинг 13-5. `doxygen_example.h`

```
#pragma once
#include <common.h>
#include <vm.h>
/** @defgroup const_pool Constant pool */
/** Free allocated memory for the pool contents
*/
void const_pool_deinit( struct vm_const_pool* pool );
/** Non-destructive constant pool combination
 * @param a First pool.
 * @param b Second pool.
 * @returns An initialized constant pool combining contents of both arguments
 * */
struct vm_const_pool const_combine(
        struct vm_const_pool const* a,
        struct vm_const_pool const* b );
/** Change the constant pool by adding the other pool's contents in its end.
 * @param[out] src The source pool which will be modified.
 * @param fresh The pool to merge with the `src` pool.
 */
void const_merge(
        struct vm_const_pool* src,
        struct vm_const_pool const* fresh );
/**@} */
```

Специально отформатированные комментарии (начинающиеся с `/**` и содержащие такие команды, как `@defgroup`) обрабатываются Doxygen для генерации документации для соответствующих сущностей кода. Подробнее см. в документации Doxygen.

13.4 Инкапсуляция

Одна из основ мышления — абстракция. В программной инженерии это процесс сокрытия деталей реализации и данных. Если мы хотим реализовать какое-то поведение, например поворот изображения, нам хотелось бы думать только о повороте изображения. Формат входного файла, формат его заголовков имеют для нас мало значения. Что действительно важно — уметь работать с точками, образующими изображение, и знать его размеры. Однако написать программу, не учитывая всю эту информацию, на самом деле не зависящую от самого алгоритма поворота, нельзя. Мы разобьём программу на части; каждая часть будет выполнять своё назначение и только его. Эту логику

можно использовать, вызывая набор открытых функций и/или набор открытых глобальных переменных. Вместе они образуют *интерфейс* для этой части программы. Однако для их реализации нам обычно приходится писать больше функций, которые лучше скрыть от конечного пользователя.

> **Работа с системами контроля версий** При работе в команде, где многие люди одновременно вносят изменения, создание функций меньшего размера очень важно. Если функция выполняет много действий, а её код огромен, несколько независимых изменений будет труднее слить автоматически.

В языках программирования, поддерживающих пакеты или классы, они используются для сокрытия частей кода и создания для них интерфейсов. К сожалению, в C нет ничего подобного; более того, в структурах нет понятия «приватных полей»: все поля видны всем. Поддержка отдельных файлов кода, называемых единицами трансляции, — единственная реальная возможность языка, помогающая нам изолировать части кода программы. Мы используем понятие модуля как синоним единицы трансляции, `.c`-файла. Стандарт C не определяет понятие модуля. В этой книге мы используем их как взаимозаменяемые, потому что для языка C они примерно эквивалентны. Как мы знаем, функции и глобальные переменные по умолчанию становятся публичными символами и потому доступны другим файлам. Разумно помечать все «приватные» функции и глобальные переменные как `static` в `.c`-файле и объявлять все «публичные» функции в `.h`-файле. В качестве примера мы напишем модуль, реализующий стек. Заголовочный файл будет описывать структуру и функции, способные оперировать её экземплярами. Это напоминает объектно-ориентированное программирование без подтипизации (без наследования). Интерфейс будет состоять из следующих функций:

- Создать или уничтожить стек;
- Поместить и снять элементы со стека.
- Проверить, пуст ли стек.
- Запустить функцию для каждого элемента стека.

Файл кода определит все функции и, вероятно, ещё несколько, недоступных за его пределами и созданных исключительно ради декомпозиции и возможности повторного использования кода. Листинги 13-6 и 13-7 показывают итоговый код. `stack.h` описывает интерфейс. У него есть защитная конструкция включения, он перечисляет все остальные заголовки (сначала стандартные заголовки, затем пользовательские) и определяет пользовательские типы.

Листинг 13-6. stack.h

```
#ifndef _STACK_H_
#define _STACK_H_
#include <stddef.h>
#include  <stdint.h>
#include  <stdbool.h>
struct list;
struct stack  {
    struct list* first;
    struct list* last;
    size_t count;
};
struct stack stack_init    ( void );
void stack_deinit( struct stack* st );
void stack_push( struct stack* s, int value );
int  stack_pop ( struct stack* s );
bool stack_is_empty( struct stack const* s );
void stack_foreach( struct stack* s, void (f)(int) );
#endif  /* _STACK_H_  */
```

Определены два типа: `list` и `stack`. Первый используется только внутренне, внутри стека, и потому мы объявили его *неполным типом*. Допустимы только указатели на экземпляры такого типа, пока его определение не будет указано позже. Для всех, кто подключает `stack.h`, тип `struct list` останется неполным. Файл реализации, `stack.c`, однако, определит структуру, дополнив тип и позволив обращаться к его полям (но только в `stack.c`). Затем определяется `struct stack`, и объявляются функции, работающие с ним (`stack_push`, `stack_pop` и т. д.) (см. листинг 13-7).

Листинг 13-7. `stack.c`

```
#include <malloc.h>
#include "stack.h"
struct list { int value; struct list* next; };
static struct list* list_new( int item, struct list* next ) {
    struct list* lst = malloc( sizeof( *lst ) );
    lst->value  =  item;
    lst->next  =  next;
    return lst;
}
void stack_push( struct stack* s, int value ) {
    s->first = list_new( value, s->first );
    if ( s->last ==  NULL  ) s->last =  s-> first;
    s->count++;
}
int stack_pop( struct stack* s ) {
    struct list* const head = s->first;
    int value;
    if ( head ) {
        if ( head->next ) s->first = head->next;
        value = head->value;
        free( head );
        if( -- s->count ) {
            s->first =  s->last =  NULL;
        }
        return value;
    }
    return 0;
}
void stack_foreach( struct stack* s, void (f)(int) ) {
    struct list* cur;
    for( cur = s->first; cur; cur = cur-> next )
        f( cur->value );
}
bool stack_is_empty( struct stack const* s ) {
    return s->count == 0;
}
struct stack stack_init( void ) {
    struct stack empty = { NULL, NULL, 0 };
    return empty;
}
void stack_deinit( struct stack* st ) {
    while( ! stack_is_empty( st ) ) stack_pop( st );
    st-> first = NULL;
```

```
    st-> last = NULL;
}
```

Этот файл определяет все функции, объявленные в заголовке. Его можно разбить на несколько `.c`-файлов, что иногда пойдёт на пользу структуре проекта; важно то, что компилятор должен принять их все, а затем скомпилированный код должен попасть в компоновщик. Статическая функция `list_new` определена, чтобы изолировать инициализацию экземпляра `struct list`. Она не открыта внешнему миру. Во время оптимизаций компилятор может не только встроить её, но даже полностью удалить саму функцию, эффективно устраняя любое возможное влияние на производительность кода. Пометка функции как статической необходима (но не достаточна) для того, чтобы эта оптимизация произошла. Кроме того, инструкции статических функций могут размещаться ближе к их соответствующим вызывающим сторонам, улучшая локальность. Разбивая программу на модули с хорошо описанными интерфейсами, вы снижаете общую сложность и достигаете лучшей возможности повторного использования. Необходимость создавать заголовочные файлы немного усложняет внесение изменений, потому что согласованность заголовков с самим кодом лежит на ответственности программиста. Однако мы можем извлечь из этого и пользу, задавая чёткое описание интерфейса, лишённое деталей реализации.

13.5 Неизменяемость

Довольно часто приходится выбирать между созданием новой изменённой копии структуры и изменением на месте. Вот некоторые преимущества и недостатки обоих вариантов.

- Создание копии: проще писать: вы случайно не передадите не тот экземпляр в функцию.
- Проще отлаживать, потому что не нужно отслеживать изменения переменной. Может быть оптимизировано компилятором.
- Дружелюбно к распараллеливанию. Может быть медленнее.
- Изменение существующего экземпляра. Быстрее.
- Может стать очень трудно отлаживать, особенно в многопоточном окружении. Иногда проще, потому что не нужно аккуратно и рекурсивно копировать структуры с несколькими указателями на другие структуры (этот процесс называется глубоким копированием).
- Для объектов с явной идентичностью этот подход может быть более интуитивным и при этом достаточно надёжным.

Наше восприятие реального мира основано на изменяемых объектах, потому что у объектов реального мира часто есть явная идентичность. Когда вы включаете свой телефон, телефон не заменяется своей копией — вместо этого меняется состояние того же самого телефона. Другими словами, идентичность телефона сохраняется, а его состояние меняется. Таким образом, в ситуациях, где у вас есть только один экземпляр определённого типа, и последовательные изменения выполняются над ним, вполне нормально изменять его вместо создания копии при каждом изменении.

13.6 Утверждения

Существует механизм, позволяющий проверять определённые условия во время выполнения программы. Когда такое условие не выполняется, порождается ошибка, и программа завершается аварийно. Чтобы использовать механизм утверждений, нужно подключить `#include <assert.h>`, а затем использовать макрос `assert`. Листинг 13-8 показывает пример.

Листинг 13-8. `assert.c`

```
#include <assert.h>
int main() {
    int x = 0;
    assert( x != 0 );
    return 0;
}
```

Условие, переданное макросу `assert`, очевидно ложно; следовательно, программа завершится аварийно и сообщит нам о несработавшем утверждении:

```
assert: assert.c:6: main: Assertion `x != 0' failed.
```

Если определён символ препроцессора NDEBUG (чего можно добиться опцией компилятора `-D NDEBUG` или директивой `#define NDEBUG`), `assert` заменяется на пустую строку и тем самым отключается. Так что утверждения не будут вызывать никаких накладных расходов, и проверки выполняться не будут. Используйте утверждения для проверки невозможных условий, сигнализирующих о несогласованности состояния программы. Никогда не используйте утверждения для проверки пользовательского ввода.

13.7 Обработка ошибок

Хотя в языках более высокого уровня есть какой-то механизм обработки ошибок (не мешающий описанию основной логики), в C его нет. Есть три основных способа справляться с ошибками:

1. Использовать коды возврата. Функции следует возвращать не сам результат, а код,

показывающий, успешно ли она отработала. В последнем случае код отражает точный тип произошедшей ошибки. Результат вычисления присваивается через указатель, принимаемый как дополнительный аргумент. Листинг 13-9 показывает пример.

Листинг 13-9. `error_code.c`

```
enum   div_res   {
      DIV_OK,
      DIV_BYZERO
};
enum div_res div( int x, int y, int* result ) {
    if ( y  != 0  ) { *result =  x/y; return DIV_OK;  }
    else return DIV_BYZERO;
}
```

Симметрично, можно возвращать значения как обычно, а код ошибки устанавливать через указатель на соответствующую переменную. Коды ошибок можно описать с помощью `enum` или нескольких `#define`. Затем их можно использовать как индексы в статическом массиве сообщений или использовать оператор `switch`. Листинг 13-10 показывает пример.

Листинг 13-10. `err_switch_arr.c`

```
enum   error_code   {
      ERROR1,
      ERROR2
};
...
enum  error_code  err;
...
switch (err) {
    case ERROR1: ... break;
    case ERROR2: ... break;
    default: ... break;
}
/* alternatively */
static const char* const messages[] = {
    "It is the first error\n",
    "The second error it is\n"
};
fprintf( stderr, messages[err] );
```

Никогда не используйте глобальные переменные для хранения кода ошибки (или для возврата значения из функции). Согласно стандарту C, существует стандартная переменноподобная сущность `errno`. Она должна быть изменяемым `lvalue` и не должна объявляться явно. Её использование напоминает глобальную переменную, хотя её значение локально для потока. Библиотечные функции используют её для хранения кода ошибки, так что, увидев неудачу от функции (например, `fopen` вернул `NULL`), нужно проверить значение `errno` на код ошибки. Man-страницы соответствующей функции перечисляют возможные значения `errno` (например, `EEXIST`). Несмотря на то что эта возможность закралась в стандартную библиотеку, она в основном считается антипаттерном, и её не следует копировать.

2. Использование обратных вызовов.

Обратные вызовы (callbacks) — это указатели на функции, передаваемые в качестве аргументов и вызываемые функцией, которая их принимает. Их можно использовать, чтобы изолировать код обработки ошибок, но они часто выглядят странно для людей, более привычных к традиционному использованию кодов возврата. Кроме того, порядок выполнения становится менее очевидным. Листинг 13-11 показывает пример.

Листинг 13-11. `div_cb.c`

```
#include <stdio.h>
int div( int x, int y, void (onerror)(int, int)) {
    if ( y != 0 )
        return x/y;
    else  {
        onerror(x,y);
        return 0;
    }
}
static void div_by_zero(int x, int y) {
    fprintf( stderr, "Division by zero: %d / %d\n", x, y );
}
int main(void) {
    printf("%d %d\n",
            div( 10, 2, div_by_zero ),
            div( 10, 0, div_by_zero ) );
    return 0;
}
```

3. Использование `longjmp`. Эта продвинутая техника будет объяснена в разделе 14.3.

Есть классическая техника восстановления после ошибок, требующая использования `goto`. Листинг 13-12 показывает пример.

Листинг 13-12. `goto_error_recover.c`

```
void foo(void)
{
    if (!doA()) goto exit;
    if (!doB()) goto revertA;
    if (!doC()) goto revertB;
    /* doA, doB and doC succeeded */
    return;
revertB:
    undoB();
revertA:
    undoA();
exit:
    return;
}
```

В этом примере выполнены три действия, и все они могут завершиться неудачей. Природа этих действий такова, что после них нужно выполнить очистку. Например, `doA` может вызвать динамическое выделение памяти. Если `doA` завершилась успешно, а `doB` — нет, нам нужно освободить эту память, чтобы предотвратить утечку памяти. Именно это делает код, помеченный `revertA`. Восстановления выполняются в обратном порядке. Если `doA` и `doB` завершились успешно, а `doC` — нет, нам нужно откатить B, а затем A. Так что мы помечаем этапы отката метками и позволяем управлению проваливаться сквозь них. Так, `goto revertB` сначала откатит `doB`, а затем провалится в код, откатывающий `doA`. Этот трюк часто можно увидеть в ядре Linux. Однако будьте осторожны: goto обычно значительно усложняют верификацию, из-за чего иногда их запрещают.

13.8 О выделении памяти

- Многие программисты советуют не использовать гибкие массивы, выделенные на стеке. Это лёгкий способ

получить переполнение стека, если вы недостаточно хорошо проверяете длину. Хуже того, нет способа узнать, *можно* ли безопасно выделить указанное количество байт на стеке или нет.

- Не злоупотребляйте `malloc`! Как вы увидите в последнем задании этой главы, `malloc`

совсем не дёшев. Всякий раз, когда вы хотите выделить что-то достаточно небольшое, делайте это на стеке, как локальную переменную. Если какой-то функции нужен адрес структуры, вы можете взять адрес структуры, выделенной на стеке, и передать его ей. Это предотвращает утечки памяти и улучшает производительность и читаемость кода.

- Глобальные переменные не представляют угрозы, пока они константы. То же самое касается статических

локальных переменных. Используйте их, если хотите ограничить использование определённой константы одной функцией.

13.9 О гибкости

Мы действительно выступаем за повторное использование кода. Однако доведение этого до крайности приводит к абсурдному количеству слоёв абстракции и шаблонному коду, присутствующему лишь для поддержки возможной будущей потребности в дополнительных возможностях (которая может никогда не наступить). Серебряной пули, в широком смысле, не существует. Каждый стиль программирования, каждая модель вычислений хороша и лаконична в одних случаях и громоздка и многословна в других. Аналогично, лучший инструмент скорее специализированный, чем мастер на все руки. Можно превратить программу для просмотра изображений в мощный редактор, способный вос-

производить видео и редактировать теги IDv3, но грань просмотра изображений наверняка пострадает, как и удобство использования. Написание более абстрактного кода может принести пользу, потому что такой код проще адаптировать к новым контекстам. В то же время это вносит сложность, которая может оказаться ненужной. Обобщайте только до той степени, что не причиняет вреда. Чтобы понять, когда остановиться, нужно ответить себе на несколько вопросов, таких как

- Какова цель вашей программы или библиотеки?
- Каковы границы функциональности, которую вы представляете для своей программы?
- Будет ли эту функцию проще писать, использовать и/или отлаживать, если она написана более общим образом?

Хотя первые два вопроса весьма субъективны, для последнего можно привести пример. Взглянем на код, показанный в листинге 13-13.

Листинг 13-13. `dump_1.c`

```
void dump( char const* filename ) {
    FILE* f = fopen( filename, "w" );
    fprintf(f, "this is the dump %d", 42 );
    fclose( f );
}
```

Сравните его с другой версией той же логики, разделённой на две функции, показанной в листинге 13-14.

Листинг 13-14. `dump_2.c`

```
void dump( FILE* f ) {
    fprintf(f, "this is the dump %d", 42 );
}
void fun( void ) {
    FILE* f = fopen( "dump.txt", "w" );
    dump( f );
    fclose( f );
}
```

Вторая версия предпочтительнее по двум причинам:

- Первая версия требует имени файла, а значит, её нельзя использовать для записи в

```
stderr or stdout.
```

- Вторая версия разделяет логику открытия файла и логику записи в файл. Если вы хотите обрабатывать ошибки, которые могут возникнуть при вызовах `fprintf`, `fopen` или `fclose`, вы будете делать это отдельно для `fopen`, оставляя функции относительно простыми. Функция dump не будет обрабатывать ошибки открытия файла: она вообще не будет вызвана, если открытие не удалось.

Листинг 13-15 показывает пример той же логики с обработкой ошибок. Как видите, в функции `dump` нет обработки ошибок открытия и закрытия файла; вместо этого она выполняется в `fun`.

Листинг 13-15. file_open_sep.c

```
#include <stdio.h>
enum stat {
    STAT_OK,
    STAT_ERR_OPEN,
    STAT_ERR_CLOSE,
    STAT_ERR_WRITE
};
enum stat dump( FILE * f ) {
    if ( fprintf( f, "this is the dump %d", 42 ) ) return STAT_OK;
    return  STAT_ERR_WRITE;
}
enum stat fun( void ) {
    enum stat dump_stat;
    FILE * f;
    f =  fopen(  "dump.txt",  "w"  );
    if (!f) return STAT_ERR_OPEN;
    dump_stat = dump( f );
    if ( dump_stat != STAT_OK ) return dump_stat;
    if (! fclose( f ) ) return STAT_ERR_CLOSE;
    return STAT_OK;
}
```

В случае нескольких записей в функции `dump` функция станет громоздкой и потому менее читаемой.

13.10 Задание: поворот изображения

Вам нужно создать программу, поворачивающую BMP-изображение любого разрешения на 90 градусов по часовой стрелке.

13.10.1 Формат файла BMP

Формат BMP (BitMaP) — растровый графический формат, а значит, он хранит изображение как таблицу цветных точек (пикселей). В этом формате цвет кодируется числами фиксированного размера (может быть 1, 4, 8, 16 или 24 бита). Если на пиксель используется 1 бит, изображение чёрно-белое. Если используется 24 бита, число возможных различных цветов составляет примерно 16 миллионов. Мы реализуем поворот только 24-битных изображений. Подмножество BMP-файлов, с которым должна работать ваша программа, описывается структурой, показанной в листинге 13-16. Она представляет заголовок файла, за которым сразу следуют данные пикселей.

Листинг 13-16. `bmp_struct.c`

```
#include  <stdint.h>
Struct __attribute__((packed))
    bmp_header {
        uint16_t bfType;
        uint32_t  bfileSize;
        uint32_t bfReserved;
        uint32_t bOffBits;
        uint32_t biSize;
        uint32_t biWidth;
        uint32_t  biHeight;
        uint16_t  biPlanes;
        uint16_t biBitCount;
        uint32_t biCompression;
        uint32_t biSizeImage;
        uint32_t biXPelsPerMeter;
        uint32_t biYPelsPerMeter;
        uint32_t biClrUsed;
        uint32_t  biClrImportant;
};
```

> **Вопрос 259** Прочитайте спецификацию файлов BMP, чтобы определить, за что отвечают эти поля.

Формат файла зависит от количества бит на пиксель. Когда используется 16 или 24 бита на пиксель, цветовых палитр нет. Каждый пиксель кодируется 24 битами, или 3 байтами, как показано в листинге 13-17. Каждый компонент — число от 0 до 255 (один байт), показывающее присутствие синего, зелёного или красного цвета в этом пикселе. Итоговый цвет — суперпозиция этих трёх базовых цветов.

Листинг 13-17. `pixel.c`

```
struct  pixel  {
    unsigned char b, g, r;
}
```

Каждая строка пикселей дополняется так, чтобы её длина была кратна 4. Например, ширина изображения — 15 пикселей. Это соответствует 15 × 3 = 45 байтам. Чтобы дополнить это, мы пропускаем 3 байта (до ближайшего кратного 4, то есть 48) перед началом новой строки пикселей. Из-за этого настоящий размер изображения будет отличаться от произведения ширины, высоты и размера пикселя (3 байта).

> **Примечание** Не забудьте открыть изображение в бинарном режиме!

13.10.2 Архитектура

Мы хотим продумать архитектуру программы так, чтобы она была расширяемой и модульной.

1. Опишите структуру пикселя `struct pixel`, чтобы не работать напрямую с растровой таблицей

(как с совершенно бесструктурными данными). Этого следует всегда избегать.

2. Отделите внутреннее представление изображения от входного формата. Поворот

выполняется над внутренним форматом изображения, который затем сериализуется обратно в BMP. Формат BMP может измениться, вы можете захотеть поддержать другие форматы, и вы не хотите жёстко привязывать алгоритм поворота к BMP. Чтобы этого добиться, определите структуру `structure image` для хранения массива пикселей (непрерывного, теперь без выравнивающих промежутков) и некоторой информации, которую действительно стоит сохранить. Например, здесь совершенно нет необходимости хранить сигнатуру BMP или любые из никогда не используемых полей заголовка. Вполне можно обойтись шириной и высотой изображения в пикселях. Вам понадобится создать функции для чтения изображения из BMP-файла и записи его в BMP-файл (вероятно, также для генерации заголовка BMP из внутреннего представления).

3. Отделите открытие файла от его чтения.
4. Сделайте обработку ошибок единообразной и обрабатывайте ошибки ровно в одном месте (для

именно этой программы этого достаточно). Чтобы этого добиться, определите функцию `from_bmp`, которая будет читать файл из потока и возвращать один из кодов, показывающих, успешно ли завершилась операция. Помните о соображениях гибкости. Ваш код должен быть легко использовать как в приложениях с графическим интерфейсом (GUI), так и в тех, что вовсе без GUI, так что разбрасывать вывод в `stderr` повсюду — не лучший вариант: ограничьте его частью кода, отвечающей за обработку ошибок. Ваш код также должен легко адаптироваться под разные входные форматы. Листинг 13-18 показывает несколько фрагментов стартового кода.

Листинг 13-18. `image_rot_stub.c`

```
#include  <stdint.h>
#include <stdio.h>
struct pixel { uint8_t b,g,r; };
struct image {
    uint64_t width, height;
    struct pixel_t* data;
};
/*  deserializer   */
enum read_status  {
    READ_OK = 0,
    READ_INVALID_SIGNATURE,
    READ_INVALID_BITS,
    READ_INVALID_HEADER
        /* more codes  */
};
enum read_status from_bmp( FILE* in, struct image* const read );
/*  image_t from_jpg( FILE* );...
 *  and other deserializers are possible
 *  All information needed will be
 *  stored in image structure */
/* makes a rotated copy */
struct image rotate( struct image const source );
/*  serializer   */
enum  write_status  {
     WRITE_OK = 0,
     WRITE_ERROR
         /* more codes */
};
enum write_status to_bmp( FILE* out, struct image const* img );
```

> **Вопрос 260** Реализуйте размытие. Оно делается очень простым способом: для каждого пикселя вы вычисляете новые

> **Примечание.** компоненты как среднее в окне 3 × 3 пикселя (называемом **ядром**). Граничные пиксели остаются нетронутыми.

> **Вопрос 261** Реализуйте поворот на произвольный угол (не только на 90 или 180 градусов).

> **Вопрос 262** Реализуйте преобразования «расширение» (dilate) и «сужение» (erode). Они похожи на размытие, но вместо вычисления среднего в окне нужно вычислить минимальное (erode) или максимальное (dilate) значение компонента.

13.11 Задание: собственный аллокатор памяти

В этом задании мы реализуем собственную версию `malloc` и `free` на основе системного вызова отображения памяти `mmap` и связного списка блоков произвольного размера. Это можно рассматривать как упрощённую версию менеджера памяти, типичного для стандартной библиотеки C, разделяющую большинство её слабых мест. В этом задании использование `malloc/calloc`, `free` и `realloc` запрещено. Как мы знаем, эти функции используются для манипуляции кучей. Куча состоит из *анонимных страниц* и на самом деле представляет собой связный список блоков. Каждый блок состоит из заголовка и самих данных. Заголовок описывается структурой, показанной в листинге 13-19.

Листинг 13-19. `mem_str.c`

```
struct mem  {
    struct mem* next;
    size_t capacity;
    bool is_free;
};
```

Сразу за заголовком следует полезная область. Нам нужно хранить и размер, и связь со следующим блоком, потому что в нашем случае в куче могут быть промежутки по двум причинам.

- Начало кучи может быть размещено между двумя уже отображёнными областями.

- Куча может расти до произвольного размера.

Выделение в куче — это разбиение первого доступного блока на два (при условии, что его размера достаточно). Первая часть помечается как не свободная, и возвращается её адрес. Если нет свободных блоков, достаточно больших для запрошенного размера, аллокатор пытается получить больше памяти у ОС, вызывая `mmap`. Нет смысла выделять блоки по 1 или 3 байта; они слишком малы. Обычно это расточительство, поскольку размер заголовка в любом случае больше. Так что мы введём константу `BLOCK_MIN_SIZE` для минимального допустимого размера блока (не считая заголовка). Получив запрос на `query` байт, мы сначала заменяем его на `BLOCK_MIN_SIZE`, если он слишком мал. Затем мы проходим по цепочке блоков и применяем следующую логику к каждому блоку:

- query <= capacity-sizeof(struct mem) - MINIMAL_BLOCK_SIZE

В этом случае мы можем разбить блок на два и использовать первую часть как выделенный блок памяти, оставив вторую свободной.

- Иначе блок недостаточно велик, чтобы вместить запрошенное количество байт.
- Если блок не последний, переходим к следующему блоку. Иначе нужно отобразить больше страниц (достаточно, чтобы выделить `query` байт).

Сначала мы пытаемся сделать это сразу после конца блока (флаг `MAP_FIXED` для `mmap`), и если получается, увеличиваем текущий блок, включая новые страницы. В конце мы разбиваем его на два и используем первый из пары. Если не удаётся отобразить больше страниц сразу в конце кучи, мы пытаемся отобразить их где угодно (достаточно, чтобы хранить `query` байт). Затем мы разбиваем его на два и используем первый из пары. Если все отображения не удаются, мы возвращаем `NULL`, точно так же, как это делает `malloc`. `free` реализовать проще. Зная начало блока, нам нужно вычислить начало соответствующего заголовка, что меняет его статус с «выделен» на «свободен». Если сразу за ним следует свободный блок, они объединяются. Это не так, когда блок последний в своей области памяти, а следующий отображён после определённого промежутка. Вы можете использовать заголовочный файл, показанный в листинге 13-20, как отправную точку.

Листинг 13-20. mem.h

```
#ifndef _MEM_H_
#define _MEM_H_
#define _USE_MISC
#include <stddef.h>
#include  <stdint.h>
#include <stdio.h>
#include <sys/mman.h>
#define  HEAP_START  ((void*)0x04040000)
struct mem;
#pragma pack(push, 1)
struct mem  {
    struct mem* next;
    size_t capacity;
    bool is_free;
};
#pragma pack(pop)
void* _malloc( size_t query );
void  _free( void* mem );
void* heap_init( size_t initial_size );
#define DEBUG_FIRST_BYTES 4
void memalloc_debug_struct_info( FILE* f,
        struct mem const* const address );
void memalloc_debug_heap( FILE* f,   struct mem  const* ptr );
#endif
```

Помните, что сложная логика требует продуманной декомпозиции на функции поменьше. Вы можете использовать код, показанный в листинге 13-21, для отладки состояния кучи. Не забывайте, что вы также можете подождать пользовательского ввода и проверить файл `/proc/PID/maps`, чтобы увидеть настоящие отображения процесса с идентификатором PID.

Листинг 13-21. `mem_debug.c`

```
#include "mem.h"
void memalloc_debug_struct_info( FILE* f,
        struct mem const* const address ) {
    size_t i;
    fprintf( f,
            "start: %p\nsize: %lu\nis_free: %d\n",
            (void*)address,
            address-> capacity,
            address-> is_free );
    for ( i = 0;
            i <  DEBUG_FIRST_BYTES  &&  i <  address-> capacity;
            ++i )
        fprintf( f, "%hhX",
                ((char*)address)[ sizeof( struct mem_t ) + i ] );
    putc( '\n', f );
}
void memalloc_debug_heap( FILE* f, struct mem const* ptr ) {
    for( ; ptr; ptr = ptr->next )
        memalloc_debug_struct_info( f, ptr );
}
```

Примерное количество строк кода — от 150 до 200. Не забудьте написать Makefile.

13.12 Итоги

В этой главе мы подробно изучили некоторые из самых важных рекомендаций относительно стиля кода и архитектуры программ. Мы рассмотрели соглашения об именовании и причины, стоящие за распространёнными рекомендациями по коду. Когда мы пишем код, нам следует придерживаться определённых ограничений, вытекающих из наших требований к коду, а также из самого процесса разработки. Мы рассмотрели такие важные понятия, как инкапсуляция. Наконец, мы предоставили ещё два более продвинутых задания, где вы можете применить свои новые знания об архитектуре программ. В следующей части мы углубимся в детали трансляции, рассмотрим некоторые возможности языка, которые проще понять на уровне ассемблера, и поговорим о производительности и оптимизациях компилятора.

Между С и ассемблером

Детали трансляции

В этой главе мы вернёмся к понятию соглашения о вызовах, чтобы углубить наше понимание и разобраться в деталях трансляции. Этот процесс требует как понимания работы программы на уровне ассемблера, так и определённой степени знакомства с C. Мы также рассмотрим некоторые классические низкоуровневые уязвимости безопасности, которые может открыть неосторожный программист. Понимание этих низкоуровневых деталей трансляции порой критически важно для устранения весьма неочевидных ошибок, которые проявляются не при каждом выполнении.

14.1 Последовательность вызова функции

В Главе 2 мы изучили, как вызывать процедуры, как они возвращают значения и как принимают аргументы. Полная последовательность вызова описана в [24], и мы настоятельно рекомендуем вам туда заглянуть. Мы вернёмся к этому процессу и добавим ценные детали.

14.1.1 Регистры XMM

Помимо регистров, о которых мы уже говорили, у современных процессоров есть несколько наборов специальных регистров, происходящих из расширений процессора. Расширение предоставляет дополнительные схемы, расширяет систему команд и иногда добавляет полезные регистры. Заметное расширение называется *SSE* (Streaming SIMD Extensions) и описывает набор регистров `xmm`: `xmm0, xmm1,..., xmm15`. Они шириной 128 бит и обычно используются для двух видов задач:

- Арифметики с плавающей точкой; и
- Инструкций SIMD (такие инструкции выполняют действие сразу над несколькими данными).

Обычная команда `mov` не может работать с регистрами `xmm`. Вместо неё используется команда `movq` для копирования данных между младшей половиной регистров `xmm` (64 из 128 бит) с одной стороны и регистрами `xmm`, регистрами общего назначения или памятью (тоже 64 бита) с другой. Чтобы заполнить весь регистр `xmm`, у вас есть два варианта: `movdqa` и `movdqu`. Первый расшифровывается как «move aligned double quad word» (переместить выровненное учетверённое слово), второй — невыровненная версия. Большинство инструкций SSE требуют, чтобы операнды в памяти были правильно выровнены. Невыровненные версии этих инструкций часто существуют под другой мнемоникой и подразумевают штраф производительности из-за невыровненного чтения. Поскольку инструкции SSE часто используются в местах, критичных к производительности,

обычно разумнее придерживаться инструкций, требующих выравнивания операндов. Мы будем использовать инструкции SSE для высокопроизводительных вычислений в разделе 16.4.1.

> **Вопрос 263** Прочитайте про инструкции `movq`, `movdqa` и `movdqu` в [15].

14.1.2 Соглашение о вызовах

Соглашение о вызовах — это набор правил о последовательности вызова функции, которых программист добровольно придерживается. Если все следуют одним и тем же правилам, гарантируется гладкая совместимость. Однако как только кто-то нарушает правила, например, вносит изменения и не восстанавливает `rbp` в определённой функции, может произойти что угодно: ничего, отложенный сбой или немедленный. Причина в том, что другие функции написаны в расчёте на соблюдение этих правил и рассчитывают, что `rbp` останется нетронутым. Соглашения о вызовах определяют, среди прочего, алгоритм передачи аргументов. В случае типичного соглашения *nix x86 64, которое мы используем (полностью описанного в [24]), следующее описание — достаточно точное приближение того, как вызывается функция.

1. Сначала сохраняются регистры, которые нужно сохранить. Все регистры, кроме

семи регистров, сохраняемых вызываемой функцией (`rbx`, `rbp`, `rsp` и `r12–r15`), могут быть изменены вызываемой функцией, так что если их значение сколько-нибудь важно, их следует сохранить (вероятно, в стеке).

2. Регистры и стек заполняются аргументами.

Размер каждого аргумента округляется вверх до 8 байт.

Аргументы разбиваются на три списка:

(a) Целочисленные или указательные аргументы.

(b) Float и double.

(c) Аргументы, передаваемые в памяти через стек («memory»).

Первые шесть аргументов из первого списка передаются в регистрах общего назначения (`rdi`, `rsi`, `rdx`, `rcx`, `r8` и `r9`). Первые восемь аргументов из второго списка передаются в регистрах `xmm0` – `xmm7`. Если из этих списков нужно передать больше аргументов, они передаются через стек в обратном порядке. Это значит, что последний аргумент окажется на вершине стека перед выполнением вызова.

Если с целыми числами и float легко справиться, со структурами всё немного хитрее.

Если структура больше 32 байт или у неё невыровненные поля, она передаётся через память.

Структура поменьше разбивается на поля, и каждое поле обрабатывается отдельно и, если оно находится во вложенной структуре, рекурсивно. Так что структуру из двух элементов можно передать так же, как два аргумента. Если одно поле структуры считается «memory», это распространяется и на саму структуру.

Регистр `rbp`, как мы увидим, используется для адресации аргументов, переданных через память, и локальных переменных.

А что насчёт возвращаемых значений? Целочисленные и указательные значения возвращаются в `rax` и `rdx`. Значения с плавающей точкой возвращаются в `xmm0` и `xmm1`. Большие структуры возвращаются через указатель, предоставляемый как дополнительный скрытый аргумент, в духе следующего примера:

```
struct s {
    char vals[100];
};
struct s f( int x ) {
    struct s mys;
    mys.vals[10] = 42;
    return mys;
}
void f( int x, struct s* ret ) {
    ret->vals[10] = 42;
}
```

3. Затем должна быть вызвана инструкция `call`. Её параметр — адрес первой инструкции вызываемой функции. Она помещает адрес возврата в стек. У каждой программы может быть несколько экземпляров одной и той же функции, запущенных одновременно, не только в разных потоках, но и из-за рекурсии. Каждый такой экземпляр функции хранится в стеке, потому что его основной принцип — «последним пришёл, первым ушёл» — соответствует тому, как функции запускаются и завершаются. Если функция `f` запущена и затем вызывает функцию `g`, `g` завершается первой (но была вызвана последней), а `f` завершается последней (хотя была вызвана первой). *Кадр стека* — часть стека, выделенная под один экземпляр функции. Он хранит значения локальных переменных, временных переменных и сохранённых регистров. Код функции обычно заключён между парой *пролог* и *эпилог*, схожими для всех функций. Пролог помогает инициализировать кадр стека, а эпилог деинициализирует его. Во время выполнения функции `rbp` остаётся неизменным и указывает на начало её кадра стека. Локальные переменные и аргументы из стека можно адресовать относительно `rbp`. Это отражено в прологе функции, показанном в листинге 14-1.

Листинг 14-1. `prologue.asm`

```
func:
push rbp
mov rbp, rsp
sub rsp, 24      ; given 24 is total size of local variables
```

Старое значение `rbp` сохраняется, чтобы позже восстановить его в эпилоге. Затем новый `rbp` устанавливается на текущую вершину стека (которая теперь, кстати, хранит старое значение `rbp`). Затем память под локальные переменные выделяется в стеке вычитанием их общего размера из `rsp`. Это и есть автоматическое выделение памяти в C — техника, которую мы использовали в самом первом задании для выделения буферов на стеке.

Функции заканчиваются эпилогом, показанным в листинге 14-2.

Листинг 14-2. `epilogue.asm`

```
mov rsp, rbp
pop rbp
ret
```

Перемещая начальный адрес кадра стека в `rsp`, мы можем быть уверены, что вся память, выделенная в стеке, освобождена. Затем восстанавливается старое значение `rbp`, и теперь `rbp` указывает на начало предыдущего кадра стека. Наконец, `ret` снимает адрес возврата со стека в `rip`.

Компилятор иногда выбирает полностью эквивалентную альтернативную форму. Она показана в листинге 14-3.

Листинг 14-3. `epilogue_alt.asm`

```
Leave
ret
```

Инструкция `leave` создана специально для разрушения кадра стека. Её аналог, `enter`, не всегда используется компиляторами, потому что она функциональнее, чем последовательность инструкций, показанная в листинге 14-1. Она нацелена на языки с поддержкой вложенных функций.

4. После выхода из функции наша работа не всегда закончена. Если были

аргументы, переданные через память (стек), нам нужно избавиться и от них тоже.

14.1.3 Пример: простая функция и её стек

Взглянем на простую функцию, вычисляющую максимум из двух значений. Мы скомпилируем её без оптимизаций и посмотрим на ассемблерный листинг. Листинг 14-4 показывает пример.

Листинг 14-4. `maximum.c`

```
int maximum( int a, int b ) {
    char buffer[4096];
    if (a < b) return b;
    return a;
}
int main(void) {
    int x = maximum( 42, 999 );
    return 0;
}
```

Листинг 14-5 показывает дизассемблирование, полученное с помощью `objdump`.

Листинг 14-5. `maximum.asm`

```
00000000004004b6 <maximum>:
4004b6:       55                      push   rbp
4004b7:       48 89 e5                mov    rbp,rsp
4004ba:       48 81 ec 90 0f 00 00    sub    rsp,0xf90
4004c1:       89 bd fc ef ff ff       mov    DWORD PTR [rbp-0x1004],edi
4004c7:       89 b5 f8 ef ff ff       mov    DWORD PTR [rbp-0x1008],esi
4004cd:       8b 85 fc ef ff ff       mov    eax,DWORD PTR [rbp-0x1004]
4004d3:       3b 85 f8 ef ff ff       cmp    eax,DWORD PTR [rbp-0x1008]
4004d9:       7d 08                   jge    4004e3 <maximum+0x2d>
4004db:       8b 85 f8 ef ff ff       mov    eax,DWORD PTR [rbp-0x1008]
4004e1:       eb 06                   jmp    4004e9 <maximum+0x33>
4004e3:       8b 85 fc ef ff ff       mov    eax,DWORD PTR [rbp-0x1004]
4004e9:       c9                      leave
4004ea:       c3                      ret
00000000004004eb <main>:
4004eb:       55                      push   rbp
4004ec:       48 89 e5                mov    rbp,rsp
4004ef:       48 83 ec 10             sub    rsp,0x10
4004f3:       be e7 03 00 00          mov    esi,0x3e7
4004f8:       bf 2a 00 00 00          mov    edi,0x2a
4004fd:       e8 b4 ff ff ff          call   4004b6 <maximum>
400502:       89 45 fc                mov    DWORD PTR [rbp-0x4],eax
```

После небольшой чистки мы получаем чистый и более читаемый ассемблерный код, показанный в листинге 14-6.

Листинг 14-6. `maximum_refined.asm`

```
mov rsi, 999
mov rdi, 42
call maximum
...
maximum:
push rbp
mov rbp, rsp
sub rsp, 3984
mov [rbp-0x1004], edi
mov [rbp-0x1008], esi
mov eax, [rbp-0x1004]
...
Leave
ret
```

> **Назначение регистров** Объяснение того, почему изменение `esi` означает изменение всего `rsi`, см. в разделе 3.4.2.

Мы проследим вызов функции и её пролог (см. листинг 14-6) и покажем содержимое стека сразу после его выполнения.

```
call maximum
push rbp
mov rbp, rsp
sub rsp, 3984
```

14.1.4 Красная зона

Красная зона — это область в 128 байт, простирающаяся от `rsp` в сторону меньших адресов. Она смягчает правило «никаких данных ниже `rsp`»; там безопасно размещать данные, и они не будут перезаписаны системными вызовами или прерываниями. Мы говорим о прямой записи в память относительно `rsp` без изменения `rsp`. Вызовы функций, однако, всё равно перезапишут красную зону. Красная зона была создана, чтобы позволить одну конкретную оптимизацию. Если функция никогда не вызывает другие функции, она может опустить создание кадра стека (изменения `rbp`). Тогда локальные переменные и аргументы будут адресоваться относительно `rsp`, а не `rbp`.

- Общий размер локальных переменных меньше 128 байт.
- Функция — листовая (не вызывает другие функции).
- Функция не изменяет `rsp`; иначе адресовать память относительно него невозможно.

Продвигая `rsp` вперёд, вы всё ещё можете получить больше свободного места для размещения данных в стеке, чем 128 байт. См. также раздел 16.1.3.

14.1.5 Переменное количество аргументов

Используемое нами соглашение о вызовах поддерживает переменное количество аргументов. Это значит, что функция может принимать произвольное число аргументов. Это возможно, потому что передача аргументов (и очистка стека после завершения функции) — ответственность вызывающей функции. Объявление таких функций содержит так называемое *многоточие (ellipsis)* — три точки вместо последнего аргумента. Типичная функция с переменным числом аргументов — наш старый знакомый `printf`.

```
void printf( char const* format, ... );
```

Как `printf` узнаёт точное количество аргументов? Она точно знает, что передан как минимум один аргумент (`char const* format`). Анализируя эту строку и подсчитывая спецификаторы, она вычислит общее количество аргументов, а также их типы (в каких регистрах они должны быть).

> **Примечание** В случае переменного числа аргументов `al` должен содержать количество регистров `xmm`, использованных аргументами.

Как видите, узнать точное количество переданных аргументов совершенно невозможно. Функция выводит его из аргументов, которые точно присутствуют (`format` в данном случае). Если спецификаторов формата больше, чем аргументов, `printf` об этом не узнает и наивно попытается получить содержимое соответствующих регистров и памяти. Разумеется, программист не может напрямую закодировать эту функциональность на C, потому что к регистрам нет прямого доступа. Однако существует переносимый механизм объявления функций с переменным числом аргументов, являющийся частью стандартной библиотеки. У каждой платформы своя реализация этого механизма. Его можно использовать после подключения файла `stdarg.h`, и он состоит из следующего:

- `va_list` — структура, хранящая информацию об аргументах.
- `va_start` — *макрос*, инициализирующий `va_list`.
- `va_end` — *макрос*, деинициализирующий `va_list`.
- `va_arg` — *макрос*, берущий следующий аргумент из списка аргументов, если дан

экземпляр `va_list` и тип аргумента. Листинг 14-7 показывает пример. Функция `printer` принимает число аргументов и произвольное их количество.

Листинг 14-7. `vararg.c`

```
#include <stdarg.h>
#include <stdio.h>
void printer( unsigned long argcount, ... ) {
    va_list args;
    unsigned long i;
    va_start( args, argcount );
    for (i = 0; i < argcount; i++ )
        printf(" %d\n", va_arg(args, int )  );
    va_end( args );
}
int main () {
    printer(10, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 );
    return 0;
}
```

Сначала `va_list` инициализируется именем последнего аргумента перед точками с помощью `va_start`. Затем каждый вызов `va_arg` получает следующий аргумент. Второй параметр — имя типа нового аргумента. В конце `va_list` деинициализируется с помощью `va_end`. Поскольку имя типа становится аргументом, а `va_list` используется по имени, но при этом изменяется, этот пример может показаться запутанным.

> **Вопрос 264** Можете ли вы представить ситуацию, в которой функция, а не макрос, принимает переменную по имени (синтаксически) и изменяет её? Каким должен быть тип такой переменной?

14.1.6 vprintf и подобные

У таких функций, как `printf`, `fprintf` и т. д., есть специальные версии. Они принимают `va_list` в качестве своего последнего аргумента. Их имена начинаются с буквы v, например,

```
int vprintf(const char *format, va_list ap);
```

Они используются внутри пользовательских функций, которые, в свою очередь, принимают произвольное число аргументов. Листинг 14-8 показывает пример.

Листинг 14-8. `vsprintf.c`

```
#include <stdarg.h>
#include <stdio.h>
void logmsg( int client_id, const char* const str, ... ) {
    va_list args;
    char buffer[1024];
    char* bufptr = buffer;
va_start( args, str );
bufptr += sprintf(bufptr, "from client %d :", client_id );
vsprintf( bufptr, str, args );
fprintf( stderr, "%s", buffer );
va_end( args );
}
```

14.2 volatile

Ключевое слово `volatile` сильно влияет на то, как компилятор оптимизирует код. Модель вычислений C — модель фон Неймана. Она не поддерживает параллельное выполнение программы, и компилятор обычно старается выполнить как можно больше оптимизаций, не меняя наблюдаемое поведение программы. Это может включать переупорядочивание инструкций и кэширование переменных в регистрах. Чтение значения из памяти, которое никуда не записывается, опускается. Однако чтение и запись в переменные `volatile` происходят всегда. Порядок операций также сохраняется. Основные случаи использования следующие:

- *ввод/вывод, отображаемый на память*, когда взаимодействие с внешними устройствами выполняется

через взаимодействие с определённой выделенной областью памяти. Запись символа в видеопамять (в результате чего он отображается на экране) действительно должна произойти.

- Обмен данными между потоками. Если память используется для взаимодействия с другими потоками,

вы не хотите, чтобы записи или чтения были оптимизированы прочь. Заметьте, что одного `volatile` недостаточно для надёжного взаимодействия между потоками. Точно так же, как квалификатор `const`, в случае указателя `volatile` можно применить как к данным, на которые он указывает, так и к самому указателю. Правило то же: `volatile` слева от звёздочки относится к данным, на которые указывают, а справа — к самому указателю.

14.2.1 Ленивое выделение памяти

Многие операционные системы отображают страницы лениво, в момент первого использования, а не сразу после вызова `mmap` (или его аналога). Если программист не хочет задержек при первом использовании страницы, он может обратиться к каждой странице по отдельности, чтобы операционная система действительно её создала, как показано в листинге 14-9.

Листинг 14-9. `lma_bad.c`

```
char* ptr;
for( ptr = start; ptr < start + size; ptr += pagesize )
*ptr;
```

Однако у этого кода нет наблюдаемого эффекта с точки зрения компилятора, так что он может быть полностью оптимизирован прочь. Однако когда указатель помечен `volatile`, этого не произойдёт. Листинг 14-10 показывает пример.

Листинг 14-10. `lma_good.c`

```
volatile char* ptr;
for( ptr = start; ptr < start + size; ptr += pagesize )
*ptr;
```

> **Volatile-указатели в стандарте языка** Если volatile-указатель указывает на неволатильную память, согласно стандарту никаких гарантий нет! Они существуют, только когда и указатель, и память volatile. Так что, согласно стандарту, приведённый выше пример некорректен. Однако, поскольку программисты используют volatile-указатели именно с такими рассуждениями, самые используемые компиляторы (MSVC, GCC, clang) не оптимизируют прочь разыменование volatile-указателей. Стандартно-соответствующего способа сделать это не существует.

14.2.2 Сгенерированный код

Рассмотрим пример, показанный в листинге 14-11.

Листинг 14-11. `volatile_ex.c`

```
#include <stdio.h>
int main( int argc, char** argv ) {
    int ordinary = 0;
    volatile int vol = 4;
    ordinary++;
    vol++;
    printf( "%d\n", ordinary );
    printf( "%d\n", vol );
    return 0;
}
```

Есть две переменные: одна volatile, другая нет. Обе увеличиваются и передаются `printf` в качестве аргументов. GCC сгенерирует следующий код (с уровнем оптимизации `-O2`), показанный в листинге 14-12:

Листинг 14-12. `volatile_ex.asm`

```
; these are two arguments for `printf`
mov    esi,0x1
mov    edi,0x4005d4
; vol = 4
mov    DWORD PTR [rsp+0xc],0x4
; vol ++
mov    eax,DWORD PTR [rsp+0xc]
add    eax,0x1
mov    DWORD  PTR  [rsp+0xc],eax
xor    eax,eax
; printf( "%d\n", ordinary )
; the `ordinary` is not even created in stack frame
; its final precomputed value 1 was placed in `rsi` in the first line!
call   4003e0 <printf@plt>
; the second argument is taken from memory, it is volatile!
mov    esi,DWORD PTR [rsp+0xc]
; First argument is the address of "%d\n"
mov    edi,0x4005d4
xor    eax,eax
; printf( "%d\n", vol )
call   4003e0 <printf@plt>
xor    eax,eax
```

Как видим, содержимое volatile-переменной действительно читается и записывается каждый раз, когда она встречается в C. `ordinary` переменная даже не будет создана: вычисления будут выполнены во время компиляции, а итоговый результат сохранён в `rsi`, ожидая использования как второй аргумент вызова.

14.3 Нелокальные переходы — setjmp

Стандартная библиотека C содержит механизм для выполнения очень хитрого трюка. Он позволяет сохранять контекст вычислений и восстанавливать его. Контекст описывает состояние выполнения программы *за исключением* следующего:

- Всё, что связано с внешним миром (например, открытые дескрипторы).
- Контекст вычислений с плавающей точкой.
- Переменные стека.

Он позволяет сохранить контекст и вернуться к нему, если нам вдруг захочется вернуться. Мы не ограничены областью видимости той же функции. Подключите `setjmp.h`, чтобы получить доступ к следующему механизму:

- `jmp_buf` — тип переменной, способной хранить контекст.
- `int setjmp( jmp_buf env )` — функция, принимающая экземпляр `jmp_buf` и сохраняющая

в нём текущий контекст. По умолчанию возвращает 0.

- `void longjmp( jmp_buf env, int val )` используется для возврата к сохранённому контексту, хранящемуся в

определённой переменной типа `jmp_buf`. При возврате из `longjmp` `setjmp` возвращает не обязательно 0, а значение `val`, переданное `longjmp`. Листинг 14-13 показывает пример. Первый `setjmp` по умолчанию вернёт 0, таким же будет и значение `val`. Однако `longjmp` принимает 1 в качестве своего аргумента, и выполнение программы продолжится с вызова `setjmp` (потому что они связаны через использование `jb`). На этот раз `setjmp` вернёт 1, и именно это значение будет присвоено `val`.

Листинг 14-13. `longjmp.c`

```
#include <stdio.h>
#include <setjmp.h>
int main(void) {
    jmp_buf jb;
    int val;
    val = setjmp( jb );
    puts("Hello!");
    if (val == 0) longjmp( jb, 1 );
    else puts("End");
    return 0;
}
```

Локальные переменные, не помеченные `volatile`, после `longjmp` будут все хранить неопределённые значения. Это источник ошибок, а также проблем с освобождением памяти: трудно анализировать поток управления при наличии `longjmp` и гарантировать, что вся динамически выделенная память освобождена. В общем случае вызывать `setjmp` как часть сложного выражения разрешено, но лишь в редких случаях. В большинстве случаев это неопределённое поведение. Так что лучше так не делать. Важно помнить, что весь этот механизм основан на использовании кадров стека. Это значит, что нельзя выполнить `longjmp` в функции с деинициализированным кадром стека. Например, код, показанный в листинге 14-14, порождает неопределённое поведение именно по этой причине.

Листинг 14-14. `longjmp_ub.c`

```
jmp_buf jb;
void f(void) {
    setjmp( jb );
}
void g(void) {
    f();
    longjmp(jb);
}
```

Функция `f` уже завершилась, но мы выполняем `longjmp` в неё. Поведение программы не определено, потому что мы пытаемся восстановить контекст внутри разрушенного кадра стека. Другими словами, можно переходить только в ту же самую функцию или в функцию, которая запущена.

14.3.1 Volatile и setjmp

Компилятор считает, что `setjmp` — просто функция. Однако на самом деле это не совсем так, потому что это точка, с которой программа может начать выполняться заново. В обычных условиях некоторые локальные переменные могли быть закэшированы в регистрах (или вообще никогда не выделены) до вызова `setjmp`. Когда мы возвращаемся в эту точку из-за вызова `longjmp`, они не будут восстановлены. Отключение оптимизаций меняет это поведение. Так что отключённые оптимизации скрывают ошибки, связанные с использованием `setjmp`. Чтобы писать корректно, помните, что только локальные переменные `volatile` хранят определённые значения после `longjmp`. Они *не* восстанавливаются к своим прежним значениям, потому что `jmp_buf` не сохраняет переменные стека, а сохраняет значения, которые были до `longjmp`. Листинг 14-15 показывает пример.

Листинг 14-15. `setjmp_volatile.c`

```
#include <stdio.h>
#include <setjmp.h>
jmp_buf buf;
int main( int argc, char** argv ) {
    int var = 0;
    volatile int b = 0;
    setjmp( buf );
    if (b < 3) {
        b++;
        var ++;
        printf( "\n\n%d\n", var );
        longjmp( buf, 1 );
    }
    return 0;
}
```

Мы скомпилируем его без оптимизаций (`gcc -O0`, листинг 14-16) и с оптимизациями (`gcc -O2`, листинг 14-17). Без оптимизаций

Листинг 14-16. `volatile_setjmp_o0.asm`

```
main:
push     rbp
mov      rbp,rsp
sub      rsp,0x20
; `argc` and `argv` are saved in stack to make `rdi` and `rsi` available
mov    DWORD PTR [rbp-0x14],edi
mov    QWORD PTR [rbp-0x20],rsi
; var = 0
mov    DWORD PTR [rbp-0x4],0x0
; b = 0
mov    DWORD PTR [rbp-0x8],0x0
; 0x600a40 is the address of `buf` (a global variable of type `jmp_buf`)
mov    edi,0x600a40
call   400470 <_setjmp@plt>
; if (b < 3), the good branch is executed
; This is encoded by skipping several instructions to the `.endlabel` if b > 2
mov    eax,DWORD PTR [rbp-0x8]
cmp    eax,0x2
jg     .endlabel
; A fair increment
; b++
mov    eax,DWORD PTR [rbp-0x8]
add    eax,0x1
mov    DWORD PTR [rbp-0x8],eax
; var++
add    DWORD PTR [rbp-0x4],0x1
; `printf` call
mov    eax,DWORD PTR [rbp-0x4]
mov    esi,eax
mov    edi,0x400684
; There are no floating point arguments, thus rax = 0
mov    eax,0x0
call   400450 <printf@plt>
; calling `longjmp`
mov    esi,0x1
mov    edi,0x600a40
call   400490 <longjmp@plt>
.endlabel:
Mov    eax,0x0
Leave
ret
```

Вывод программы будет `123`

С оптимизациями

Листинг 14-17. `volatile_setjmp_o2.asm`

```
main:
; allocating memory in stack
sub    rsp,0x18
; a `setjmp` argument, the address of `buf`
mov    edi,0x600a40
; b = 0
mov    DWORD PTR [rsp+0xc],0x0
; instructions are placed in the order different
; from C statements to make better use of pipeline and other inner
; CPU mechanisms.
call   400470 <_setjmp@plt>
; `b` is read and checked in a fair way
mov    eax,DWORD PTR [rsp+0xc]
cmp    eax,0x2
jle    .branch
; return 0
xor    eax,eax
add    rsp,0x18
ret
.branch:
mov    eax,DWORD PTR [rsp+0xc]
; the second argument of `printf` is var + 1
; It was not even read from memory nor allocated.
; The computations were performed in compile time
mov    esi,0x1
; The first argument of `printf`
mov    edi,0x400674
; b = b + 1
add    eax,0x1
mov    DWORD PTR [rsp+0xc],eax
xor    eax,eax
call   400450 <printf@plt>
; longjmp( buf, 1 )
mov    esi,0x1
mov    edi,0x600a40
call   400490 <longjmp@plt>
```

Вывод программы будет `111` Volatile-переменная `b`, как видите, повела себя как задумано (иначе цикл никогда бы не закончился). Переменная `var` всегда была равна 1, несмотря на то что она «увеличивалась» согласно тексту программы.

> **Вопрос 265** Как реализовать конструкции наподобие «try–catch» с помощью `setjmp` и `longjmp`?

14.4 inline

`inline` — квалификатор функции, введённый в C99. Он имитирует поведение своего аналога в C++. Прежде чем читать объяснение, пожалуйста, *не* считайте, что это ключевое слово используется для принудительного встраивания функции! До C99 существовал квалификатор `static`, часто используемый в следующем сценарии:

- Заголовочный файл включает не объявление функции, а полное *определение* функции,

помеченное как `static`.

- Затем заголовок подключается в нескольких единицах трансляции. Каждая из них получает копию

сгенерированного кода, но, поскольку соответствующий символ локален для объектного файла, компоновщик не воспринимает это как конфликт множественного определения. В крупном проекте это даёт компилятору доступ к исходному коду функции, что позволяет ему при необходимости действительно встроить функцию. Разумеется, компилятор может и решить, что функцию лучше оставить не встроенной. В этом случае мы начинаем получать клоны этой функции практически везде. Каждый файл вызывает свою собственную копию, что плохо для локальности и раздувает как образ памяти, так и сам исполняемый файл. Ключевое слово `inline` решает эту проблему. Правильное его использование выглядит так:

- Опишите функцию `inline` в соответствующем заголовке, например,

```
inline int inc( int x ) { return x+1; }
```

- Ровно в одной единице трансляции (то есть в одном `.c`-файле) добавьте внешнее объявление

```
extern inline int inc( int x ) ;
```

Этот файл будет содержать код функции, на который будут ссылаться все остальные файлы, где функция не была встроена.

> **Изменение семантики** В GCC до версии 4.2.1 ключевое слово `inline` имело немного другой смысл. Подробный анализ см. в статье [14].

14.5 restrict

`restrict` — ключевое слово, схожее с `volatile` и `const`, впервые появившееся в стандарте C99. Оно используется для пометки указателей и потому ставится справа от звёздочки, вот так:

```
int x;
int* restrict p_x = &x;
```

Создавая ограниченный (restricted) указатель на объект, мы даём обещание, что все обращения к этому объекту будут проходить через значение этого указателя. Компилятор может либо проигнорировать это, либо воспользоваться этим для определённых оптимизаций, что часто возможно. Другими словами, любая запись через другой указатель не повлияет на значение, хранящееся по ограниченному указателю. Нарушение этого обещания приводит к неочевидным ошибкам и является явным случаем неопределённого поведения. Без `restrict` любой указатель — источник возможного алиасинга памяти, когда к одним и тем же ячейкам памяти можно обратиться, используя для них разные имена. Рассмотрим очень простой пример, показанный в листинге 14-18. Равно ли тело `f *x += 2 * (*add);?`

Листинг 14-18. `restrict_motiv.c`

```
void f(int* x, int* add) {
    *x += *add;
    *x += *add;
}
```

Ответ, как ни удивительно, нет, они не равны. Что, если `add` и `x` указывают на один и тот же адрес? В этом случае изменение `*x` изменяет и `*add` тоже. Так что если `x == add`, функция прибавит `*x` к `*x`, удвоив исходное значение, а затем повторит это, учетверив исходное значение. Однако когда `x != add`, даже если `*x == *add`, итоговое `*x` будет утроенным исходным значением. Компилятор прекрасно об этом знает, и даже с включёнными оптимизациями он не оптимизирует прочь два чтения, как показано в листинге 14-19.

Листинг 14-19. `restrict_motiv_dump.asm`

```
0000000000000000 <f>:
0:   8b 06                   mov   eax,DWORD PTR [rsi]
2:   03 07                   add   eax,DWORD PTR [rdi]
4:   89 07                   mov   DWORD PTR [rdi],eax
6:   03 06                   add   eax,DWORD PTR [rsi]
8:   89 07                   mov   DWORD PTR [rdi],eax
a:   c3                      ret
```

Однако добавьте `restrict`, как показано в листинге 14-20, и дизассемблирование продемонстрирует улучшение, как показано в листинге 14-21. Второй аргумент читается ровно один раз, умножается на 2 и прибавляется к разыменованному первому аргументу.

Листинг 14-20. `restrict_motiv1.c`

```
void f(int* restrict x, int* restrict add) {
    *x += *add;
    *x += *add;
}
```

Листинг 14-21. `restrict_motiv_dump1.asm`

```
0000000000000000 <f>:
   0:   8b 06                   mov   eax,DWORD PTR [rsi]
   2:   01 c0                   add   eax,eax
   4:   01 07                   add   DWORD PTR [rdi],eax
   6:   c3                      ret
```

Используйте `restrict`, только если уверены в том, что делаете. Написать чуть менее эффективную программу гораздо лучше, чем написать некорректную. `restrict` также важно использовать для документирования кода. Например, сигнатура `memcpy`, функции, копирующей *n* байт с некоторого начального адреса `s2` в блок, начинающийся с `s1`, изменилась в C99:

```
void*
memcpy(void*       restrict s1,
       const void* restrict s2,
       size_t               n );
```

Это отражает тот факт, что эти две области не должны пересекаться; иначе корректность не гарантируется. Ограниченные указатели можно копировать один в другой, создавая иерархию указателей. Однако стандарт ограничивает это случаями, когда копия не находится в том же блоке, что и исходный указатель. Листинг 14-22 показывает пример.

Листинг 14-22. `restrict_hierarchy.c`

```
struct s {
    int* x;
} inst;
void f(void) {
    struct s* restrict p_s = &inst;
    int* restrict p_x = p_s->x; /* Bad */
    {
        int* restrict p_x2 = p_s->x; /* Fine, other block scope */
    }
}
```

14.6 Строгий алиасинг

До появления `restrict` программисты иногда добивались того же эффекта, используя разные имена структур. Компилятор считает, что разные типы данных подразумевают, что соответствующие указатели не могут указывать на одни и те же данные (это известно как *правило строгого алиасинга*). Предположения включают следующее:

- Указатели на разные встроенные типы не алиасятся.
- Указатели на структуры или объединения *с разными метками* не алиасятся (так что `struct foo` и `struct bar` никогда не используются один вместо другого).
- Псевдонимы типов, созданные с помощью `typedef`, *могут* ссылаться на одни и те же данные.
- Тип `char*` — исключение (знаковый он или нет). Компилятор всегда предполагает, что `char*` может алиасить другие типы, но *не* наоборот. Это значит, что можно создать буфер `char`, использовать его для получения данных, а затем сослаться на него как на экземпляр некоторого `struct packet`.

Нарушение этих правил может привести к неочевидным ошибкам оптимизации, потому что вызывает неопределённое поведение. Пример, показанный в листинге 14-18, можно переписать, чтобы добиться того же эффекта без ключевого слова `restrict`. Идея в том, чтобы использовать правила строгого алиасинга в свою пользу, упаковав оба параметра в структуры с разными метками. Листинг 14-23 показывает изменённый исходный код.

Листинг 14-23. `restrict-hack.c`

```
struct a {
    int v;
};
struct b {
    int v;
};
void f(struct a* x, struct b* add) {
    x->v += add->v;
    x->v += add->v;
}
```

К нашему удовлетворению, компилятор оптимизирует чтения прочь именно так, как мы хотели. Листинг 14-24 показывает дизассемблирование.

Листинг 14-24. `restrict-hack-dump`

```
0000000000000000 <f>:
   0:   8b 06                   mov   eax,DWORD PTR [rsi]
   2:   01 c0                   add   eax,eax
   4:   01 07                   add   DWORD PTR [rdi],eax
   6:   c3                      ret
```

Мы не рекомендуем использовать правила алиасинга в целях оптимизации в коде для C99 и более новых стандартов, потому что `restrict` делает намерение более очевидным и не вводит лишних имён типов.

14.7 Проблемы безопасности

C не создавался как язык для написания надёжного программного обеспечения. Он позволяет работать с памятью напрямую и не имеет средств контроля корректности — ни статических, как Rust, ни динамических, как Java. Мы рассмотрим несколько классических дыр в безопасности, которые теперь можем объяснить во всех деталях.

14.7.1 Переполнение буфера на стеке

Предположим, программа использует функцию `f` с локальным буфером, как показано в листинге 14-25.

Листинг 14-25. `buffer_overrun.c`

```
#include <stdio.h>
void f( void ) {
    char buffer[16];
    gets( buffer );
}
int main( int argc, char** argv ) {
    f();
    return 0;
}
```

После инициализации раскладка кадра стека будет выглядеть следующим образом: функция `gets` читает строку из `stdin` и помещает её в буфер, чей адрес принимается как аргумент. К сожалению, она вообще не контролирует размер буфера и потому может его превысить. Если строка слишком длинная, она перезапишет буфер, затем сохранённое значение `rbp`, а затем адрес возврата. Когда выполнится инструкция `ret`, программа, скорее всего, аварийно завершится. Хуже того, если злоумышленник сформирует хитрую строку, он может переписать адрес возврата специфичными байтами, образующими действительный адрес. Если злоумышленник решит перенаправить адрес возврата прямо внутрь переполняемого буфера, он может передать исполняемый код прямо в этом буфере. Такой код часто называют *шелл-кодом (shellcode)*, потому что он маленький и обычно лишь открывает удалённую оболочку для работы. Очевидно, это не только изъян `gets`, но и особенность самого языка. Мораль в том, чтобы никогда не использовать `gets` и всегда предусматривать способ проверить границы целевого блока памяти.

14.7.2 return-to-libc

Как мы уже разобрали, злонамеренный пользователь может переписать адрес возврата, если программа позволяет ему переполнить буфер на стеке. Атака return-to-libc выполняется, когда адресом возврата становится адрес функции в стандартной библиотеке C. Особый интерес представляет одна функция, `int system(const char*command)`. Эта функция позволяет выполнить произвольную команду оболочки. Хуже того, она будет выполнена с теми же привилегиями, что и атакуемая программа. Когда текущая функция завершается выполнением команды `ret`, мы начнём выполнять функцию из libc. Остаётся вопрос: как сформировать для неё действительный аргумент? При наличии *ASLR* (address space layout randomization, рандомизации раскладки адресного пространства) выполнить такую атаку нетривиально (но всё же возможно).

14.7.3 Уязвимости форматного вывода

Функции форматного вывода могут быть источником весьма неприятных ошибок. В стандартной библиотеке есть несколько таких функций; таблица 14-1 их показывает.

Таблица 14-1. Функции форматирования строк

Функция Описание

Выводит форматированную строку.

```
printf
```

`fprintf` Записывает результат printf в файл.

Выводит в строку.

```
sprintf
```

`snprintf` Выводит в строку с проверкой длины.

Выводит структуру `va_arg` в файл.

```
vfprintf
```

`vprintf` Выводит структуру `va_arg` в `stdout`.

Выводит `va_arg` в строку.

```
vsprintf
```

Выводит `va_arg` в строку с проверкой длины.

```
vsnprintf
```

Листинг 14-26 показывает пример. Предположим, пользователь вводит менее 100 символов. Можете ли вы обрушить эту программу или вызвать другие интересные эффекты?

Листинг 14-26. `printf_vuln.c`

```
#include <stdio.h>
int main(void) {
    char buffer[1024];
    gets(buffer);
    printf( buffer );
    return 0;
}
```

Уязвимость возникает не из-за использования `gets`, а из-за использования строки формата, взятой у пользователя. Пользователь может предоставить строку, содержащую спецификаторы формата, что приведёт к интересному поведению. Мы упомянём несколько потенциально нежелательных видов поведения.

- Спецификаторы `"%x"` и им подобные можно использовать для просмотра содержимого стека. Первые 5 `"%x"`

возьмут аргументы из регистров (`rdi` уже занят адресом строки формата), а следующие покажут содержимое стека. Скомпилируем пример, показанный в листинге 14-26, и посмотрим на его реакцию на ввод `"%x %x %x %x %x %x %x %x`

```
%x %x %x".
> %x %x %x %x %x %x %x %x %x %x
b1b6701d b19467b0 fbad2088 b1b6701e 0 25207825 20782520 78252078 25207825
```

Как видим, это действительно дало нам четыре числа, обладающих определённым неформальным сходством, 0 и ещё два числа. Наша гипотеза в том, что последние два числа уже взяты из стека. Зайдя в `gdb` и исследовав память рядом с вершиной стека сразу после вызова `printf`, мы получим результаты, подтверждающие нашу точку зрения. Листинг 14-27 показывает вывод.

Листинг 14-27. `gdb_printf`

```
(gdb) x/10 $rsp
0x7fffffffdfe0: 0x25207825   0x78252078   0x20782520   0x25207825
0x7fffffffdff0: 0x78252078   0x20782520   0x25207825   0x00000078
0x7fffffffe000: 0x00000000   0x00000000
```

- Спецификатор формата `"%s"` используется для вывода строк. Поскольку строка определяется адресом

своего начала, это означает обращение к памяти по указателю. Так что если действительный указатель не предоставлен, будет разыменован недействительный указатель.

> **Вопрос 266** Каким будет результат запуска кода, показанного в листинге 14-26, на входе `"%s %s %s %s %s"`?

- Спецификатор формата `"%n"` немного экзотичен, но всё же вреден. Он позволяет записать целое число

в память. Функция `printf` принимает *указатель* на целое число, которое будет переписано количеством символов, выведенных на данный момент (до появления `"%n"`). Листинг 14-28 показывает пример его использования.

Листинг 14-28. `printf_n.c`

```
#include <stdio.h>
int main(void) {
    int count;
    printf( "hello%n world\n", &count);
    printf( "%d\n", count );
    return 0;
}
```

Это выведет 5, потому что перед `"%n"` было выведено пять символов. Это нетривиальная длина строки, потому что перед этим могут быть другие спецификаторы формата, приводящие к выводу переменной длины (например, вывод целого числа может дать семь или десять символов). Листинг 14-29 показывает пример.

Листинг 14-29. `printf_n_ex.c`

```
int x;
printf("%d %n", 10, &x);  /* x = 3 */
printf("%d %n", 200, &x); /* x = 4 */
```

Чтобы этого избежать, не используйте строку, полученную от пользователя, в качестве строки формата. Вы всегда можете написать `printf("%s", buffer)`, что безопасно, пока buffer не `NULL` и является действительной строкой, завершающейся нулём. Не забывайте и про такие функции, как `puts` или `fputs`, которые не только быстрее, но и безопаснее.

14.8 Механизмы защиты

Переписывание адреса возврата может привести к одному из следующих двух последствий:

- Программа аварийно завершается.
- Злоумышленник выполняет произвольный код.

В первом случае мы можем стать жертвой атаки DoS (Denial of Service, отказ в обслуживании), когда программа, предоставляющая определённый сервис, становится недоступной. Однако второй вариант гораздо хуже.

14.8.1 Защитная метка (security cookie)

Защитная метка (security cookie, stack guard, canary) призвана защитить нас от выполнения произвольного кода, вынуждая программу аварийно завершиться, как только адрес возврата изменён. Защитная метка — случайное значение, находящееся в кадре стека рядом с сохранённым `rbp` и адресом возврата. Переполнение буфера перепишет защитную метку. Перед инструкцией `ret` компилятор вставляет специальную проверку, удостоверяющуюся в целостности защитной метки, и если она изменена, обрушивает программу. Инструкция `ret` не выполняется. И у MSVC, и у GCC этот механизм включён по умолчанию.

14.8.2 Рандомизация раскладки адресного пространства

Загрузка каждой секции программы в случайное место адресного пространства делает почти невозможным угадать правильный адрес возврата для выполнения осмысленного перехода. Большинство широко используемых операционных систем это поддерживают; однако эту возможность нужно включить во время компиляции. В этом случае информация о поддержке ASLR будет храниться в самом исполняемом файле, что заставит загрузчик выполнить правильное перемещение.

14.8.3 DEP

Мы уже обсуждали *Data Execution Prevention (предотвращение выполнения данных)* в Главе 4. Эта технология защищает некоторые страницы от выполнения инструкций, хранящихся на этих страницах. Чтобы её включить, программы также должны компилироваться с включённой поддержкой. Печальный факт в том, что она плохо работает с программами, использующими just-in-time компиляцию, формирующую исполняемый код прямо во время выполнения программы. Это не так редко, как может показаться; например, практически все браузеры используют движки JavaScript, поддерживающие just-in-time компиляцию.

14.9 Итоги

В этой главе мы вернулись к соглашению о вызовах, используемому в *nix на Intel 64. Мы рассмотрели примеры использования более продвинутых возможностей C, а именно квалификаторов типа `volatile` и `restrict` и нелокальных переходов. Наконец, мы дали краткий обзор нескольких классических уязвимостей, возможных из-за способа организации кадров стека, и возможностей компилятора, разработанных для автоматического противодействия им. Следующая глава объяснит более низкоуровневые детали, связанные с созданием и использованием динамических библиотек, чтобы укрепить наше понимание их.

Вопрос 267 Что такое регистры `xmm` ? Сколько их?

Вопрос 268 Что такое инструкции SIMD?

Вопрос 269 Почему некоторые инструкции SSE требуют, чтобы операнды в памяти были выровнены?

Вопрос 270 Какие регистры используются для передачи аргументов функциям?

Вопрос 271 Почему при передаче аргументов функции иногда используется `rax` ?

Вопрос 272 Как используется регистр `rbp` ?

Вопрос 273 Что такое кадр стека?

Вопрос 274 Почему мы не адресуем локальные переменные относительно `rsp` ?

Вопрос 275 Что такое пролог и эпилог?

Вопрос 276 Каково назначение инструкций `enter` и `leave` ?

Вопрос 277 Подробно опишите, как кадр стека меняется во время выполнения функции.

Вопрос 278 Что такое красная зона?

Вопрос 279 Как объявить и использовать функцию с переменным числом аргументов?

Вопрос 280 Какой контекст хранит `va_list` ?

Вопрос 281 Зачем используются такие функции, как `vfprintf` ?

Вопрос 282 Каково назначение переменных `volatile` ?

Вопрос 283 Почему только `volatile` -переменные стека сохраняются после `longjmp` ?

Вопрос 284 Все ли локальные переменные выделяются на стеке?

Вопрос 285 Для чего используется `setjmp` ?

Вопрос 286 Каково возвращаемое значение `setjmp` ?

Вопрос 287 В чём польза `restrict` ?

Вопрос 288 Может ли компилятор проигнорировать `restrict` ?

Вопрос 289 Как добиться того же результата без использования ключевого слова `restrict` ?

Вопрос 290 Объясните механизм эксплуатации переполнения буфера на стеке.

Вопрос 291 Когда использование `printf` небезопасно?

Вопрос 292 Что такое защитная метка? Решает ли она проблему обрушения программы при переполнении буфера?

Разделяемые объекты и модели кода

Глава 5 уже дала краткий обзор динамических библиотек (также известных как разделяемые объекты). Эта глава вернётся к динамическим библиотекам и расширит наши знания, введя понятия таблицы связей процедур (Program Linkage Table) и таблицы глобальных смещений (Global Offset Table). В результате мы сможем собрать разделяемую библиотеку на чистом ассемблере и на C, сравнить результаты и изучить её структуру. Мы также изучим понятие моделей кода, о котором редко говорят, но которое даёт цельное представление о нескольких важных деталях генерации ассемблерного кода.

15.1 Динамическая загрузка

Как вы, возможно, помните, ELF-файл (Executable and Linkable Format) содержит три заголовка:

- Главный заголовок, находящийся по смещению ноль. Он определяет общую информацию о файле, включая точку входа и смещения до двух таблиц, разобранных ниже. Его можно посмотреть с помощью команды `readelf -h`.
- Таблица заголовков секций, содержащая информацию о разных секциях ELF. Её можно посмотреть с помощью команды `readelf -S`.
- Таблица заголовков программы, содержащая информацию о сегментах файла. Каждый сегмент — структура времени выполнения, содержащая одну или несколько секций, определённых в таблице заголовков секций. Её можно посмотреть с помощью команды `readelf -l`.

Начальный этап загрузки исполняемого файла — создание адресного пространства и выполнение отображений памяти согласно таблице заголовков программы с соответствующими правами доступа. Это выполняется ядром операционной системы. Как только виртуальное адресное пространство настроено, должна вмешаться другая программа (то есть динамический загрузчик). Последний должен быть исполняемой программой, полностью перемещаемой (то есть способной загружаться по любому нужному нам адресу). Назначение динамического компоновщика:

- Определить все зависимости и загрузить их.
- Выполнить *перемещение* приложения и зависимостей.
- Инициализировать приложение и его зависимости и передать управление приложению. Теперь начнётся выполнение программы.

Определение зависимостей и их загрузка относительно просты: это сводится к рекурсивному поиску зависимостей и проверке, был ли объект уже загружен. Инициализация тоже не особо загадочна. А вот перемещение представляет для нас интерес. Есть два вида перемещений:

- Связи с местами внутри того же объекта. Все такие перемещения выполняет статический

компоновщик, потому что они известны на момент компоновки.

- Зависимости от символов, которые обычно находятся в другом объекте.

Второй вид перемещения затратнее и выполняется динамическим компоновщиком. Перед выполнением перемещений нам сначала нужно выполнить поиск, чтобы найти символы, которые мы хотим связать. Существует понятие *области поиска (lookup scope)* объектного файла — упорядоченного списка, содержащего некоторые другие загруженные объекты. Область поиска объектного файла используется для разрешения нужных ему символов. Способ её вычисления описан в [24] и довольно сложен, так что мы отсылаем вас к соответствующему документу в случае необходимости. Область поиска состоит из трёх частей, перечисленных в обратном порядке поиска — то есть символ сначала ищется в третьей части области.

1. Глобальная область поиска, состоящая из исполняемого файла и всех его зависимостей,

включая зависимости зависимостей и т. д. Они перечисляются в порядке поиска в ширину, то есть:

- Сам исполняемый файл.
- Его зависимости.
- Зависимости его первой зависимости, затем второй и т. д. Каждый объект загружается только один раз.

2. Часть, строящаяся, если в метаданных исполняемого ELF-файла установлен флаг `DF_SYMBOLIC`.

Она считается устаревшей; её использование не рекомендуется, так что мы не будем её здесь изучать.

3. Объекты, загруженные динамически со всеми своими зависимостями с помощью вызова функции `dlopen`.

Они не участвуют в обычных поисках. Каждый объектный файл содержит хеш-таблицу, используемую для поиска.[28] Эта таблица хранит информацию о символах и используется для быстрого поиска символа по имени. Компонуется первый объект в области поиска, содержащий нужный символ, что позволяет перегружать символы — например, с помощью механизма `LD_PRELOAD`, который мы рассмотрим в разделе 15.5. Размер хеш-таблицы и число экспортируемых символов влияют на время поиска. Когда для *компоновщика* указан флаг `-O`,[29] он пытается оптимизировать эти параметры ради лучшей скорости поиска. Помните, что в таких языках, как C++, имена символов вычисляются не только

на основе, например, имени функции, но и кодируют все свои пространства имён (и имя класса), что легко может привести к именам в несколько сотен символов. В случае коллизий в хеш-таблицах (которые обычно часты) нужно выполнить сравнение строк между именем искомого символа и всеми символами в корзине, выбранной вычислением его хеша. Современные хеш-таблицы в стиле GNU предоставляют дополнительную эвристику — использование фильтра Блума[30] для быстрого ответа на вопрос: «а определён ли вообще этот символ в этом объектном файле?» Это делает ненужные поиски намного реже, что положительно сказывается на производительности.

15.2 Перемещения и PIC

Итак, какие перемещения выполняются? Мы видели процесс перемещений при статической компоновке в Главе 5. Можем ли мы сделать то же самое, перемещая все элементы кода и данных? Ответ да, можем, и пока распространённые архитектуры не добавили специальные возможности для облегчения написания позиционно-независимого кода, этот подход широко использовался. Однако у него есть следующие недостатки:

- Перемещения выполняются медленно, особенно когда зависимостей много. Это может задержать запуск приложения.
- Секцию *.text* нельзя разделять, потому что её нужно патчить. В то время как статическая компоновка подразумевает патчинг содержимого объектного файла при сборке итогового объектного файла, динамическая компоновка подразумевает патчинг объектных файлов в памяти. Это не только тратит память впустую, но и создаёт риск безопасности, потому что, например, шелл-код может переписать программу прямо в памяти, чтобы изменить её поведение.

Сегодня PIC — рекомендуемый способ, позволяющий держать *.text* доступной только для чтения (тогда как *.data* разделить всё равно нельзя). Число перемещений будет меньше, потому что перемещения кода выполняться не будут. PIC подразумевает использование двух служебных таблиц: таблицы глобальных смещений (Global Offset Table, GOT) и таблицы связей процедур (Program Linkage Table, PLT).

15.3 Пример: динамическая библиотека на C

Прежде чем начать изучать GOT и PLT, давайте создадим минимальный рабочий пример динамической библиотеки на C. На самом деле это довольно просто. Наша программа будет состоять из двух файлов: `mainlib.c` (показан в листинге 15-1) и `dynlib.c` (показан в листинге 15-2).

Листинг 15-1. `mainlib.c`

```
extern void libfun( int value );
int global = 100;
int main( void ) {
    libfun( 42 );
    return 0;
}
```

Листинг 15-2. `dynlib.c`

```
#include <stdio.h>
extern int global;
void libfun(int value) {
    printf( "param: %d\n", value );
    printf( "global: %d\n", global );
}
```

Как видим, в главном файле есть глобальная переменная, которой мы хотим поделиться с библиотекой; библиотека явно объявляет её как `extern`. В главном файле есть объявление функции библиотеки (обычно оно размещается в заголовочном файле, поставляемом со скомпилированной библиотекой). Чтобы скомпилировать эти файлы, нужно выполнить следующие команды:

```
> # creating object file for the main part
> gcc -c  -o mainlib.o mainlib.c
> # creating object file for the library
> gcc -c -fPIC -o dynlib.o  dynlib.c
> gcc -o dynlib.so -shared dynlib.o # creating dynamic library itself
> # creating an executable and linking it with the dynamic library
> gcc -o main  mainlib.o dynlib.so
```

Сначала мы, как обычно, создаём объектные файлы. Затем мы собираем динамическую библиотеку с помощью флага `-shared`. При сборке исполняемого файла мы указываем все динамические библиотеки, от которых он зависит, потому что эта информация

должна быть включена в метаданные ELF. Обратите внимание на использование флага `-fPIC`, заставляющего генерировать позиционно-независимый код. Эффект этого флага на ассемблер мы увидим позже. Проверим зависимости файла с помощью `ldd`.

```
> ldd main
        linux-vdso.so.1 => (0x00007fffcd428000)
        lib.so => not found
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007ff988d60000)
        /lib64/ld-linux-x86-64.so.2 (0x00007ff989200000)
```

Наша свежая библиотека присутствует в списке зависимостей, но `ldd` не может её найти. Попытка запустить исполняемый файл проваливается с ожидаемым сообщением:

```
./main: error while loading shared libraries:
    lib.so: cannot open shared object file: No such file or directory
```

Библиотеки ищутся в местах по умолчанию (таких как `/lib/`). Нашей там нет, так что у нас есть другой вариант: переменная окружения `LD_LIBRARY_PATH` разбирается, чтобы получить список дополнительных каталогов, где могут находиться библиотеки. Как только мы устанавливаем её в текущий каталог, `ldd` находит библиотеку. Заметьте, что поиск начинается с каталогов, определённых в `LD_LIBRARY_PATH`, и продолжается стандартными каталогами.

```
> export LD_LIBRARY_PATH=.
> ldd main
        linux-vdso.so.1 =>  (0x00007ffff1315000)
        lib.so => ./lib.so (0x00007f3a7bc70000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f3a7b890000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f3a7c000000)
```

Запуск даёт ожидаемые результаты.

```
> ./main
   param: 42
   global: 100
```

15.4 GOT и PLT

15.4.1 Доступ к внешним переменным

Чтобы держать *.text* доступной только для чтения и никогда не патчить её из-за перемещений, мы добавляем уровень косвенности при обращении к любому символу, гарантированно не определённому в том же объекте — другими словами, к каждому символу, определённому в исполняемом или разделяемом объектном файле после статической компоновки. Эта косвенность выполняется через специальную *таблицу глобальных смещений*. Важны два факта, чтобы код PIC заработал.

- Intel 64 позволяет адресовать операнды инструкций относительно регистра `rip`. Получить текущее значение `rip` можно и с помощью пары инструкций `call` и `pop`, но аппаратная поддержка, безусловно, помогает с производительностью.
- Смещение между секцией *.text* и секцией *.data* известно на момент компоновки, то есть когда динамическая библиотека *создаётся*. Это также значит, что расстояние между `rip` и началом секции *.data* тоже известно. Так что мы размещаем таблицу глобальных смещений в секции *.data* или рядом с ней. Она будет хранить абсолютные адреса глобальных переменных.

Мы обращаемся к ячейке GOT относительно `rip` и получаем оттуда абсолютный адрес глобальной переменной — см. рисунок 15-1.

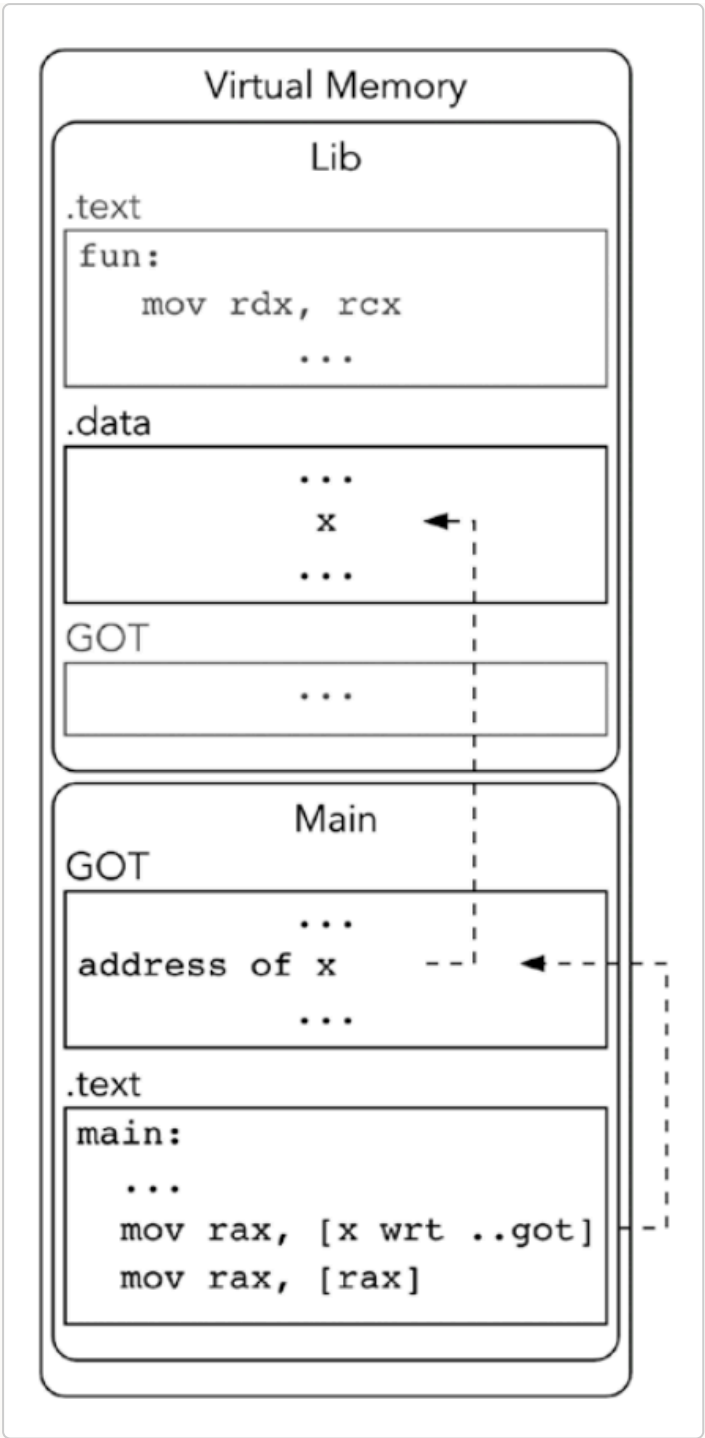


Рисунок 15-1. Доступ к глобальной переменной через GOT

Посмотрим, как переменная `global`, созданная в главном исполняемом файле, адресуется в динамической библиотеке. Для этого мы изучим фрагмент вывода `objdump -D -Mintel-mnemonic`, показанный в листинге 15-3.

Листинг 15-3. `libfun`

```
00000000000006d0 <libfun>:
# Function prologue
 6d0: 55                      push   rbp
 6d1: 48 89 e5                mov    rbp,rsp
 6d4: 48 83 ec 10             sub    rsp,0x10
# Second argument for printf( "param: %d\n", value );
 6d8: 89 7d fc                mov    DWORD PTR [rbp-0x4],edi
 6db: 8b 45 fc                mov    eax,DWORD PTR [rbp-0x4]
 6de: 89 c6                   mov    esi,eax
# First argument for printf( "param: %d\n", value );
 6e0: 48 8d 3d 32 00 00 00    lea    rdi,[rip+0x32]
# Printf call; no XMM registers used
 6e7: b8 00 00 00 00          mov    eax,0x0
 6ec: e8 bf fe ff ff          call   5b0 <printf@plt>
# Second argument for printf( "global: %d\n", global );
 6f1: 48 8b 05 e0 08 20 00    mov    rax,QWORD PTR [rip+0x2008e0]
 6f8: 8b 00                   mov    eax,DWORD PTR [rax]
 6fa: 89 c6                   mov    esi,eax
# First argument for printf( "global: %d\n", global );
 6fc: 48 8d 3d 21 00 00 00    lea    rdi,[rip+0x21]
# Printf call; no XMM registers used
 703: b8 00 00 00 00          mov    eax,0x0
 708: e8 a3 fe ff ff          call   5b0 <printf@plt>
# Function epilogue
 70d: 90                      nop
 70e: c9                      leave
 70f: c3                      ret
```

Помните, что исходный код показан в листинге 15-2. Нас интересует, как обращаются к глобальным переменным. Сначала заметьте, что первый аргумент `printf` (адрес строки формата, находящейся в *.rodata*) обращается не типичным способом. В таких случаях у нас обычно было абсолютное значение адреса (которое заполнил бы компоновщик во время перемещения, как объяснено в разделе 5.3.2). Однако здесь используется адрес относительно `rip`. Как мы понимаем, `rdi` как первый аргумент должен хранить адрес строки формата. Так что этот адрес хранится *в памяти* по адресу `[rip + 0x32]`. Это место — часть GOT. Теперь посмотрим, как `global` обращается из кода динамической библиотеки. На самом деле механизм абсолютно тот же, хотя требуется ещё одно чтение памяти. Сначала мы читаем содержимое GOT в

```
mov rax,QWORD PTR [rip+0x2008e0]
```

чтобы получить адрес `global`, затем мы читаем его значение, снова обращаясь к памяти в

```
mov eax,DWORD PTR [rax].
```

Довольно просто для глобальных переменных. Для функций, однако, реализация чуть сложнее.

15.4.2 Вызов внешних функций

Хотя точно такой же подход мог бы сработать и для функций, для реализации ленивого поиска функций по требованию добавлена дополнительная возможность. Сначала обсудим причины этого. Поиск определений символов нетривиален, как мы уже видели в этой главе. Обычно функций гораздо больше, чем экспортируемых глобальных переменных, и лишь малая их часть на самом деле вызывается во время выполнения программы (например, функции обработки ошибок). В общем случае, когда программисты получают динамическую библиотеку для использования в своей программе, они часто получают стороннюю библиотеку, в которой функций гораздо больше, чем им реально нужно вызывать. Мы добавляем ещё один уровень косвенности через специальную таблицу связей процедур (Program Linkage Table, PLT). Она находится в секции *.text*. У каждой функции, вызываемой разделяемой библиотекой, есть запись в PLT. Каждая запись — небольшой фрагмент исполняемого кода, скомпонованный статически и потому вызываемый напрямую. Вместо вызова функции, чей адрес хранился бы в GOT, мы вызываем заглушку для неё. Чтобы это проиллюстрировать, мы набросаем PLT в листинге 15-4.

Листинг 15-4. `plt_sketch.asm`

```
; somewhere in the program
call func@plt
; PLT
PLT_0:           ; the common part
call resolver
...
PLT_n:     func@plt:
jmp [GOT_n]
PLT_n_first:
; here the arguments for resolver are prepared
jmp PLT_0
GOT:
...
GOT_n:
dq PLT_n_first
```

Итак, что там происходит?

- Вызов функции ссылается на запись PLT в обход GOT.
- Нулевая запись PLT определяет «общий код» для всех записей. Все они в итоге

переходят к этой записи.

- *n*-я запись начинается с перехода по адресу, хранящемуся в *n*-й записи GOT. Значение

этой записи по умолчанию — адрес следующей инструкции после этого перехода! В нашем примере он обозначен меткой `PLT_n_first`. Так что при первом вызове функции мы переходим к следующей инструкции, фактически выполняя операцию `NOP`.

- Этот код готовит аргументы для динамического загрузчика и переходит к общему коду

```
in PLT_0.
```

- В `PLT_0` вызывается загрузчик. Он выполняет поиск и разрешает адрес функции,

заполняя `GOT_n` настоящим адресом функции. Следующий вызов функции уже не будет задействовать динамический загрузчик: будет вызвана заглушка `PLT_n`, которая немедленно перейдёт к разрешённой функции, чей адрес теперь находится в GOT. Схему изменений в PLT из-за процесса разрешения символов см. на рисунках 15-2 и 15-3.

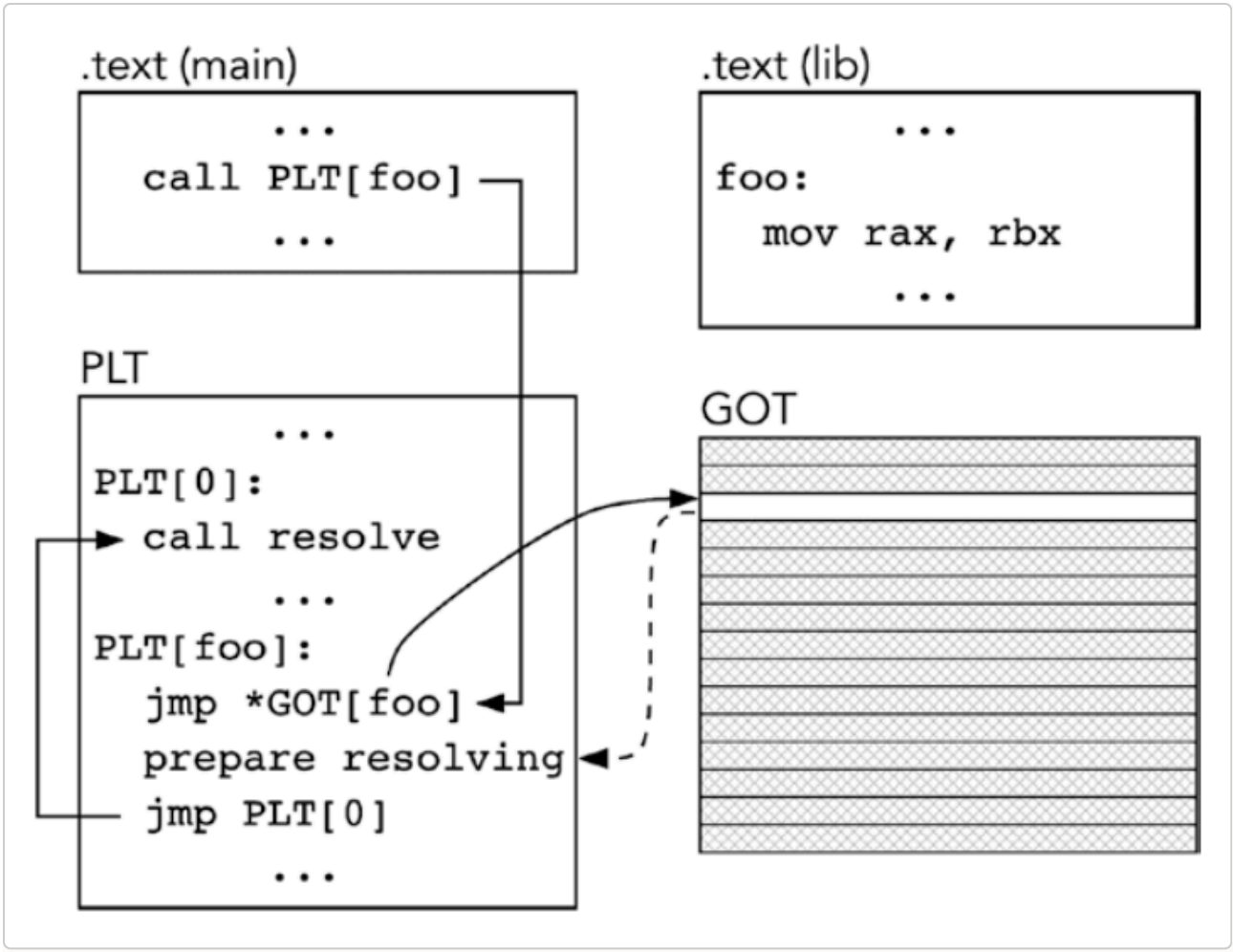


Рисунок 15-2. PLT до компоновки функции во время выполнения

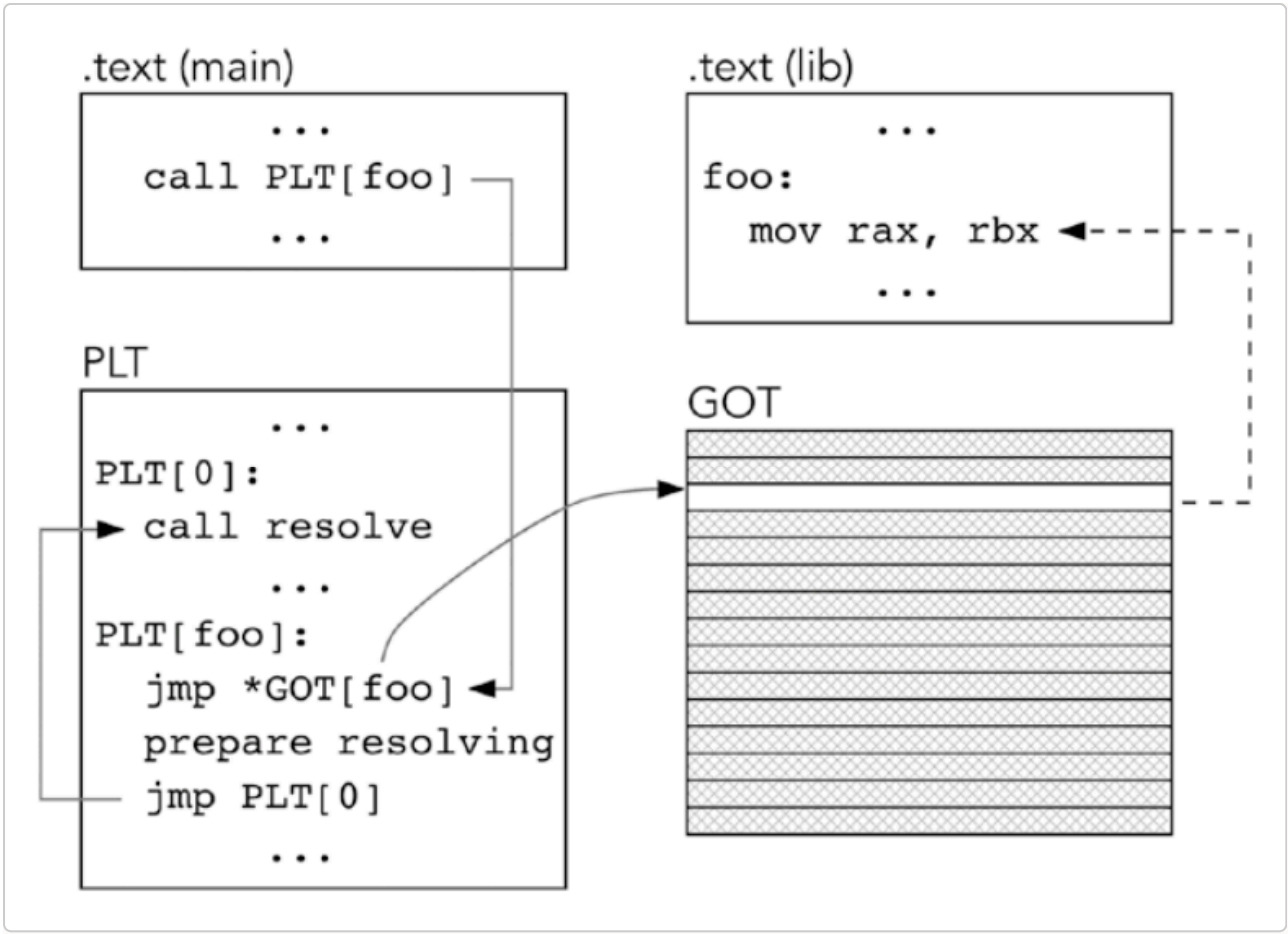


Рисунок 15-3. PLT после компоновки функции во время выполнения

> **Вопрос 293** Прочитайте в man `ld.so` про переменные окружения (такие как `LD_BIND_NOT`), способные изменить поведение загрузчика.

15.4.3 Пример PLT

Чтобы быть до конца честными, мы изучим код, сгенерированный для примера, показанного в разделе 15.3. Функция `main` вызывает `libfun`, что, как мы и ожидали, выполняется через PLT.

```
Disassembly of section .text:
00000000004006a6 <main>:
  push   rbp
  mov    rbp,rsp
  mov    edi,0x2a
  call   400580 <libfun@plt>
  mov    eax,0x0
  pop    rbp
  ret
```

Далее посмотрим, как выглядит PLT. Запись PLT для `libfun` называется `libfun@plt`. Найдите её в листинге 15-5.

Листинг 15-5. `plt_rw.asm`

```
Disassembly of section .init:
0000000000400550 <_init>:
sub    rsp,0x8
mov    rax,QWORD PTR [rip+0x200a9d]        # 600ff8 <_DYNAMIC+0x1e0>
test   rax,rax
je     400565 <_init+0x15>
call   4005a0 <__libc_start_main@plt+0x10>
add    rsp,0x8
ret
Disassembly of section .plt:
0000000000400570 <libfun@plt-0x10>:
push   QWORD PTR [rip+0x200a92]       # 601008 <_GLOBAL_OFFSET_TABLE_+0x8>
jmp    QWORD PTR [rip+0x200a94]       # 601010 <_GLOBAL_OFFSET_TABLE_+0x10>
nop    DWORD PTR [rax+0x0]
0000000000400580 <libfun@plt>:
imp    QWORD PTR [rip+0x200a92]       # 601018 <_GLOBAL_OFFSET_TABLE_+0x18>
push   0x0
jmp    400570 <_init+0x20>
0000000000400590 <__libc_start_main@plt>:
jmp    QWORD PTR [rip+0x200a8a]        # 601020 <_GLOBAL_OFFSET_TABLE_+0x20>
push   0x1
jmp    400570 <_init+0x20>
Disassembly of section .got:
0000000000600ff8 <.got>:
...
Disassembly of section .got.plt:
0000000000601000 <_GLOBAL_OFFSET_TABLE_>:
...
```

Первая инструкция — переход в GOT к её третьему элементу (потому что каждая запись занимает 8 байт, а смещение — `0x18`). Затем выполняется инструкция `push`, чей операнд — номер функции в PLT. Для `libfun` это 0x0, для `libc_start_main` — 0x1. Следующая инструкция в `libfun@plt` — переход к `_init+0x20`, что странно, но если проверить настоящий адрес `_init`, мы увидим, что

- `_init` находится по адресу 0x400550.
- `_init+0x20` находится по адресу 0x400570.
- `libfun@plt-0x10` тоже находится по адресу 0x400570, так что это один и тот же адрес.

- Этот адрес также является началом секции `.plt` и, согласно объяснению выше, должен соответствовать «общему» коду, разделяемому всеми записями PLT. Он помещает в стек ещё одно значение GOT и берёт из GOT адрес динамического загрузчика, чтобы перейти к нему.

Комментарии, выданные `objdump`, показывают, что последние два значения ссылаются на адреса 0x601008 и 0x601010. Как видим, они должны храниться где-то в секции `.got.plt`, которая является частью GOT, относящейся к записям PLT. Листинг 16 показывает содержимое этой секции.

Листинг 15-6. `got_plt_dump_ex.c`

```
Contents of section .got.plt:
0x601000   180e6000 00000000 00000000 00000000
0x601010   00000000 00000000 86054000 00000000
0x601020   96054000 00000000
```

Внимательно приглядевшись, мы видим, что начиная с адреса 0x601018 расположены следующие байты:

```
86 05 40 00 00 00 00 00
```

Помня о том, что Intel 64 использует little endian, мы заключаем, что настоящее учетверённое слово, хранящееся здесь, — `0x400586`, что на самом деле является адресом `libfun@plt + 6`, другими словами, адресом инструкции `push0x0`. Это иллюстрирует тот факт, что начальные значения для функций в GOT указывают на вторые инструкции их соответствующих записей PLT.

15.5 Предзагрузка

Установка переменной `LD_PRELOAD` позволяет предзагрузить разделяемые объекты раньше любой другой библиотеки (включая стандартную библиотеку C). Функции из этой библиотеки будут иметь приоритет при поиске, так что они могут переопределять функции, определённые в обычно загружаемых разделяемых объектах. Динамический загрузчик игнорирует значение `LD_PRELOAD`, если эффективный идентификатор пользователя не совпадает с реальным идентификатором пользователя. Это сделано из соображений безопасности. Мы напишем и скомпилируем простую программу, показанную в листинге 15-7.

Листинг 15-7. `preload_launcher.c`

```
#include <stdio.h>
int main(void) {
    puts("Hello, world!");
    return 0;
}
```

Она не делает ничего впечатляющего, но важно, что она использует функцию `puts`, определённую в стандартной библиотеке C. Мы перепишем её нашей версией `puts`, игнорирующей свой ввод и просто выводящей фиксированную строку. При запуске этой программы выполняется стандартная функция `puts`. Теперь давайте сделаем простую динамическую библиотеку с содержимым, показанным в листинге 15-8. Она подменяет функцию `puts` своей альтернативой, которая игнорирует свой аргумент и всегда выводит фиксированную строку.

Листинг 15-8. `prelib.c`

```
#include <stdio.h>
int puts( const char* str ) {
    return printf("We took control over your C library! \n");
}
```

Мы компилируем её следующими командами:

```
> gcc -o preload_launcher preload_launcher.c
> gcc -c -fPIC prelib.c
> gcc -o prelib.so -shared prelib.o
```

Заметьте, что исполняемый файл *не* был скомпонован с динамической библиотекой. Листинг 15-9 показывает эффект от установки переменной `LD_PRELOAD`.

Листинг 15-9. `ld_preload_effect`

```
> export LD_PRELOAD=
> ./a.out
Hello, world!
> export LD_PRELOAD=$PWD/prelib.so
> ./a.out
We took control over your C library!
```

Как видим, если `LD_PRELOAD` содержит путь к разделяемому объекту, определяющему некоторые функции, они переопределят другие функции, присутствующие в адресном пространстве процесса.

> **Вопрос 294** Обратитесь к заданию. Используйте эту технику, чтобы протестировать свою реализацию `malloc` против некоторых стандартных утилит из `coreutils`.

> **Вопрос 295** Прочитайте про функции `dlopen`, `dlsym`, `dlclose`.

15.6 Итоги по адресации символов

Прежде чем перейти к примерам на ассемблере и C, подведём итоги возможных случаев адресации символов. Главный исполняемый файл обычно не перемещаемый и не позиционно-независимый и загружается по фиксированному абсолютному адресу, скажем, `0x40000`.[31] Динамическая библиотека сегодня строится с использованием позиционно-независимого кода, и потому её *.text* можно разместить где угодно; в других секциях перемещения могут понадобиться. Символ может быть:

1. Определён в исполняемом файле и используется там же локально.

Это тривиально, потому что символы будут привязаны к абсолютным адресам. Адресация данных будет абсолютной, переходы и вызовы кода обычно будут генерироваться со смещениями относительно `rip`.

2. Определён в динамической библиотеке и используется только там же локально (недоступен внешним объектам). При наличии PIC это делается с помощью адресации относительно `rip` (для данных) или относительных смещений (для вызовов функций). Более общий случай будет обсуждаться позже, в разделе 15.10. NASM использует ключевое слово `rel` для достижения адресации относительно `rip`. Это не задействует ни GOT, ни PLT.
3. Определён в исполняемом файле и используется глобально. Это требует использования GOT (а также PLT для функций), если пользователь внешний. Для внутреннего использования правила те же: нам не нужны GOT или PLT для адресации внутри того же объектного файла.
4. Определён в динамической библиотеке и используется глобально.

Должно быть скорее пунктом связного списка, чем отдельным абзацем.

15.7 Примеры

Вполне возможно написать динамическую библиотеку на языке ассемблера, которая будет позиционно-независимой и будет использовать таблицы GOT и PLT.

> **Компоновка с помощью gcc** Рекомендуемый способ компоновки библиотек — использование GCC. Однако в этой главе мы иногда будем использовать более примитивный `ld`, чтобы подробнее показать, что происходит на самом деле. Когда задействован рантайм C, никогда не используйте `ld`.

Мы также, как всегда, ограничимся Intel 64. Писать код PIC было немного сложнее до появления адресации относительно `rip`.

15.7.1 Вызов функции

В первом примере будут показаны следующие возможности:

- Адресация данных динамической библиотеки внутри той же библиотеки.
- Вызов функции динамической библиотеки из главного исполняемого файла.

Этот пример состоит из `main.asm` (листинг 15-10) и `lib.asm` (листинг 15-11). `Makefile` приведён в листинге 15-12, чтобы показать процесс сборки. Заметьте, что явное указание динамического компоновщика *обязательно*, если только вы не используете GCC для компоновки файлов (который позаботится о правильном пути к динамическому компоновщику). Подробнее см. в разделе 15.7.2.

Листинг 15-10. `ex1-main.asm`

```
extern _GLOBAL_OFFSET_TABLE_
global _start
extern sofun
section .text
_start:
call sofun wrt ..plt
; `exit` system call
mov rdi, 0
mov rax, 60
syscall
```

Первое, что мы замечаем, — `extern _GLOBAL_OFFSET_TABLE_` обычно импортируется в каждом файле, компонуемом динамически.[32] Главный файл импортирует символ `sofun`. Затем вызов содержит не только имя функции, но и квалификатор `wrt ..plt`. Обращение к символу с помощью `wrt ..plt` заставляет компоновщик создать запись PLT.

Соответствующее выражение будет вычислено в смещение до записи PLT относительно текущей позиции в коде. До статической компоновки это смещение неизвестно, но его заполнит статический компоновщик. Тип этого вида перемещения должен быть перемещением относительно `rip` (как в инструкциях `call` или `jmp`-подобных). Структура ELF не предоставляет средств адресовать записи PLT по их абсолютным адресам.

Листинг 15-11. `ex1-lib.asm`

```
extern _GLOBAL_OFFSET_TABLE_
global sofun:function
section .rodata
msg: db "SO function called", 10
.end:
section .text
sofun:
mov rax, 1
mov rdi, 1
lea rsi, [rel msg]
mov rdx, msg.end - msg
syscall
ret
```

Заметьте, что глобальный символ `sofun` помечен как `:func` (перед двоеточием не должно быть пробела). Очень важно помечать экспортируемые функции именно так, если к ним нужно динамически обращаться из других объектов. Метка `.end` позволяет нам статически вычислить длину строки, чтобы передать её системному вызову `write`. Важное изменение — использование ключевого слова `rel`.

Код позиционно-независим, так что абсолютный адрес `msg` может быть произвольным. Его смещение относительно этой точки в коде (инструкция `lea rsi, [rel msg]`) фиксировано. Так что мы можем использовать `lea` для вычисления его адреса как смещения от `rip`. Эта строка скомпилируется в `lea rsi, [rip + offset]`, где `offset` — константа, которую заполнит статический компоновщик. Последняя форма `([rip + offset])` синтаксически некорректна в NASM. Листинг 15-12 показывает `Makefile`, используемый для сборки этого примера. Перед запуском убедитесь, что переменная окружения `LD_LIBRARY_PATH` включает текущий каталог, иначе можно просто написать

```
export LD_LIBRARY_PATH=.
```

для целей тестирования, а затем запустить исполняемый файл.

Листинг 15-12. `ex1-makefile`

```
main: main.o lib.so
      ld --dynamic-linker=/lib64/ld-linux-x86-64.so.2 main.o lib.so -o main
lib.so: lib.o
    ld -shared lib.o -o lib.so
lib.o:
   nasm -felf64 lib.asm -o lib.o
main.o: main.asm
     nasm -felf64 main.asm -o main.o
```

> **Вопрос 296** Проведите эксперимент. Уберите конструкцию `wrt ..plt` из вызова и перекомпилируйте всё. Затем используйте `objdump -D -Mintel-mnemonic` на итоговом главном исполняемом файле, чтобы проверить, задействован ли всё ещё PLT. Попробуйте его запустить.

15.7.2 О разных динамических компоновщиках

Динамический компоновщик — не высеченная в камне константа. Он закодирован как часть метаданных в ELF-файле и его можно посмотреть с помощью `ldd`. Во время компоновки можно управлять тем, какой динамический компоновщик будет выбран, например,

```
ld --dynamic-linker=/lib64/ld-linux-x86-64.so.2
```

Если вы его не укажете, `ld` выберет путь по умолчанию, который в вашем случае может вести к несуществующему файлу. Если динамический компоновщик не существует, попытка загрузить библиотеку приведёт к загадочному сообщению, лишённому какого-либо смысла. Предположим, вы собрали исполняемый файл `main`, использующий библиотеку `so_lib`, и `LD_LIBRARY_PATH` настроена правильно.

```
./main
bash: no such file or directory: ./main
> ldd ./main
linux-vdso.so.1 => (0x00007ffcf7f9f000)
so_lib.so => ./so_lib.so (0x00007f0e1cc0a000)
```

Проблема в том, что компоновка была выполнена без указания подходящего динамического компоновщика, и метаданные ELF не хранят к нему правильный путь. Перекомпоновка объектных файлов с подходящим путём к динамическому компоновщику должна

решить эту проблему. Например, в дистрибутиве Debian Linux, установленном на виртуальной машине, поставляемой с этой книгой, динамический компоновщик — `/lib64/ld-linux-x86-64.so.2`.

15.7.3 Доступ к внешней переменной

Для следующего примера мы разместим строку сообщения в главном исполняемом файле; за исключением этого, код останется тем же. Это позволит нам показать, как обращаться к внешней переменной. Главный файл показан в листинге 15-13, а исходный код библиотеки — в листинге 15-14.

Листинг 15-13. `ex2-main.asm`

```
extern _GLOBAL_OFFSET_TABLE_
global _start
extern sofun
global msg:data (msg.end - msg)
section .rodata
msg: db "SO function called -- message is stored in 'main'", 10
.end:
section .text
_start:
call sofun  wrt ..plt
mov rdi, 0
mov rax, 60
syscall
```

Листинг 15-14. `ex2-lib.asm`

```
extern _GLOBAL_OFFSET_TABLE_
global sofun:func
extern msg
section  .text
sofun:
mov rax, 1
mov rdi, 1
mov rsi, [rel msg wrt ..got]
mov rdx, 50
syscall
ret
```

Очень важно помечать объявление динамически разделяемых данных их размером. Размер задаётся как выражение, которое может включать метки и операции над ними, такие как вычитание. Без размера символ будет считаться глобальным для статического компоновщика (видимым другим модулям на этапе статической компоновки), но не будет экспортирован динамической библиотекой. Когда переменная объявлена как `global` с её размером и типом (`:data`), она будет жить в секции *.data* исполняемого файла, а не библиотеки! Из-за этого вам *всегда придётся обращаться к ней через GOT, даже в том же файле*. GOT, как мы знаем, хранит *адреса* переменных, глобальных для процесса. Так что, если мы хотим узнать адрес `msg`, нам нужно прочитать запись из GOT. Однако, поскольку динамическая библиотека позиционно-независима, нам также нужно адресовать её GOT относительно `rip`. Если мы хотим прочитать её значение, нам нужно дополнительное чтение памяти после получения её адреса из GOT. Если переменная объявлена в динамической библиотеке, а обращение к ней происходит из главного исполняемого файла, это следует делать точно такой же конструкцией: её адрес можно прочитать из `[rel varname wrt ..got]`. Если вам нужно сохранить адрес переменной GOT, используйте следующий квалификатор:

```
othervar: dq global_var wrt ..sym
```

За дополнительной информацией обратитесь к разделу 7.9.3 [27].

15.7.4 Полный пример на ассемблере

Листинг 15-15 и листинг 15-16 показывают полный пример со всеми обычными возможностями, нужными от динамической библиотеки.

Листинг 15-15. `ex3-main.asm`

```
extern _GLOBAL_OFFSET_TABLE_
extern fun1
global commonmsg:data commonmsg.end - commonmsg
global mainfun:function
global _start
section .rodata
commonmsg: db "fun2", 10, 0
.end:
mainfunmsg: db "mainfun", 10, 0
section .text
_start:
    call fun1 wrt ..plt
    mov rax, 60
    mov rdi, 0
    syscall
mainfun:
    mov rax, 1
    mov rdi, 1
    mov rsi, mainfunmsg
    mov rdx, 8
    syscall
    ret
```

Листинг 15-16. `ex3-lib.asm`

```
extern _GLOBAL_OFFSET_TABLE_
extern commonmsg
extern mainfun
global fun1:function
section .rodata
msg: db "fun1", 10
section .text
fun1:
    mov rax, 1
    mov rdi, 1
    lea rsi, [rel msg]
    mov rdx, 6
    syscall
    call fun2
    call mainfun wrt ..plt
    ret
fun2:
    mov rax, 1
    mov rdi, 1
    mov rsi, [rel commonmsg wrt ..got]
    mov rdx, 5
    syscall
    ret
```

15.7.5 Смешивание C и ассемблера

Отказ от ответственности: мы приведём пример, специфичный для конкретного компилятора и архитектуры, так что в вашем случае процесс может отличаться. Однако основные идеи останутся более или менее теми же. Что может усложнить смешивание кода на C и ассемблере, так это необходимость учитывать стандартную библиотеку C и правильно всё скомпоновать. Проще всего собрать объектные файлы отдельно с помощью GCC и NASM соответственно, а затем скомпоновать их тоже с помощью GCC. Помимо этого, бояться особо нечего. Листинг 15-17 и листинг 15-8 показывают пример вызова библиотеки на ассемблере из C.

Листинг 15-17. `ex4-main.c`

```
#include <stdio.h>
extern int sofun( void );
extern const char sostr[];
int main( void ) {
    printf( "%d\n", sofun() );
    puts( sostr );
    return 0;
}
```

В главном файле из динамической библиотеки вызывается внешняя функция `sofun`. Её результат выводится в `stdout` функцией `printf`. Затем строка, взятая из динамической библиотеки, выводится функцией `puts`. Заметьте, что глобальная строка — это глобальный символьный буфер, а не указатель!

Листинг 15-18. `ex4-lib.asm`

```
extern _GLOBAL_OFFSET_TABLE_
extern puts
global sostr:data (sostr.end - sostr)
global sofun:function
section .rodata
sostr: db "sostring", 10, 0
.end:
localstr: db "localstr", 10, 0
section .text
sofun:
    lea rdi, [rel localstr]
    call puts wrt ..plt
    mov rax, 42
    ret
```

В библиотеке определена функция `sofun`, а также глобальная строка `sostr`. `sofun` вызывает `puts`, функцию стандартной библиотеки C, с адресом `localstr` в качестве аргумента. Поскольку библиотека написана позиционно-независимым образом, адрес нужно вычислять как смещение от `rip`; отсюда использование команды `lea`. Эта функция всегда возвращает 42. Листинг 15-19 показывает соответствующий `Makefile`.

Листинг 15-19. `ex4-Makefile`

```
all: main
main: main.o lib.so
   gcc -o main main.o lib.so
lib.so: lib.o
   gcc -shared lib.o -o lib.so
lib.o: lib.asm
   nasm -felf64 lib.asm -o lib.o
main.o: main.asm
   gcc -ansi -c main.c -o main.o
clean:
   rm -rf *.o *.so main
```

15.8 Какие объекты компонуются?

Стандартная библиотека C обычно реализована как одна или несколько статических библиотек (которые, например, определяют `_start`) и динамическая библиотека, содержащая функции, которые мы привыкли вызывать. Структура библиотеки строго зависит от архитектуры, но мы проведём несколько экспериментов, чтобы её исследовать. Соответствующую документацию для нашего конкретного случая можно найти в [3]. Как нам узнать, с какими библиотеками GCC компонует исполняемый файл? Мы можем

провести эксперимент, используя GCC с аргументом `-v`. Ниже приведён список дополнительных аргументов, которые GCC неявно примет во время финальной компоновки согласно Makefile, показанному в листинге 15-19:

```
/usr/lib/gcc/x86_64-linux-gnu/4.9/collect2
-plugin
/usr/lib/gcc/x86_64-linux-gnu/4.9/liblto_plugin.so
-plugin-opt=/usr/lib/gcc/x86_64-linux-gnu/4.9/lto-wrapper
-plugin-opt=-fresolution=/tmp/ccqEOGnU.res
-plugin-opt=-pass-through=-lgcc
-plugin-opt=-pass-through=-lgcc_s
-plugin-opt=-pass-through=-lc
-plugin-opt=-pass-through=-lgcc
-plugin-opt=-pass-through=-lgcc_s
--sysroot=/
--build-id
--eh-frame-hdr
-m elf_x86_64
--hash-style=gnu
-dynamic-linker /lib64/ld-linux-x86-64.so.2
-o main
/usr/lib/gcc/x86_64-linux-gnu/4.9/../../../x86_64-linux-gnu/crt1.o
/usr/lib/gcc/x86_64-linux-gnu/4.9/../../../x86_64-linux-gnu/crti.o
/usr/lib/gcc/x86_64-linux-gnu/4.9/crtbegin.o
-L/usr/lib/gcc/x86_64-linux-gnu/4.9
-L/usr/lib/gcc/x86_64-linux-gnu/4.9/../../../x86_64-linux-gnu
-L/usr/lib/gcc/x86_64-linux-gnu/4.9/../../../../lib
-L/lib/x86_64-linux-gnu
-L/lib/../lib
-L/usr/lib/x86_64-linux-gnu
-L/usr/lib/../lib
-L/usr/lib/gcc/x86_64-linux-gnu/4.9/../../..
main.o
lib.so
-lgcc
--as-needed  -lgcc_s
--no-as-needed -lc
-lgcc
--as-needed  -lgcc_s
--no-as-needed /usr/lib/gcc/x86_64-linux-gnu/4.9/crtend.o
/usr/lib/gcc/x86_64-linux-gnu/4.9/../../../x86_64-linux-gnu/crtn.o
```

Сокращение `lto` соответствует «link-time optimizations» (оптимизации на этапе компоновки), что нас не интересует. Интересная часть состоит из скомпонованных дополнительных библиотек. Это:

- crti.o
- crtbegin.o
- crtend.o
- crtn.o
- crt1.o

Как мы знаем, ELF-файлы поддерживают несколько секций. Отдельная секция *.init* используется для хранения кода, выполняемого перед `main`, другая секция, *.fini*, используется для хранения кода, вызываемого при завершении программы. Содержимое этих секций разбито на несколько файлов. `crti` и `crto` содержат пролог и эпилог функции `__init` (и аналогично для функции `__fini`). Эти две функции вызываются соответственно до и после выполнения программы. `crtbegin` и `crtend` содержат другой служебный код, включённый в секции *.init* и *.fini*. Они присутствуют не всегда. Хотим повторить, что их порядок важен. `crt1.o` содержит функцию `_start`. Чтобы доказать наши утверждения, мы дизассемблируем файлы `crti.o`, `crtn.o` и `crt1.o` с помощью доброго старого

```
objdump  -D  -Mintel-mnemonic.
```

Листинги 15-20, 15-22 и 15-21 показывают очищенное дизассемблирование.

Листинг 15-20. `da_crti`

```
/usr/lib/x86_64-linux-gnu/crti.o:      file format elf64-x86-64
Disassembly of section .init:
0000000000000000 <_init>:
0:   sub    rsp, 0x8
4:   mov    rax, QWORD PTR [rip+0x0]         # b <_init+0xb>
b:   test   rax, rax
e:   je     15 <_init+0x15>
10: call   15 <_init+0x15>
Disassembly of section .fini:
0000000000000000 <_fini>:
0:   sub    rsp, 0x8
```

Листинг 15-21. `da_crtn`

```
/usr/lib/x86_64-linux-gnu/crtn.o:      file format elf64-x86-64
Disassembly of section .init:
0000000000000000 <.init>:
0: add    rsp,0x8
4: ret
Disassembly of section .fini:
0000000000000000 <.fini>:
0: add    rsp,0x8
4: ret
```

Листинг 15-22. `da_crt1`

```
/usr/lib/x86_64-linux-gnu/crt1.o:      file format elf64-x86-64
Disassembly of section .text:
0000000000000000 <_start>:
0:       xor    ebp,ebp
2:       mov    r9,rdx
5:       pop    rsi
6:       mov    rdx,rsp
9:       and    rsp,0xfffffffffffffff0
d:       push   rax
e:       push   rsp
f:       mov    r8,0x0
16:      mov    rcx,0x0
1d:      mov    rdi,0x0
24:      call   29 <_start+0x29>
29:      hlt
```

Как видим, эти функции-формы в итоге попадают в исполняемый файл. Чтобы увидеть полностью скомпонованный и перемещённый код, мы возьмём часть вывода `objdump -D -Mintel-mnemonic` для итогового файла, как показано в

Листинге 15-23.

Листинг 15-23. `dasm_init_fini`

```
Disassembly of section .init:
00000000004005d8 <_init>:
4005d8:  sub    rsp,0x8
4005dc:  mov    rax,QWORD PTR [rip+0x200a15]          # 600ff8 <_DYNAMIC+0x1e0>
4005e3:  test   rax,rax
4005e6:  je     4005ed <_init+0x15>
4005e8:  call   400650 <__libc_start_main@plt+0x10>
4005ed:  add    rsp,0x8
4005f1:  ret
Disassembly of section .text:
0000000000400660 <_start>:
400660:  xor    ebp,ebp
400662:  mov    r9,rdx
400665:  pop    rsi
400666:  mov    rdx,rsp
400669:  and    rsp,0xfffffffffffffff0
40066d:  push   rax
40066e:  push   rsp
40066f:  mov    r8,0x400800
400676:  mov    rcx,0x400790
40067d:  mov    rdi,0x400756
400684:  call   400640 <__libc_start_main@plt>
400689:  hlt
Disassembly of section .fini:
0000000000400804 <_fini>:
400804:  sub    rsp,0x8
400808:  add    rsp,0x8
40080c:  ret
```

15.9 Оптимизации

Что влияет на производительность при работе с динамической библиотекой? Прежде всего, никогда не забывайте опцию компилятора `-fPIC`.[33] Без неё будет перемещаться даже секция *.text*, что делает использование динамических библиотек намного менее привлекательным. Также критически важно отключить некоторые оптимизации, которые могут помешать динамическим библиотекам работать корректно. Как мы видели, когда функция объявлена `static` в динамической библиотеке и потому не экспортируется, её можно вызывать напрямую без накладных расходов PLT. Всегда используйте `static`, чтобы ограничить видимость одним файлом. Также можно управлять видимостью символов способом, зависящим от компилятора. Например, GCC распознаёт четыре

типа видимости (default, hidden, internal, protected), из которых нас интересуют только первые два. Видимостью всех символов сразу можно управлять с помощью переключателя компилятора `-fvisibility`, вот так:

```
> gcc -fvisibility=hidden ... # will hide all symbols from shared object
```

Уровень видимости «default» подразумевает, что все нестатические символы видны снаружи разделяемого объекта. С помощью директивы `__attribute__` можно тонко управлять видимостью для каждого символа отдельно. Листинг 15-24 показывает пример.

Листинг 15-24. `visibility_symbol.c`

```
int
__attribute__ (( visibility( "default" ) ))
func(int x) { return 42; }
```

Хорошая практика — скрыть все символы разделяемого объекта и явно пометить символы с видимостью default. Так вы полностью опишете интерфейс. Это особенно хорошо, потому что никакие другие символы не будут раскрыты, и вы будете свободны менять внутренности библиотеки, не нарушая бинарную совместимость какого бы то ни было рода. Перемещения данных могут немного замедлить работу. Каждый раз, когда переменная в *.data* хранит адрес другой переменной, её должен инициализировать динамический компоновщик, как только станет известен абсолютный адрес второй. По возможности избегайте таких ситуаций. Поскольку доступ к локальным символам обходит PLT, вы можете захотеть ссылаться внутри своего кода только на «скрытые» функции и делать публично доступные обёртки для функций, которые хотите экспортировать. PLT будет использоваться только для вызовов обёрток. Листинг 15-25 показывает пример.

Листинг 15-25. `so_adapter.c`

```
static int _function( int x ) { return x + 1; }
void otherfunction( ) {
    printf(" %d \n", _function( 41 ) );
}
int function( int x ) { return _function( x ); }
```

Чтобы устранить возможные накладные расходы функций-обёрток, существует техника написания псевдонимов символов (тоже зависящая от компилятора). GCC обрабатывает это с помощью атрибута `alias`. Листинг 15-26 показывает пример.

Листинг 15-26. `gcc_alias.c`

```
#include <stdio.h>
int global = 42;
extern int global_alias
__attribute__ ((alias ("global"), visibility ("hidden" ) ));
void fun( void ) {
    puts("1337\n");
}
extern void fun_alias( void )
__attribute__ ((alias ("fun"), visibility ("hidden" ) ));
int tester(void) {
    printf( "%d\n", global );
    printf( "%d\n", global_alias );
    fun();
    fun_alias();
    return 0;
}
```

Когда мы компилируем это с помощью `gcc -shared-O3 -fPIC` и дизассемблируем, мы видим код, показанный в листинге 15-27 (дизассемблирование функции `tester`).

Листинг 15-27. `gcc_aliased_gain.asm`

```
;  global -> rsi
787:   mov    rax,QWORD  PTR  [rip+0x20084a]      # 200fd8 <_DYNAMIC+0x1c8>
78e:   mov    eax,DWORD PTR [rax]
790:   mov    esi,eax
792:   lea    rdi,[rip+0x46]          # 7df <_fini+0xf>
799:   mov    eax,0x0
79e:   call   650 <printf@plt>
;  global_alias -> rsi
7a3:   mov    eax,DWORD PTR [rip+0x20088f]          # 201038 <global>
7a9:   mov    esi,eax
7ab:   lea    rdi,[rip+0x2d]        # 7df <_fini+0xf>
7b2:   mov    eax,0x0
7b7:   call   650 <printf@plt>
;  calling global `fun`
7bc:   call   640 <fun@plt>
;  calling aliased `fun` directly
7c1:   call   770 <fun>
```

`global` и `global_aliased` обрабатываются по-разному; для второго требуется на одно чтение памяти меньше. Вызов функции `fun` также обрабатывается эффективнее, минуя PLT и тем самым экономя лишний переход. Наконец, помните, что глобальные переменные, инициализированные нулём, всегда инициализируются быстрее. Однако мы настоятельно не рекомендуем использовать глобальные переменные. Больше информации об оптимизациях разделяемых объектов можно найти в [13].

> **Примечание** Обычный способ компоновки с библиотеками — использование ключа `-l`, например, `gcc -lhello`. Единственные два отличия от указания полного пути к файлу:

- `-lhello` будет искать библиотеку с именем `libhello.a` (то есть с префиксом `lib` и

> **Примечание.** расширением `.a`).

- Библиотека ищется в стандартном списке каталогов. Она также ищется в пользовательских

> **Примечание.** каталогах, которые можно указать с помощью опции `-L`. Например, чтобы включить каталог `/usr/libcustom` и текущий каталог, можно написать:

```
> gcc -lhello -L. -L/usr/libcustom main.c
```

> **Примечание.** Помните, порядок, в котором вы указываете библиотеки, важен.

15.10 Модели кода

Модели кода — тема, о которой редко говорят. [24] можно считать справочником по этому вопросу, и мы обсудим модели кода в этом разделе. Отправная точка для обсуждения — тот факт, что адресация относительно `rip` ограничена. [15] поясняет, что смещение должно быть непосредственным значением максимум в 32 бита. Это оставляет нам смещения ± 2 ГБ. Напрямую использовать 64-битные смещения расточительно, поскольку большая часть кода никогда не задействует лишние биты; однако такие смещения кодируются напрямую в самих инструкциях, из-за чего код занимает больше места, что плохо для кэша инструкций. Размер адресного пространства намного больше 32 бит, так что же делать, когда 32 бит не хватает? Модель кода — это соглашение, которого придерживаются и программист, и компилятор; она описывает ограничения на программу, которая будет использовать компилируемый в данный момент объектный файл.

От неё зависит генерация кода. Короче говоря, когда программа относительно небольшая, нет вреда в использовании 32-битных смещений. Однако когда она может быть достаточно большой, следует использовать более медленные 64-битные смещения, обрабатываемые несколькими инструкциями. 32-битным смещениям соответствует *малая модель кода*; 64-битным смещениям соответствует *большая модель кода*. Есть также своего рода компромисс, называемый *средней моделью кода*. Все эти модели трактуются по-разному в контексте позиционно-зависимого и позиционно-независимого кода, так что мы рассмотрим все шесть возможных комбинаций. Могут существовать и другие модели кода, такие как модель кода ядра, но мы оставим их за рамками этого тома. Если вы делаете собственную операционную систему, можете придумать свою на здоровье. Соответствующая опция GCC — `-mcmodel`, например, `-mcmodel=large`. Модель по умолчанию — малая.[34] Руководство GCC говорит следующее об опции `-mcmodel`[35]:

```
-mcmodel=small
      Generate code for the small code model: the program and its symbols must be linked in
the lower 2 GB of the address space. Pointers are 64 bits. Programs can be statically or
dynamically linked. This is the default code model.
-mcmodel=kernel
      Generate code for the kernel code model. The kernel runs in the negative 2 GB of the
address space. This model has to be used for Linux kernel code.
-mcmodel=medium
      Generate code for the medium model: the program is linked in the lower 2 GB of the
address space. Small symbols are also placed there. Symbols with sizes larger than -mlarge-
data-threshold are put into large data or BSS sections and can be located above 2GB.
Programs can be statically or dynamically linked.
-mcmodel=large
      Generate code for the large model. This model makes no assumptions about addresses and
sizes of sections.
```

Чтобы проиллюстрировать различия в скомпилированном коде при использовании разных моделей кода, мы возьмём простой пример, показанный в листинге 15-28.

Листинг 15-28. `cm-example.c`

```
char glob_small[100] = {1};
char glob_big[10000000] = {1};
static char loc_small[100] = {1};
static char loc_big[10000000] = {1};
int global_f(void) { return 42; }
static int local_f(void) { return 42; }
int main(void) {
    glob_small[0] = 42;
    glob_big[0] = 42;
    loc_small[0] = 42;
```

```
    loc_big[0] = 42;
    global_f();
    local_f();
    return 0;
}
```

Мы используем следующую строку для его компиляции:

```
gcc -O0 -g cm-example.c
```

Флаг `-g` добавляет отладочную информацию, такую как секция *.line*, описывающую соответствие между ассемблерными инструкциями и строками исходного кода. В этом примере есть массивы побольше и поменьше. Это имеет значение только для средней модели кода, поэтому мы опустим обращения к большому массиву в остальных листингах дизассемблирования.

15.10.1 Малая модель кода (без PIC)

В малой модели кода программа ограничена по размеру. Все объекты, компонуемые вместе, должны находиться в пределах 4 ГБ друг от друга. Компоновка может быть выполнена как статически, так и динамически. Поскольку это модель кода по умолчанию, мы не увидим здесь ничего интересного. Передав `objdump` ключ `-S`, мы перемежим ассемблерный код со строками исходного кода на C (если соответствующий файл был скомпилирован с флагом `-g`). Полная последовательность команд будет выглядеть так:

```
gcc -O0 -g cm-example.c -o example
objdump -D -Mintel-mnemonic -S example
```

Листинг 15-29 показывает скомпилированный ассемблер.

Листинг 15-29. `mc-small`

```
;     glob_small[0] = 42;
4004f0:  c6 05 49 0b 20 00 2a     mov     BYTE PTR [rip+0x200b49],0x2a
;     loc_small[0] = 42;
4004fe:   c6 05 3b a2 b8 00 2a    mov     BYTE PTR [rip+0xb8a23b],0x2a
;     global_f();
40050c:   e8 c5 ff ff ff          call    4004d6 <global_f>
;     local_f();
400511:   e8 cb ff ff ff          call    4004e1 <local_f>
```

Второй столбец показывает нам шестнадцатеричные коды байт, соответствующих каждой инструкции. Обращения к массиву выполняются явно относительно `rip`, а вызовы принимают смещения (тоже неявно относительные к `rip`). Мы видим, что размер инструкций обращения к данным — 7 байт, из которых 1 байт — значение (`0x2a`), а 4 байта кодируют смещение относительно `rip`. Это иллюстрирует основную идею малой модели кода: адресацию относительно `rip`.

15.10.2 Большая модель кода (без PIC)

Теперь скомпилируем тот же код, используя большую модель кода (`-mcmodel=large`).

```
;     glob_small[0] = 42;
   4004f0:   48 b8 40 10 60 00 00    mov     rax,0x601040
   4004f7:   00 00 00
   4004fa:   c6 00 2a                mov     BYTE PTR [rax],0x2a
;     loc_small[0] = 42;
   40050a:   48 b8 40 a7 f8 00 00    mov     rax,0xf8a740
   400511:   00 00 00
   400514:   c6 00 2a                mov     BYTE PTR [rax],0x2a
;     global_f();
   400524:   48 b8 d6 04 40 00 00    mov     rax,0x4004d6
   40052b:   00 00 00
   40052e:   ff d0                   call    rax
;     local_f();
   400530:   48 b8 e1 04 40 00 00    mov     rax,0x4004e1
   400537:   00 00 00
   40053a:   ff d0                   call    rax
```

И обращения к данным, и вызовы выполняются единообразно. Мы всегда начинаем с перемещения непосредственного значения в один из регистров общего назначения, а затем обращаемся к памяти, используя адрес, хранящийся в этом регистре.[36] Ценой более объёмного ассемблерного кода (и, вероятно, чуть более медленного) мы выбираем самый безопасный из возможных путей, позволяющий обратиться к чему угодно в любой части 64-битного виртуального адресного пространства.

15.10.3 Средняя модель кода (без PIC)

В средней модели кода массивы размером больше, чем указано параметром компилятора `-mlarge-data-threshold`, размещаются в специальных секциях *.ldata* и *.lbss*. Эти секции можно размещать выше отметки 2 ГБ. По сути, это малая модель кода, за исключением больших блоков данных, размещаемых отдельно. С точки зрения производительности это лучше, чем обращаться ко всему через 64-битные указатели, из-за локальности. Дизассемблирование для исходников, скомпилированных с `-mcmodel=medium`, выглядит так:

```
  glob_small[0] = 42;
400530:   c6 05 09 0b 20 00 2a     mov      BYTE PTR [rip+0x200b09],0x2a
  glob_big[0] = 42;
400537:   48 b8 40 11 a0 00 00     movabs   rax,0xa01140
40053e:   00 00 00
400541:   c6 00 2a                 mov      BYTE PTR [rax],0x2a
  loc_small[0]  =  42;
400544:   c6 05 75 0b 20 00 2a     mov      BYTE PTR [rip+0x200b75],0x2a
```

```
  loc_big[0] = 42;
40054b:   48 b8 c0 a7 38 01 00     movabs   rax,0x138a7c0
400552:   00 00 00
400555:   c6 00 2a                 mov      BYTE PTR [rax],0x2a
  global_f();
400558:   e8 b9 ff ff ff           call     400516 <global_f>
  local_f();
40055d:   e8 bf ff ff ff           call     400521 <local_f>
```

Как видим, сгенерированный код использует большую модель для обращения к большим массивам и малую для остальных обращений. Это довольно умно и может вас выручить, если вам нужно работать лишь с одним большим блоком статически выделенных данных.

15.10.4 Малая PIC-модель кода

Теперь мы исследуем позиционно-независимые аналоги этих трёх моделей кода. Как и раньше, малая модель нас не удивит, потому что до сих пор мы работали только с малой моделью кода. Для удобства мы приводим пример кода, скомпилированного с `gcc -g -O0 -mcmodel=small -fpic`.

```
  glob_small[0] = 42;
4004f0:   48 8d 05 49 0b 20 00      lea      rax,[rip+0x200b49]
  # 601040 <glob_small>
4004f7:   c6 00 2a                  mov      BYTE PTR [rax],0x2a
  glob_big[0] = 42;
4004fa:   48 8d 05 bf 0b 20 00      lea      rax,[rip+0x200bbf]
  # 6010c0 <glob_big>
400501:   c6 00 2a                  mov      BYTE PTR [rax],0x2a
  loc_small[0] = 42;
400504:   c6 05 35 a2 b8 00 2a      mov      BYTE PTR [rip+0xb8a235],0x2a
  # f8a740 <loc_small>
  loc_big[0] = 42;
40050b:   c6 05 ae a2 b8 00 2a      mov      BYTE PTR [rip+0xb8a2ae],0x2a
  # f8a7c0 <loc_big>
  global_f();
400512:   e8 bf ff ff ff            call     4004d6 <global_f>
  local_f();
400517:   e8 c5 ff ff ff            call     4004e1 <local_f>
```

К массивам `static` обращаются легко, относительно `rip`, как и ожидалось. К глобально видимым массивам обращаются через GOT, что подразумевает дополнительное чтение из самой таблицы, чтобы получить её адрес.

15.10.5 Большая PIC-модель кода

Интересные вещи начинают происходить при использовании большой модели кода с позиционно-независимым кодом. Теперь мы не можем использовать адресацию относительно `rip`, чтобы добраться до GOT, потому что она может находиться дальше 2 ГБ в адресном пространстве! Из-за этого нам нужно выделить регистр для хранения

её адреса (`rbx` в нашем случае).

```
# Standard prologue
400594:   55                            push    rbp
400595:   48 89 e5                      mov     rbp,rsp
# What is that?
400598:   41 57                         push    r15
40059a:   53                            push    rbx
40059b:   48 8d 1d f9 ff ff ff          lea     rbx,[rip+0xfffffffffffffff9]
# 40059b <main+0x7>
4005a2:   49 bb 65 0a 20 00 00          movabs  r11,0x200a65
4005a9:   00 00 00
4005ac:   4c 01 db                      add     rbx,r11
# Accessing global symbols
  glob_small[0] = 42;
4005af:   48 b8 e8 ff ff ff ff          movabs  rax,0xffffffffffffffe8
4005b6:   ff ff ff
4005b9:   48 8b 04 03                   mov     rax,QWORD PTR [rbx+rax*1]
4005bd:   c6 00 2a                      mov     BYTE PTR [rax],0x2a
# Accessing local symbols
  loc_small[0] = 42;
4005d1:   48 b8 40 97 98 00 00          movabs  rax,0x989740
4005d8:   00 00 00
4005db:   c6 04 03 2a                   mov     BYTE  PTR  [rbx+rax*1],0x2a
# Calling global function
  global_f();
4005ed:   49 89 df                      mov     r15,rbx
4005f0:   48 b8 56 f5 df ff ff          movabs  rax,0xffffffffffdff556
4005f7:   ff ff ff
4005fa:   48 01 d8                      add     rax,rbx
4005fd:   ff d0                         call    rax
# Calling local function
  local_f();
4005ff:   48 b8 75 f5 df ff ff          movabs  rax,0xffffffffffdff575
400606:   ff ff ff
400609:   48 8d 04 03                   lea     rax,[rbx+rax*1]
40060d:   ff d0                         call    rax
    return 0;
  40060f:   b8 00 00 00 00               mov      eax,0x0
}
400614:   5b                            pop     rbx
400615:   41 5f                         pop     r15
400617:   5d                            pop     rbp
400618:   c3                            ret
```

Этот пример нужно изучить внимательно. Сначала разберём необычный код в прологе функции.

```
400598:     41 57                   push   r15
40059a:     53                      push   rbx
40059b:     48 8d 1d f9 ff ff ff    lea    rbx,[rip+0xfffffffffffffff9]
# 40059b <main+0x7>
4005a2:   49 bb 65 0a 20 00 00      movabs  r11,0x200a65
4005a9:   00 00 00
4005ac:   4c 01 db                  add    rbx,r11
```

Мы используем `rbx` и `r15`, потому что они *сохраняются вызываемой функцией*. Они используются здесь, чтобы собрать адрес GOT из следующих двух компонентов:

- Адрес текущей инструкции, вычисленный в `lea rbx,[rip+0xfffffffffffffff9]`. Операнд равен -6, тогда как сама инструкция занимает 6 байт. Когда она выполняется, значение `rip` указывает на адрес, следующий сразу за инструкцией.
- Затем к `rbx` прибавляется число `0x200a65`. Это делается через другой регистр, потому что прибавление непосредственного 64-битного операнда не поддерживается инструкцией `add` (проверьте описание инструкции в [15]!).
- Это число — смещение GOT относительно адреса `lea rbx,[rip+0xfffffffffffffff9]`, которое, как мы знаем, всегда известно на момент компоновки в позиционно-независимом коде.[37]

ABI считает, что `r15` должен всегда хранить адрес GOT. `rbx` тоже используется GCC для удобства. Абсолютный адрес GOT неизвестен на момент компоновки, поскольку код написан позиционно-независимым. Теперь об обращениях к данным: к глобальному символу обращаются через GOT так же, как и в не-PIC коде; однако, поскольку адрес GOT хранится в `rbx`, нам приходится вычислять адрес записи большим числом инструкций.

```
# Accessing global symbols
  glob_small[0] = 42;
4005af:   48 b8 e8 ff ff ff ff      movabs   rax,0xfffffffffffffe8
4005b6:   ff  ff  ff
4005b9:   48 8b 04 03               mov      rax,QWORD PTR [rbx+rax*1]
4005bd:   c6 00 2a                  mov      BYTE PTR [rax],0x2a
```

Запись находится по отрицательному смещению `-24` относительно значения `rbx (r15)`. Это смещение может быть произвольной длины, так что нам нужно хранить его в регистре, учитывая случаи, когда оно не помещается в 32 бита. Затем мы загружаем запись GOT в `rax` и используем этот адрес для наших целей (в данном случае мы сохраняем значение в начало массива).

К переменным, не видимым как другие объекты, тоже обращаются через GOT. Однако мы не читаем их адреса из GOT. Вместо этого мы используем значение `rbx` как базу (поскольку оно указывает куда-то в сегмент данных). У каждой глобальной переменной фиксированное смещение от этой базы, так что мы можем просто взять это смещение и использовать режим базово-индексной адресации.

```
# Accessing local symbols
  loc_small[0] = 42;
4005d1:   48 b8 40 97 98 00 00     movabs     rax,0x989740
4005d8:   00 00 00
4005db:   c6 04 03 2a              mov        BYTE PTR [rbx+rax*1],0x2a
```

Это, очевидно, быстрее, так что всегда, когда можете, предпочитайте ограничивать видимость символов, как объяснено в разделе 15.9. Локальные функции вызываются так же. Их адрес вычисляется относительно GOT и сохраняется в регистре. Мы не можем просто использовать команду `call`, потому что её непосредственный операнд ограничен 32 битами (в её описании, приведённом в [15], есть только типы операндов `rel16` и `rel32`, но нет `rel64`).

```
# Calling local  function
  local_f();
4005ff:   48 b8 75 f5 df ff ff     movabs     rax,0xffffffffffdff575
400606:   ff ff ff
400609:   48 8d 04 03              lea        rax,[rbx+rax*1]
40060d:   ff d0                    call       rax
```

Вызов глобальных функций выполняется более традиционным способом. Используется её запись PLT, чей адрес тоже вычисляется как фиксированное смещение до известной позиции в GOT.

```
# Calling global function
  global_f();
4005ed:   49 89 df                 mov     r15,rbx
4005f0:   48 b8 56 f5 df ff ff     movabs  rax,0xffffffffffdff556
4005f7:   ff ff ff
4005fa:   48 01 d8                 add     rax,rbx
4005fd:   ff d0                    call    rax
```

15.10.6 Средняя PIC-модель кода

Средняя модель кода, как и в не-PIC коде, — смесь большой и малой моделей кода. Можно считать её малой PIC-моделью кода с добавлением больших массивов, размещённых отдельно.

```
int main(void) {
  40057a:   55                      push   rbp
  40057b:   48 89 e5                mov    rbp,rsp
# Different from small model: we save GOT address locally.
  40057e:   48 8d 15 7b 0a 20 00    lea    rdx,[rip+0x200a7b]
    glob_small[0] = 42;
  400585:   48 8d 05 b4 0a 20 00    lea    rax,[rip+0x200ab4]
  40058c:   c6 00 2a                mov    BYTE PTR [rax],0x2a
    glob_big[0] = 42;
  40058f:   48 8b 05 62 0a 20 00    mov    rax,QWORD PTR [rip+0x200a62]
  400596:   c6 00 2a                mov    BYTE PTR [rax],0x2a
    loc_small[0] = 42;
  400599:   c6 05 20 0b 20 00 2a    mov    BYTE PTR [rip+0x200b20],0x2a
    loc_big[0] = 42;
  4005a0:   48 b8 c0 97 d8 00 00    movabs rax,0xd897c0
  4005a7:   00 00 00
  4005aa:   c6 04 02 2a             mov    BYTE PTR [rdx+rax*1],0x2a
    global_f();
  4005ae:   e8 a3 ff ff ff          call   400556 <global_f>
    local_f();
  4005b3:   e8 b0 ff ff ff          call   400568 <local_f>
    return 0;
  4005b8:   b8 00 00 00 00          mov    eax,0x0
}
    4005bd: 5d                      pop    rbp
  4005be:   c3                      ret
```

Адрес GOT тоже досягаем адресацией относительно `rip`, так что его адрес загружается одной инструкцией.

```
40057e:   48 8d 15 7b 0a 20 00     lea    rdx,[rip+0x200a7b]
```

Так что не всегда нужно выделять для него регистр, поскольку этот адрес используется не везде. Считается, что ссылки на код достижимы 32-битными смещениями относительно `rip`. Так что вызов любых функций тривиален.

```
    global_f();
  4005ae:   e8 a3 ff ff ff      call     400556 <global_f>
    local_f();
  4005b3:  e8 b0 ff ff ff       call     400568 <local_f>
```

Что касается обращений к данным, обращения к глобальным переменным выполняются единообразно независимо от размера. GOT задействуется в любом случае, и она содержит 64-битные адреса глобальных переменных, так что у нас есть возможность бесплатно адресовать что угодно.

```
  glob_small[0] = 42;
400585:   48 8d 05 b4 0a 20 00     lea     rax,[rip+0x200ab4]
40058c:   c6 00 2a                 mov     BYTE PTR [rax],0x2a
  glob_big[0] = 42;
40058f:   48 8b 05 62 0a 20 00     mov     rax,QWORD PTR [rip+0x200a62]
400596:   c6 00 2a                 mov     BYTE PTR [rax],0x2a
```

Локальные переменные, однако, отличаются. К маленьким массивам можно обращаться относительно `rip`.

```
  loc_small[0] = 42;
400599:   c6 05 20 0b 20 00 2a     mov     BYTE PTR [rip+0x200b20],0x2a
```

Локальные большие массивы находятся относительно начальных адресов GOT, как в большой модели.

```
  loc_big[0] = 42;
4005a0:   48 b8 c0 97 d8 00 00     movabs     rax,0xd897c0
4005a7:   00 00 00
4005aa:   c6 04 02 2a              mov        BYTE PTR [rdx+rax*1],0x2a
```

15.11 Итоги

В этой главе мы получили знания, нужные для понимания механизма загрузки и использования динамических библиотек. Мы написали библиотеку на языке ассемблера и на C и успешно скомпоновали её с исполняемым файлом. Для дальнейшего чтения мы отсылаем вас прежде всего к классической статье [13] и к описанию ABI [24]. В следующей главе мы поговорим об оптимизациях компилятора и их влиянии на производительность, а также о специализированных расширениях системы команд (SSE/AVX), нацеленных на ускорение определённых видов вычислений.

> **Вопрос 297** В чём разница между статической и динамической компоновкой?

> **Вопрос 298** Что делает динамический компоновщик?

> **Вопрос 299** Можем ли мы разрешить все зависимости на момент компоновки? С какой системой нам нужно работать, чтобы это было возможно?

> **Вопрос 300** Всегда ли нужно перемещать секцию **.data**?

> **Вопрос 301** Всегда ли нужно перемещать секцию **.text**?

> **Вопрос 302** Что такое PIC?

> **Вопрос 303** Можем ли мы разделять секцию **.text** между процессами, когда она перемещается?

> **Вопрос 304** Можем ли мы разделять секцию **.data** между процессами, когда она перемещается?

> **Вопрос 305** Можем ли мы разделять секцию **.data**, когда она перемещается?

> **Вопрос 306** Почему мы компилируем динамические библиотеки с флагом `-fPIC` ?

Вопрос 307 Напишите с нуля простую динамическую библиотеку на C и продемонстрируйте вызов функции из неё.

Вопрос 308 Для чего используется `ldd` ?

Вопрос 309 Где ищутся библиотеки?

Вопрос 310 Для чего нужна переменная окружения `LD_LIBRARY_PATH` ?

Вопрос 311 Что такое GOT? Зачем она нужна?

Вопрос 312 Что делает использование GOT эффективным?

Вопрос 313 Как получается, что позиционно-независимый код может напрямую адресовать GOT, но не может напрямую адресовать глобальные переменные?

Вопрос 314 Уникальна ли GOT для каждого процесса?

Вопрос 315 Что такое PLT?

Вопрос 316 Почему мы не используем GOT для вызова функций из разных объектов (или используем)?

Вопрос 317 На что указывает начальная запись GOT для функции?

Вопрос 318 Как предзагрузить библиотеку и для чего это можно использовать?

Вопрос 319 Как в ассемблере адресуется символ, если он определён в исполняемом файле и используется там же?

Вопрос 320 Как в ассемблере адресуется символ, если он определён в библиотеке и используется там же?

Вопрос 321 Как в ассемблере адресуется символ, если он определён в исполняемом файле и используется отовсюду?

Вопрос 322 Как в ассемблере адресуется символ, если он определён в библиотеке и используется отовсюду?

Вопрос 323 Как управлять видимостью символа в динамической библиотеке? Как сделать его приватным для библиотеки, но доступным из любого места в ней?

Вопрос 324 Зачем иногда пишут функции-обёртки для тех, что используются в библиотеке?

Вопрос 325 Как скомпоноваться с библиотекой, хранящейся в `libdir`?

Вопрос 326 Что такое модель кода и почему нас должны заботить модели кода?

Вопрос 327 Какие ограничения накладывает малая модель кода?

Вопрос 328 Какие накладные расходы несёт большая модель кода?

Вопрос 329 В чём компромисс между большой и малой моделями кода?

Вопрос 330 Когда средняя модель наиболее полезна?

Вопрос 331 Чем отличаются большие модели кода для PIC и не-PIC кода?

Вопрос 332 Чем отличаются средние модели кода для PIC и не-PIC кода?

Производительность

В этой главе мы изучим, как писать более быстрый код. Для этого мы рассмотрим инструкции SSE (Streaming SIMD Extensions), изучим оптимизации компилятора и работу аппаратного кэша. Заметьте, что эта глава — лишь введение в тему и не сделает вас экспертом по оптимизации. Не существует серебряной пули, магическим образом ускоряющей всё. Оборудование стало настолько сложным, что даже обоснованная догадка о том, какой код замедляет выполнение программы, может оказаться неверной. Всегда следует проводить тестирование и профилирование, а производительность следует измерять воспроизводимым образом. Это значит, что всё об окружении должно быть описано настолько подробно, чтобы кто угодно мог воспроизвести условия эксперимента и получить схожие результаты.

16.1 Оптимизации

В этом разделе мы хотим обсудить самые важные оптимизации, происходящие во время процесса трансляции. Их понимание критически важно для того, чтобы понимать, как писать качественный код. Почему? Распространённый тип принятия решений в программировании — баланс между читаемостью кода и производительностью. Знание оптимизаций необходимо, чтобы принимать хорошие решения. Иначе, выбирая между двумя версиями кода, мы можем выбрать менее читаемую только потому, что она «выглядит» так, будто выполняет меньше действий. На самом же деле обе версии будут оптимизированы в точности в одну и ту же последовательность ассемблерных инструкций. В этом случае мы просто сделали код менее читаемым безо всякой выгоды.

> **Примечание** В листингах, приведённых в этом разделе, мы будем часто использовать директиву GCC `__attribute__ ((noinline))`. Применённая к определению функции, она подавляет встраивание для этой функции. Примеры функций часто маленькие, что подталкивает компиляторы встраивать их, а мы этого не хотим, чтобы лучше показать различные эффекты оптимизаций. В качестве альтернативы мы могли бы скомпилировать примеры с опцией `-fno-inline`.

16.1.1 Миф о быстрых языках

Существует распространённое заблуждение, что язык определяет скорость выполнения программы. Это не так. Лучшие и более полезные тесты производительности обычно узкоспециализированы. Они измеряют производительность в очень конкретных случаях. Это удерживает нас от смелых обобщений. Так что при высказывании утверждений о производительности разумно давать максимально подробное описание сценария и результатов тестов. Описания должно хватать, чтобы построить похожую систему и запустить похожие тесты, получив сопоставимые результаты.

Бывают случаи, когда программу, написанную на C, обгоняет другая программа, выполняющая похожие действия, но написанная, скажем, на Java. Это никак не связано с самим языком. Например, у типичной реализации `malloc` есть особое свойство: трудно предсказать время её выполнения. В общем случае оно зависит от текущего состояния кучи: сколько существует блоков, насколько куча фрагментирована и т. д. В любом случае оно, скорее всего, больше, чем выделение памяти на стеке. В типичной реализации виртуальной машины Java, однако, выделение памяти быстрое. Это происходит потому, что у Java более простая структура кучи. С некоторыми упрощениями, это просто область памяти и указатель внутри неё, отделяющий занятую область от свободной. Выделение памяти означает продвижение этого указателя дальше в свободную часть, что быстро. Однако у этого есть своя цена: чтобы избавиться от блоков памяти, которые больше не нужны, выполняется сборка мусора, способная приостановить программу на неизвестный период времени. Представим ситуацию, в которой сборка мусора никогда не происходит, например, программа выделяет память, выполняет вычисления и завершается, уничтожая всё адресное пространство без вызова сборщика мусора. В этом случае возможно, что программа на Java работает быстрее просто из-за накладных расходов на аккуратное выделение памяти, которые накладывает `malloc`. Однако если мы используем собственный аллокатор памяти, подходящий под наши конкретные нужды для конкретной задачи, мы можем провернуть тот же трюк в C, кардинально изменив результат. Кроме того, поскольку Java обычно интерпретируется и компилируется во время выполнения, у виртуальной машины есть доступ к оптимизациям времени выполнения, основанным на том, как именно программа выполняется. Например, методы, часто выполняемые один за другим, можно разместить рядом друг с другом в памяти, так что они попадут в кэш вместе. Для этого нужно собрать определённую информацию о трассе выполнения программы, что возможно только во время выполнения. Что действительно отличает C от других языков, так это очень прозрачная модель издержек. Что бы вы ни писали, легко представить, какие ассемблерные инструкции будут сгенерированы. В противоположность этому, языки, предназначенные в первую очередь для работы внутри среды выполнения (Java, C#), или предоставляющие множество дополнительных абстракций, такие как C++ с его механизмом виртуального наследования, предсказать труднее. Единственные две настоящие абстракции, которые предоставляет C, — это структуры/объединения и функции. При наивном переводе в машинные инструкции программа на C работает очень медленно. Ей не сравниться с кодом, сгенерированным хорошим оптимизирующим компилятором. Обычно у программиста нет больше знаний о низкоуровневых деталях архитектуры, чем у компилятора, а эти знания очень нужны для выполнения низкоуровневых оптимизаций, так что тягаться с компилятором он не сможет. Впрочем, иногда для конкретной платформы и компилятора можно изменить программу, обычно снижая её читаемость и сопровождаемость, но так, чтобы ускорить код. И снова, тесты производительности обязательны для всех.

16.1.2 Общие советы

При программировании нам обычно не стоит сразу же заботиться об оптимизациях. Преждевременная оптимизация — зло по многим причинам.

- Большинство программ написаны так, что лишь часть их кода выполняется

многократно. Именно этот код определяет, насколько быстро будет выполняться программа, и он может замедлять всё остальное. Ускорение остальных частей в таких обстоятельствах даст мало эффекта или не даст никакого.

Находить такие части кода лучше всего с помощью *профилировщика* — служебной программы, измеряющей, как часто и как долго выполняются разные части кода.

- Оптимизация кода вручную практически всегда делает его менее читаемым и труднее

поддерживаемым.

- Современные компиляторы осведомлены о распространённых шаблонах в коде на языках

высокого уровня. Такие шаблоны хорошо оптимизируются, потому что авторы компиляторов вложили в это много труда, поскольку это того стоит. Самая важная часть оптимизации часто — выбор правильного алгоритма. Низкоуровневые оптимизации на уровне ассемблера редко бывают настолько же полезны. Например, обращение к элементам связного списка по индексу медленное, потому что приходится проходить его с начала, перескакивая от узла к узлу. Массивы выгоднее, когда логика программы требует обращения к элементам по индексу. Однако вставка в связный список проще, чем в массив, потому что для вставки элемента в i-ю позицию массива сначала приходится сдвигать все следующие элементы (а то и вовсе сначала перевыделять под него память и копировать всё). Простой, чистый код часто оказывается и самым эффективным. Затем, если производительность неудовлетворительна, нам нужно найти код, выполняющийся чаще всего, с помощью профилировщика и попробовать оптимизировать его вручную. Проверьте на наличие дублирующихся вычислений и попробуйте запоминать и повторно использовать результаты вычислений. Изучите ассемблерные листинги и проверьте, приносит ли пользу принудительное встраивание некоторых из используемых функций. Общие соображения об оборудовании, такие как локальность и использование кэша, стоит учитывать именно на этом этапе. Мы поговорим о них в разделе 16.2. Затем стоит рассмотреть оптимизации компилятора. Базовые из них мы изучим далее в этом разделе. Включение или выключение конкретных оптимизаций для отдельного файла или области кода может положительно сказаться на производительности. По умолчанию они обычно все включены при компиляции с флагом `-O3`. И только потом идут более низкоуровневые оптимизации: ручное добавление инструкций SSE или AVX (Advanced Vector Extensions), встраивание ассемблерного кода, запись данных в обход аппаратного кэша,

предварительная загрузка данных в кэш перед использованием и т. д. Оптимизациями компилятора грубо управляют с помощью флагов компилятора `-O0`, `-O1`, `-O2`, `-O3`, `-Os` (оптимизация использования пространства для получения максимально маленького исполняемого файла). Индекс рядом с `-O` растёт по мере роста набора включённых оптимизаций. Конкретные оптимизации можно включать и выключать. У каждого типа оптимизации для этого есть две связанные опции компилятора, например, `-fforward-propagate` и `-fno-forward-propagate`.

16.1.3 Опустить указатель кадра стека

Соответствующие опции GCC: `-fomit-frame-pointer` Иногда нам на самом деле не нужно сохранять старое значение `rbp` и инициализировать его новым базовым значением. Это происходит, когда

- Нет локальных переменных.
- Локальные переменные помещаются в красную зону, И функция не вызывает другие функции.

Однако есть и обратная сторона: это значит, что во время выполнения хранится меньше информации о состоянии программы. У нас возникнут проблемы с разворачиванием стека вызовов и получением значений локальных переменных, потому что нам не хватает информации о том, где начинается кадр. Это особенно проблематично в ситуациях, когда программа обрушилась и нужно проанализировать дамп её состояния. Такие дампы часто сильно оптимизированы и лишены отладочной информации. С точки зрения производительности эффект этих оптимизаций часто незначителен [26]. Код, показанный в листинге 16-1, демонстрирует, как развернуть стек и вывести адреса указателей кадров для всех функций, запущенных на момент вызова `unwind`. Скомпилируйте его с `-O0`, чтобы предотвратить оптимизации.

Листинг 16-1. `stack_unwind.c`

```
void unwind();
void f( int count ) {
    if ( count ) f( count-1 ); else unwind();
}
int main(void) {
    f( 10 ); return 0;
}
```

Листинг 16-2 показывает пример.

Листинг 16-2. `stack_unwind.asm`

```
extern printf
global unwind
section .rodata
format : db "%x ", 10, 0
section .code
unwind:
push rbx
; while (rbx != 0) {
    ;     print rbx; rbx = [rbx];
    ; }
    mov rbx, rbp
    .loop:
    test rbx, rbx
    jz .end
    mov rdi, format
    mov rsi, rbx
    call printf
    mov rbx, [rbx]
    jmp .loop
    .end:
    pop rbx
    ret
```

Как это использовать? Попробуйте это как последнее средство для улучшения производительности кода, задействующего огромное количество не поддающихся встраиванию вызовов функций.

16.1.4 Хвостовая рекурсия

```
Related GCC options: -fomit-frame-pointer -foptimize-sibling-calls
```

Рассмотрим функцию, показанную в листинге 16-3.

Листинг 16-3. `factorial_tailrec.c`

```
 __attribute__ (( noinline ))
     int factorial( int acc, int arg ) {
         if ( arg == 0 ) return acc;
         return factorial( acc * arg, arg-1 );
     }
int main(int argc, char** argv) { return factorial(1, argc); }
```

Она вызывает саму себя рекурсивно, но этот вызов особенный. Как только вызов завершается, функция немедленно возвращается. Мы говорим, что функция *хвостово рекурсивна*, если функция либо

- Возвращает значение, не задействующее рекурсивный вызов, например, `return 4`;.
- Запускает саму себя рекурсивно с другими аргументами и немедленно возвращает результат, не выполняя над ним дальнейших вычислений. Например, `return factorial`

```
( acc * arg, arg-1 );.
```

Функция не является хвостово рекурсивной, когда результат рекурсивного вызова затем используется в вычислениях. Листинг 16-4 показывает пример нехвостово-рекурсивного вычисления факториала. Результат рекурсивного вызова умножается на `arg` перед возвратом, а значит, хвостовой рекурсии здесь нет.

Листинг 16-4. `factorial_nontailrec.c`

```
 __attribute__ (( noinline ))
     int factorial( int arg ) {
        if ( arg == 0 ) return acc;
        return arg * factorial( arg-1 );
    }
int main(int argc, char** argv) { return factorial(argc); }
```

В Главе 2 мы рассмотрели Вопрос 20, предлагающий решение в духе хвостовой рекурсии. Когда последнее, что делает функция, — это вызов другой функции, сразу за которым следует возврат, мы можем выполнить переход к началу указанной функции. Другими словами, следующий шаблон инструкций может быть предметом оптимизации:

```
; somewhere else:
    call f
...
...
f:
   ...
   call g
   ret      ; 1
g:
   ...
   ret      ; 2
```

Инструкции `ret` в этом листинге помечены как первая и вторая. Выполнение `call g` поместит адрес возврата в стек. Это адрес первой инструкции `ret`. Когда `g` завершает выполнение, она выполняет вторую инструкцию `ret`, которая снимает адрес возврата, оставляя нас на первой `ret`. Таким образом, две инструкции `ret` будут выполнены подряд, прежде чем управление перейдёт функции, вызвавшей `f`. Однако почему бы не вернуться к вызывающей стороне `f` немедленно? Для этого мы заменяем `call g` на `jmp g`. Теперь `g` мы никогда не вернёмся в функцию `f`, и мы не поместим бесполезный адрес возврата в стек. Вторая `ret` заберёт адрес возврата из `call f`, который должен был где-то произойти, и вернёт нас прямо туда.

```
; somewhere else:
    call f
...
...
f:
   ...
   jmp g
g:
   ...
   ret      ; 2
```

Когда `g` и `f` — одна и та же функция, это в точности случай хвостовой рекурсии. Без оптимизации `factorial(5, 1)` запустит саму себя пять раз, засоряя стек пятью кадрами стека. Последний вызов завершится выполнением `ret` пять раз подряд, чтобы избавиться от всех адресов возврата. Современные компиляторы обычно осведомлены

о хвостовых рекурсивных вызовах и умеют оптимизировать хвостовую рекурсию в цикл. Ассемблерный листинг, сгенерированный GCC для хвостово-рекурсивного факториала (листинг 16-3), показан в

Листинге 16-5.

Листинг 16-5. `factorial_tailrec.asm`

```
0000000004004c6 <factorial>:
4004c6:  89 f8                   mov     eax,edi
4004c8:  85 f6                   test    esi,esi
4004ca:  74 07                   je      4004d3 <factorial+0xd>
4004cc:  0f af c6                imul    eax,esi
4004cf:  ff ce                   dec     esi
4004d1:  eb f5                   jmp     4004c8 <factorial+0x2>
4004d3:  c3                      ret
```

Как видим, хвостовой рекурсивный вызов состоит из двух этапов.

- Заполнение регистров новыми значениями аргументов.
- Переход к началу функции.

Циклы быстрее рекурсии, потому что последней нужно дополнительное место в стеке (что также может привести к переполнению стека). Так почему бы тогда всегда не придерживаться циклов? Рекурсия часто позволяет выразить некоторые алгоритмы более связно и элегантно. Если мы можем написать функцию так, чтобы она стала ещё и хвостово рекурсивной, эта рекурсия не скажется на производительности. Листинг 16-6 показывает пример функции, обращающейся к элементу связного списка по индексу.

Листинг 16-6. `tail_rec_example_list.c`

```
#include <stdio.h>
#include <malloc.h>
struct llist {
    struct llist* next;
    int value;
};
struct llist* llist_at(
        struct llist* lst,
        size_t idx ) {
    if ( lst && idx ) return llist_at( lst->next, idx-1 );
    return lst;
}
struct llist* c( int value, struct llist* next) {
    struct llist* lst = malloc( sizeof(struct llist*) );
    lst->next = next;
    lst->value = value;
    return lst;
}
int main( void ) {
    struct llist* lst = c( 1, c( 2, c( 3, NULL )));
    printf("%d\n", llist_at( lst, 2 )->value );
    return 0;
}
```

Компиляция с `-Os` даст нерекурсивный код, показанный в листинге 16-7.

Листинг 16-7. `tail_rec_example_list.asm`

```
0000000000400596 <llist_at>:
400596:       48 89 f8                    mov      rax,rdi
400599:       48 85 f6                    test     rsi,rsi
40059c:       74 0d                       je       4005ab <llist_at+0x15>
40059e:       48 85 c0                    test     rax,rax
4005a1:       74 08                       je       4005ab <llist_at+0x15>
4005a3:       48 ff ce                    dec      rsi
4005a6:       48 8b 00                    mov      rax,QWORD PTR [rax]
4005a9:       eb ee                       jmp      400599 <llist_at+0x3>
4005ab:       c3                          ret
```

Как это использовать? Никогда не бойтесь использовать хвостовую рекурсию, если она делает код более читаемым, ведь она не влечёт штрафа производительности.

16.1.5 Устранение общих подвыражений

Соответствующие опции GCC: `-fgcse` и другие, содержащие подстроку `cse`. Вычисление двух выражений с общей частью не приводит к двукратному вычислению этой части. Это значит, что с точки зрения производительности нет смысла вычислять эту часть заранее, сохранять её результат в переменную и использовать её в обоих выражениях. В примере, показанном в листинге 16-8, подвыражение $x^2 + 2x$ вычисляется один раз, хотя наивный подход предполагает обратное.

Листинг 16-8. `common_subexpression.c`

```
#include <stdio.h>
__attribute__ ((noinline))
    void test(int x) {
        printf("%d %d",
                x*x + 2*x + 1,
                x*x + 2*x - 1 );
    }
int main(int argc, char** argv) {
    test( argc );
    return 0;
}
```

В качестве доказательства листинг 16-9 показывает скомпилированный код, который не вычисляет $x^2 + 2x$ дважды.

Листинг 16-9. `common_subexpression.asm`

```
0000000000400516 <test>:
; rsi = x + 2
400516:       8d 77 02                    lea       esi,[rdi+0x2]
400519:       31 c0                       xor       eax,eax
40051b:       0f af f7                    imul      esi,edi
; rsi = x*(x+2)
40051e:       bf b4 05 40 00              mov       edi,0x4005b4
; rdx = rsi-1 = x*(x+2) - 1
400523:       8d 56 ff                    lea       edx,[rsi-0x1]
; rsi = rsi + 1 = x*(x+2) - 1
400526:       ff c6                       inc       esi
400528:       e9 b3 fe ff ff              jmp       4003e0 <printf@plt>
```

Как это использовать? Не бойтесь писать красивые формулы с одинаковыми общими подвыражениями: они будут вычислены эффективно. Отдавайте предпочтение читаемости кода.

16.1.6 Распространение констант

Соответствующие опции GCC: `-fipa-cp, -fgcse, -fipa-cp-clone` и др. Если компилятор может доказать, что переменная имеет конкретное значение в определённом месте программы, он может опустить чтение её значения и подставить его туда напрямую. Иногда он даже генерирует специализированные версии функций, частично применённые к некоторым аргументам, если знает точное значение аргумента (опция `-fipa-cp-clone`). Например, листинг 16-10 показывает типичный случай, когда для sum будет создана специализированная версия функции всего с одним аргументом вместо двух, а значение второго аргумента зафиксировано и равно 42.

Листинг 16-10. `constant_propagation.c`

```
__attribute__ ((noinline))
static int sum(int x, int y) { return x + y; }
int main( int argc, char** argv ) {
    return sum( 42, argc );
}
```

Листинг 16-11 показывает транслированный ассемблерный код.

Листинг 16-11. `constant_propagation.asm`

```
00000000004004c0 <sum.constprop.0>:
4004c0:       8d 47 2a                    lea       eax,[rdi+0x2a]
4004c3:       c3                          ret
```

Становится ещё лучше, когда компилятор вычисляет за вас сложные выражения (включая вызовы функций). Листинг 16-2 показывает пример.

Листинг 16-12. `cp_fact.c`

```
#include <stdio.h>
int fact( int n ) {
    if (n == 0) return 1;
    else return n * fact( n-1 );
}
int main(void) {
    printf("%d\n", fact( 4 ) );
    return 0;
}
```

Очевидно, что функция факториала всегда будет вычислять один и тот же результат, потому что это значение не зависит от пользовательского ввода. GCC достаточно умён, чтобы предварительно вычислить это значение, стерев вызов и подставив значение `fact(4)` прямо как 24, как показано в листинге 16-13. Инструкция `mov edx, 0x18` помещает $24_{10} = 18_{16}$ прямо в `rdx`.

Листинг 16-13. `cp_fact.asm`

```
0000000000400450 <main>:
400450:  48 83 ec 08                    sub     rsp,0x8
400454:  ba 18 00 00 00                 mov     edx,0x18
400459:  be 44 07 40 00                 mov     esi,0x400744
40045e:  bf 01 00 00 00                 mov     edi,0x1
400463:  31 c0                          xor     eax,eax
400465:  e8 c6 ff ff ff                 call    400430 <__printf_chk@plt>
40046a:  31 c0                          xor     eax,eax
40046c:  48 83 c4 08                    add     rsp,0x8
400470:  c3                             ret
```

Как это использовать? Именованные константы не вредны, как и константные переменные. Компилятор может и будет предварительно вычислять всё, что способен, включая функции без побочных эффектов, запущенные с известными аргументами. Несколько копий функции для каждого отдельного значения аргумента могут навредить локальности и увеличат размер исполняемого файла. Учитывайте это, если столкнётесь с проблемами производительности.

16.1.7 Оптимизация возвращаемого (именованного) значения

Отбрасывание копирования и оптимизация возвращаемого значения позволяют устранить лишние операции копирования. Напомним, что, наивно говоря, локальные переменные создаются внутри кадра стека функции. Так что если функция возвращает экземпляр структурного типа, ей сначала нужно создать его в собственном кадре стека, а затем скопировать во внешний мир (если только он не помещается в два регистра общего назначения, `rax` и `rdx`). Листинг 16-14 показывает пример.

Листинг 16-14. `nrvo.c`

```
struct p  {
    long x;
    long y;
    long z;
};
__attribute__ ((noinline))
    struct p f(void) {
        struct p copy;
        copy.x = 1;
        copy.y = 2;
        copy.z = 3;
        return copy;
    }
int main(int argc, char** argv) {
    volatile struct p inst = f();
    return 0;
}
```

Экземпляр `struct p` по имени `copy` создаётся в кадре стека `f`. Его поля заполняются значениями 1, 2 и 3, а затем он копируется во внешний мир, предположительно по указателю, принятому `f` как скрытый аргумент. Листинг 16-15 показывает итоговый ассемблерный код.

Листинг 16-15. `nrvo_off.asm`

```
00000000004004b6 <f>:
; prologue
4004b6:  55                             push   rbp
4004b7:  48 89 e5                       mov    rbp,rsp
; A hidden argument is the address of a structure which will hold the  result.
; It is saved into stack.
4004ba:  48 89 7d d8                    mov    QWORD PTR [rbp-0x28],rdi
; Filling the fields of `copy` local variable
4004be:  48 c7 45 e0 01 00 00           mov    QWORD PTR [rbp-0x20],0x1
4004c5:  00
4004c6:  48 c7 45 e8 02 00 00           mov    QWORD PTR [rbp-0x18],0x2
4004cd:  00
4004ce:  48 c7 45 f0 03 00 00           mov    QWORD PTR [rbp-0x10],0x3
4004d5:  00
; rax = address of the destination struct
4004d6:  48 8b 45 d8                    mov    rax,QWORD PTR [rbp-0x28]
; [rax] = 1 (taken from `copy.x`)
4004da:  48 8b 55 e0                    mov    rdx,QWORD PTR [rbp-0x20]
4004de:  48 89 10                       mov    QWORD PTR [rax],rdx
; [rax + 8] = 2 (taken from `copy.y`)
4004da:  48 8b 55 e0                    mov    rdx,QWORD PTR [rbp-0x20]
4004e1:  48 8b 55 e8                    mov    rdx,QWORD PTR [rbp-0x18]
4004e5:  48 89 50 08                    mov    QWORD PTR [rax+0x8],rdx
; [rax + 10] = 3 (taken from `copy.z`)
4004e9:  48 8b 55 f0                    mov    rdx,QWORD PTR [rbp-0x10]
4004ed:  48 89 50 10                    mov    QWORD PTR [rax+0x10],rdx
; rax =  address where we have put the structure contents
; (it was the hidden argument)
4004f1:  48 8b 45 d8                    mov    rax,QWORD PTR [rbp-0x28]
4004f5:  5d                             pop    rbp
4004f6:  c3                             ret
00000000004004f7 <main>:
4004f7:  55                             push   rbp
4004f8:  48 89 e5                       mov    rbp,rsp
4004fb:  48 83 ec 30                    sub    rsp,0x30
4004ff:  89 7d dc                       mov    DWORD PTR [rbp-0x24],edi
400502:  48 89 75 d0                    mov    QWORD PTR [rbp-0x30],rsi
400506:  48 8d 45 e0                    lea    rax,[rbp-0x20]
40050a:  48 89 c7                       mov    rdi,rax
40050d:  e8 a4 ff ff ff                 call   4004b6 <f>
400512:  b8 00 00 00 00                 mov    eax,0x0
400517:  c9                             leave
```

```
400518:  c3                             ret
400519:  0f 1f 80 00 00 00 00           nop    DWORD PTR [rax+0x0]
```

Компилятор может сгенерировать более эффективный код, как показано в листинге 16-16.

Листинг 16-16. `nrvo_on.asm`

```
0000000004004b6 <f>:
4004b6:  48 89 f8                       mov     rax,rdi
4004b9:  48 c7 07 01 00 00 00           mov     QWORD PTR [rdi],0x1
4004c0:  48 c7 47 08 02 00 00           mov      QWORD PTR [rdi+0x8],0x2
4004c7:  00
4004c8:  48 c7 47 10 03 00 00           mov QWORD PTR [rdi+0x10],0x3
4004cf:  00
4004d0:  c3                             ret
0000000004004d1 <main>:
4004d1:  48 83 ec 20                    sub     rsp,0x20
4004d5:  48 89 e7                       mov     rdi,rsp
4004d8:  e8 d9 ff ff ff                 call    4004b6 <f>
4004dd:  b8 00 00 00 00                 mov     eax,0x0
4004e2:  48 83 c4 20                    add     rsp,0x20
4004e6:  c3                             ret
4004e7:  66 0f 1f 84 00 00 00           nop     WORD PTR [rax+rax*1+0x0]
4004ee:  00 00
```

Мы вообще не выделяем место в кадре стека под `copy`! Вместо этого мы оперируем напрямую структурой, переданной нам через скрытый аргумент. *Как это использовать?* Если вы хотите написать функцию, заполняющую определённую структуру, обычно невыгодно передавать ей указатель на предварительно выделенную область памяти напрямую (или выделять её с помощью `malloc`, что тоже медленнее).

16.1.8 Влияние предсказания переходов

На уровне микрокода действия, выполняемые CPU (центральным процессором), ещё более примитивны, чем машинные инструкции; они тоже переупорядочиваются для лучшего использования всех ресурсов CPU. Предсказание переходов — аппаратный механизм, нацеленный на повышение скорости выполнения программы. Когда CPU видит инструкцию условного перехода (такую как `jg`), он может

- Начать выполнять обе ветви одновременно; или
- Угадать, какая ветвь будет выполнена, и начать её выполнять.

Это происходит, когда результат вычисления (например, значение флага `GF` в `jg [rax]`), от которого зависит место назначения перехода, ещё не готов, так что мы начинаем выполнять код спекулятивно, чтобы не тратить время впустую. Блок предсказания переходов может ошибиться, выдав *неверное предсказание (misprediction)*. В этом случае, как только вычисление завершится, CPU выполнит дополнительную работу по откату изменений, сделанных инструкциями из неверной ветви. Это медленно и реально сказывается на производительности программы, но неверные предсказания относительно редки. Точная логика предсказания переходов зависит от модели CPU. В общем случае существует два типа предсказания [6]: *статическое* и *динамическое*.

- Если у CPU нет информации о переходе (когда он выполняется впервые), используется

статический алгоритм. Возможный простой алгоритм таков:

- Если это переход вперёд, мы предполагаем, что он произойдёт. Если это переход назад, мы предполагаем, что он не произойдёт.

В этом есть смысл, потому что переходы, используемые для реализации циклов, скорее произойдут, чем нет.

- Если переход уже случался в прошлом, CPU может использовать более сложные

алгоритмы. Например, можно использовать кольцевой буфер, хранящий информацию о том, произошёл переход или нет. Другими словами, он хранит историю переходов. При использовании такого подхода небольшие циклы длиной, делящей длину буфера, хорошо предсказываются. Лучший источник актуальной информации о конкретной модели CPU можно найти в [16]. К сожалению, большая часть информации о внутреннем устройстве CPU не раскрывается публично. *Как это использовать?* При использовании if-then-else или switch начинайте с самых вероятных случаев. Можно также использовать специальные подсказки, такие как директивы GCC `__builtin_expect`, реализованные как специальные префиксы инструкций для инструкций перехода (см. [6]).

16.1.9 Влияние исполнительных блоков

CPU состоит из множества частей. Каждая инструкция выполняется в несколько этапов, и на каждом этапе её обрабатывают разные части CPU. Например, первый этап обычно называется выборкой инструкции и состоит из загрузки инструкции из памяти[38] вообще без учёта её семантики.

Часть CPU, выполняющая операции и вычисления, называется *исполнительным блоком (execution unit)*. Она реализует разные виды операций, которые CPU нужно обрабатывать: выборку инструкций, арифметику, трансляцию адресов, декодирование инструкций и т. д. На самом деле CPU может использовать её более или менее независимым об-

разом. Разные инструкции выполняются за разное число этапов, и каждый из этих этапов может выполняться разным исполнительным блоком. Это позволяет интересно использовать схемотехнику, например:

- Выборка одной инструкции сразу после выборки другой (но выполнение которой ещё не завершилось).
- Одновременное выполнение нескольких арифметических действий, несмотря на то что в ассемблерном коде они описаны последовательно.

CPU семейства Pentium IV уже были способны при подходящих обстоятельствах одновременно выполнять четыре арифметические инструкции. Как нам использовать знание о существовании исполнительных блоков? Взглянем на пример, показанный в

Листинге 16-17.

Листинг 16-17. `cycle_nonpar_arith.asm`

```
looper:
    mov      rax,[rsi]
    ; The next instruction depends on the previous one.
    ; It means that we can not swap them because
    ; the program behavior will change.
    xor     rax, 0x1
    ; One more dependency here
    add     [rdi],rax
    add     rsi,8
    add     rdi,8
    dec     rcx
    jnz     looper
```

Можем ли мы сделать его быстрее? Мы видим зависимости между инструкциями, мешающие оптимизатору микрокода CPU. Мы развернём цикл так, чтобы две итерации старого цикла стали одной итерацией нового. Листинг 16-18 показывает результат.

Листинг 16-18. `cycle_par_arith.asm`

```
looper:
mov      rax,  [rsi]
mov      rdx,  [rsi + 8]
xor      rax,  0x1
xor      rdx,  0x1
add      [rdi], rax
add      [rdi+8], rdx
add      rsi, 16
add      rdi, 16
sub      rcx, 2
jnz      looper
```

Теперь зависимостей нет, инструкции двух итераций теперь перемешаны. В таком порядке они будут выполняться быстрее, потому что это усиливает одновременное использование разных исполнительных блоков CPU. Зависимые инструкции стоит размещать подальше друг от друга, чтобы между ними могли выполняться другие инструкции.

> **Вопрос 333** Что такое конвейер инструкций и суперскалярная архитектура?

Мы не можем сказать вам, какие исполнительные блоки есть в вашем CPU, потому что это сильно зависит от модели. Нужно читать руководства по оптимизации для конкретного CPU, такие как [16]. Дополнительные источники часто полезны; например, процессоры Haswell хорошо описаны в [17].

16.1.10 Группировка чтений и записей в коде

Оборудование лучше работает с последовательностями чтений и записей, не перемежающимися друг с другом. По этой причине код, показанный в листинге 16-19, обычно медленнее своего аналога, показанного в листинге 16-20. У второго последовательности последовательных чтений и записей сгруппированы, а не перемежаются.

Листинг 16-19. `rwgroup_bad.asm`

```
mov rax,[rsi]
mov [rdi],rax
mov rax,[rsi+8]
mov [edi+4],eax
mov rax,[rsi+16]
mov [rdi+16],rax
mov rax,[esi+24]
mov [rdi+24],eax
```

Листинг 16-20. `rwgroup_good.asm`

```
mov rax, [rsi]
mov rbx, [rsi+8]
mov rcx, [rsi+16]
mov rdx, [rsi+24]
mov [rdi], rax
mov [rdi+8], rbx
mov [rdi+16], rcx
mov [rdi+24], rdx
```

16.2 Кэширование

16.2.1 Как эффективно использовать кэш?

Кэширование — один из важнейших механизмов повышения производительности. Мы говорили об общих понятиях кэширования в Главе 4. Этот раздел подробнее рассмотрит, как эффективно использовать эти понятия. Начнём с того, что, вопреки духу архитектуры фон Неймана, обычные CPU используют отдельные кэши для инструкций и данных уже как минимум 25 лет. Инструкции и код практически всегда живут в разных областях памяти, что объясняет, почему раздельные кэши эффективнее. Нас интересует кэш данных. По умолчанию все операции с памятью задействуют кэш, за исключением страниц, помеченных битами cache-write-through и cache-disable (см. Главу 4). Кэш содержит небольшие блоки памяти по 64 байта, называемые *строками кэша*, выровненные по границе в 64 байта. Кэш-память отличается от основной памяти на уровне схемотехники. Каждая строка кэша идентифицируется *тегом* — адресом соответствующего блока памяти. С помощью специальной схемотехники строку кэша можно получить по её адресу очень быстро (но только для маленьких кэшей, вроде 4 МБ на процессор, иначе это слишком дорого). При попытке прочитать значение из памяти CPU снача-

ла попробует прочитать его из кэша. Если его там нет, соответствующий блок памяти будет загружен в кэш. Такая ситуация называется *промахом кэша (cache-miss)* и часто оказывает *огромное* влияние на производительность программы. Обычно есть несколько уровней кэша; каждый следующий больше и медленнее. *LL-кэш (Last Level Cache)* — последний уровень кэша, ближайший к основной памяти. Для программ с хорошей локальностью кэширование работает хорошо. Однако когда для части кода локальность нарушена, имеет смысл обходить кэш. Например, запись значений в большой блок памяти, к которому в ближайшее время обращений не будет, лучше выполнять без использования кэша. CPU пытается предсказать, к каким адресам памяти будут обращения в ближайшем будущем, и заранее загружает соответствующие части памяти в кэш. Он благоволит последовательным обращениям к памяти. Это даёт нам два важных эмпирических правила, нужных для эффективного использования кэшей.

- Старайтесь обеспечивать локальность.
- Отдавайте предпочтение последовательному доступу к памяти (и проектируйте структуры данных с учётом этого).

16.2.2 Предвыборка

Можно дать CPU специальную подсказку, указывающую, что к определённой области памяти скоро будет обращение. В Intel 64 это делается с помощью инструкции `prefetch`. Она принимает адрес в памяти; CPU сделает всё возможное, чтобы в ближайшем будущем загрузить его в кэш. Это используется для предотвращения промахов кэша. Использование `prefetch` может быть достаточно эффективным, но его следует сочетать с тестированием. Её нужно выполнять перед самими обращениями к данным, но не слишком близко. Предзагрузка кэша выполняется асинхронно, а значит, она выполняется одновременно с выполнением следующих инструкций. Если `prefetch` расположена слишком близко к обращениям к данным, у CPU не хватит времени предзагрузить данные в кэш, и промах кэша всё равно произойдёт. Более того, очень важно понимать, что «близко» и «далеко» от обращения к данным означают позицию инструкции в трассе выполнения. Не обязательно размещать `prefetch` близко с точки зрения структуры программы (в той же функции), но нужно выбрать место, предшествующее обращению к данным. Оно может находиться в совершенно другом модуле, например, в модуле логирования, который просто *обычно оказывается выполненным перед обращением к данным*. Это, конечно, очень плохо для читаемости кода, вносит неочевидные зависимости между модулями и является техникой «последнего средства». Чтобы использовать `prefetch` в C, можно использовать одну из встроенных функций GCC:

```
Void __builtin_prefetch (const void *addr, ...)
```

Она будет заменена архитектурно-специфичной инструкцией предвыборки. Помимо адреса, она также принимает два параметра, которые должны быть целочисленными константами.

1. Будем ли мы читать по этому адресу (0, по умолчанию) или писать (1)?
2. Насколько сильна локальность? От трёх для максимальной локальности до нуля для минимальной. Ноль означает, что значение можно вытеснить из кэша после использования, 3 означает, что все уровни кэша должны продолжать его хранить.

Предвыборку выполняет сам CPU, *если* он способен предсказать, где, вероятно, произойдёт следующее обращение к памяти. Хотя это хорошо работает для непрерывных обращений к памяти, таких как обход массивов, это начинает терять эффективность, как только шаблон обращения к памяти начинает казаться предсказателю случайным.

16.2.3 Пример: бинарный поиск с предвыборкой

Рассмотрим пример, показанный в листинге 16-21.

Листинг 16-21. `prefetch_binsearch.c`

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#define SIZE 1024*512*16
int binarySearch(int *array, size_t number_of_elements, int key) {
    size_t low = 0, high = number_of_elements-1, mid;
    while(low <= high) {
        mid = (low + high)/2;
#ifdef DO_PREFETCH
        // low path
        __builtin_prefetch (&array[(mid + 1 + high)/2], 0, 1);
        // high path
        __builtin_prefetch (&array[(low + mid - 1)/2], 0, 1);
#endif
        if(array[mid] < key)
            low = mid + 1;
        else if(array[mid] == key)
            return mid;
        else if(array[mid] > key)
            high = mid-1;
    }
    return -1;
}
int main() {
    size_t i = 0;
    int NUM_LOOKUPS = SIZE;
    int *array;
    int *lookups;
    srand(time(NULL));
    array =  malloc(SIZE*sizeof(int));
    lookups = malloc(NUM_LOOKUPS * sizeof(int));
    for (i=0;i<SIZE;i++) array[i] = i;
    for (i=0;i<NUM_LOOKUPS;i++) lookups[i] = rand() % SIZE;
    for (i=0;i<NUM_LOOKUPS;i++)
        binarySearch(array, SIZE, lookups[i]);
    free(array);
    free(lookups);
}
```

Шаблон обращения к памяти при бинарном поиске трудно предсказать. Он крайне непоследователен: прыжок от начала к концу, затем к середине, затем к четверти и т. д. Посмотрим на разницу во времени выполнения. Листинг 16-22 показывает результаты

выполнения с отключённой предвыборкой.

Листинг 16-22. `binsearch_prefetch_off`

```
> gcc -O3 prefetch.c -o prefetch_off && /usr/bin/time -v ./prefetch_off
   Command being timed: "./prefetch_off"
   User time (seconds): 7.56
   System time (seconds): 0.02
   Percent of CPU  this job got: 100%
   Elapsed (wall clock) time (h:mm:ss or m:ss): 0:07.58
   Average shared text size (kbytes): 0
   Average unshared data size (kbytes): 0
   Average stack size (kbytes): 0
   Average total size (kbytes): 0
   Maximum resident set size (kbytes): 66432
   Average resident set size (kbytes): 0
   Major (requiring I/O) page faults: 0
   Minor (reclaiming a frame) page faults: 16444
   Voluntary context switches: 1
   Involuntary context switches: 51
   Swaps: 0
   File system inputs: 0
   File system outputs: 0
   Socket messages sent: 0
   Socket messages received: 0
   Signals delivered: 0
   Page size (bytes): 4096
   Exit status: 0
```

Листинг 16-23 показывает результаты выполнения с включённой предвыборкой.

Листинг 16-23. `binsearch_prefetch_on`

```
> gcc -O3 prefetch.c -o prefetch_off && /usr/bin/time -v ./prefetch_off
   Command being timed: "./prefetch_on"
   User time  (seconds):  6.56
   System time (seconds): 0.01
   Percent of CPU  this job got: 100%
   Elapsed (wall clock) time (h:mm:ss or m:ss): 0:06.57
   Average shared text size (kbytes): 0
   Average unshared data size (kbytes): 0
   Average stack size (kbytes): 0
   Average total size (kbytes): 0
   Maximum resident set size (kbytes): 66512
   Average resident set size (kbytes): 0
   Major (requiring I/O) page faults: 0
   Minor (reclaiming a frame) page faults: 16443
   Voluntary context switches: 1
   Involuntary context switches: 42
   Swaps: 0
   File system inputs: 0
   File system outputs: 0
   Socket messages sent: 0
   Socket messages received: 0
   Signals  delivered: 0
   Page size (bytes): 4096
   Exit status: 0
```

С помощью утилиты `valgrind` и модуля `cachegrind` мы можем проверить количество промахов кэша. Листинг 16-24 показывает результаты без предвыборки, а листинг 16-25 показывает результаты с предвыборкой. `I` соответствует кэшу инструкций, `D` — кэшу данных, `LL` – *Last Level Cache*). Промахов кэша данных почти 100%, что очень плохо.

Листинг 16-24. `binsearch_prefetch_off_cachegrind`

```
==25479== Cachegrind, a cache and branch-prediction profiler
==25479== Copyright (C) 2002-2015, and GNU GPL'd, by Nicholas Nethercote et al.
==25479== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==25479== Command: ./prefetch_off
==25479==
--25479-- warning: L3 cache found, using its data for the LL simulation.
==25479==
==25479== I   refs:      2,529,064,580
==25479== I1  misses:              778
==25479== LLi misses:              774
==25479== I1  miss rate:          0.00%
==25479== Lli miss rate:          0.00%
==25479==
==25479== D   refs:         404,809,999  (335,588,367 rd   + 69,221,632 wr)
==25479== D1  misses:       160,885,105  (159,835,971 rd   +  1,049,134 wr)
==25479== LLd misses:       133,467,980  (132,418,879 rd   +  1,049,101 wr)
==25479== D1  miss rate:           39.7%  (       47.6%     +        1.5%  )
==25479== LLd miss rate:           33.0%  (       39.5%     +        1.5%  )
==25479==
==25479== LL refs:          160,885,883  (159,836,749 rd   +  1,049,134 wr)
==25479== LL misses:        133,468,754  (132,419,653 rd   +  1,049,101 wr)
==25479== LL miss rate:             4.5%  (        4.6%     +        1.5%  )
```

Листинг 16-25. `binsearch_prefetch_on_cachegrind`

```
==26238== Cachegrind, a cache and branch-prediction profiler
==26238== Copyright (C) 2002-2015, and GNU GPL'd, by Nicholas Nethercote et al.
==26238== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==26238== Command: ./prefetch_on
==26238==
--26238-- warning: L3 cache found, using its data for the LL simulation.
==26238==
==26238== I   refs:     3,686,688,760
==26238== I1  misses:             777
==26238== LLi misses:             773
==26238== I1  miss rate:         0.00%
==26238== LLi miss rate:         0.00%
==26238==
==26238== D   refs:       404,810,009   (335,588,374  rd   + 69,221,635 wr)
==26238== D1  misses:     160,887,823   (159,838,690  rd   +  1,049,133 wr)
==26238== LLd misses:     133,488,742   (132,439,642  rd   +  1,049,100 wr)
==26238== D1  miss  rate:        39.7%  (       47.6%      +        1.5%  )
==26238== LLd miss rate:         33.0%  (       39.5%      +        1.5%  )
==26238==
==26238== LL refs:        160,888,600   (159,839,467  rd   +  1,049,133 wr)
==26238== LL misses:      133,489,515   (132,440,415  rd   +  1,049,100 wr)
==26238== LL miss rate:           3.3%  (        3.3%      +        1.5%  )
```

Как видим, количество промахов кэша резко уменьшилось.

16.2.4 Обход кэша

Существует способ писать в память в обход кэша, работающий даже в пользовательском режиме, так что для его использования нам не нужен доступ к записям таблицы страниц, хранящим бит `CWT`. В Intel 64 это позволяет сделать инструкция `movntps`. Сама операционная система обычно устанавливает бит `CWT` (cache-write-through) в таблицах страниц, когда используется ввод/вывод, отображаемый на память (когда виртуальная память служит интерфейсом для внешних устройств). В этом случае любое чтение или запись в память приведёт к инвалидации кэша, что только вредит производительности, не давая никакой выгоды. У GCC есть встроенные функции, транслируемые в машинно-специфичные инструкции, выполняющие операции с памятью без задействования кэша. Листинг 16-26 их показывает.

Листинг 16-26. `cache_bypass_intrinsics.c`

```
#include <emmintrin.h>
void _mm_stream_si32(int *p, int a);
void _mm_stream_si128(int *p, __m128i a);
void _mm_stream_pd(double *p, __m128d a);
#include <xmmintrin.h>
void _mm_stream_pi(__m64 *p, __m64 a);
void _mm_stream_ps(float *p, __m128 a);
#include <ammintrin.h>
void _mm_stream_sd(double *p, __m128d a);
void _mm_stream_ss(float *p, __m128 a);
```

Обход кэша полезен, если мы уверены, что довольно долго не будем трогать соответствующую область памяти. Подробнее см. в [12].

16.2.5 Пример: инициализация матрицы

Чтобы проиллюстрировать хороший шаблон доступа к памяти, мы возьмём огромную матрицу со значениями 42. Матрица хранится построчно. Одна программа, показанная в листинге 16-27, инициализирует каждую строку; другая, показанная в листинге 16-28, инициализирует каждый столбец. Какая будет быстрее?

Листинг 16-27. `matrix_init_linear.c`

```
#include <stdio.h>
#include <malloc.h>
#define DIM (16*1024)
int main( int argc, char** argv ) {
    size_t i, j;
    int* mat = (int*)malloc( DIM * DIM * sizeof( int ) );
    for( i = 0; i < DIM; ++i )
        for( j = 0; j < DIM; ++j )
            mat[i*DIM+j] = 42;
    puts("TEST DONE");
    return 0;
}
```

Листинг 16-28. `matrix_init_ra.c`

```
#include <stdio.h>
#include <malloc.h>
#define DIM (16*1024)
int main( int argc, char** argv ) {
    size_t i, j;
    int* mat = (int*)malloc( DIM * DIM * sizeof( int ) );
    for( i = 0; i < DIM; ++i )
        for( j = 0; j < DIM; ++j )
            mat[j*DIM+i] = 42;
    puts("TEST DONE");
    return 0;
}
```

Мы снова используем утилиту `time` (не встроенную в оболочку), чтобы проверить время

выполнения.

```
> /usr/bin/time -v ./matrix_init_ra
   Command being timed: "./matrix_init_ra"
   User time (seconds): 2.40
   System time (seconds): 1.01
   Percent of CPU this job got: 86%
   Elapsed (wall clock) time (h:mm:ss or m:ss): 0:03.94
   Average shared text size (kbytes): 0
   Average unshared data size (kbytes): 0
   Average stack size (kbytes): 0
   Average total size (kbytes): 0
   Maximum resident set size (kbytes): 889808
   Average resident set size (kbytes): 0
   Major (requiring I/O) page faults: 2655
   Minor (reclaiming a frame) page faults: 275963
   Voluntary context switches: 2694
   Involuntary context switches: 548
   Swaps: 0
   File system inputs: 132368
   File system outputs: 0
   Socket messages sent: 0
   Socket messages received: 0
   Signals delivered: 0
   Page size (bytes): 4096
   Exit status: 0
> /usr/bin/time -v ./matrix_init_linear
   Command being timed: "./matrix_init_linear"
   User time (seconds): 0.12
   System time (seconds): 0.83
   Percent of CPU this job got: 92%
   Elapsed (wall clock) time (h:mm:ss or m:ss): 0:01.04
   Average shared text size (kbytes): 0
   Average unshared data size (kbytes): 0
   Average stack size (kbytes): 0
   Average total size (kbytes): 0
   Maximum resident set size (kbytes): 900280
   Average resident set size (kbytes): 0
   Major (requiring I/O) page faults: 4
    Minor (reclaiming a frame) page faults: 262222
   Voluntary context switches: 29
   Involuntary context switches: 449
   Swaps: 0
   File system inputs: 176
   File system outputs: 0
```

```
    Socket messages sent: 0
    Socket messages received: 0
    Signals  delivered: 0
    Page size (bytes): 4096
    Exit status: 0
```

Выполнение настолько медленнее из-за промахов кэша, что можно проверить с помощью утилиты `valgrind` с модулем `cachegrind`, как показано в листинге 16-29.

Листинг 16-29. `cachegrind_matrix_bad`

```
> valgrind --tool=cachegrind ./matrix_init_ra
==17022== Command: ./matrix_init_ra
==17022==
--17022-- warning: L3 cache found, using its data for the LL simulation.
==17022==
==17022== I   refs:     268,623,230
==17022== I1  misses:           809
==17022== LLi misses:           804
==17022== I1  miss rate:       0.00%
==17022== Lli miss rate:       0.00%
==17022==
==17022== D   refs:      67,163,682   (40,974 rd  + 67,122,708 wr)
==17022== D1  misses:    67,111,793   ( 2,384 rd  + 67,109,409 wr)
==17022== LLd misses:    67,111,408   ( 2,034 rd  + 67,109,374 wr)
==17022== D1  miss rate:       99.9%  (   5.8%    +      100.0%  )
==17022== LLd miss rate:      99.9%   (   5.0%    +      100.0%  )
==17022==
==17022== LL refs:       67,112,602   ( 3,193 rd  + 67,109,409 wr)
==17022== LL misses:     67,112,212   ( 2,838 rd  + 67,109,374 wr)
==17022== LL miss rate:        20.0%  (   0.0%    +      100.0%  )
```

Как видим, последовательный доступ к памяти радикально снижает число промахов кэша:

```
==17023== Command: ./matrix_init_linear
==17023==
--17023-- warning: L3 cache found, using its data for the LL simulation.
==17023==
==17023== I   refs:      336,117,093
==17023== I1  misses:            813
==17023== LLi misses:            808
==17023== I1  miss rate:        0.00%
==17023== LLi miss rate:        0.00%
==17023==
==17023== D   refs:       67,163,675   (40,970 rd  + 67,122,705 wr)
==17023== D1  misses:     16,780,146   ( 2,384 rd  + 16,777,762 wr)
==17023== LLd misses:     16,779,760   ( 2,033 rd  + 16,777,727 wr)
==17023== D1  miss rate:        25.0%  (   5.8%    +       25.0%  )
==17023== LLd  miss rate:       25.0%  (   5.0%    +       25.0%  )
==17023==
==17023== LL refs:        16,780,959   ( 3,197 rd   + 16,777,762 wr)
==17023== LL misses:      16,780,568   ( 2,841 rd   + 16,777,727 wr)
==17023== LL miss rate:          4.2%  (   0.0%     +      25.0%   )
```

> **Вопрос 334** Загляните на страницы `man` GCC, раздел «Optimizations».

16.3 Класс инструкций SIMD

Модель вычислений фон Неймана по своей природе последовательна. Она не предполагает, что некоторые операции можно выполнять параллельно. Однако со временем стало очевидно, что для достижения лучшей производительности необходимо выполнять действия параллельно. Это возможно, когда вычисления независимы друг от друга. Например, чтобы просуммировать 1 миллион целых чисел, можно вычислить сумму блоков по 100,000 чисел на десяти процессорах, а затем сложить результаты. Это типичная задача, хорошо решаемая техникой *map-reduce* [5]. Реализовать параллельное выполнение можно двумя способами.

- Параллельное выполнение нескольких последовательностей инструкций. Этого можно достичь добавлением дополнительных ядер процессора. Многопоточное программирование, использующее несколько ядер, мы обсудим в Главе 17.
- Параллельное выполнение действий, нужных для завершения *одной инструкции*. В этом случае у нас могут быть инструкции, вызывающие несколько независимых

вычислений, охватывающих разные части схем процессора, которые можно задействовать параллельно. Чтобы реализовать такие инструкции, CPU должен включать несколько АЛУ, чтобы получить реальный выигрыш в производительности, но ему не обязательно уметь по-настоящему одновременно выполнять несколько инструкций. Такие инструкции называются *SIMD* (Single Instruction, Multiple Data — одна инструкция, много данных).

В этом разделе мы рассмотрим расширения CPU под названием *SSE (Streaming SIMD Extensions)* и их более новый аналог *AVX (Advanced Vector Extensions)*. В отличие от инструкций SIMD, большинство изученных нами до сих пор инструкций относятся к типу SISD (Single Instruction, Single Data — одна инструкция, одни данные).

16.4 Расширения SSE и AVX

Инструкции SIMD — основа для расширений системы команд SSE и AVX. Большинство из них используются для выполнения операций сразу над несколькими парами данных; например, `mulps` может перемножить четыре пары 32-битных float за раз. Однако их аналоги с одной парой операндов (такие как `mulss`) теперь являются рекомендуемым способом выполнения всей арифметики с плавающей точкой. По умолчанию GCC будет генерировать инструкции SSE для операций с числами с плавающей точкой. Они принимают операнды либо в регистрах `xmm`, либо в памяти.

> **Согласованность** Мы опускаем описание устаревшего специализированного стека для чисел с плавающей точкой ради краткости. Однако хотим отметить, что все части программы должны транслироваться одним и тем же методом арифметики с плавающей точкой: либо стеком чисел с плавающей точкой, либо инструкциями SSE.

Начнём с примера, показанного в листинге 16-30.

Листинг 16-30. `simd_main.c`

```
#include <stdlib.h>
#include <stdio.h>
void sse( float[static 4], float[static 4] );
int main() {
    float x[4] = {1.0f, 2.0f, 3.0f, 4.0f };
    float y[4] = {5.0f, 6.0f, 7.0f, 8.0f };
    sse( x, y );
    printf( "%f %f %f %f\n", x[0], x[1], x[2], x[3] );
    return 0;
}
```

В этом примере есть функция `sse`, определённая где-то ещё, принимающая два массива `floats`. Каждый из них должен быть шириной как минимум четыре элемента. Эта функция выполняет вычисления и изменяет первый массив. Мы называем значения *упакованными (packed)*, если они заполняют регистр `xmm` последовательными ячейками памяти одного размера. В листинге 16-30 `float x[4]` — четыре упакованных значения float одинарной точности. Мы определим функцию `sse` в файле ассемблера, показанном в листинге 16-31.

Листинг 16-31. `simd_asm.asm`

```
section .text
global sse
; rdi = x, rsi = y
sse:
    movdqa xmm0, [rdi]
    mulps  xmm0, [rsi]
    addps  xmm0, [rsi]
    movdqa [rdi], xmm0
    ret
```

Этот файл определяет функцию `sse`. Она выполняет четыре инструкции SSE:

- `movdqa` (*MOV*e *D*ouble *Q*word *A*ligned — переместить выровненное учетверённое слово) копирует 16 байт из памяти, на которую указывает `rdi`,

в регистр `xmm0`. Мы уже видели эту инструкцию в разделе 14.1.1.

- `mulps` (*MUL*tiply *P*acked *S*ingle precision floating point values — перемножить упакованные значения с плавающей точкой одинарной точности) умножает содержимое

`xmm0` на четыре последовательных значения float, хранящихся в памяти по адресу, взятому из `rsi`.

- `addps` (*ADD P*acked *S*ingle precision floating point — сложить упакованные значения с плавающей точкой одинарной точности) складывает содержимое с четырьмя

последовательными значениями float, хранящимися в памяти по адресу, снова взятому из `rsi`.

- `movdqa` копирует `xmm0` в память, на которую указывает `rdi`.

Другими словами, четыре пары float перемножаются, а затем второй float каждой пары прибавляется к первому. Шаблон именования общий: семантика действия (`mov`, `add`, `mul`...) с суффиксами. Первый суффикс — либо *P* (packed, упакованный), либо *S* (scalar, скалярный, для одиночных значений). Второй — либо *D* для значений двойной точности

(`double` в C), либо *S* для значений одинарной точности (`float` в C). Хотим ещё раз подчеркнуть, что большинство инструкций SSE принимают только выровненные операнды в памяти. Чтобы выполнить задание, вам нужно будет изучить документацию по следующим инструкциям, используя Intel Software Developer Manual [15]:

- `movsd` – переместить скалярное значение с плавающей точкой двойной точности.
- `movdqa` – переместить выровненное учетверённое слово.
- `movdqu` – переместить невыровненное учетверённое слово.
- `mulps` – перемножить упакованные значения с плавающей точкой одинарной точности.
- `mulpd` – перемножить упакованные значения с плавающей точкой двойной точности.
- `addps` – сложить упакованные значения с плавающей точкой одинарной точности.
- `haddps` – горизонтальное сложение упакованных значений float.
- `shufps` – перетасовать упакованные значения с плавающей точкой одинарной точности.
- `unpcklps` – распаковать и чередовать младшие упакованные значения с плавающей точкой двойной точности.
- `packswb` – упаковать со знаковым насыщением.
- `cvtdq2pd` – преобразовать упакованные целые числа dword в упакованные значения FP двойной точности.

Эти инструкции — часть расширений SSE. Intel ввела новое расширение под названием AVX, добавляющее новые регистры `ymm0`, `ymm1`, … , `ymm15`. Их ширина — 256 бит, к их младшим 128 битам (младшей половине) можно обращаться как к старым регистрам `xmm`. Новые инструкции по большей части имеют префикс v, например `vbroadcastss`. Важно понимать, что если ваш CPU поддерживает инструкции AVX, это не значит, что они быстрее SSE! Разные процессоры одного семейства отличаются не системой команд, а количеством схемотехники. У более дешёвых процессоров, вероятно, меньше АЛУ. Возьмём в качестве примера `mulps` с регистрами `ymm`. Она используется для умножения 8 пар float. У лучших CPU хватит АЛУ (арифметико-логических устройств), чтобы перемножить все восемь пар одновременно. У более дешёвых CPU будет, скажем, только четыре АЛУ, так что им придётся дважды пройтись на уровне микрокода, сначала перемножив первые четыре пары, затем последние. Программист этого не заметит, при использовании инструкции семантика та же, но с точки зрения производительности разница будет заметна. Одна версия AVX для `mulps` с регистрами `ymm` и восемью парами float может быть даже медленнее, чем две версии SSE для `mupls` с регистрами `xmm` по четыре пары каждая!

16.4.1 Задание: сепия-фильтр

В этом задании мы создадим программу для применения сепия-фильтра к изображению. Сепия-фильтр заставляет изображение с яркими цветами выглядеть как старая, состарившаяся фотография. В большинство графических редакторов включён сепия-фильтр. Сам фильтр закодировать не сложно. Он пересчитывает красный, зелёный и синий компоненты каждого пикселя на основе старых значений красного, зелёного и синего. Математически, если думать о пикселе как о трёхмерном векторе, преобразование — не что иное, как умножение вектора на матрицу. Пусть новое значение пикселя будет (B G R) (где верхний индекс T означает транспонирование). B, G и R означают уровни синего, зелёного и красного. В векторной форме преобразование можно описать так:

$$\begin{pmatrix} B \\ G \\ R \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{pmatrix} \times \begin{pmatrix} b \\ g \\ r \end{pmatrix}$$

В скалярной форме перепишем это как $B = bc_{11} + gc_{12} + rc_{13}$

$G = bc_{21} + gc_{22} + rc_{23}$ $R = bc_{31} + gc_{32} + rc_{33}$

В задании из раздела 13.10 мы написали программу для поворота изображения. Если вы хорошо продумали её архитектуру, будет легко повторно использовать большую часть её кода. *Нам понадобится использовать арифметику с насыщением (saturation arithmetic)*. Это значит, что все операции, такие как сложение и умножение, ограничены фиксированным диапазоном между минимальным и максимальным значением. Наша типичная машинная арифметика модульная: если результат больше максимального значения, мы придём с другой стороны диапазона. Например, для `unsigned char`: 200 + 100 = 300 mod 256 = 44. Арифметика с насыщением подразумевает, что для того же диапазона от 0 до 255 включительно 200 + 100 = 255, поскольку это максимальное значение в диапазоне. В C такая арифметика не реализована, так что нам придётся вручную проверять переполнения. В SSE есть инструкции, преобразующие значения с плавающей точкой в однобайтовые целые числа с насыщением. Выполнить преобразование на C легко. Оно требует прямого кодирования умножения матрицы на вектор с учётом насыщения. Листинг 16-32 показывает код.

Листинг 16-32. `image_sepia_c_example.c`

```
#include <inttypes.h>
struct pixel { uint8_t b, g, r; };
struct image {
    uint32_t width, height;
    struct pixel* array;
};
static unsigned char sat( uint64_t x) {
    if (x < 256) return x; return 255;
}
static void sepia_one( struct pixel* const pixel ) {
    static const float c[3][3] =  {
    { .393f, .769f, .189f },
    { .349f, .686f, .168f },
    { .272f, .543f, .131f } };
struct pixel const old = *pixel;
pixel->r = sat(
        old.r * c[0][0] + old.g * c[0][1] + old.b * c[0][2]
        );
pixel->g = sat(
        old.r * c[1][0] + old.g * c[1][1] + old.b  * c[1][2]
        );
pixel->b = sat(
        old.r * c[2][0] + old.g * c[2][1] + old.b * c[2][2]
        );
}
void sepia_c_inplace( struct image* img ) {
    uint32_t x,y;
    for( y = 0; y < img->height; y++ )
        for( x = 0; x < img->width; x++ )
            sepia_one( pixel_of( *img, x, y ) );
}
```

Заметьте, что использование `uint8_t` или `unsignedchar` очень важно. В этом задании вам нужно

- Реализовать в отдельном файле процедуру применения фильтра к большей части изображения (кроме,

возможно, последних пикселей). Она будет оперировать блоками из нескольких пикселей за раз, используя инструкции SSE. Последние несколько пикселей, не заполнившие последний блок, можно обработать по одному, используя код на C, приведённый в листинге 16-32.

- Убедитесь, что версии на C и на ассемблере дают похожие результаты.
- Скомпилируйте две программы; первая должна использовать наивный подход на C, а вторая — инструкции SSE.
- Сравните время выполнения C и SSE, используя в качестве входных данных огромное изображение (желательно сотни мегабайт).
- Повторите сравнение несколько раз и вычислите средние значения для SSE и C.

Чтобы получить заметную разницу, нам нужно выполнять как можно больше операций параллельно. Каждый пиксель состоит из 3 байт; после преобразования его компонентов в float он займёт 12 байт. Каждый регистр `xmm` шириной 16 байт. Если мы хотим быть эффективными, нам придётся использовать и последние 4 байта тоже. Чтобы этого добиться, мы используем кадр из 48 байт, соответствующий трём регистрам `xmm`, 12 компонентам пикселей и 4 пикселям. Пусть нижний индекс обозначает индекс пикселя. Тогда изображение выглядит так:

$b_1 g_1 r_1 b_2 g_2 r_2 b_3 g_3 r_3 b_4 g_4 r_4$ ...

Мы хотим вычислить первые четыре компонента. Три из них соответствуют первому пикселю, четвёртый соответствует второму. Чтобы выполнить нужные преобразования, полезно сначала поместить в регистры следующие значения:

$xmm_0 = b_1 b_1 b_1 b_2$

$xmm_1 = g_1 g_1 g_1 g_2$

$xmm_2 = r_1 r_1 r_1 r_2$

Мы будем хранить коэффициенты матрицы либо в регистрах `xmm`, либо в памяти, но важно хранить *столбцы*, а не строки. Чтобы продемонстрировать алгоритм, мы используем следующие начальные значения:

$xmm_3 = c_{11}* | c_{21} | c_{31} | c*_{11}$

$xmm_4 = c_{12}* | c_{22} | c_{32} | c*_{12}$

$xmm_5 = c_{13}* | c_{23} | c_{33} | c*_{13}$

Мы используем `mulps`, чтобы перемножить эти упакованные значения с `xmm0` ... `xmm2`.

$xmm_3 = b_1 c_{11}* | b_1 c_{21} | b_1 c_{31} | b_2 c*_{11}$

$xmm_4 = g_1 c_{12}* | g_1 c_{22} | g_1 c_{32} | g_2 c*_{12}$

$xmm_5 = r_1 c_{13}* | r_1 c_{23} | r_1 c_{33} | r_2 c*_{13}$

Следующий шаг — сложить их с помощью инструкций `addps`. Похожие действия нужно выполнить с двумя другими 16-байтовыми частями кадра, содержащими

$g_2r_2b_3g_3$ и $r_3b_4g_4r_4$

Эта техника с использованием транспонированной матрицы коэффициентов позволяет нам обойтись без инструкций горизонтального сложения, таких как `haddps`. Она подробно описана в [19]. Чтобы измерить время, используйте `getrusage(RUSAGE_SELF, &r)` (сначала прочитайте страницы `man getrusage`). Она заполняет структуру `r` типа `struct rusage`, чьё поле `r.ru_utime` содержит поле типа `struct timeval`. Оно, в свою очередь, содержит пару значений — потраченные секунды и потраченные миллисекунды. Сравнивая эти значения до преобразования и после, мы можем вычислить время, потраченное на преобразование. Листинг 16-33 показывает пример однократного измерения времени.

Листинг 16-33. `execution_time.c`

```
#include <sys/time.h>
#include <sys/resource.h>
#include <stdio.h>
#include <unistd.h>
#include <stdint.h>
int main(void) {
    struct rusage r;
    struct timeval start;
    struct timeval end;
    getrusage(RUSAGE_SELF, &r );
    start = r.ru_utime;
    for( uint64_t i = 0; i < 100000000; i++ );
    getrusage(RUSAGE_SELF, &r );
    end = r.ru_utime;
    long res = ((end.tv_sec - start.tv_sec) * 1000000L) +
        end.tv_usec - start.tv_usec;
    printf( "Time elapsed in microseconds: %ld\n", res );
    return 0;
}
```

Используйте таблицу для быстрого преобразования из `unsigned char` в `float`.

```
float const byte_to_float[] = {
    0.0f, 1.0f, 2.0f, ..., 255.0f };
```

> **Вопрос 335** Почитайте о методах вычисления доверительного интервала и вычислите 95%-й доверительный интервал для достаточно большого числа измерений.

16.5 Итоги

В этой главе мы поговорили об оптимизациях компилятора и о том, зачем они нужны. Мы увидели, насколько далеко транслированный оптимизированный код может уйти от своей исходной версии. Затем мы изучили, как извлечь максимум пользы из кэширования и как распараллеливать вычисления с плавающей точкой на уровне инструкций с помощью инструкций SSE. В следующей главе мы увидим, как распараллеливать выполнение последовательностей инструкций, создавать несколько потоков и как меняется наше представление о памяти в присутствии многопоточности.

> **Вопрос 336** Какие опции GCC глобально управляют параметрами оптимизации?

> **Вопрос 337** Какие виды оптимизаций потенциально приносят больше всего пользы?

> **Вопрос 338** Какие преимущества и недостатки может принести отказ от указателя кадра?

> **Вопрос 339** Чем хвостово рекурсивная функция отличается от обычной рекурсивной функции?

> **Вопрос 340** Можно ли переписать любую рекурсивную функцию как хвостово рекурсивную без использования дополнительных структур данных?

> **Вопрос 341** Что такое устранение общих подвыражений? Как это влияет на то, как мы пишем код?

> **Вопрос 342** Что такое распространение констант?

> **Вопрос 343** Зачем нужно помечать функции `static`, когда это возможно, чтобы помочь оптимизациям компилятора?

> **Вопрос 344** Какую пользу приносит оптимизация именованного возвращаемого значения?

Вопрос 345 Что такое предсказание переходов?

Вопрос 346 Что такое динамическое предсказание переходов, глобальные и локальные таблицы истории?

Вопрос 347 Проверьте заметки о предсказании переходов для вашего CPU в [16].

Вопрос 348 Что такое исполнительный блок и почему они для нас важны?

Вопрос 349 Как связаны скорость инструкций AVX и количество исполнительных блоков?

Вопрос 350 Какие шаблоны доступа к памяти хороши?

Вопрос 351 Зачем у нас несколько уровней кэша?

Вопрос 352 В каких случаях `prefetch` может дать выигрыш в производительности и почему?

Вопрос 353 Для чего используются инструкции SSE?

Вопрос 354 Почему большинству инструкций SSE нужны выровненные операнды?

Вопрос 355 Как скопировать данные из регистров общего назначения в регистры `xmm`?

Вопрос 356 В каких случаях стоит использовать инструкции SIMD?

Многопоточность

В этой главе мы рассмотрим возможности многопоточности, предоставляемые языком C. Многопоточность — тема для отдельной книги, так что мы сосредоточимся на возможностях языка и соответствующих свойствах абстрактной машины, а не на хороших практиках и вопросах архитектуры программ. До C11 поддержка многопоточности была внешней по отношению к самому языку — через библиотеки и нестандартные трюки. Часть этого (атомарные операции) теперь реализована во многих компиляторах и предоставляет соответствующий стандарту способ писать многопоточные приложения. К сожалению, по сей день поддержка самих потоков не реализована в большинстве наборов инструментов, так что мы будем использовать библиотеку `pthreads` для записи примеров кода. При этом мы будем использовать соответствующие стандарту атомарные операции. Эта глава ни в коем случае не исчерпывающее руководство по многопоточному программированию — теме, достойной отдельной книги, — но она познакомит с самыми важными понятиями и соответствующими возможностями языка. Если вы хотите стать в этом мастером, мы рекомендуем много практики, специализированные статьи, книги вроде [34] и код-ревью от более опытных коллег.

17.1 Процессы и потоки

Важно понимать разницу между двумя ключевыми понятиями, фигурирующими в большинстве разговоров о многопоточности: потоками и процессами. *Процесс* — это контейнер ресурсов, собирающий всевозможную информацию времени выполнения и ресурсы, нужные программе для выполнения. Процесс содержит следующее:

- Адресное пространство, частично заполненное исполняемым кодом, данными, разделяемыми библиотеками, другими отображёнными файлами и т. д. Части его могут быть общими с другими процессами.
- Все прочие виды связанного состояния, такие как открытые дескрипторы файлов, регистры и т. д.
- Информацию, такую как ID процесса, ID группы процессов, ID пользователя, ID группы...
- Другие ресурсы, используемые для межпроцессного взаимодействия: каналы, семафоры, очереди сообщений...

Поток — это поток инструкций, который операционная система может планировать для выполнения. Операционная система планирует не процессы, а потоки. Каждый поток живёт как часть процесса и обладает частью состояния процесса, принадлежащей только ему.

- Регистры.

- Стек (технически он определяется регистром указателя стека; однако, поскольку все потоки процесса разделяют одно и то же адресное пространство, один из них может обратиться к стекам других потоков, хотя это редко бывает хорошей идеей).
- Свойства, важные для планировщика, такие как приоритет.
- Ожидающие и заблокированные сигналы.
- Маска сигналов.

Когда процесс закрывается, освобождаются все связанные с ним ресурсы, включая все его потоки, открытые дескрипторы файлов и т. д.

17.2 Что делает многопоточность сложной?

Многопоточность позволяет задействовать несколько ядер процессора (или несколько процессоров) для одновременного выполнения потоков. Например, если один поток читает файл с диска (что очень медленная операция), другой может использовать эту паузу для выполнения ресурсоёмких вычислений, распределяя нагрузку CPU (центрального процессора) более равномерно во времени. Так что это может быть быстрее, если ваша программа способна из этого извлечь выгоду. Потокам часто приходится работать с одними и теми же данными. Пока данные не изменяются ни одним из них, работать с ними не составляет проблем, потому что чтение данных никак не влияет на выполнение других потоков. Однако если общие данные изменяются одним (или несколькими) потоками, мы сталкиваемся с несколькими проблемами, такими как:

- Когда поток *A* увидит изменения, выполненные *B*?
- В каком порядке потоки изменяют данные? (Как мы видели в Главе 16,

инструкции могут быть переупорядочены в целях оптимизации.)

- Как нам выполнять операции над сложными фрагментами данных без вмешательства

других потоков? Когда эти проблемы не решены должным образом, возникает весьма проблемный род ошибок, который трудно отловить (потому что он проявляется лишь изредка, когда инструкции разных потоков выполняются в определённом, невезучем порядке). Мы попробуем разобраться в этих проблемах и изучить, как их можно решить.

17.3 Порядок выполнения

Когда мы начинали изучать абстрактную машину C, мы привыкли думать, что последовательность операторов C соответствует действиям, выполняемым скомпилированными машинными инструкциями — естественно, в том же порядке. Теперь пора углубиться в более прагматичные детали того, почему на самом деле это не так. Мы склонны описывать алгоритмы так, чтобы их было проще понять, и это почти всегда хорошо. Однако порядок, заданный программистом, не всегда оптимален с точки зрения производительности. Например, компилятор может захотеть улучшить *локальность*, не меняя семантику кода. Листинг 17-1 показывает пример.

Листинг 17-1. `ex_locality_src.c`

```
char x[1000000], y[1000000];
...
x[4] = y[4];
x[10004] = y[10004];
x[5] = y[5];
x[10005] = y[10005];
```

Листинг 17-2 показывает возможный результат трансляции.

Листинг 17-2. `ex_locality_asm1.asm`

```
mov al,[rsi + 4]
mov [rdi+4],al
mov al,[rsi + 10004]
mov [rdi+10004],al
mov al,[rsi + 5]
mov [rdi+5],al
mov al,[rsi + 10005]
mov [rdi+10005],al
```

Однако очевидно, что этот код можно переписать так, чтобы обеспечить лучшую локальность, то есть сначала присвоить `x[4]` и `x[5]`, затем `x[10004]` и `x[10005]`, как показано в листинге 17-3.

Листинг 17-3. `ex_locality_asm2.asm`

```
mov al,[rsi + 4]
mov [rdi+4],al
mov al,[rsi + 5]
mov [rdi+5],al
mov al,[rsi + 10004]
mov [rdi+10004],al
mov al,[rsi + 10005]
mov [rdi+10005],al
```

Эффекты этих двух последовательностей инструкций схожи, *если абстрактная машина учитывает только один CPU*: при любом начальном состоянии машины итоговое состояние после их выполнения будет одинаковым. Второй результат трансляции часто работает быстрее, так что компилятор может предпочесть его. Это простой случай *переупорядочивания памяти (memory reordering)* — ситуации, когда обращения к памяти переупорядочиваются по сравнению с исходным кодом. Для однопоточных приложений, выполняемых «действительно последовательно», мы часто можем ожидать, что порядок операций не важен, пока наблюдаемое поведение остаётся неизменным. Эта свобода заканчивается, как только мы начинаем обмениваться данными между потоками. Большинство неопытных программистов особо об этом не задумываются, потому что ограничиваются однопоточным программированием. В наши дни мы больше не можем позволить себе не думать о параллелизме — настолько он всепроникающ и часто является единственным, что реально может повысить производительность программы. Так что в этой главе мы поговорим о переупорядочиваниях памяти и о том, как правильно их настроить.

17.4 Строгая и слабая модели памяти

Переупорядочивания памяти может выполнять компилятор (как показано выше) или сам процессор, на уровне микрокода. Обычно выполняются оба, и нас будут интересовать оба вида. Для всех них можно создать единообразную классификацию. *Модель памяти* говорит нам, каких видов переупорядочивания инструкций `load` и `store` можно ожидать. Нас большую часть времени не интересуют точные инструкции, используемые для обращения к памяти (`mov`, `movq` и т. д.), важен только сам факт чтения или записи в память. Есть два крайних полюса: слабая и строгая модели памяти. Точно так же, как с сильной и слабой типизацией, большинство существующих соглашений находятся где-то между ними, ближе к одному или другому полюсу. Мы сочли полезной классификацию, созданную Джеффом Прешингом [31], и будем придерживаться её в этой книге. Согласно ей, модели памяти можно разделить на четыре категории, перечисленные от более слабых к более строгим.

1. Действительно слабые.

В этих моделях может произойти любое переупорядочивание памяти (пока, разумеется, наблюдаемое поведение однопоточной программы остаётся неизменным).

2. Слабые с упорядочиванием по зависимости данных (например, аппаратная модель памяти

ARM v7).

Здесь мы говорим об одном конкретном виде зависимости данных: зависимости между `loads`. Она возникает, когда нам нужно извлечь адрес из памяти, а затем использовать его для выполнения ещё одного извлечения, например,

```
Mov rdx, [rbx]
mov rax, [rdx]
```

В C это ситуация, когда мы используем оператор, чтобы обратиться к полю определённой структуры через указатель на эту структуру.

Действительно слабые модели памяти не гарантируют упорядочивание по зависимости данных.

3. Обычно строгие (например, аппаратная модель памяти Intel 64).

Это значит, что гарантируется выполнение всех `stores` в том же порядке, в котором они заданы. Некоторые `loads`, однако, могут быть перемещены.

Intel 64 обычно попадает в эту категорию.

4. Последовательно согласованные.

Это можно описать как то, что вы видите, отлаживая неоптимизированную программу пошагово на одном ядре процессора. Никакого переупорядочивания памяти никогда не происходит.

17.5 Пример переупорядочивания

Листинг 17-4 показывает пример ситуации, когда переупорядочивание памяти может испортить нам день. Здесь два потока выполняют операторы, содержащиеся соответственно в функциях `thread1` и `thread2`.

Листинг 17-4. `mem_reorder_sample.c`

```
int x = 0;
int y = 0;
void thread1(void) {
    x = 1;
    print(y);
}
void thread2(void) {
    y = 1;
    print(x);
}
```

Оба потока разделяют переменные `x` и `y`. Один из них выполняет запись в `x`, а затем читает значение `y`, тогда как другой делает то же самое, но с `y` и `x` наоборот. Нас интересуют два типа обращений к памяти: `load` и `store`. В наших примерах мы для простоты часто будем опускать все остальные действия. Поскольку эти инструкции полностью независимы (они оперируют разными данными), их можно переупорядочить внутри каждого потока, не меняя наблюдаемого поведения, что даёт нам четыре варианта: `store + load` или `load + store` для каждого из двух потоков. Это то, что компилятор может сделать по своим собственным причинам. Для каждого варианта существует *шесть* возможных порядков выполнения. Они изображают, как оба потока продвигаются во времени относительно друг друга. Мы показываем их как последовательности 1 и 2; если шаг сделал первый поток, мы пишем 1; иначе шаг сделал второй.

1. 1-1-2-2
2. 1-2-1-2
3. 2-1-1-2
4. 2-1-2-1
5. 2-2-1-1
6. 1-2-2-1

Например, 1-1-2-2 означает, что первый процесс выполнил два шага, а затем то же самое сделал второй процесс. Каждая последовательность соответствует четырём разным сценариям. Например, последовательность 1-2-1-2 кодирует одну из трасс, показанных в таблице 17-1:

Таблица 17-1. Возможные последовательности выполнения инструкций, если процессы чередуются как 1-2-1-2

ID ПОТОКА	ТРАССА 1	ТРАССА 2	ТРАССА 3	ТРАССА 4
1	store x	store x	load y	load y
2	store y	load x	store y	load x
1	load y	load y	store x	store x
2	load x	store y	load x	store y

Если рассмотреть все возможные трассы для каждого порядка выполнения, мы получим *24* сценария (некоторые из которых будут эквивалентны). Как видите, даже для небольших примеров эти числа могут быть достаточно большими. На самом деле все эти возможные трассы нам не нужны; нас интересует позиция `load` относительно `store` для каждой переменной. Даже в таблице 17-1 присутствует много возможных комбинаций: и `x`, и `y` могут быть сохранены, а затем прочитаны, или прочитаны, а затем сохранены. Очевидно, результат `load` зависит от того, была ли перед ним `store`. Не будь переупорядочиваний в игре, мы были бы ограничены: любому из двух указанных `loads` должна была предшествовать запись, потому что так написано в исходном коде; планирование инструкций иным образом не может этого изменить. Однако, поскольку переупорядочивания присутствуют, иногда можно получить интересный исход: если у обоих этих потоков инструкции переупорядочены, мы приходим к ситуации, показанной в листинге 17-5.

Листинг 17-5. `mem_reorder_sample_happened.c`

```
int x = 0;
int y = 0;
void thread1(void) {
    print(y);
    x = 1;
}
void thread2(void) {
    print(x);
    y = 1;
}
```

Если выбрана стратегия 1-2-- (где * обозначает любой из потоков), мы сначала выполняем `load x` и `load y`, из-за чего они будут казаться равными 0 для всех, кто использует результаты этих `loads`. Это действительно возможно, если *компилятор* переупорядочил эти операции. Но даже если это хорошо контролируется или отключено, переупорядочи-

вания памяти, выполняемые CPU, всё ещё могут дать такой эффект. Этот пример демонстрирует, что исход такой программы крайне непредсказуем. Позже мы изучим, как ограничить переупорядочивания компилятором и CPU; мы также приведём код, демонстрирующий это переупорядочивание в оборудовании.

17.6 Что является volatile, а что нет

Используемая нами модель памяти C довольно слабая. Рассмотрим следующий код:

```
int x,y; x = 1;
y = 2;
x = 3;
```

Как мы уже видели, компилятор может переупорядочить инструкции. Более того, компилятор может вывести, что первое присваивание — мёртвый код, потому что за ним следует другое присваивание той же переменной `x`. Поскольку оно бесполезно, компилятор может даже удалить этот оператор. Ключевое слово `volatile` решает эту проблему. Оно заставляет компилятор никогда не оптимизировать записи и чтения указанной переменной, а также подавляет любые возможные переупорядочивания инструкций. Однако оно накладывает эти ограничения только на одну конкретную переменную и не даёт никаких гарантий о порядке, в котором выполняются записи в разные volatile-переменные. Например, в предыдущем коде даже изменение типа и `x`, и `y` на `volatile int` наложит порядок на присваивания каждой из них по отдельности, но всё ещё позволит нам свободно чередовать записи, как показано:

```
volatile int x, y;
x = 1;
x = 3;
y = 2;
```

Или вот так:

```
volatile int x, y;
y = 2;
x = 1;
x = 3;
```

Очевидно, этих гарантий недостаточно для многопоточных приложений. Нельзя использовать переменные `volatile`, чтобы организовать доступ к общим данным, потому что эти обращения всё ещё можно достаточно свободно переставлять. Чтобы безопасно

обращаться к общим данным, нам нужны две гарантии.

- Чтение или запись действительно происходит. Компилятор мог бы просто закэшировать значение в регистре и никогда не записать его обратно в память. Это гарантия, которую предоставляет `volatile`. Её достаточно для выполнения ввода/вывода, отображаемого на память, но не для многопоточных приложений.
- Никакого переупорядочивания памяти не должно происходить. Представим, что мы используем переменную `volatile` как флаг, указывающий, что некоторые данные готовы к чтению. Код подготавливает данные, а затем устанавливает флаг; однако переупорядочивание может поместить это присваивание до подготовки данных. Здесь имеют значение и аппаратные переупорядочивания, и переупорядочивания компилятора. Эта гарантия переменными `volatile` не предоставляется.

В этой главе мы изучим два механизма, предоставляющих обе следующие гарантии:

- Барьеры памяти.
- Атомарные переменные, введённые в C11.

Переменные volatile используются крайне редко. Они подавляют оптимизацию, чего обычно делать не хочется.

17.7 Барьеры памяти

Барьер памяти — специальная инструкция или оператор, накладывающий ограничения на то, как могут выполняться переупорядочивания. Как мы видели в Главе 16, компиляторы или оборудование могут использовать множество трюков для повышения производительности в среднем случае, включая переупорядочивание, отложенные операции с памятью, спекулятивные загрузки или предсказание переходов, кэширование переменных в регистрах и т. д. Контролировать их жизненно важно, чтобы гарантировать, что определённые операции уже выполнены, потому что от этого зависит логика другого потока. В этом разделе мы хотим представить разные виды барьеров памяти и дать общее представление об их возможных реализациях в Intel 64. Пример барьера памяти, предотвращающего переупорядочивания *компилятором*, — следующая директива GCC:

```
asm volatile("" ::: "memory")
```

Директива `asm` используется для включения встроенного ассемблерного кода прямо в программы на C. Ключевое слово `volatile` вместе с ограничением `"memory"` описывает, что этот (пустой) фрагмент встроенного ассемблера нельзя оптимизировать прочь или переместить и что он выполняет чтение и/или запись в память. Из-за этого компилятор вынужден сгенерировать код, фиксирующий все операции в памяти (например, сохраняющий значения локальных переменных, закэшированных в регистрах).

Это не мешает процессору выполнять спекулятивные чтения после этого оператора, так что это *не* барьер памяти для самого процессора. Очевидно, барьеры памяти и для компилятора, и для CPU *дорогостоящи*, потому что они мешают оптимизациям. Именно поэтому мы не хотим использовать их после каждой инструкции. Есть несколько видов барьеров памяти. Мы поговорим о тех, что определены в документации ядра Linux, но эта классификация применима в большинстве ситуаций.

1. Барьер памяти для записи. Он гарантирует, что все операции `store`, указанные в коде до барьера, будут казаться произошедшими раньше всех операций `store`, указанных после барьера. GCC использует `asm volatile("":::"memory")` как общий барьер памяти. Intel 64 использует инструкцию `sfence`.
2. Барьер памяти для чтения.

Аналогично, он гарантирует, что все операции `load`, указанные в коде до барьера, будут казаться произошедшими раньше всех операций `load`, указанных после барьера. Это более строгая форма барьера зависимости данных.

GCC использует `asm volatile("":::"memory")` как общий барьер памяти. Intel 64 использует инструкцию `lfence`.

3. Барьеры зависимости данных.

Барьер зависимости данных учитывает зависимые чтения, описанные в разделе 17.4. Его можно считать более слабой формой барьера чтения памяти. Никаких гарантий о независимых `load` или о `store` любого вида не предоставляется.

4. Общие барьеры памяти

Это предельный барьер, заставляющий зафиксировать каждое изменение памяти, указанное в коде до него. Он также не даёт всем последующим операциям переупорядочиться так, чтобы казаться выполненными до барьера.

GCC использует `asm volatile("":::"memory")` как общий барьер памяти. Intel 64 использует инструкцию `mfence`.

5. Операции acquire.

Это класс операций, объединённых свойством, называемым *семантикой acquire*. Если операция выполняет *чтения* из общей памяти и гарантированно не переупорядочивается со *следующими за ней чтениями и записями* в исходном коде, говорят, что у неё есть это свойство.

Другими словами, это похоже на общий барьер памяти, но следующий за ним код не будет переупорядочен так, чтобы выполниться до этого барьера.

6. Операции release.

Семантика release — свойство таких операций. Если операция выполняет *записи* в общую память и гарантированно не переупорядочивается с *предыдущими чтениями и записями* в исходном коде, говорят, что у неё есть это свойство.

Другими словами, это похоже на общий барьер памяти, но всё же *позволяет* более поздним операциям переупорядочиться на позицию перед операцией release. Таким образом, операции acquire и release — своего рода однонаправленные барьеры для переупорядочиваний. Ниже пример одной ассемблерной команды `mfence`, встроенной GCC:

```
asm ("mfence" )
```

Комбинируя её с барьером компилятора, мы получаем строку, которая и предотвращает переупорядочивание компилятором, и действует как полный барьер памяти.

```
asm volatile("mfence" ::: "memory")
```

Любой вызов функции, *чьё определение недоступно в текущей единице трансляции* и которая *не является интринсиком* (кроссплатформенной заменой конкретной ассемблерной инструкции), — это *барьер памяти компилятора*.

17.8 Введение в pthreads

POSIX threads (`pthreads`) — стандарт, описывающий определённую модель выполнения программы. Он предоставляет средства для параллельного выполнения кода и управления этим выполнением. Он реализован как библиотека `pthreads`, которую мы будем использовать на протяжении этой главы. Библиотека содержит типы C, константы и процедуры (с префиксом `pthread_`). Их объявления доступны в заголовке `pthread.h`. Предоставляемые ей функции делятся на следующие группы:

- Базовое управление потоками (создание, уничтожение).
- Управление мьютексами.
- Условные переменные.
- Синхронизация с помощью блокировок и барьеров.

В этом разделе мы рассмотрим несколько примеров, чтобы освоиться с `pthreads`. Для выполнения многопоточных вычислений у вас есть следующие два варианта:

- Породить несколько потоков в одном процессе. Потоки разделяют одно и то же адресное пространство, так что обмен данными относительно прост и быстр. Когда процесс завершается, завершаются и все его потоки.

- Породить несколько процессов; у каждого из них свой поток по умолчанию. Эти потоки общаются через механизмы, предоставляемые операционной системой (такие как каналы). Это не так быстро; также порождение процесса медленнее, чем порождение просто потока, потому что создаёт больше структур операционной системы (и отдельное адресное пространство). Взаимодействие между процессами часто подразумевает одну или несколько (иногда неявных) операций копирования. Однако разделение логики программы на отдельные процессы может положительно сказаться на безопасности и надёжности, потому что каждый поток видит только открытую часть процессов, отличных от своего собственного.

`Pthreads` позволяет породить несколько потоков в одном процессе, и это обычно то, что вам нужно.

17.8.1 Когда использовать многопоточность

Иногда многопоточность удобна для логики программы. Например, обычно не стоит принимать сетевой пакет и рисовать графический интерфейс в одном и том же потоке. Графический интерфейс должен реагировать на действия пользователя (клики по кнопкам) и постоянно перерисовываться (например, когда соответствующее окно перекрывается другим окном, а затем открывается снова). Сетевое же действие заблокирует рабочий поток, пока не завершится. Поэтому удобно разделить эти действия на разные потоки, чтобы выполнять их как бы одновременно. Многопоточность естественным образом может улучшить производительность, но не всегда. Есть задачи, ограниченные CPU (CPU bound), и задачи, ограниченные вводом/выводом (IO bound).

- Код, ограниченный CPU, можно ускорить, дав ему больше времени CPU. Он тратит большую часть времени CPU на вычисления, а не на чтение данных с диска или взаимодействие с устройствами.
- Код, ограниченный вводом/выводом, нельзя ускорить, дав ему больше времени CPU, потому что его замедляет чрезмерное использование памяти или внешних устройств.

Использование многопоточности для ускорения программ, ограниченных CPU, может быть полезным. Распространённый шаблон — использовать очередь с запросами, распределяемыми между рабочими потоками из пула потоков — набора созданных потоков, которые либо работают, либо ждут работы, но не пересоздаются каждый раз, когда в них появляется потребность. Подробнее см. Главу 7 [23]. Что до того, сколько потоков нам нужно, универсального рецепта нет. Создание потоков, переключение между ними и их планирование создают накладные расходы. Это может замедлить всю программу, если для потоков не так уж много работы. В вычислительно тяжёлых задачах некоторые советуют порождать $n - 1$ потоков, где n — общее число ядер процессора. В задачах, последовательных по своей природе (где каждый шаг напрямую зависит от предыдущего),

порождение нескольких потоков не поможет. Мы рекомендуем *всегда* экспериментировать с количеством потоков при разных нагрузках, чтобы выяснить, какое число больше всего подходит для данной задачи.

17.8.2 Создание потоков

Создавать потоки легко. Листинг 17-6 показывает пример.

Листинг 17-6. `pthread_create_mwe.c`

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
void* threadimpl( void* arg ) {
    for(int i = 0; i < 10; i++ ) {
        puts( arg );
        sleep(1);
    }
    return NULL;
}
int main( void ) { pthread_t t1, t2;
    pthread_create( &t1, NULL, threadimpl, "fizz" );
    pthread_create( &t2, NULL, threadimpl, "buzzzz" );
    pthread_exit( NULL );
    puts("bye");
    return 0;
}
```

Заметьте, что код, использующий библиотеку `pthread`, нужно компилировать с флагом `-pthread`, например,

```
> gcc -O3 -pthread main.c
```

Указание `-lpthread` не даст желаемого результата. Компоновки с одной лишь `libpthread.a` недостаточно: флаг `-pthread` включает несколько опций препроцессора (например, `_REENTRANT`). Так что всегда, когда доступна опция `-pthread`,[39] используйте её.

Изначально есть только один поток, начинающий выполнение функции `main`. Тип `pthread_t` хранит всю информацию о каком-то другом потоке, так что мы можем управлять им, используя этот экземпляр как хендл. Затем потоки инициализируются функцией `pthread_create` со следующей сигнатурой:

```
int pthread_create(
    pthread_t *thread,
    const pthread_attr_t *attr,
    void *(*start_routine) (void *),
    void *arg);
```

Первый аргумент — указатель на инициализируемый экземпляр `pthread_t`. Второй — набор атрибутов, которых мы коснёмся позже — пока безопасно передать вместо него `NULL`. Стартовая функция потока должна принимать указатель и возвращать указатель. Она принимает указатель `void*` на свой аргумент. Разрешён только один аргумент; однако вы легко можете создать структуру или массив, инкапсулирующий несколько аргументов, и передать указатель на него. Возвращаемое значение `start_routine` тоже указатель и может использоваться для возврата результата работы потока.[40] Последний аргумент — фактический указатель на аргумент, который будет передан функции `start_routine`. В нашем примере каждый поток реализован одинаково: он принимает указатель (на строку), а затем многократно выводит её с интервалом примерно в одну секунду. Функция `sleep`, объявленная в `unistd.h`, приостанавливает текущий поток на заданное число секунд. После десяти итераций поток возвращается. Это эквивалентно вызову функции `pthread_exit` с аргументом. Возвращаемое значение обычно является результатом вычислений, выполненных потоком; верните `NULL`, если он вам не нужен. Позже мы увидим, как можно получить это значение из родительского потока.

> **Приведение к `void`** Конструкции вроде `(void)argc` служат лишь одной цели: подавить предупреждения о неиспользуемой переменной или аргументе `argc`. Иногда их можно встретить в исходном коде.

Однако наивный `return` из функции `main` приведёт к завершению процесса. А что, если другие потоки всё ещё существуют? Главному потоку следует сначала дождаться их завершения! Именно это делает `pthread_exit`, когда её вызывают *в главном потоке*: она ждёт завершения всех остальных потоков, а затем завершает программу. Весь следующий код выполнен не будет, так что вы не увидите сообщение `bye` в `stdout`. Эта программа десять раз выведет пару строк `buzz` и `fizz` в случайном порядке, а затем завершится. Невозможно предсказать, какой поток — первый или второй — будет запланирован первым, так что каждый раз порядок может отличаться. Листинг 17-7 показывает пример вывода.

Листинг 17-7. `pthread_create_mwe_out`

```
> ./main
fizz
buzzzz
buzzzz
fizz
fizz
buzzzz
fizz
buzzzz
fizz
```

```
buzzzz
buzzzz
fizz
buzzzz
fizz
buzzzz
fizz
buzzzz
fizz
buzzzz
fizz
```

Как видите, строка `bye` не выводится, потому что соответствующий вызов `puts` находится ниже вызова `pthread_exit`.

> **Где находятся аргументы?** Важно отметить, что указатель на аргумент, переданный потоку, должен указывать на данные, остающиеся живыми до завершения этого самого потока. Передавать указатель на переменную, выделенную на стеке, может быть рискованно, поскольку после уничтожения кадра стека функции обращение к этим освобождённым переменным даёт неопределённое поведение. Если только аргументы не гарантированно константны (или вы не собираетесь использовать их для целей синхронизации), не передавайте их разным потокам. В примере, показанном в листинге 17-6, строки, принимаемые `threadimpl`, размещены в глобальной

> **Примечание.** памяти только для чтения (**.rodata**). Так что передавать указатель на неё безопасно.

Максимальное число порождаемых потоков зависит от реализации. В Linux, например, можно использовать `ulimit -a`, чтобы получить соответствующую информацию. Потоки могут создавать другие потоки; ограничений на это нет. Реализация `pthreads` действительно гарантирует, что вызов `pthread_create` действует как полный барьер памяти компилятора, а также полный аппаратный барьер памяти. `pthread_attr_init` используется для инициализации экземпляра непрозрачного типа `pthread_attr_t` (реализованного как *неполный тип*). Атрибуты предоставляют дополнительные параметры для потоков, такие как размер стека или адрес. Следующие функции используются для установки атрибутов:

- `pthread_attr_setaffinity_np` – поток будет предпочитать выполняться на конкретном

ядре CPU.

- `pthread_attr_setdetachstate` – сможем ли мы вызвать `pthread_join` для этого потока,

или он будет отсоединённым (detached, в противоположность joinable). Назначение `pthread_join` будет объяснено в следующем разделе.

- `pthread_attr_setguardsize` – устанавливает пространство перед границей стека как область

запрещённых адресов заданного размера, чтобы отлавливать переполнения стека.

- `pthread_attr_setinheritsched` – наследуются ли следующие два параметра от

родительского потока (того, где произошло создание), или берутся из самих атрибутов?

- `pthread_attr_setschedparam` – прямо сейчас касается только приоритета планирования, но

в будущем могут быть добавлены дополнительные параметры.

- `pthread_attr_setschedpolicy` – как будет выполняться планирование. Политики планирования с соответствующими описаниями можно увидеть в `man sched`.
- `pthread_attr_setscope` – относится к системе области конкуренции (contention scope), определяющей набор потоков, с которыми этот поток будет конкурировать за CPU (или другие ресурсы).
- `pthread_attr_setstackaddr` – где будет расположен стек?
- `pthread_attr_setstacksize` – каким будет размер стека потока?
- `pthread_attr_setstack` – устанавливает и адрес стека, и размер стека.

У всех них есть аналоги «get» (например, `pthread_attr_getscope`).

> **Вопрос 357** Прочитайте страницы `man` для перечисленных выше функций.

```
■Question 358What will sysconf(_SC_NPROCESSORS_ONLN) return?
```

17.8.3 Управление потоками

Того, что мы уже узнали, достаточно для параллельного выполнения работы. Однако у нас пока нет средств синхронизации, так что, распределив работу между потоками, мы не можем по-настоящему использовать результаты вычислений одного потока в других потоках. Простейшая форма синхронизации — присоединение потока (thread joining). Идея проста: вызывая `thread_join` для экземпляра `pthread_t`, мы переводим текущий поток в состояние ожидания, пока другой поток не завершится. Листинг 17-8 показывает пример.

Листинг 17-8. `thread_join_mwe.c`

```
#include <pthread.h>
#include <unistd.h>
#include <stdio.h>
void* worker( void* param ) {
    for( int i = 0; i < 3; i++ ) {
        puts( (const char*) param );
        sleep(1);
    }
    return (void*)"done";
}
int main( void ) {
    pthread_t t;
    void* result;
    pthread_create( &t, NULL, worker, (void*) "I am a worker!" );
    pthread_join( t, &result );
    puts( (const char*) result );
    return 0;
}
```

`thread_join` принимает два аргумента: сам поток и адрес переменной `void*`, которая будет инициализирована результатом выполнения потока. Присоединение потока действует как полный барьер, потому что перед присоединением нельзя размещать никакие чтения или записи, запланированные на после него. По умолчанию потоки создаются присоединяемыми (joinable), но можно создать и отсоединённый (detached) поток. Это может принести определённую выгоду: ресурсы отсоединённого потока могут быть освобождены сразу после его завершения. Присоединяемый же поток будет ждать присоединения, прежде чем его ресурсы смогут быть освобождены. Чтобы создать отсоединённый поток

- Создайте экземпляр атрибутов `pthread_attr_t attr` ;
- Инициализируйте его с помощью pthread_attr_init(&attr);
- Вызовите pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED); и
- Создайте поток с помощью `pthread_create` , передав `&attr` как аргумент атрибутов.

Текущий поток можно явно перевести из присоединяемого в отсоединённый, вызвав `pthread_detach()` . Сделать обратное невозможно.

17.8.4 Пример: распределённая факторизация

Мы выбрали простую программу, ограниченную CPU, — подсчёт делителей числа. Сначала мы решим её самым тривиальным методом полного перебора на одном ядре. Листинг 17-9 показывает код.

Листинг 17-9. `dist_fact_sp.c`

```
#include <pthread.h>
#include <unistd.h>
#include <inttypes.h>
#include <stdio.h>
#include <malloc.h>
uint64_t factors( uint64_t num ) {
    uint64_t result = 0;
    for (uint64_t i = 1; i <= num; i++ )
        if ( num % i == 0 ) result++;
    return result;
}
int main( void ) {
    /* volatile to prevent constant propagation */
    volatile uint64_t input = 2000000000;
    printf( "Factors of %"PRIu64": %"PRIu64"\n", input, factors(input) );
    return 0;
}
```

Код довольно прост: мы наивно перебираем все числа от 1 до входного значения и проверяем, являются ли они делителями. Заметьте, что входное значение помечено volatile, чтобы предотвратить вычисление всего результата во время компиляции. Скомпилируйте код следующей командой:

```
> gcc -O3 -std=c99 -o fact_sp dist_fact_sp.c
```

Мы начнём распараллеливание с упрощённой версии многопоточного кода, которая всегда будет выполнять вычисления в двух потоках и не будет архитектурно красивой. Листинг 17-10 её показывает.

Листинг 17-10. `dist_fact_mp_simple.c`

```
#include <pthread.h>
#include <inttypes.h>
#include <stdio.h>
int input = 0;
int result1 = 0;
void* fact_worker1( void* arg ) {
    result1 = 0;
    for( uint64_t i = 1; i < input/2; i++ )
        if ( input % i == 0 ) result1++;
    return NULL;
}
int result2 = 0;
void* fact_worker2( void* arg ) {
    result2 = 0;
    for( uint64_t i = input/2; i <= input; i++ )
        if ( input % i == 0 ) result2++;
    return NULL;
}
uint64_t factors_mp( uint64_t num ) {
    input = num;
    pthread_t thread1, thread2;
    pthread_create( &thread1, NULL, fact_worker1, NULL );
    pthread_create( &thread2, NULL, fact_worker2, NULL );
    pthread_join( thread1, NULL );
    pthread_join( thread2, NULL );
    return result1 + result2;
}
int main( void ) {
    uint64_t input = 2000000000;
    printf( "Factors of %"PRIu64": %"PRIu64"\n",
            input, factors_mp(input ));
    return 0;
}
```

При запуске он даёт тот же результат, что хороший знак.

```
Factors of 2000000000: 110
```

Что делает эта программа? Что ж, мы разбиваем диапазон (0*, n*] на две половины. Два рабочих потока вычисляют количество делителей в своих соответствующих половинах. Затем, когда оба присоединены, мы гарантированно знаем, что они уже выполнили все вычисления. Результаты остаётся только сложить. Затем, в листинге 17-11, мы покажем многопоточную программу, использующую произвольное число потоков для вычисления того же результата. У неё более продуманная архитектура.

Листинг 17-11. `dist_fact_mp.c`

```
#include <pthread.h>
#include <unistd.h>
#include <inttypes.h>
#include <stdio.h>
#include <malloc.h>
#define THREADS 4
struct fact_task {
    uint64_t num;
    uint64_t from, to;
    uint64_t result;
};
void* fact_worker( void* arg ) {
    struct fact_task* const task =  arg;
    task-> result = 0;
    for( uint64_t i = task-> from; i < task-> to; i++ )
        if ( task->num %  i ==  0 ) task-> result ++;
    return NULL;
}
/* assuming threads_count < num */
uint64_t factors_mp( uint64_t num, size_t threads_count ) {
    struct fact_task* tasks = malloc( threads_count * sizeof( *tasks ) );
    pthread_t* threads = malloc( threads_count * sizeof( *threads ) );
    uint64_t start = 1;
    size_t step = num / threads_count;
    for( size_t i = 0; i < threads_count; i++ ) {
        tasks[i].num = num;
        tasks[i].from = start;
        tasks[i].to = start + step;
        start += step;
    }
    tasks[threads_count-1].to = num+1;
    for ( size_t i = 0; i < threads_count; i++ )
        pthread_create( threads + i, NULL, fact_worker, tasks + i );
    uint64_t result = 0;
    for ( size_t i = 0; i < threads_count; i++ ) {
        pthread_join( threads[i], NULL );
        result  +=  tasks[i].result;
    }
    free( tasks );
    free( threads );
    return result;
}
int main( void ) {
```

```
    uint64_t input = 2000000000;
    printf( "Factors of %"PRIu64": %"PRIu64"\n",
            input, factors_mp(input, THREADS ) );
    return 0;
}
```

Предположим, мы используем t потоков. Чтобы посчитать количество делителей n, мы разбиваем диапазон от 1 до n на t равных частей. Мы вычисляем количество делителей в каждом из этих интервалов, а затем складываем результаты. Мы создаём тип для хранения информации об отдельной задаче, называемый `struct fact_task`. Он включает само число, границы диапазона `from` и `to`, а также слот для результата, который будет количеством делителей `num` между `from` и `to`. Все обработчики, вычисляющие количество делителей, реализованы одинаково, как процедура `fact_worker`, принимающая указатель на `struct fact_task`, вычисляющая количество делителей и заполняющая поле `result`. Код, выполняющий запуск потоков и сбор результатов, находится в функции `factors_mp`, которая для заданного числа потоков

- Выделяет описания задач и экземпляры потоков;
- Инициализирует описания задач;
- Запускает все потоки;
- Ждёт завершения выполнения каждого потока с помощью `join` и прибавляет его результат к общему аккумулятору `result`; и
- Освобождает всю выделенную память.

Так что мы помещаем создание потоков в чёрный ящик, что позволяет нам извлечь выгоду из многопоточности. Этот код можно скомпилировать следующей командой:

```
> gcc -O3 -std=c99 -pthread -o fact_mp dist_fact_mp.c
```

Несколько потоков сокращают общее время выполнения на многоядерной системе для этой задачи, ограниченной CPU. Чтобы проверить время выполнения, мы снова воспользуемся утилитой `time` (программой, а не встроенной командой оболочки). Чтобы убедиться, что используется именно программа, а не встроенная команда оболочки, мы добавим перед ней обратный слэш.

```
> gcc -O3 -o sp -std=c99 dist_fact_sp.c && \time ./sp
Factors of 2000000000: 110
21.78user 0.03system 0:21.83elapsed 99%CPU (0avgtext+0avgdata 524maxresident)k
0inputs+0outputs (0major+207minor)pagefaults 0swaps
> gcc -O3 -pthread -o mp -std=c99 dist_fact_mp.c && \time ./mp
Factors of 2000000000: 110
25.48user 0.01system 0:06.58elapsed 387%CPU (0avgtext+0avgdata 656maxresident)k
0inputs+0outputs (0major+250minor)pagefaults 0swaps
```

Многопоточная программа выполнялась 6.5 секунд, тогда как однопоточная версия — почти 22 секунды. Это большое улучшение. Чтобы говорить о производительности, мы введём понятие ускорения. *Ускорение (speedup)* — это улучшение скорости выполнения программы, выполняемой на двух похожих архитектурах с разными ресурсами. Добавляя больше потоков, мы делаем доступным больше ресурсов, отсюда и возможное улучшение. Разумеется, для первого примера мы выбрали задачу, которую легко и эффективно решать параллельно. Ускорение не всегда будет настолько значительным, если оно вообще будет; однако, как мы видим, итоговый код достаточно компактен (мог бы быть даже компактнее, если бы мы не учитывали расширяемость — например, зафиксировав число потоков вместо использования его как параметра).

> **Вопрос 359** Поэкспериментируйте с числом потоков и найдите оптимальное в вашем собственном окружении.

> **Вопрос 360** Прочитайте про функции `pthread_self` и `pthread_equal`. Почему нельзя сравнивать потоки простым оператором равенства `==`?

17.8.5 Мьютексы

Хотя присоединение потоков — доступная техника, она не даёт средств управлять выполнением потока «на ходу». Иногда мы хотим гарантировать, что действия, выполняемые в одном потоке, не выполняются до того, как выполнено некоторое другое действие в других потоках. Иначе мы получим ситуацию, в которой система работает не всегда стабильно: её вывод станет зависеть от фактического порядка, в котором будут выполнены инструкции из разных потоков. Это происходит при работе с изменяемыми данны-

ми, общими для потоков. Такие ситуации называются *гонками данных (data races)*, потому что потоки соревнуются за ресурсы, и любой поток может выиграть и добраться до них первым. Чтобы предотвратить такие ситуации, существует ряд инструментов, и мы начнём с мьютексов. *Мьютекс* (сокращение от «mutual exclusion», взаимное исключение) — объект, который может находиться в двух состояниях: заблокирован и разблокирован. Мы работаем с ними с помощью двух запросов.

- Блокировка. Меняет состояние с разблокированного на заблокированное. Если мьютекс заблокирован, то

пытающийся его заблокировать поток ждёт, пока мьютекс не разблокируют другие потоки.

- Разблокировка. Если мьютекс заблокирован, он становится разблокированным.

Мьютексы часто используются для предоставления эксклюзивного доступа к общему ресурсу (например, общему фрагменту данных). Поток, который хочет работать с ресурсом, блокирует мьютекс, используемый исключительно для контроля доступа к ресурсу. После завершения работы с ресурсом поток разблокирует мьютекс. Блокировка и разблокировка мьютекса действуют и как барьер памяти компилятора, и как полный аппаратный барьер памяти, так что никакие чтения или записи не могут быть переупорядочены до блокировки или после разблокировки. Листинг 17-12 показывает пример программы, которой нужен мьютекс.

Листинг 17-12. `mutex_ex_counter_bad.c`

```
#include <pthread.h>
#include <inttypes.h>
#include <stdio.h>
#include <unistd.h>
pthread_t t1, t2;
uint64_t value = 0;
void* impl1( void* _ ) {
    for (int n = 0; n < 10000000; n++) {
        value +=  1;
    }
    return NULL;
}
int main(void) {
    pthread_create(  &t1, NULL, impl1, NULL );
    pthread_create(  &t2, NULL, impl1, NULL );
    pthread_join( t1, NULL );
    pthread_join( t2, NULL );
    printf( "%"PRIu64"\n", value );
    return 0;
}
```

В этой программе два потока, реализованных функцией `impl1`. Оба потока постоянно увеличивают общую переменную `value 10000000` раз. Эту программу следует скомпилировать с отключёнными оптимизациями, чтобы предотвратить превращение этого цикла инкремента в единственный оператор `value += 10000000` (или можно сделать `value volatile`).

```
gcc -O0 -pthread mutex_ex_counter_bad.c
```

Итоговый вывод, однако, не 20000000, как мы могли бы подумать, и отличается при каждом запуске исполняемого файла:

```
> ./a.out
11297520
> ./a.out
10649679
> ./a.out
13765500
```

Проблема в том, что инкремент переменной не является атомарной операцией с точки зрения C. Сгенерированный ассемблерный код соответствует этому описанию, используя несколько инструкций, чтобы прочитать значение, прибавить единицу, а затем записать его обратно. Это позволяет планировщику передать CPU другому потоку «посреди» выполняющейся операции инкремента. У оптимизированного кода поведение может быть таким же, а может и не быть. Чтобы избежать этого беспорядка, мы используем мьютекс, чтобы предоставить одному потоку исключительное право работать с `value`. Так мы обеспечиваем корректное поведение. Листинг 17-13 показывает изменённую программу.

Листинг 17-13. `mutex_ex_counter_good.c`

```
#include <pthread.h>
#include <inttypes.h>
#include <stdio.h>
#include <unistd.h>
pthread_mutex_t m; //
pthread_t t1, t2;
uint64_t value = 0;
void* impl1( void* _ ) {
    for (int n = 0; n < 10000000; n++) {
        pthread_mutex_lock( &m );//
        value += 1;
        pthread_mutex_unlock( &m );//
    }
    return NULL;
}
int main(void) {
    pthread_mutex_init( &m, NULL );     //
    pthread_create(  &t1, NULL, impl1, NULL );
    pthread_create(  &t2, NULL, impl1, NULL );
    pthread_join( t1, NULL );
    pthread_join( t2, NULL );
    printf( "%"PRIu64"\n", value );
    pthread_mutex_destroy( &m ); //
    return 0;
}
```

Его вывод согласован (хотя вычисляется дольше):

```
> ./a.out
20000000
```

Мьютекс `m` *программистом* связан с общей переменной `value`. Никакие изменения `value` не должны выполняться вне участка кода между блокировкой и разблокировкой `m`. Если это ограничение соблюдается, никакой другой поток не может изменить `value`, пока удерживается блокировка. Блокировка также действует как барьер памяти. Из-за этого `value` будет перечитана после взятия блокировки, и её можно безопасно кэшировать в регистре. Нет необходимости делать переменную `value` volatile, поскольку это лишь подавит оптимизации, а программа и так корректна. Прежде чем использовать мьютекс, его нужно инициализировать с помощью `pthread_mutex_init`, как видно в функции `main`. Она принимает атрибуты, точно так же, как и функция `pthread_create`, которые можно использовать для создания рекурсивного мьютекса, мьютекса с обнаружением взаимоблокировок, управления устойчивостью (что произойдёт, если поток-владелец мьютекса умрёт?) и многого другого. Чтобы избавиться от мьютекса, используется вызов `pthread_mutex_unlock`.

> **Вопрос 361** Что такое рекурсивный мьютекс? Чем он отличается от обычного?

17.8.6 Взаимоблокировки

Одиночный мьютекс редко становится причиной проблем. Однако когда вы блокируете несколько мьютексов одновременно, могут случиться разные странные ситуации. Взгляните на пример, показанный в листинге 17-14.

Листинг 17-14. `deadlock_ex`

```
mutex A, B;
thread1 () {
    lock(A);
    lock(B);
    unlock(B);
    unlock(A);
}
thread2() {
    lock(B);
    lock(A);
    unlock(A);
    unlock(B);
}
```

Этот псевдокод демонстрирует ситуацию, в которой оба потока могут зависнуть навсегда. Представим, что из-за неудачного планирования произошла следующая последовательность действий:

- Поток 1 заблокировал *A*; управление передалось потоку 2.
- Поток 2 заблокировал *B*; управление передалось потоку 1.

После этого потоки попытаются сделать следующее:

- Поток 1 попытается заблокировать *B*, но *B* уже заблокирован потоком 2.
- Поток 2 попытается заблокировать *A*, но *A* уже заблокирован потоком 1.

Оба потока навсегда застрянут в этом состоянии. Когда потоки застревают в заблокированном состоянии, ожидая друг от друга разблокировки, такая ситуация называется *взаимоблокировкой (deadlock)*. Причина взаимоблокировки — разный порядок, в котором разные потоки берут блокировки. Отсюда простое правило, которое в большинстве случаев спасёт нас, когда нужно заблокировать несколько мьютексов одновременно.

> **Предотвращение взаимоблокировок** Упорядочьте все мьютексы в вашей программе в воображаемую последовательность. Блокируйте мьютексы только в том же порядке, в котором они появляются в этой последовательности. Например, предположим, у нас есть мьютексы A, B, C и D. Мы устанавливаем для них естественный порядок: A < B < C < D. Если вам нужно заблокировать и D, и B, всегда блокируйте их в одном и том же порядке, то есть сначала B, затем D.

> **Примечание.** Если этот **инвариант** соблюдается, никакие два потока не заблокируют пару мьютексов в разном порядке.

> **Вопрос 362** Что такое условия Коффмана? Как их можно использовать для диагностики взаимоблокировок?

> **Вопрос 363** Как использовать `Helgrind` для обнаружения взаимоблокировок?

17.8.7 Живые блокировки

Живая блокировка (livelock) — ситуация, в которой два потока застряли, но не в состоянии ожидания разблокировки мьютекса. Их состояния меняются, но реального прогресса не происходит. Например, `pthreads` не позволяет проверить, заблокирован мьютекс или нет. Было бы бесполезно предоставлять информацию о состоянии мьютекса, потому что как только вы её получаете, она уже может быть изменена другим потоком.

```
if ( mutex is not locked ) {
    /* We still do not know if the mutex is locked or not.
       Other thread might have locked or unlocked it
       several times already.  */
}
```

Однако `pthread_mutex_trylock` разрешена: она либо блокирует мьютекс, либо возвращает ошибку, если он уже заблокирован кем-то ещё. В отличие от `pthread_mutex_lock`, она не блокирует текущий поток в ожидании разблокировки. Использование `pthread_mutex_trylock` может привести к ситуациям живой блокировки. Листинг 17-15 показывает простой пример на псевдокоде.

Листинг 17-15. `livelock_ex`

```
mutex m1, m2;
thread1() {
    lock( m1 );
    while ( mutex_trylock m2 indicates LOCKED ) {
        unlock( m1 );
        wait for some time;
        lock( m1 );
    }
    // now we are good because both locks are taken
}
thread2() {
    lock( m2 );
    while ( mutex_trylock m1 indicates LOCKED ) {
        unlock( m2 );
        wait for some time;
        lock( m2 );
    }
    // now we are good because both locks are taken
}
```

Каждый поток пытается нарушить принцип «блокировки всегда должны выполняться в одном и том же порядке». Оба хотят заблокировать два мьютекса `m1` и `m2`. Первый поток действует так:

- Блокирует мьютекс `m1`.
- Пытается заблокировать мьютекс `m2`. При неудаче разблокирует `m1`, ждёт и снова блокирует `m1`.

Эта пауза призвана дать другому потоку время заблокировать `m1` и `m2` и сделать всё, что ему нужно. Однако мы можем застрять в цикле, когда

1. Поток 1 блокирует `m1`, поток 2 блокирует `m2`.
2. Поток 1 видит, что `m2` заблокирован, и на время разблокирует `m1`.
3. Поток 2 видит, что `m1` заблокирован, и на время разблокирует `m2`.
4. Возврат к шагу один.

Этот цикл может выполняться бесконечно долго или вызывать значительные задержки; это полностью зависит от планировщика операционной системы. Так что проблема с этим кодом в том, что существуют трассы выполнения, навсегда препятствующие прогрессу потоков.

17.8.8 Условные переменные

Условные переменные используются вместе с мьютексами. Они как провода, передающие импульс для пробуждения спящего потока, ожидающего выполнения некоторого условия. Мьютексы реализуют синхронизацию, контролируя доступ потоков к ресурсу; условные же переменные позволяют потокам синхронизироваться на основе дополнительных правил. Например, в случае общих данных таким правилом может быть фактическое значение данных. Ядро использования условных переменных состоит из трёх новых сущностей:

- Самой условной переменной типа `pthread_cond_t`.
- Функции для отправки сигнала пробуждения через условную переменную

```
pthread_cond_signal.
```

- Функции для ожидания прихода сигнала пробуждения через условную переменную `pthread_`

```
cond_wait.
```

Эти две функции следует использовать только между блокировкой и разблокировкой одного и того же мьютекса. Вызывать `pthread_cond_signal` до `pthread_cond_wait` — ошибка, иначе программа может застрять. Рассмотрим минимальный рабочий пример, показанный в листинге 17-16.

Листинг 17-16. `condvar_mwe.c`

```
#include <pthread.h>
#include <stdio.h>
#include <stdbool.h>
#include <unistd.h>
pthread_cond_t condvar = PTHREAD_COND_INITIALIZER;
pthread_mutex_t m;
bool sent = false;
void* t1_impl( void* _ ) {
    pthread_mutex_lock( &m );
    puts( "Thread2 before wait" );
    while (!sent)
        pthread_cond_wait( &condvar, &m );
    puts( "Thread2 after wait" );
    pthread_mutex_unlock( &m );
    return  NULL;
}
void* t2_impl( void* _ ) {
    pthread_mutex_lock( &m );
    puts( "Thread1 before signal" );
    sent = true;
    pthread_cond_signal( &condvar );
    puts( "Thread1 after signal" );
    pthread_mutex_unlock( &m );
    return NULL;
}
int main( void ) {
    pthread_t t1, t2;
    pthread_mutex_init( &m, NULL );
    pthread_create( &t1, NULL, t1_impl, NULL );
    sleep( 2 );
    pthread_create( &t2, NULL, t2_impl, NULL );
    pthread_join( t1, NULL );
    pthread_join( t2, NULL );
    pthread_mutex_destroy( &m );
    return 0;
}
```

Запуск этого кода даст следующий вывод:

```
./a.out
Thread2 before wait
Thread1 before signal
Thread1 after signal
Thread2 after wait
```

Инициализировать условную переменную можно либо присваиванием специальной константы препроцессора `PTHREAD_COND_INITIALIZER`, либо вызовом `pthread_cond_init`. Последняя может принимать указатель на атрибуты типа `pthread_condattr_t`, аналогично `pthread_create` или `pthread_mutex_init`. В этом примере создаются два потока: `t1`, выполняющий инструкции из `t1_impl`, и `t2`, выполняющий инструкции из `t2_impl`. Первый поток блокирует мьютекс `m`. Затем он ждёт сигнала, который может быть передан через условную переменную `condvar`. Заметьте, что `pthread_cond_wait` также принимает указатель на текущий заблокированный мьютекс. Теперь `t1` спит, ожидая прихода сигнала. Мьютекс `m` при этом немедленно разблокируется! Когда поток получает сигнал, он автоматически заново заблокирует мьютекс и продолжит выполнение со следующего оператора после вызова `pthread_cond_wait`. Другой поток блокирует тот же мьютекс `m` и посылает сигнал через `condvar`. `pthread_cond_signal` посылает сигнал через `condvar`, разблокируя как минимум один из потоков, заблокированных на условной переменной `condvar`. Функция `pthread_cond_broadcast` разблокировала бы все потоки, ожидающие эту условную переменную, заставляя их соревноваться за соответствующий мьютекс, как будто все они вызвали `pthread_mutex_lock`. Планировщику решать, в каком порядке они получат доступ к CPU. Как видим, условные переменные позволяют нам блокироваться до получения сигнала. Альтернативой было бы «активное ожидание», когда значение переменной постоянно проверяется (что убивает производительность и увеличивает ненужное энергопотребление), вот так:

```
while (somecondition == false);
```

Разумеется, можно усыпить поток на время, но так мы всё равно будем просыпаться либо слишком редко, чтобы вовремя отреагировать на событие, либо слишком часто:

```
while (somecondition == false)
    sleep(1); /* or something else that lets us sleep for less time */
```

Условные переменные позволяют нам подождать ровно столько, сколько нужно, и продолжить выполнение потока в заблокированном состоянии. Стоит объяснить один важный момент. Зачем мы ввели общую переменную `sent?` Зачем мы используем

её вместе с условной переменной? Зачем мы ждём внутри цикла `while (!sent)`? Важнейшая причина в том, что реализации разрешено *выдавать ложные пробуждения (spurious wake-ups) ожидающему потоку*. Это значит, что поток может проснуться от ожидания сигнала не только после его получения, но и в *любой момент*. В этом случае, поскольку переменная `sent` устанавливается только перед отправкой сигнала, ложное пробуждение проверит её значение, и если оно всё ещё равно `false`, снова выполнит `pthread_cond_wait`.

17.8.9 Спинлоки

Мьютекс — надёжный способ синхронизации. Попытка заблокировать мьютекс, уже занятый другим потоком, переводит текущий поток в спящее состояние. У усыпления потока и его пробуждения есть своя цена, особенно из-за переключения контекста, но если ожидание долгое, эти затраты себя оправдывают. Мы тратим немного времени на засыпание и пробуждение, но в продолжительном спящем состоянии поток не использует CPU. Какая была бы альтернатива? Активное ожидание, описываемое следующим простым псевдокодом:

```
while ( locked == true ) {
    /* do nothing */
}
locked = true;
```

Переменная `locked` — флаг, показывающий, взял ли какой-то поток блокировку или нет. Если блокировку взял другой поток, текущий поток будет постоянно опрашивать её значение, пока оно не изменится обратно. Иначе он переходит к тому, чтобы взять блокировку самому. Это тратит время CPU впустую (и увеличивает энергопотребление), что плохо. Однако это может повысить производительность, если ожидается, что время ожидания будет очень коротким. Этот механизм называется *спинлоком (spinlock)*. *Спинлоки имеют смысл только на многоядерных и многопроцессорных системах*. Использование спинлока на одном ядре бесполезно. Представьте, что поток входит в цикл внутри спинлока. Он продолжает ждать, пока другой поток изменит значение `locked`, но в этот самый момент никакой другой поток не выполняется, потому что есть только одно ядро, переключающееся между потоками. В конце концов планировщик усыпит текущий поток и позволит выполниться другим потокам, но это лишь означает, что мы впустую потратили циклы CPU, выполняя пустой цикл безо всякой причины! В этом случае немедленное засыпание всегда лучше, а значит, спинлок бесполезен. Этот сценарий, конечно, может произойти и на многоядерной системе, но также есть (обычно) хороший шанс, что другой поток разблокирует спинлок до истечения кванта времени, выделенного текущему потоку. В целом использование спинлоков может быть полезным или нет; это зависит от конфигурации системы, логики программы и нагрузки. В случае сомнений тестируйте и предпочитайте использовать мьютексы (которые часто реализованы

так, что сначала берут спинлок на несколько итераций, а затем переходят в спящее состояние, если разблокировки не произошло). Реализовать *быстрый* и корректный спинлок на практике не так тривиально. Есть вопросы, на которые нужно ответить, например:

- Нужен ли нам барьер памяти при блокировке и/или разблокировке? Если да, какой? У Intel 64,

например, есть `lfence`, `sfence` и `mfence`.

- Как гарантировать, что изменение флага атомарно? В Intel 64, например, для этого

достаточно инструкции `xchg` (с префиксом `lock` в случае нескольких процессоров). `pthreads` предоставляет нам тщательно продуманный и переносимый механизм спинлоков. Подробнее см. страницы `man` для следующих функций:

- pthread_spin_lock
- pthread_spin_destroy
- pthread_spin_unlock

17.9 Семафоры

Семафор — общая целочисленная переменная, над которой можно выполнять три действия.

- Инициализация с аргументом *N*. Устанавливает её значение в *N*.
- Wait (войти). Если значение не ноль, уменьшает его. Иначе ждёт, пока кто-то

другой его увеличит, а затем продолжает с уменьшением.

- Post (выйти). Увеличивает её значение.

Разумеется, значение этой переменной, недоступное напрямую, не может опуститься ниже 0. Семафоры *не* являются частью спецификации `pthreads`; мы работаем с семафорами, чей интерфейс описан в стандарте POSIX. Однако код, использующий семафоры, следует компилировать с

```
-pthread flag.
```

Большинство Unix-подобных операционных систем реализуют и стандартные возможности `pthreads`, и семафоры. Использование семафоров довольно распространено для синхронизации между потоками. Листинг 17-17 показывает пример использования семафора.

Листинг 17-17. `semaphore_mwe.c`

```
#include <semaphore.h>
#include <inttypes.h>
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
sem_t sem;
uint64_t counter1 = 0;
uint64_t counter2 = 0;
pthread_t t1, t2, t3;
void* t1_impl( void* _ ) {
    while( counter1 < 10000000 ) counter1++;
    sem_post( &sem );
    return NULL;
}
void* t2_impl( void* _ ) {
    while( counter2 < 20000000 ) counter2++;
    sem_post( &sem );
    return NULL;
}
void* t3_impl( void* _ ) {
    sem_wait( &sem );
    sem_wait( &sem );
    printf("End: counter1 = %" PRIu64 " counter2 = %" PRIu64 "\n",
            counter1, counter2 );
    return NULL;
}
int main(void) {
    sem_init( &sem, 0, 0 );
    pthread_create( &t3, NULL, t3_impl, NULL );
    sleep( 1 );
    pthread_create( &t1, NULL, t1_impl, NULL );
    pthread_create( &t2, NULL, t2_impl, NULL );
    sem_destroy( &sem );
    pthread_exit( NULL );
    return 0;
}
```

Функция `sem_init` инициализирует семафор. Её второй аргумент — флаг: 0 соответствует семафору, локальному для процесса (который могут использовать разные потоки), ненулевое значение настраивает семафор, видимый нескольким процессам.[41] Третий аргумент устанавливает начальное значение семафора. Семафор удаляется с помощью функции `sem_destroy`. В примере создаются два счётчика и три потока. Потоки `t1` и `t2` уве-

личивают соответствующие счётчики до 1000000 и 20000000, а затем увеличивают значение семафора `sem`, вызывая `sem_post. t3` блокирует себя, дважды уменьшая значение семафора. Затем, когда семафор был дважды увеличен другими потоками, `t3` выводит счётчики в `stdout`. Вызов `pthread_exit` гарантирует, что главный поток не завершится преждевременно, пока все остальные потоки не закончат работу. Семафоры пригождаются в таких задачах, как

- Запрет более чем *n* процессам одновременно выполнять участок кода.
- Заставить один поток ждать, пока другой завершит определённое действие, тем самым налагая

порядок на их действия.

- Удержание не более фиксированного числа рабочих потоков, параллельно выполняющих определённую задачу.

Больше потоков, чем нужно, может снизить производительность. Неверно, что семафор с двумя состояниями полностью аналогичен мьютексу. В отличие от мьютекса, который может разблокировать только тот же поток, что его заблокировал, семафоры может свободно менять любой поток. Мы увидим ещё один пример использования семафора в листинге 17-18, чтобы заставить два потока начинать каждую итерацию цикла одновременно (а когда тело цикла выполняется, они ждут завершения итерации другими циклами). Манипуляции с семафорами, очевидно, действуют и как барьеры памяти компилятора, и как аппаратные барьеры памяти. Подробнее о семафорах см. страницы `man` для следующих функций:

- em_close
- sem_destroy
- sem_getvalue
- sem_init
- sem_open
- sem_post
- sem_unlink
- sem_wait

> **Вопрос 364** Что такое именованный семафор? Почему его нужно обязательно отвязывать (unlink), даже если процесс завершён?

17.10 Насколько строг Intel 64?

Абстрактным машинам со слабой моделью памяти бывает трудно следовать. Записи не по порядку, возвращаемые значения из будущего и спекулятивные чтения сбивают с толку. Intel 64 обычно считается строгой. В большинстве случаев она гарантирует выполнение довольно многих ограничений, включая, среди прочего, следующие:

- Store не переупорядочиваются с более старыми store.
- Store не переупорядочиваются с более старыми load.
- Load не переупорядочиваются с другими load.
- В многопроцессорной системе store в одно и то же место имеют общий порядок.

Есть и исключения, такие как:

- Запись в память в обход кэша такими инструкциями, как `movntdq`, может быть переупорядочена с другими store.
- Строковые инструкции вроде `rep movs` могут быть переупорядочены с другими store.

Полный список гарантий можно найти в томе 3, разделе 8.2.2 [15]. Однако, согласно [15], «чтения могут быть переупорядочены с более старыми записями в другие места, но не с более старыми записями в то же самое место». Так что не обманывайтесь: переупорядочивания памяти действительно происходят. Простая программа, показанная в листинге 17-18, демонстрирует переупорядочивание памяти, выполняемое оборудованием. Она реализует пример, уже показанный в листинге 17-4, где есть два потока и две общие переменные `x` и `y`. Первый поток выполняет `store x` и `load y`, второй выполняет `store y` и `load x`. Барьер компилятора гарантирует, что эти два оператора транслируются в ассемблер в одном и том же порядке. Как подсказывает раздел 17.10, store и load в независимые места могут быть переупорядочены. Так что мы не можем исключить здесь аппаратное переупорядочивание памяти, поскольку `x` и `y` независимы!

Листинг 17-18. `reordering_cpu_mwe.c`

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <inttypes.h>
#include <stdint.h>
#include <stdlib.h>
#include <time.h>
sem_t sem_begin0, sem_begin1, sem_end;
int x, y, read0, read1;
void *thread0_impl( void *param )
{
    for (;;) {
        sem_wait( &sem_begin0 );
        x = 1;
        // This only disables compiler reorderings:
        asm volatile("" ::: "memory");
        // The following line disables also hardware reorderings:
        // asm volatile("mfence" ::: "memory");
        read0 = y;
        sem_post( &sem_end );
}
    return NULL;
};
void *thread1_impl( void *param )
{
    for (;;) {
        sem_wait( &sem_begin1 );
        y = 1;
        // This only disables compiler reorderings:
        asm volatile("" ::: "memory");
        // The following line disables also hardware reorderings
        // asm volatile("mfence" ::: "memory");
        read1 = x;
        sem_post( &sem_end );
    }
    return NULL;
};
int main( void ) {
    sem_init( &sem_begin0, 0, 0);
    sem_init( &sem_begin1, 0, 0);
    sem_init( &sem_end, 0, 0);
    pthread_t thread0, thread1;
    pthread_create(  &thread0, NULL, thread0_impl, NULL);
```

```
    pthread_create(  &thread1, NULL, thread1_impl, NULL);
    for (uint64_t i = 0; i < 100000; i++)
    {
        x = 0;
        y = 0;
        sem_post( &sem_begin0 );
        sem_post( &sem_begin1 );
        sem_wait( &sem_end );
        sem_wait( &sem_end );
        if (read0 == 0 && read1 == 0 ) {
            printf( "reordering happened on iteration %" PRIu64 "\n", i );
            exit(0);
        }
}
        puts("No reordering detected during 100000 iterations");
        return 0;
}
```

Чтобы это проверить, мы проводим несколько экспериментов. Функция `main` действует так:

1. Инициализирует потоки и два начальных семафора, а также один завершающий.
2. x = 0, y = 0
3. Уведомляет потоки, что им нужно начать выполнять транзакцию.
4. Ждёт завершения транзакции обоими потоками.
5. Проверяет, произошло ли переупорядочивание памяти. Это видно, когда и `load x`, и `load y` вернули нули (потому что они были переупорядочены и оказались перед

```
store s).
```

6. Если переупорядочивание памяти обнаружено, мы получаем об этом уведомление, и процесс завершается. Иначе он пробует снова начиная с шага (2), максимум до 100000 попыток.

Каждый поток ждёт сигнала на старт от `main`, выполняет транзакцию и уведомляет об этом `main`. Затем всё начинается заново. После запуска вы увидите, что 100000 итераций достаточно, чтобы наблюдать переупорядочивание памяти.

```
> gcc -pthread -o ordering -O2 ordering.c
> ./ordering
reordering happened on iteration 128
> ./ordering
reordering happened on iteration 12
> ./ordering
reordering happened on iteration 171
> ./ordering
reordering happened on iteration 80
> ./ordering
reordering happened on iteration 848
> ./ordering
reordering happened on iteration 366
> ./ordering
reordering happened on iteration 273
> ./ordering
reordering happened on iteration 105
> ./ordering
reordering happened on iteration 14
> ./ordering
reordering  happened  on  iteration 5
> ./ordering
reordering happened on iteration 414
```

Это может показаться волшебством, но здесь виден уровень даже ниже языка ассемблера, вносящий редко наблюдаемые (но всё же устойчивые) ошибки в программное обеспечение. Такие ошибки в многопоточном ПО *очень* трудно отловить. Представьте себе ошибку, проявляющуюся только после четырёх месяцев непрерывной работы, повреждающую кучу и обрушивающую программу через 42 выделения после того, как она сработала! Так что для написания высокопроизводительного многопоточного ПО в неблокирующем стиле требуется огромный опыт. Итак, что нам нужно сделать — это добавить инструкцию `mfence`. Замена барьера компилятора на полный барьер памяти `asm volatile( "mfence":::"memory");` решает проблему, и переупорядочивания полностью исчезают. Если мы это сделаем, никаких переупорядочиваний не будет обнаружено, сколько бы попыток мы ни делали.

17.11 Что такое неблокирующее программирование?

Мы увидели, как можно обеспечить согласованность операций при работе в многопоточной среде. Каждый раз, когда нам нужно выполнить сложную операцию над общими данными или ресурсами без вмешательства других потоков, мы блокируем мьютекс, связанный с этим ресурсом или блоком памяти. Мы говорим, что код *неблокирующий (lock-free)*, если выполнены следующие два условия:

- Мьютексы не используются.
- Систему нельзя заблокировать на неопределённое время. Это включает и живые блокировки.

Другими словами, это семейство техник, обеспечивающих безопасную работу с общими данными без использования мьютексов. Мы почти всегда ожидаем, что свойством неблокируемости обладает лишь часть кода программы. Например, структуру данных, такую как очередь, можно считать неблокирующей, если функции, используемые для работы с ней, неблокирующие. Так что это не мешает нам полностью заблокироваться, но в той мере, в какой мы вызываем такие функции, как `enqueue` или `dequeue`, прогресс будет достигнут. С точки зрения программиста, неблокирующее программирование отличается от традиционного использования мьютексов, потому что вносит две сложности, обычно покрываемые мьютексами.

1. Переупорядочивания. Хотя манипуляции с мьютексами подразумевают барьеры памяти

компилятора и оборудования, без них вам придётся точно указывать, где размещать барьеры памяти. Вы не захотите размещать их после каждого оператора, потому что это убивает производительность.

2. Неатомарные операции. Операции между блокировкой и разблокировкой мьютекса

безопасны и в некотором смысле атомарны. Никакой другой поток не может изменить данные, связанные с мьютексом (если только нет небезопасных манипуляций с данными вне участка блокировки-разблокировки). Без этого механизма у нас остаётся очень мало атомарных операций, которые мы изучим далее в этой главе. На большинстве современных процессоров чтения и записи естественно выровненных встроенных типов атомарны. Естественное выравнивание означает выравнивание переменной по границе, соответствующей её размеру. В Intel 64 нет гарантии, что чтения и записи больше 8 байт атомарны. Другие взаимодействия с памятью обычно неатомарны. Сюда входят, среди прочего,

- 16-байтовые чтения и записи, выполняемые инструкциями SSE (Streaming SIMD Extensions).
- Строковые операции (инструкция `movsb` и подобные).
- Многие операции атомарны в однопроцессорной системе, но не в многопроцессорной

системе (например, инструкция `inc`). Чтобы сделать их атомарными, нужен специальный префикс `lock`, предотвращающий выполнение другими процессорами своей собственной последовательности чтение-изменение-запись между этапами указанных инструкций. Например, инструкции `inc` <`addr`> нужно прочитать байты из памяти и записать обратно их увеличенное значение. Без префикса `lock` между этими этапами могут вмешаться другие процессоры, что может привести к потере значения. Вот несколько примеров неатомарных операций:

```
char buf[1024];
uint64_t* data = (uint64_t*)(buf + 1);
/* not atomic: unnatural alignment */
*data = 0;
/* not atomic: increment can need a read and a write */
++global_aligned_var;
/* atomic write */
global_aligned_var = 0;
void f(void) {
/* atomic read */
int64_t local_variable = global_aligned_var;
}
```

Эти случаи специфичны для архитектуры. Нам также иногда нужно атомарно выполнять более сложные операции (например, инкремент счётчика). Чтобы делать это безопасно без использования мьютексов, инженеры придумали интересные базовые операции, такие как compare-and-swap (CAS, сравнение с обменом). Как только эта операция реализована как машинная инструкция на конкретной архитектуре, её можно использовать в сочетании с более тривиальными неатомарными чтениями и записями для реализации многих неблокирующих алгоритмов и структур данных. Инструкция CAS действует как атомарная последовательность операций, описываемая следующей эквивалентной функцией на C:

```
bool cas(int* p , int old, int new) {
    if (*p != old) return false;
    *p = new;
    return true;
}
```

Общий счётчик, значение которого вы читаете и записываете обратно изменённым, — типичный случай, когда для атомарного инкремента или декремента нужна инструкция CAS. Листинг 17-19 показывает функцию для его выполнения.

Листинг 17-19. `cas_counter.c`

```
int add(int* p, int add ) {
    bool done = false;
    int value;
    while (!done) {
        value = *p;
            done = cas(p, value, value + add );
    }
    return value + add;
}
```

Этот пример показывает типичный шаблон, встречающийся во многих алгоритмах на основе CAS. Они читают определённую ячейку памяти, вычисляют её изменённое значение и раз за разом пытаются подставить новое значение обратно, если текущее значение памяти равно старому. Это не удаётся, если эту ячейку памяти изменил другой поток; тогда весь цикл чтение-изменение-запись повторяется. Intel 64 реализует инструкции CAS `cmpxchg`, `cmpxchg8b` и `cmpxchg16b`. При наличии нескольких процессоров для них тоже требуется префикс `lock`. Особый интерес представляет инструкция `cmpxchg`. Она принимает два операнда: регистр или память и регистр. Она сравнивает `rax` [42] с первым операндом. Если они равны, устанавливается флаг `zf`, значение второго операнда загружается в первый. Иначе фактическое значение первого операнда загружается в `rax`, а `zf` сбрасывается. Эти инструкции можно использовать как часть реализации мьютексов и семафоров. Как мы увидим в разделе 17.12.2, теперь есть соответствующий стандарту способ использования операций compare-and-set (а также манипуляции с атомарными переменными). Мы рекомендуем придерживаться его, чтобы избежать непереносимого кода, и использовать атомарные операции всегда, когда можете. Когда вам нужно атомарно выполнить сложные операции, используйте мьютексы или придерживайтесь реализаций неблокирующих структур данных, сделанных экспертами: написание неблокирующих структур данных оказалось непростой задачей.

> **Вопрос 365** Что такое проблема ABA?

> **Вопрос 366** Прочитайте описание `cmpxchg` в документации Intel [15].

17.12 Модель памяти C11

17.12.1 Обзор

Чаще всего мы хотим писать код, корректный на любой архитектуре. Чтобы этого добиться, мы опираемся на модель памяти, описанную в стандарте C11. Компилятор может реализовать некоторые операции напрямую или сгенерировать специальные инструкции для обеспечения определённых гарантий, если реальная аппаратная архитектура слабее. В отличие от Intel 64, модель памяти C11 довольно слабая. Она гарантирует упорядочивание по зависимости данных, но не более того, так что в классификации, упомянутой в разделе 17.4, она соответствует второй: слабая с упорядочиванием по зависимости. Есть и другие аппаратные архитектуры, предоставляющие схожие слабые гарантии, например ARM. Из-за слабости C, чтобы писать переносимый код, мы *не можем предполагать*, что он будет выполняться на обычно строгой архитектуре, такой как Intel 64, по двум причинам.

- При перекомпиляции для другой, более слабой архитектуры наблюдаемое поведение программы

изменится из-за того, как работают аппаратные переупорядочивания.

- При перекомпиляции для той же архитектуры могут произойти переупорядочивания

компилятора, не нарушающие правила слабого упорядочивания, налагаемые стандартом. Это может изменить наблюдаемое поведение программы, по крайней мере для некоторых трасс выполнения.

17.12.2 Атомарные операции

Важная возможность C11, которую можно использовать для написания быстрых многопоточных программ, — атомарные операции (см. раздел 7.17 [7]). Это специальные типы переменных, которые можно изменять атомарно. Чтобы их использовать, подключите заголовок `stdatomic.h`.

Разумеется, для их эффективной реализации нужна поддержка архитектуры. В худшем случае, когда архитектура не поддерживает подобных операций, каждая такая переменная будет связана с мьютексом, который будет блокироваться для выполнения любого изменения переменной или даже для её чтения. Атомарные операции позволяют нам в некоторых случаях выполнять потокобезопасные операции над общими данными, часто позволяя обойтись без тяжёлой машинерии с мьютексами. Однако написать такую структуру данных, как очередь, неблокирующим способом — задача не из лёгких.

Для этого мы настоятельно советуем использовать существующие реализации как «чёрные ящики». C11 определяет новый спецификатор типа `_Atomic()`. Атомарное целое число можно объявить так:

```
_Atomic(int) counter;
```

`_Atomic` превращает имя типа в имя атомарного типа. В качестве альтернативы можно использовать атомарные типы напрямую, вот так:

```
atomic_int counter;
```

Полное соответствие между `_Atomic(T)` и прямыми формами типа `atomic_T` можно найти в разделе 7.17.6 [7]. Атомарные *локальные* переменные *не* следует инициализировать напрямую; вместо этого следует использовать макрос `ATOMIC_VAR_INIT`. Это понятно, поскольку на некоторых архитектурах с меньшими аппаратными возможностями каждую такую переменную нужно связывать с мьютексом, который тоже нужно создать и инициализировать. Глобальные атомарные переменные гарантированно находятся в корректном начальном состоянии. `ATOMIC_VAR_INIT` следует использовать при объявлении переменной вместе с инициализацией; однако если вы хотите инициализировать переменную позже, используйте макрос `atomic_init`.

```
void f(void) {
    /* Initialization during declaration */
    atomic_int x = ATOMIC_VAR_INIT( 42 );
    atomic_int y;
    /* initialization later */
    atomic_init( &y, 42 );
}
```

Ваша ответственность — гарантировать, что инициализация атомарной переменной завершается прежде, чем с ней будет сделано что-либо ещё. Другими словами, конкурентный доступ к инициализируемой переменной — гонка данных. С атомарными переменными следует работать только через интерфейс, определённый в стандарте языка. Он состоит из нескольких операций, таких как load, store, exchange и т. д. Каждая из них существует в двух версиях.

- Явная версия, принимающая дополнительный аргумент, описывающий упорядочивание памяти. Её имя заканчивается на `_explicit`. Например, операция load —

```
T atomic_load_explicit( _Atomic(T) *object, memory_order order );
```

- Неявная версия, подразумевающая самое строгое упорядочивание памяти (последовательно согласованное). Суффикса `_explicit` нет. Например,

```
T atomic_load( _Atomic(T) *object );
```

17.12.3 Упорядочивания памяти в C11

Упорядочивание памяти описывается одной из констант перечисления (в порядке возрастания строгости).

- `memory_order_relaxed` подразумевает самую слабую модель: возможно любое переупорядочивание памяти,

пока наблюдаемое поведение однопоточной программы остаётся неизменным.

- `memory_order_consume` — более слабая версия `memory_order_acquire`.
- `memory_order_acquire` означает, что операция load обладает семантикой acquire.
- `memory_order_release` означает, что операция store обладает семантикой release.
- `memory_order_acq_rel` объединяет семантику acquire и release.
- `memory_order_seq_cst` подразумевает, что переупорядочивание памяти не выполняется *для*

всех операций, помеченных им, независимо от того, к какой атомарной переменной идёт обращение. Предоставляя явную константу упорядочивания памяти, мы можем контролировать, как мы хотим разрешить *наблюдаемое* переупорядочивание операций. Это включает и переупорядочивания компилятора, и аппаратные переупорядочивания, так что когда компилятор видит, что переупорядочивания компилятора не дают всех нужных нам гарантий, он также сгенерирует платформенно-специфичные инструкции, такие как `sfence`. Опция `memory_order_consume` используется редко. Она опирается на понятие «операции потребления». Эта операция — событие, происходящее, когда значение читается из памяти, а затем используется впоследствии в нескольких операциях, создавая зависимость данных. На более слабых архитектурах, таких как PowerPC или ARM, её использование может дать лучшую производительность за счёт задействования зависимостей данных для наложения определённого порядка на обращения к памяти. Так экономится дорогостоящая инструкция аппаратного барьера памяти, потому что эти архитектуры гарантируют упорядочивание по зависимости данных без явных барьеров. Однако из-за того, что это упорядочивание настолько трудно эффективно и корректно реализовать в компиляторах, оно обычно напрямую отображается на `memory_order_acquire`, немного более строгую версию. Мы не рекомендуем его использовать. Подробнее см. [30]. *Семантика acquire и release* этих опций упорядочивания памяти напрямую соответствует понятиям, которые мы изучили в разделе 17.7. `memory_order_seq_cst` соответствует понятию последовательной согласованности, которое

мы разобрали в разделе 17.4. Поскольку все неявные операции с атомарными переменными принимают его как значение упорядочивания памяти по умолчанию, атомарные переменные C11 по умолчанию последовательно согласованы. Это самый безопасный путь, к тому же обычно более быстрый, чем мьютексы. Более слабые упорядочивания труднее реализовать правильно, но они также позволяют добиться лучшей производительности. `atomic_thread_fence(memory_order order)` позволяет нам вставить барьер памяти (и компилятора, и оборудования) с силой, соответствующей указанному упорядочиванию памяти. Например, эта операция не имеет эффекта для `memory_order_relaxed`, но для последовательно согласованного упорядочивания в Intel 64 будет сгенерирована инструкция `mfence` (вместе с барьером компилятора).

17.12.4 Операции

Над атомарными переменными можно выполнять следующие операции (`T` обозначает неатомарный тип, `U` относится к типу другого аргумента для арифметических операций; для всех типов, кроме указателей, он совпадает с `T`, для указателей это `ptrdiff_t`).

```
void atomic_store(volatile _Atomic(T)* object, T  value);
T atomic_load(volatile _Atomic(T)* object);
T atomic_exchange(volatile _Atomic(T)* object, desired);
T atomic_fetch_add(volatile _Atomic(T)* object, U operand);
T atomic_fetch_sub(volatile _Atomic(T)* object, U operand);
T atomic_fetch_or (volatile _Atomic(T)* object, U operand);
T atomic_fetch_xor(volatile _Atomic(T)* object, U operand);
T atomic_fetch_and(volatile _Atomic(T)* object, U operand);
bool atomic_compare_exchange_strong(
    volatile _Atomic(T)* object, T * expected, T desired);
bool atomic_compare_exchange_weak(
    volatile _Atomic(T)* object, T * expected, T desired);
```

Все эти операции можно использовать с суффиксом `_explicit`, чтобы указать упорядочивание памяти как дополнительный аргумент. Функции load и store не нуждаются в дальнейших объяснениях; остальные мы кратко обсудим. `atomic_exchange` — комбинация load и store: она заменяет значение атомарной переменной на `desired` и возвращает её старое значение. Семейство операций `fetch_op` используется для атомарного изменения значения атомарной переменной. Представьте, что вам нужно инкрементировать атомарный счётчик. Без `fetch_add` это невозможно, поскольку для инкремента нужно прибавить единицу к его старому значению, которое сначала нужно прочитать. Эта операция выполняется в три шага: чтение, сложение, запись. Между этими этапами могут вмешаться другие потоки, что разрушает атомарность. `atomic_compare_exchange_strong` предпочтительнее своего слабого аналога, поскольку слабая версия может завершиться

неудачей без причины. У последней лучше производительность на некоторых платформах. Функция `atomic_compare_exchange_strong` примерно эквивалентна следующему псевдокоду:

```
if ( *object == *expected )
    *object = desired;
else
    *expected = *object;
```

Как видим, это типичная инструкция CAS, которую мы обсуждали в разделе 17.11. Макрос `atomic_is_lock_free` используется, чтобы проверить, использует ли конкретная атомарная переменная блокировки или нет. Помните, что без явного указания упорядочивания памяти *все* эти операции считаются последовательно согласованными, что в Intel 64 означает инструкции `mfence` по всему коду. Это может сильно убить производительность. У булева общего флага есть специальный тип с именем `atomic_flag`. У него два состояния: установлен и сброшен. Гарантируется, что операции над ним атомарны без использования блокировок. Флаг следует инициализировать макросом `ATOMIC_FLAG_INIT` вот так:

```
atomic_flag is_working = ATOMIC_FLAG_INIT;
```

Соответствующие функции — `atomic_flag_test_and_set` и `atomic_flag_clear`, у обеих есть аналоги с `_explicit`, принимающие описания упорядочивания памяти.

```
■Question 367Read man pages for atomic_flag_test_and_set and atomic_flag_clear.
```

17.13 Итоги

В этой главе мы изучили основы многопоточного программирования. Мы рассмотрели разные модели памяти и проблемы, возникающие из-за того, что оптимизации компилятора и оборудования вмешиваются в порядок выполнения инструкций. Мы научились их контролировать, размещая различные барьеры памяти, и увидели, почему `volatile` не решает проблем, возникающих из-за многопоточности. Затем мы представили `pthreads`, самый распространённый стандарт написания многопоточных приложений в Unix-подобных системах. Мы рассмотрели управление потоками, использовали мьютексы и условные переменные и узнали, почему спинлоки имеют смысл только на многоядерных и многопроцессорных системах. Мы увидели, что переупорядочивания памяти нужно учитывать, даже работая на обычно строгой архитектуре, такой как Intel 64, и увидели пределы её строгости. Наконец, мы изучили атомарные переменные —

очень полезную возможность C11, позволяющую избавиться от явного использования мьютексов и во многих случаях повысить производительность, сохраняя корректность. Мьютексы всё ещё важны, когда нужно выполнять сложные манипуляции над нетривиальными структурами данных.

Вопрос 368 Какие проблемы возникают из-за использования многопоточности?

Вопрос 369 Что делает несколько потоков полезными?

Вопрос 370 Стоит ли использовать многопоточность, даже если программа не выполняет много вычислений? Если да, приведите пример применения.

Вопрос 371 Что такое переупорядочивание компилятором? Зачем оно выполняется?

Вопрос 372 Почему у однопоточной программы нет средств наблюдать переупорядочивания памяти компилятором?

Вопрос 373 Какие бывают виды моделей памяти?

Вопрос 374 Как написать код, последовательно согласованный в отношении манипуляции двумя общими переменными?

Вопрос 375 Последовательно ли согласованы переменные `volatile`?

Вопрос 376 Приведите пример, когда переупорядочивания памяти могут привести к весьма неожиданному поведению программы.

Вопрос 377 Какие есть аргументы против использования переменных volatile?

Вопрос 378 Что такое барьер памяти?

Вопрос 379 Какие виды барьеров памяти вы знаете?

Вопрос 380 Что такое семантика acquire?

Вопрос 381 Что такое семантика release?

Вопрос 382 Что такое зависимость данных? Можете ли вы написать код, где зависимость данных не навязывает порядок операциям?

Вопрос 383 В чём разница между `mfence`, `sfence` и `lfence`?

Вопрос 384 Зачем нам нужны инструкции, кроме `mfence`?

Вопрос 385 Какие вызовы функций действуют как барьеры компилятора?

Вопрос 386 Являются ли вызовы `inline`-функций барьерами компилятора?

Вопрос 387 Что такое поток?

Вопрос 388 В чём разница между потоками и процессами?

Вопрос 389 Что составляет состояние процесса?

Вопрос 390 Что составляет состояние потока?

Вопрос 391 Зачем использовать флаг `-pthread` при компиляции с `pthreads`?

Вопрос 392 `pthreads` — статическая или динамическая библиотека?

Вопрос 393 Как нам узнать, в каком порядке планировщик будет выполнять потоки?

Вопрос 394 Может ли один поток получить доступ к стеку другого потока?

Вопрос 395 Что делает `pthread_join` и как её использовать?

Вопрос 396 Что такое мьютекс? Зачем он нам нужен?

Вопрос 397 Должна ли каждая общая константная переменная быть связана с мьютексом?

Вопрос 398 Должна ли каждая общая изменяемая переменная, которая никогда не меняется, быть связана с мьютексом?

Вопрос 399 Должна ли каждая общая изменяемая переменная, которая меняется, быть связана с мьютексом?

Вопрос 400 Можем ли мы работать с общей переменной, вообще не используя мьютекс?

Вопрос 401 Что такое взаимоблокировка?

Вопрос 402 Как предотвратить взаимоблокировку?

Вопрос 403 Что такое живая блокировка? Чем она отличается от взаимоблокировки?

Вопрос 404 Что такое спинлок? Чем он отличается от живой блокировки и взаимоблокировки?

Вопрос 405 Стоит ли использовать спинлоки на одноядерной системе? Почему?

Вопрос 406 Что такое условная переменная?

Вопрос 407 Зачем нам условные переменные, если у нас есть мьютексы?

Вопрос 408 Какие гарантии Intel 64 предоставляет для переупорядочиваний памяти?

Вопрос 409 Какие важные гарантии Intel 64 не предоставляет для переупорядочиваний памяти?

Вопрос 410 Исправьте программу, показанную в листинге 17-18, так, чтобы переупорядочивание памяти не происходило.

Вопрос 411 Исправьте программу, показанную в листинге 17-18, так, чтобы переупорядочивание памяти не происходило, используя атомарные переменные.

Вопрос 412 Что такое неблокирующее программирование? Почему оно сложнее традиционного многопоточного программирования с блокировками?

Вопрос 413 Что такое операция CAS? Как её можно реализовать в Intel 64?

Вопрос 414 Насколько строга модель памяти C?

Вопрос 415 Можно ли контролировать строгость модели памяти C?

Вопрос 416 Что такое атомарная переменная?

Вопрос 417 Может ли любой тип данных быть атомарным?

Вопрос 418 Какие атомарные переменные следует инициализировать явно?

Вопрос 419 Какие упорядочивания памяти распознаёт C11?

Вопрос 420 Чем функции манипуляции атомарными переменными с суффиксом `_explicit` отличаются от своих обычных аналогов?

Вопрос 421 Как выполнить атомарный инкремент атомарной переменной?

Вопрос 422 Как выполнить атомарный XOR над атомарной переменной?

Вопрос 423 В чём разница между слабой и строгой версиями `compare_exchange` ?

Приложение A. Использование gdb

Отладчик — очень мощный инструмент в вашем распоряжении. Он позволяет выполнять программы пошагово и следить за их состоянием, включая значения регистров и содержимое памяти. В этой книге мы используем отладчик под названием `gdb`. Это приложение — введение, призванное облегчить ваши первые шаги с ним. Отладка — это процесс поиска ошибок и изучения поведения программы. Для этого мы обычно выполняем отдельные шаги, наблюдая за интересующей нас частью состояния программы. Мы также можем запускать программу до выполнения определённого условия или до достижения определённой позиции в коде. Такая позиция в коде называется точкой останова. Рассмотрим пример программы, показанный в листинге 18-1. Мы уже видели её в Главе 2. Этот код выводит содержимое регистра `rax` в stdout.

Листинг 18-1. `print_rax_2.asm`

```
section .data
codes:
db      '0123456789ABCDEF'
section .text
global _start
_start:
mov rax, 0x1122334455667788
mov rdi, 1
mov rdx, 1
mov rcx, 64
.loop:
push rax
sub rcx, 4
sar rax, cl
and rax, 0xf
lea rsi, [codes + rax]
mov rax, 1
push rcx
syscall
pop rcx
```

```
pop rax
test rcx, rcx
jnz .loop
mov     rax, 60          ; invoke 'close' system call
xor rdi, rdi
syscall
```

Мы скомпилируем из неё исполняемый файл `print_rax` и запустим `gdb`.

```
> nasm -o print_rax.o -f elf64  print_rax.asm
> ld -o print_rax print_rax.o
> gdb print_rax
...
(gdb)
```

У `gdb` своя система команд, и взаимодействие с ним происходит через эти команды. Так что всякий раз, когда `gdb` запущен и вы видите его приглашение командной строки `(gdb)`, вы можете вводить команды, и он будет реагировать соответствующим образом. Загрузить исполняемый файл можно командой `file` с последующим вводом имени файла, либо передав его как аргумент.

```
(gdb) file print_rax
Reading symbols from print_rax...(no debugging symbols found)...done.
```

Клавиша `<tab>` в командной строке `gdb` выполняет функцию автодополнения. У многих команд также есть сокращения. Две самые важные команды:

- `quit` для выхода из gdb.
- `help cmd` для показа справки по команде `cmd`.

Файл `~/.gdbinit` хранит команды, которые будут автоматически выполнены при запуске `gdb`. Такой файл можно создать и в текущем каталоге, но из соображений безопасности эта возможность по умолчанию отключена.

> **Примечание** Чтобы разрешить загрузку файла `.gdbinit` из любого каталога, добавьте следующую строку в файл `~/.gdbinit` в вашем домашнем каталоге:

```
set auto-load safe-path /
```

По умолчанию `gdb` использует синтаксис ассемблера AT&T. В нашей книге мы придерживаемся синтаксиса Intel; чтобы изменить настройки `gdb` по умолчанию относительно синтаксиса ассемблера, добавьте следующую строку в файл `~/.gdbinit`:

```
set disassembly-flavor intel
```

Другие полезные команды включают следующие:

- `run` запускает выполнение программы.
- `break x` создаёт точку останова рядом с меткой `x`. При выполнении `run` или `continue` мы остановимся на первой достигнутой точке останова, что позволит нам изучить состояние программы.
- `break *address` для размещения точки останова по указанному адресу.
- с `ontinue` для продолжения выполнения программы
- `stepi` или `si` для выполнения одной инструкции;
- `ni` или `nexti` тоже выполнит одну инструкцию, но не войдёт в функции, если инструкцией был `call`. Вместо этого он позволит вызванной функции завершиться и остановится на следующей инструкции.

Выполним следующие действия:

```
(gdb) break _start
Breakpoint 1 at 0x4000b0
(gdb) start
Function "main" not defined.
Make breakpoint pending on future shared library load? (y or [n]) n
Starting program: /home/stud/test/print_rax
Breakpoint 1, 0x00000000004000b0 in _start ()
```

Мы остановились на точке останова, которую разместили на метке `_start`. Переключимся в псевдографический режим с помощью команд:

```
layout asm
layout regs
```

Результат показан на рисунке 18-1. Раскладка состоит из трёх окон:

- Верхняя часть показывает регистры и их текущие значения.
- Средняя часть показывает дизассемблированный код.
- Нижняя часть — интерактивное приглашение.
- В каждый момент времени в фокусе одно из этих окон. `Ctrl-X` и `Ctrl-O` позволяют переключаться между ними.
- Клавиши со стрелками можно использовать для прокрутки текущего окна вверх и вниз.
- `print /FMT <val>` позволяет посмотреть содержимое регистра или значения памяти. Перед именами регистров ставится знак доллара, например: `$rax`.
- `x /FMT <address>` — ещё одна очень полезная команда для проверки содержимого памяти. В отличие от `print`, она ожидает уровень косвенности, то есть принимает

указатель.

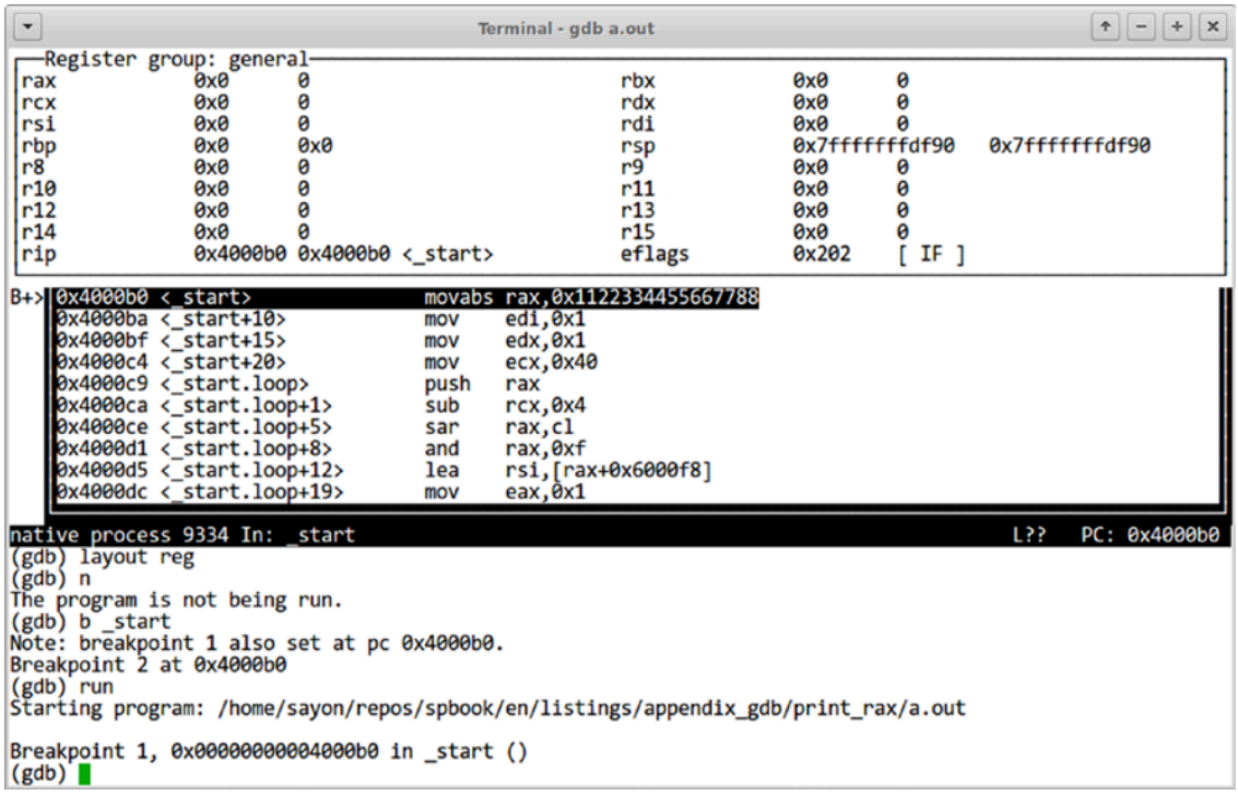


Рисунок 18-1. Пользовательский интерфейс `gdb` : раскладка `asm` + `regs`

`FMT` (используемый командами `print` и `x`) — закодированное описание формата. Он позволяет явно выбрать тип данных, чтобы правильно интерпретировать содержимое памяти. FMT состоит из буквы формата и буквы размера. Самые полезные буквы формата:

- `x` (шестнадцатеричный)
- `a` (адрес)
- `i` (инструкция, пытается выполнить дизассемблирование)
- `c` (символ)
- `s` (строка, завершающаяся нулём)

Самые полезные буквы размера — `b` (байт) и `g` (giant, гигантский, 8 байт). Чтобы взять адрес переменной, используйте символ `&`. Примеры покажут, когда это удобно. Ниже несколько примеров на основе программы, показанной в листинге 18-1:

- Показать содержимое `rax` :

```
(gdb) print $rax
$1 = 1234605616436508552
```

- Показать первый символ `codes`:

```
(gdb) print /c codes
$2 = '0'
```

- Дизассемблировать инструкцию по адресу `_start`:

```
(gdb) x /i &_start
   0x4000b0 <_start>:    movabs rax,0x11223344556677 88
```

- Дизассемблировать текущую инструкцию:

```
(gdb) x /i $rip
=> 0x4000e9 <_start.loop+32>:   jne    0x4000c9 <_start.loop>
```

- Проверить содержимое `codes`. Часть / `FMT` команды `x` может начинаться с количества элементов. В нашем случае / `12cb` означает «12 символов по одному байту каждый».

```
(gdb) x /12cb &codes
0x6000f8 <codes>:     48 '0' 49 '1' 50 '2' 51 '3' 52 '4' 53 '5' 54 '6'
55 '7'
0x600100:       56 '8' 57 '9' 65 'A' 66 'B'
```

- Изучить верхние 8 байт стека:

```
(gdb) x /x $rsp
0x7fffffffdf90: 0x01
```

- Изучить второе `qword`, хранящееся в стеке:

```
(gdb) x/x $rsp+8
0x7fffffffdf98: 0xc1
```

> **Вопрос 424** Изучите вывод команды `help x`.

Чтобы продуктивно использовать `gdb` с программами на C, не забывайте всегда использовать опцию компилятора `-ggdb`. Она генерирует дополнительную информацию, которую может использовать `gdb`, такую как секция .line или символы для локальных переменных. Подходящая раскладка для работы с кодом на C — `src`; наберите `layout src`, чтобы переключиться на неё. Рисунок 18-2 изображает эту раскладку.



Рисунок 18-2. Пользовательский интерфейс `gdb`: раскладка `src`

Ещё одна полезная возможность заключается в изучении и навигации по стеку вызовов. Каждый раз, когда вызывается какая-либо функция, она использует часть стека для хранения своих локальных переменных. Чтобы продемонстрировать навигацию, мы используем простую программу, показанную в листинге 18-2.

Листинг 18-2. `call_stack.c`

```
#include <stdio.h>
void g(int garg) {
    int glocal = 99;
    puts("Inside  g");
}
void f(int farg) {
    int flocal = 44;
    g( flocal );
}
int main( void ) {
    f( 42 );
    return 0;
}
```

Мы скомпилируем программу и запустим на ней gdb следующим образом:

```
> gcc -ggdb call_stack.c -o call_stack
> gdb call_stack
```

Затем мы размещаем точку останова на функции `g` и запускаем программу следующим образом:

```
(gdb) break g
Breakpoint 1 at 0x400531: file call_stack.c, line 5.
(gdb) run
Starting program: .../call_stack
Breakpoint 1, g (garg=0) at call_stack.c:5
5      puts("Inside g");
```

Мы можем при желании ввести `layout src`. Программа выполнится и остановится на строке 4, где начинается функция `g`. Мы можем изучить локальные переменные или аргументы с помощью команды `print`. В большинстве случаев `gdb` сама выведет правильные типы.

```
 (gdb) print garg
$1  = 44
```

Мы хотим увидеть, какие функции сейчас запущены. Для этого служит команда `backtrace`.

```
 (gdb) backtrace
#0   g (garg=44) at call_stack.c:4
#1   0x0000000000400561 in f (farg=42) at call_stack.c:10
#2   0x0000000000400572 in main () at call_stack.c:14
```

Есть три кадра стека, о которых знает `gdb`, и мы можем переключаться между ними командой `frame<idx>`. Наше текущее состояние изображено на рисунке 18-3. Мы уверены, что функция `f` запустила функцию `g`, как и говорит backtrace, так что у этого экземпляра `f` должна быть локальная переменная `flocal`. Мы хотим узнать её значение. Если мы попробуем сразу же её вывести, `gdb` пожалуется, что такой переменной не существует. Однако если мы сначала выберем подходящий кадр стека командой `frame 1`, мы получим доступ ко всем его локальным переменным. Рисунок 18-4 изображает это изменение.

```
 (gdb) print farg
No  symbol "farg" in current context.
```



Рисунок 18-3. Внутри функции `g`

```
(gdb) frame 1
#1  0x0000000000400561 in f (farg=42) at call_stack.c:10
(gdb) print farg
$3  =  42
```

> **Вопрос 425** Что делает `info locals` ?

Помимо этого, `gdb` поддерживает вычисление выражений с обычными арифметическими операциями, запуск функций, написание скриптов автоматизации на Python и многое другое. Для дальнейшего чтения обратитесь к [1].

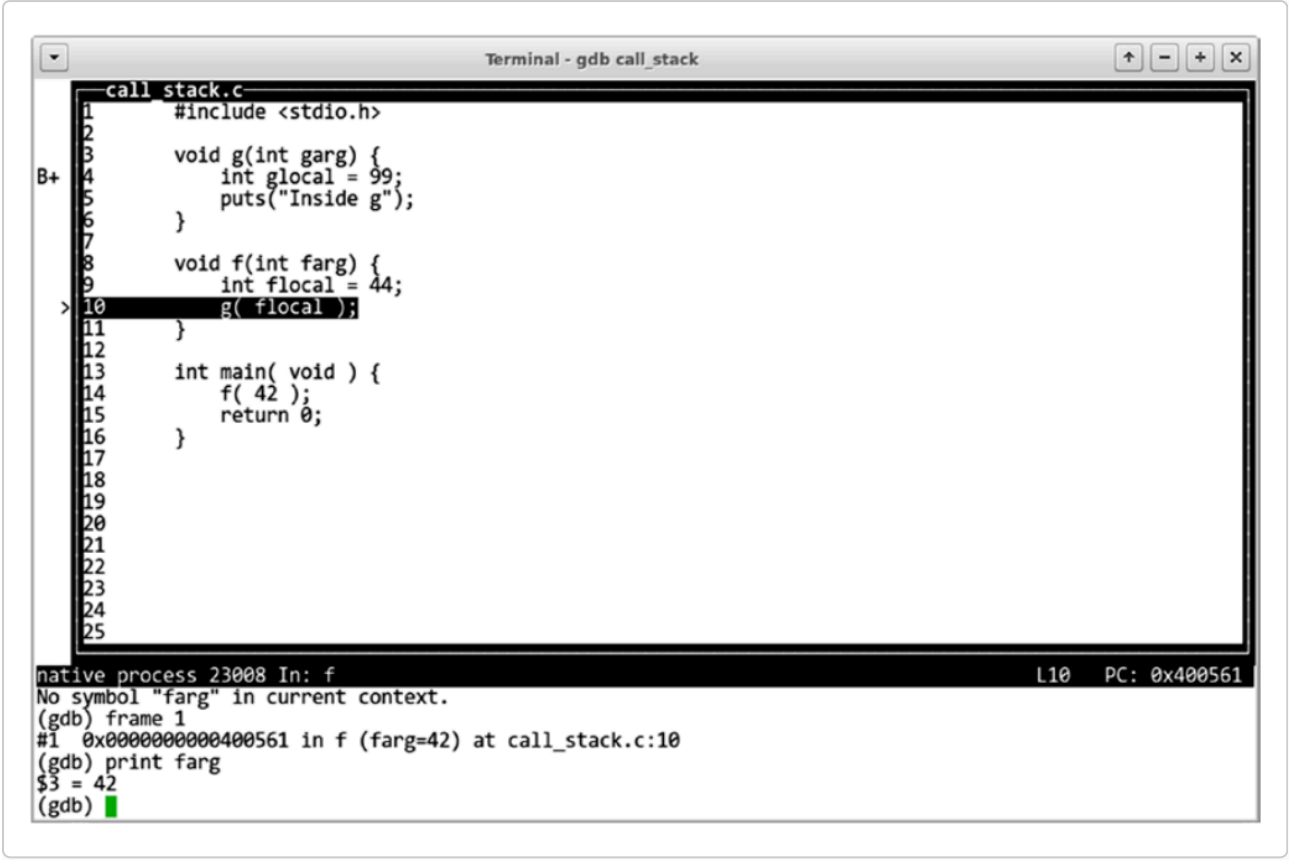


Рисунок 18-4. Внутри функции `f`

Приложение B. Использование Make

Это приложение познакомит вас с самыми базовыми понятиями написания Makefile. Подробнее см. [2]. Для сборки программы может понадобиться выполнить несколько действий: запустить компилятор с правильными флагами, возможно, для каждого исходного файла, и использовать компоновщик. Иногда также приходится запускать скрипты, написанные для генерации файлов исходного кода. Порой программа состоит из нескольких частей, написанных на разных языках программирования! Более того, если вы изменили только часть программы, вы можете не захотеть пересобирать всё, а только те части, которые зависят от изменённого исходного файла. Сборка огромных программ может занимать часы времени CPU (центрального процессора)! В этой книге мы будем использовать GNU Make. Это распространённый инструмент, используемый для управления генерацией артефактов, таких как исполняемые файлы, динамические библиотеки, файлы ресурсов и т. д.

19.1 Простой Makefile

Когда вы пишете программу, вам следует написать для неё специальный `Makefile`, чтобы можно было использовать Make для её сборки. Этот текстовый файл декларативно описывает исходные файлы и зависимости между ними. Затем `make` выберет правильный порядок обработки файлов так, чтобы к моменту обработки каждого файла его зависимости уже были обработаны. Чтобы начать процесс сборки, выполните `make` в каталоге, где создан `Makefile`. Обычно это корневой каталог вашего проекта. Можно явно выбрать другой `Makefile`, передав флаг `-f`, например: `make -f Makefile_other`. Базовый `Makefile` состоит из следующих блоков, каждый из которых называется *правилом*:

```
<target> : <prerequisites>
[tab] <recipe>
```

Правило описывает, как сгенерировать конкретный файл, являющийся `<target>`. `<prerequisites>` описывают, какие другие цели нужно сгенерировать сначала. `recipe` состоит из одного или нескольких действий, выполняемых `make`. Перед каждой строкой рецепта должен стоять символ `[tab]`! Допустим, у нас есть простая программа, состоящая из двух файлов ассемблера: `main.asm` и `lib.asm`. Мы хотим получить объектный файл для каждого из них, а затем скомпоновать их в исполняемый файл `program`. Листинг 19-1 показывает пример простого `Makefile`.

Листинг 19-1. `Makefile_simple`

```
program: main.o lib.o
    ld -o program main.o lib.o
lib.o: lib.asm
    nasm -f elf64 -o lib.o lib.asm
main.o: main.asm
    nasm -f elf64 -o main.o main.asm
clean:
    rm main.o lib.o program
```

Когда создан `Makefile` с таким содержимым, выполнение `make` в том же каталоге запустит рецепт для первой описанной цели. Если присутствует цель по имени `all`, вместо этого будет выполнен её рецепт. Иначе набор `make targetname` выполнит рецепт для цели `targetname`. Цель `program` должна создать файл `program`. Для этого сначала нужно собрать файлы `main.o` и `lib.o`. Если мы изменим файл `main.o` и снова запустим `make`, перед обновлением `program` будет пересобран только `main.o`, но не `lib.o`. Тот же механизм заставляет пересобирать `lib.o`, когда изменяется `lib.asm`. Так что рецепт запускается, когда нет файла, соответствующего имени цели, или этот файл нужно изменить (потому что одна из его зависимостей была обновлена). По традиции у каждого `Makefile` есть цель `clean`, чтобы избавиться от всех созданных файлов, оставив только исходники. Такие цели, как `clean`, называются *фиктивными целями (Phony Targets)*, потому что они не соответствуют конкретному файлу. Лучше всего перечислять их в отдельном рецепте, соответствующем специальной цели `.PHONY`, вот так:

```
clean:
    rm -f *.o
help:
    echo 'This is the help'
.PHONY: clean help
```

19.2 Добавляем переменные

Дублировать много текста в Makefile не очень уместно. Как только исходных файлов, компилируемых одинаково, становится много, нам надоедает постоянно копировать одни и те же опции компиляции. Переменные решают эту проблему. Переменные объявляются так:

```
variable = value
```

Это не то же самое, что переменные окружения, такие как `PWD`. Их значения подставляются с помощью знака доллара и пары скобок, вот так:

```
$(variable)
```

Теперь мы будем использовать переменные как минимум в следующих случаях:

- Чтобы абстрагировать компилятор (мы сможем легко переключаться между Clang, GCC, MSVC или любым другим компилятором, пока они поддерживают один и тот же набор флагов).
- Чтобы абстрагировать флаги компиляции.

По традиции, в случае C, эти переменные называют

- `CC` для «C compiler» (компилятор C).
- `CFLAGS` для «C compiler flags» (флаги компилятора C).
- `LD` для «link editor» (редактор связей, компоновщик).
- `AS` для «assembly language compiler» (компилятор языка ассемблера).
- `ASFLAGS` для «assembly language compiler flags» (флаги компилятора языка ассемблера).

Дополнительное преимущество в том, что когда мы хотим выбрать флаги компиляции, нам нужно сделать это только в одном месте. Листинг 19-2 показывает изменённый Makefile.

Листинг 19-2. `Makefile_vars`

```
AS = nasm
LD = ld
ASFLAGS = -f elf64
program: main.o lib.o
    $(LD) -o program main.o lib.o
lib.o: lib.asm
    $(AS) $(ASFLAGS) -o lib.o lib.asm
main.o: main.asm
    $(AS) $(ASFLAGS) -o main.o main.asm
clean:
    rm main.o lib.o program
.PHONY: clean
```

Переменную можно оставить пустой, и она развернётся в пустую строку:

```
EMPTYVAR  =
```

Переменная может включать значения других переменных:

```
INCLUDEDIR   = include
CFLAGS       = -c -std=c99 -I$(INCLUDEDIR) -ggdb -Wno-attributes
```

Имена целей поддерживают символ подстановки `%`. В имени цели должен быть только один такой символ. Подстрока, которой соответствует `%`, называется *основой (stem)*. Вхождения `%` в предпосылках заменяются в точности основой. Например, это правило

```
%.o : %.c
    echo "Building an object file"
```

указывает, как собрать любой объектный файл из `.c`-файла с соответствующим именем. Однако прямо сейчас мы не знаем, как использовать эти правила, потому что как только мы попробуем написать команду для компиляции файла, мы столкнёмся с проблемой: мы не знаем точных имён задействованных файлов, а основа недоступна внутри рецепта. Автоматические переменные решают эту проблему.

19.3 Автоматические переменные

Автоматические переменные — особая возможность `make`. Они заново вычисляются для каждого выполняемого правила, и их значения зависят от цели и её предпосылок. Их можно использовать только внутри самого рецепта, но не внутри предпосылок или самой цели. Представьте, что вы хотите скомпилировать каждый `.c`-файл в `.o`-файл с одними и теми же флагами. Неужели нам действительно придётся дублировать все правила? Нет, можно использовать символы подстановки вместе с автоматическими переменными. Автоматических переменных много, но самые часто используемые:

- `$*` Основа.
- `$@` Имя файла цели правила.
- `$<` Имя первой предпосылки.
- `$^` Имена всех предпосылок, разделённые пробелами.
- `$?` Имена всех предпосылок, более новых, чем цель.

Листинг 19-3 показывает пример Makefile, использующий все знания из этого руководства.

Листинг 19-3. `makefile_autovars`

```
CC = gcc
CFLAGS = -std=c11 -Wall
LD = gcc
all: main
main: main.o lib.o
   $(LD) $^ -o $@
%.o: %.c %.h
   $(CC) $(CFLAGS) -c $< -o $@
clean:
   rm -f *.o main
.PHONY: clean
```

Он предполагает следующее дерево проекта:

```
.
 lib.c
 lib.h
 main.c
 main.h
 Makefile
0 directories, 8 files
```

Чистый `make` выполнит следующие команды:

```
> make
gcc -std=c11 -Wall -c main.c -o main.o
gcc -std=c11 -Wall -c lib.c -o lib.o
gcc  main.o lib.o -o main
```

За дальнейшими инструкциями обратитесь к хорошо написанному руководству GNU Make Manual [2].

Приложение С. Системные вызовы

На протяжении этой книги мы использовали несколько системных вызовов. В этом приложении мы собрали информацию о них.

> **Примечание.** Всегда полезно сначала прочитать страницы `man`, например, `man -s 2 write`.

Точные значения флагов и параметров зависят от системы и никогда не должны использоваться в непосредственной форме. Если вы пишете на C, используйте соответствующие заголовки (указанные на страницах `man` для интересующего системного вызова). Если вы пишете на ассемблере, вам придётся использовать LXR или другую онлайн-систему с аннотированным кодом ядра, либо просмотреть эти заголовки C самостоятельно и создать собственные соответствующие `%define`. Приведённые значения верны для следующей системы:

```
> uname -a
Linux 3.16-2-amd64 #1 SMP Debian 3.16.3-2 (2014-09-20) x86_64 GNU/Linux
```

Выполнить системный вызов в ассемблере просто: инициализируйте нужные регистры правильными значениями параметров (в любом порядке) и выполните инструкцию `syscall`. Если вам нужны флаги, вам придётся сначала определить их самостоятельно; мы предоставили вам их точные значения. Помните, что NASM также может вычислять константные выражения, такие как `O_TRUNC|O_RDWR`. Выполнение системного вызова в C обычно происходит как вызов функции, чьё объявление предоставлено в некоторых файлах для включения.

> **Примечание.** В C никогда не используйте значения флагов напрямую, например, подставляя вместо `O_APPEND` 0x1000. Используйте определения (defines), предоставленные в заголовочных файлах, потому что они и более читаемы, и более переносимы. Поскольку у нас нет соответствующих заголовков для ассемблера, нам приходится определять их вручную в файлах ассемблера.

20.1 read

```
ssize_t read(int fd, void *buf, size_t count);
```

Описание Читает из дескриптора файла.

RAX	RDI	RSI	RDX	R10	R8	R9
0	int fd	void* buf	size_t count			

20.1.1 Аргументы

1. `fd` Дескриптор файла, из которого мы читаем. 0 для stdin; используйте системный вызов `open`, чтобы открыть файл по имени.
2. `buf` Адрес первого байта последовательности байт. Полученные байты будут помещены туда.
3. `count` Мы попытаемся прочитать столько байт.

Возвращает `rax` = количество успешно прочитанных байт, -1 при ошибке. Включения для использования в C:

```
#include <unistd.h>
```

20.2 write

```
ssize_t write(int fd, const void *buf, size_t count);
```

Описание Записывает в дескриптор файла.

RAX	RDI	RSI	RDX	R10	R8	R9
1	int fd	const void* buf	size_t count			

20.2.1 Аргументы

1. `fd` Дескриптор файла, в который мы пишем. 1 для stdout, 2 для stderr; используйте системный вызов `open`, чтобы открыть файл по имени.
2. `buf` Адрес первого байта последовательности байт для записи.
3. count Мы попытаемся записать столько байт.

Возвращает `rax` = количество успешно записанных байт, -1 при ошибке. Включения для использования в C:

```
#include <unistd.h>
```

20.3 open

```
int open(const char *pathname, int flags, mode_t mode);
```

Описание Открывает файл с заданным именем (строкой, завершающейся нулём)

RAX	RDI	RSI	RDX	R10	R8	R9
2	const char* filename	int flags	int mode			

20.3.1 Аргументы

1. `filename`	Имя открываемого файла (строка, завершающаяся нулём).
2. `flags`	Описаны ниже. Их можно комбинировать с помощью `|`, например, O_CREAT| O_WRONLY|O_TRUNC.
3. `mode`	Целое число, кодирующее права доступа пользователя, группы и всех остальных. Они похожи на используемые командой `chmod`.
Возвращает `rax` = новый дескриптор для заданного файла, -1 при ошибке.	
Включения для использования в C:	

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

20.3.2 Флаги

- O_APPEND = 0x1000 Дописывать в файл при каждой записи.
- O_CREAT = 0x40 Создать новый файл.

- O_TRUNC = 0x200 Если файл уже существует, является обычным файлом, и режим доступа разрешает запись, он будет усечён до длины 0.
- O_RDWR = 2 Чтение и запись.
- O_WRONLY = 1 Только запись.
- O_RDONLY = 0 Только чтение.

20.4 close

```
int close(int fd);
```

Описание Закрывает файл с заданным именем (строкой, завершающейся нулём)

RAX	RDI	RSI	RDX	R10	R8	R9
2	const char* filename	int flags	int mode			

20.4.1 Аргументы

1. `fd` действительный дескриптор файла, который нужно закрыть.

Возвращает `rax` = ноль при успехе, -1 при ошибке. Глобальная переменная `errno` хранит код ошибки. Включения для использования в C:

```
#include <unistd.h>
```

20.5 mmap

```
void *mmap(
   void *addr, size_t length,
   int prot, int flags,
   int fd, off_t offset);
```

Описание Отображает страницы в виртуальном адресном пространстве на что-либо. Это может быть что угодно, что стоит за «файлом» (устройства, файлы на диске и т. д.) или просто физическая память. В последнем случае страницы анонимны, они не соответствуют ничему, присутствующему в файловой системе. Такие страницы хранят кучу и стеки процесса.

RAX	RDI	RSI	RDX	R10	R8	R9
9	void* addr	size_t len	int prot	int flags	int fd	off_t off

20.5.1 Аргументы

1. addr Подсказка для начального виртуального адреса только что отображаемой области. Мы пытаемся отобразить по этому адресу, а если не получается, позволяем операционной системе (ОС) выбрать его самой. Если 0, всегда будет выбран ОС.
2. `len` Длина отображаемой области в байтах.
3. prot Флаги защиты (см. ниже). Их можно комбинировать с помощью `|`.
4. `flags` Флаги поведения (см. далее). Их можно комбинировать с помощью `|`.
5. `fd` Действительный дескриптор отображаемого файла, игнорируется, если используется флаг поведения `MAP_ANONYMOUS`.
6. `off` Начальное смещение в файле `fd`. Мы пропускаем все байты до этого смещения и отображаем файл начиная с него. Игнорируется, если используется флаг поведения `MAP_ANONYMOUS`.

Возвращает `rax` = указатель на отображённую область, -1 при ошибке. Включения для использования в C:

```
#include <sys/mman.h>
```

20.5.2 Флаги защиты

- PROT_EXEC = 0x4 Страницы можно выполнять.
- `PROT_READ = 0x1` Страницы можно читать.
- PROT_WRITE = 0x2 Страницы можно записывать.
- `PROT_NONE = 0x0` К страницам нельзя обращаться.

20.5.3 Флаги поведения

- MAP_SHARED = 0x1 Страницы можно разделять между процессами.

- `MAP_PRIVATE = 0x2` Страницы не разделяются с другими процессами.

- MAP_ANONYMOUS = 0x20 Страницы не соответствуют никакому файлу в файловой системе.

- `MAP_FIXED = 0x10` Не интерпретировать `addr` как подсказку, а как приказ. Если мы не можем

отобразить страницы, начиная с этого адреса, завершиться неудачей.

> **Примечание.** Чтобы иметь возможность использовать флаг `MAP_ANONYMOUS`, вам может понадобиться определить флаг `_DEFAULT_SOURCE` непосредственно перед подключением соответствующего заголовочного файла, вот так:

```
#define _DEFAULT_SOURCE
#include <sys/mman.h>
```

20.6 munmap

```
int munmap(void *addr, size_t length);
```

Описание Отменяет отображение области памяти заданной длины. Можно отобразить огромную область с помощью `mmap`, а затем отменить отображение её части с помощью `munmap`.

RAX	RDI	RSI	RDX	R10	R8	R9
11	void* addr	size_t len				

20.6.1 Аргументы

1. `addr` Начало области, для которой отменяется отображение.

2. length Длина области, для которой отменяется отображение.

Возвращает `rax` = ноль при успехе, -1 при ошибке. Глобальная переменная `errno` хранит код ошибки. Включения для использования в C:

```
#include <sys/mman.h>
```

20.7 exit

```
void _exit(int status);
```

Описание Завершает процесс.

RAX	RDI	RSI	RDX	R10	R8	R9
60	int status					

20.7.1 Аргументы

1. `status` Код возврата. Сохраняется в переменную окружения `$?`.

Возвращает Ничего. Включения для использования в C:

```
#include <unistd.h>
```

Приложение D. Информация о тестах производительности

Все тесты производительности были проведены на следующей системе:

```
> uname -a
Linux perseus 3.16-2-amd64 #1 SMP Debian 3.16.3-2 (2014-09-20) x86_64 GNU/Linux
> cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 69
model name      : Intel(R) Core(TM) i5-4210U CPU @ 1.70GHz
stepping        : 1
microcode       : 0x1d
cpu MHz         : 2394.458
cache size      : 3072 KB
physical id     : 0
siblings        : 1
core id         : 0
cpu cores       : 1
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic
sep mtrr pge mca cmov pat pse36 clflush dts mmx fxsr sse
sse2 ss syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon
pebs bts nopl xtopology tsc_reliable nonstop_tsc aperfmperf
pni pclmulqdq ssse3 fma cx16 pcid sse4_1 sse4_2 x2apic movbe
popcnt aes xsave avx f16c rdrand hypervisor lahf_lm ida arat
epb pln pts dtherm fsgsbase smep
```

```
bogomips        : 4788.91
clflush size    : 64
cache_alignment : 64
address sizes   : 40 bits physical, 48 bits virtual
power management:
processor       : 1
vendor_id       : GenuineIntel
cpu family      : 6
model           : 69
model name      : Intel(R) Core(TM) i5-4210U CPU @ 1.70GHz
stepping        : 1
microcode       : 0x1d
cpu MHz         : 2394.458
cache size      : 3072 KB
physical id     : 2
siblings        : 1
core id         : 0
cpu cores       : 1
apicid          : 2
initial apicid  : 2
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic
sep mtrr pge mca cmov pat pse36 clflush dts mmx fxsr sse
sse2 ss syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon
pebs bts nopl xtopology tsc_reliable nonstop_tsc aperfmperf
pni pclmulqdq ssse3 fma cx16 pcid sse4_1 sse4_2 x2apic movbe
popcnt aes xsave avx f16c rdrand hypervisor lahf_lm ida arat
epb pln pts dtherm fsgsbase smep
bogomips        : 4788.91
clflush size    : 64
cache_alignment : 64
address sizes   : 40 bits physical, 48 bits virtual
power management:
> cat /proc/meminfo
MemTotal:        1017348 kB
MemFree:          516672 kB
MemAvailable:     565600 kB
Buffers:           32756 kB
Cached:           114944 kB
SwapCached:        10044 kB
Active:           376288 kB
Inactive:          49624 kB
```

```
Active(anon):     266428 kB
Inactive(anon):    12440 kB
Active(file):     109860 kB
Inactive(file):    37184 kB
Unevictable:           0 kB
Mlocked:               0 kB
SwapTotal:        901116 kB
SwapFree:         868356 kB
Dirty:                44 kB
Writeback:             0 kB
AnonPages:        270964 kB
Mapped:            43852 kB
Shmem:               648 kB
Slab:              45980 kB
SReclaimable:      29016 kB
SUnreclaim:        16964 kB
KernelStack:        4192 kB
PageTables:         6100 kB
NFS_Unstable:          0 kB
Bounce:                0 kB
WritebackTmp:          0 kB
CommitLimit:     1409788 kB
Committed_AS:    1212356 kB
VmallocTotal:   34359738367 kB
VmallocUsed:       145144 kB
VmallocChunk:   34359590172 kB
HardwareCorrupted:     0 kB
AnonHugePages:         0 kB
HugePages_Total:       0
HugePages_Free:        0
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
DirectMap4k:       49024 kB
DirectMap2M:      999424 kB
DirectMap1G:           0 kB
```

Библиография

[1] *Debugging with gdb*. Доступно: `http://sourceware.org/gdb/current/`

```
onlinedocs/gdb/. 2017.
```

[2] *Gnu make manual*. Доступно: `www.gnu.org/software/make/manual/`. 2016.

[3] *How initialization functions are handled*. Доступно: `https://gcc.gnu.org/`

```
onlinedocs/gccint/Initialization.html. 2017.
```

[4] *Using ld, the gnu linker*. Доступно: `www.math.utah.edu/docs/info/ld_3.html`. 1994.

[5] *What is map-reduce?* Доступно: `www-01.ibm.com/software/data/infosphere/`

```
hadoop/mapreduce/. 2017.
```

[6] Jeff Andrews. *Branch and loop reorganization to prevent mispredicts*.

```
Available: https://software.intel.com/en-us/articles/branch-and-loop-
reorganization-to-prevent-mispredicts. May 2011.
```

[7] C11 language standard — черновик комитета. `www.open-std.org/jtc1/sc22/wg14/www/` standards. Апрель 2011.

[8] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.* 17(4):471–523. Декабрь 1985.

[9] Ryan A. Chapman. *Linux 3.2.0-33 syscall table, x86 64*. `www.cs.utexas.`

```
edu/~bismith/test/syscalls/syscalls64_orig.html.
```

[10] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to algorithms*. Нью-Йорк: McGraw-Hill Higher Education, 2-е изд., 2001.

[11] Russ Cox. *Regular expression matching can be simple and fast*. `https://swtch.`

```
com/~rsc/regexp/regexp1.html. November 2007.
```

[12] Ulrich Drepper. *What every programmer should know about memory*. `https://`

```
people.freebsd.org/~lstewart/articles/cpumemory.pdf. November 2007.
```

[13] Ulrich Drepper. *How to write shared libraries*. `https://software.intel.com/`

```
sites/default/files/m/a/1/e/dsohowto.pdf. December 2011.
```

[14] Jens Gustedt. *Myth and reality about inline in c99*. `https://gustedt.wordpress.`

```
com/2010/11/29/myth-and-reality-about-inline-in-c99/.  2010.
```

Intel Corporation. Intel® 64 and IA-32 architectures software developer's manual. [15]

```
Available: www.intel.com/content/dam/www/public/us/en/documents/
manuals/64-ia-32-architectures-software-developer-manual-325462.pdf.
```

Сентябрь 2014. Intel Corporation. *Intel® 64 and IA-32 architectures optimization reference* [16]

```
manual. Available: www.intel.com/content/www/us/en/architecture-and-
technology/64-ia-32-architectures-optimization-manual.html. June 2016.
```

[17] David Kanter. *Intel's Haswell CPU microarchitecture*. Доступно:

```
www.realworldtech.com/haswell-cpu/1.
```

[18] Brian W. Kernighan. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2-е издание, 1988. [19] Petter Larsson and Eric Palmer. Image processing acceleration techniques using intel streaming simd extensions and intel advanced vector extensions. Январь 2010. [20] Doug Lea. *A memory allocator*. `http://g.oswego.edu/dl/html/malloc.html`. 2000. [21] Michael E. Lee. *Optimization of computer programs in c*. Доступно:

```
http://leto.net/docs/C-optimization.php.
```

[22] Lomont, Chris. «Fast inverse square root.» Tech-315 nical Report (2003):

32. Февраль 2003.

[23] Love, Robert. Linux Kernel Development (Novell Press). Novell Press, 2005 год. [24] Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. System V Application Binary Interface. AMD64 Architecture Processor Supplement. Черновая версия 0.99.6, 2013 год. [25] McKenney, Paul E. «Memory barriers: a hardware view for software hackers.» Linux Technology Center, IBM Beaverton (2010 год). [26] Pawell Moll. How do debuggers (really) work? В сборнике *Embedded Linux Conference*

```
Europe, October 2015. http://events.linuxfoundation.org/sites/events/
files/slides/slides_16.pdf.
```

[27] *The netwide assembler: NASM manual*. Доступно: `www.nasm.us/doc/`. [28] N. N. Nepeyvoda and I. N. Skopin. *Foundations of programming*. RHD Москва-Ижевск, 2003. [29] Benjamin C. Pierce. *Types and programming languages*. Кембридж, Массачусетс: MIT Press, 1-е изд. 2002. [30] Jeff Preshing*. The purpose of memory order consume in c++11.* `http://preshing.`

```
com/20140709/the-purpose-of-memory_order_consume-in-cpp11/. 2014.
```

[31] Jeff Preshing. *Weak vs. strong memory models* `http://preshing.com/20120930/`

```
weak-vs-strong-memory-models/. 2012.
```

[32] Brad Rodriguez. *Moving forth: a series on writing Forth kernels*. `http://www.bradrodriguez.com/papers/moving1.html`. The Computer Journal #59 (Январь/Февраль 1993). [33] Uresh, Vahalia. *«UNIX Internals: The New Frontiers.»* (2005). Dorling Kindersley Pvt. Limited. 2008.

[34] Anthony Williams. *C++ concurrency in action: practical multithreading*. Шелтер-Айленд, Нью-Йорк: Manning. 2012 год.

[35] Glynn Winskel. *The formal semantics of programming languages: an introduction*. Кембридж, Массачусетс: MIT Press. 1993 год.

Сноски

Примечания

1. Обратите внимание, как часто решения, придуманные инженерами, продиктованы экономическими причинами, а не техническими ограничениями. ↩
2. Также известна как x86_64 и AMD64. © Igor Zhirkov 2017 I. Zhirkov, Low-Level Programming, DOI 10.1007/978-1-4842-2403-8_2 ↩
3. В руководстве NASM также используется название «псевдоинструкция» для определённого подмножества директив. ↩
4. Помните: весь исходный код, включая листинги, можно найти на `www.apress.com/us/book/9781484224021`, а также он хранится в домашнем каталоге предварительно настроенной виртуальной машины! ↩
5. Даже если нет, вскоре последовательное выполнение приведёт процессор к концу выделенных виртуальных адресов, см. раздел 4.2. В конце концов операционная система завершит программу, потому что маловероятно, что она сможет от этого оправиться. ↩
6. Нижний индекс обозначает основание системы счисления. ↩
7. Это действие невозможно закодировать с помощью команды `mov`. Проверьте документацию Intel, чтобы убедиться, что оно не реализовано. ↩
8. На самом деле, уменьшая `rsp`, вы выделяете память на стеке. ↩
9. Мы считаем пробел, табуляцию и переводы строк пробельными символами. Их коды — 0x20, 0x9 и 0x10 соответственно. © Igor Zhirkov 2017 I. Zhirkov, Low-Level Programming, DOI 10.1007/978-1-4842-2403-8_3 ↩
10. В этой книге мы немного упрощаем, потому что некоторые структуры данных могут иметь разный формат в зависимости от размера страницы и т. д. Документация даст вам самые точные ответы (читайте том 3, главу 3 [15] ↩
11. В этом случае документация называет этот флаг `B`. © Igor Zhirkov 2017 I. Zhirkov, Low-Level Programming, DOI 10.1007/978-1-4842-2403-8_4 ↩
12. Происходит прерывание #PF (Page Fault, страничный отказ). ↩
13. Чтобы найти идентификатор процесса, используйте такие стандартные программы, как `ps` или `top`. ↩
14. Теоретически мы могли бы поддерживать все 64 бита физических адресов, но пока нам не нужно так много адресов. © Igor Zhirkov 2017 I. Zhirkov, Low-Level Programming, DOI 10.1007/978-1-4842-2403-8_5 ↩
15. Д. Кнут доводит эту идею до крайности в своём подходе под названием Literate Programming (грамотное программирование) ↩
16. Простая формула для суммы первых n натуральных чисел: nn + ↩

17. Не путать с символами препроцессора! © Igor Zhirkov 2017 I. Zhirkov, Low-Level Programming, DOI 10.1007/978-1-4842-2403-8_6 ↩
18. См. раздел 6.3.1 третьего тома [15] © Igor Zhirkov 2017 I. Zhirkov, Low-Level Programming, DOI 10.1007/978-1-4842-2403-8_7 © Igor Zhirkov 2017 I. Zhirkov, Low-Level Programming, DOI 10.1007/978-1-4842-2403-8_8 ↩
19. Один из самых известных трюков называется «устройство Даффа» (Duff's device) и включает цикл, определённый внутри `switch` и содержащий несколько case. ↩
20. Здесь мы говорим о памяти абстрактной машины C. Разумеется, компилятор вправе оптимизировать переменные и никогда не выделять для них настоящую память на уровне ассемблера. Программист, однако, этим не ограничен и может считать, что каждая переменная — это адрес ячейки памяти. © Igor Zhirkov 2017 I. Zhirkov, Low-Level Programming, DOI 10.1007/978-1-4842-2403-8_9 ↩
21. Этот недостаток в дизайне языка исправлен в C++, где у `'x'` тип `char`. ↩
22. Ключевое слово — обычные арифметические преобразования. ↩
23. До C99; но даже сегодня многие не рекомендуют массивы переменной длины, потому что если размер массива достаточно велик, стек не сможет его вместить, и программа будет завершена. ↩
24. POSIX — семейство стандартов, определённых IEEE Computer Society. Оно включает описание утилит, интерфейса прикладного программирования (API) и т. д. Его цель — облегчить переносимость программного обеспечения, в основном между разными ветвями систем, произошедших от UNIX. ↩
25. Заметьте, что это может не заработать «из коробки» для более широких типов из-за возможных промежутков между полями `struct`. © Igor Zhirkov 2017 I. Zhirkov, Low-Level Programming, DOI 10.1007/978-1-4842-2403-8_10 © Igor Zhirkov 2017 I. Zhirkov, Low-Level Programming, DOI 10.1007/978-1-4842-2403-8_11 ↩
26. Если быть точным, результат такой операции не вполне определён. © Igor Zhirkov 2017 I. Zhirkov, Low-Level Programming, DOI 10.1007/978-1-4842-2403-8_12 ↩
27. Для парсеров языков программирования гораздо проще брать в качестве терминальных символов ключевые слова и классы слов (такие как идентификаторы или литералы). Разбиение их на отдельные символы вносит ненужную сложность. © Igor Zhirkov 2017 I. Zhirkov, Low-Level Programming, DOI 10.1007/978-1-4842-2403-8_13 © Igor Zhirkov 2017 I. Zhirkov, Low-Level Programming, DOI 10.1007/978-1-4842-2403-8_14 © Igor Zhirkov 2017 I. Zhirkov, Low-Level Programming, DOI 10.1007/978-1-4842-2403-8_15 ↩
28. Мы не будем рассказывать, что такое хеш-таблицы и как они реализованы, но если вы о них не знаете, мы настоятельно советуем вам о них почитать! Это абсолютно классическая структура данных, используемая повсюду. Хорошее объяснение можно найти в [10] ↩
29. Не путайте с флагом -O компилятора! ↩
30. Широко используемая вероятностная структура данных. Она позволяет быстро проверить, содержится ли элемент в определённом множестве, но ответ «да» подлежит дополнительной проверке, тогда как «нет» всегда достоверно. ↩
31. Так бывает не всегда, например, OS X рекомендует делать все исполняемые файлы позиционно-независимыми. ↩
32. Это имя специфично для ELF, и для других систем его следует изменить. См. раздел 9.2.1 [27]. ↩
33. Опция `-fpic` подразумевает ограничение на размер GOT для некоторых архитектур, что часто быстрее. ↩

34. Не все компиляторы и версии GCC поддерживают большую модель. ↩
35. Заметьте, что для разных архитектур существуют разные описания. ↩
36. Если вам встретится инструкция `movabs`, считайте её эквивалентной инструкции `mov`. ↩
37. Очевидно, здесь `r15` и `rbx` хранят не начало GOT, а её конец, но это не важно. © Igor Zhirkov 2017 I. Zhirkov, Low-Level Programming, DOI 10.1007/978-1-4842-2403-8_16 ↩
38. Мы опускаем разговор о кэше инструкций ради краткости. © Igor Zhirkov 2017 I. Zhirkov, Low-Level Programming, DOI 10.1007/978-1-4842-2403-8_17 ↩
39. Эта опция задокументирована как платформо-специфичная, так что на некоторых платформах она может быть недоступна. ↩
40. Не забывайте не возвращать адрес локальной переменной! ↩
41. В этом случае сам семафор будет размещён на общей странице, которая не будет физически продублирована после выполнения системного вызова `fork()` ↩
42. Или `eax, ax, al` — в зависимости от размера операнда © Igor Zhirkov 2017 I. Zhirkov, Low-Level Programming, DOI 10.1007/978-1-4842-2403-8_18 © Igor Zhirkov 2017 I. Zhirkov, Low-Level Programming, DOI 10.1007/978-1-4842-2403-8_19 © Igor Zhirkov 2017 I. Zhirkov, Low-Level Programming, DOI 10.1007/978-1-4842-2403-8_20 © Igor Zhirkov 2017 I. Zhirkov, Low-Level Programming, DOI 10.1007/978-1-4842-2403-8_21 ↩