EXPERT INSIGHT

Экстремальный Си

Параллелизм, ООП и продвинутые возможности

Камран Амини

Packt

Extreme C

Taking you to the limit in Concurrency, OOP, and the most advanced capabilities of C

Kamran Amini

Packt>

BIRMINGHAM - MUMBAI

Камран Амини

Экстремальный Си

Параллелизм, ООП
и продвинутые возможности

ПИТЕР®

Санкт-Петербург • Москва • Минск

2021

ББК 32.973.2-018.1
УДК 004.43
А62

Амини Камран

А62 Экстремальный Си. Параллелизм, ООП и продвинутые возможности. — СПб.: Питер, 2021. — 752 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-4461-1694-2

Для того чтобы овладеть языком Си, знания одного лишь синтаксиса недостаточно. Специалист в области разработки должен обладать четким научным пониманием принципов и методик. Книга «Экстремальный Си» научит вас пользоваться продвинутыми низкоуровневыми возможностями языка для создания эффективных систем, чтобы вы смогли стать экспертом в программировании на Си.

Вы освоите директивы препроцессора, макрокоманды, условную компиляцию, указатели и многое другое. Вы по-новому взглянете на алгоритмы, функции и структуры. Узнаете, как выжимать максимум производительности из приложений с ограниченными ресурсами.

В XXI веке Си остается ключевым языком в машиностроении, авиации, космонавтикн и многих других отраслях. Вы узнаете, как язык работает с Unix, как реализовывать принципы объектно-ориентированного программирования, и разберетесь с многопроцессной обработкой.

Камран Амини научит вас думать, сомневаться и экспериментировать. Эта книга просто необходима для всех, кто хочет поднять знания Си на новый уровень.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018.1
УДК 004.43

ISBN 978-1789343625 англ.
ISBN 978-5-4461-1694-2

Краткое содержание

Оглавление

Об авторе

Камран Амини специализируется на ядре и встроенных системах. Он работал инженером, архитектором, консультантом и техническим директором во множестве известных иранских компаний. В 2017 году переехал в Европу, где трудился старшим архитектором и инженером в таких солидных компаниях, как Jeppesen, Adecco, TomTom и ActiveVideo Networks. Во время своего пребывания в Амстердаме Камран и написал эту книгу. Его больше всего интересуют следующие темы: теория алгоритмов, распределенные системы, машинное обучение, теория информации и квантовые вычисления.

Хочу поблагодарить маму Эхтирам, которая посвятила жизнь воспитанию меня и моего брата Ашкана. Она всегда нас поддерживает.

Хочу также поблагодарить мою прекрасную и любимую жену Афсанех, которая поддерживала меня на каждом этапе, особенно во время работы над этой книгой. Без ее терпения и поддержки я бы никогда не справился.

О научных редакторах

Алиакбар Аббаси — разработчик ПО с более чем восьмилетним опытом использования различных технологий и языков программирования. Специалист в ООП, C/C++ и Python. Любит читать техническую литературу и расширять свой кругозор в области программирования. В настоящее время проживает в Амстердаме и работает старшим программистом в компании TomTom.

Рохит Талвалкар — очень опытный специалист в программировании на языках C, C++ и Java. На его счету разработка приложений, драйверов и сервисов для проприетарной версии *RTOS (Real Time OS)*, Windows, устройств на основе Windows Mobile и платформы Android.

Рохит получил диплом бакалавра технических наук в индийском Технологическом институте города Мумбаи, а также диплом магистра CS. В настоящее время занимает должность ведущего разработчика приложений в сфере смешанной реальности. Рохит успел поработать в Motorola и BlackBerry и сейчас является сотрудником компании Magic Leap, которая выпускает очки смешанной реальности и специализируется на пространственных вычислениях. В свое время редактировал книгу *C++ for the Impatient* Брайана Оверленда.

Хочу поблагодарить доктора Кловиса Тондо, который научил меня C, C++, Java и многому другому.

Введение

В современном мире мы регулярно имеем дело с умопомрачительными технологиями, которые невозможно было представить еще несколько десятилетий назад. На улицах начинают появляться беспилотные автомобили. Достижения в физике и других науках меняют наши представления о реальности как таковой. Мы читаем новости о том, как исследователи делают первые шаги в квантовых вычислениях, о блокчейне и криптовалютах, о планах колонизации других планет. Невероятно, но в основе таких разнообразных по своей природе достижений лежат всего несколько технологий, одной из которых посвящена эта книга. Речь идет о языке C.

Я начал программировать на C++ в девятом классе, присоединившись к команде юных разработчиков, занимавшейся созданием 2D-симулятора игры в футбол. Вскоре после C++ я начал изучать Linux и C. Стоит признать, что в те времена важность C и Unix не была для меня столь очевидной, но, постепенно получая опыт использования этих технологий в разных проектах и все больше узнавая о них, я осознал, насколько важную роль они играют. Чем ближе я знакомился с C, тем сильнее уважал этот язык программирования. В конце концов я решил стать профессиональным программистом на C. Мне хотелось делиться своими знаниями с другими людьми, чтобы они тоже понимали всю важность этой технологии. Данная книга стала результатом моих амбиций.

Бытует заблуждение, будто C — мертвый язык, и в целом многие технические специалисты имеют о нем туманное представление. Чтобы убедиться в обратном, достаточно взглянуть на рейтинг TIOBE по адресу www.tiobe.com/tiobe-index. На самом деле C наряду с Java — один из самых известных языков за прошедшие 15 лет, и в последние годы он только набирал популярность.

Я подошел к написанию этой книги, имея многолетний опыт в разработке и проектировании с помощью C, C++, Golang, Java и Python на разных платформах, включая различные версии BSD Unix, Linux и Microsoft Windows. Моей основной целью было вывести навыки читателей на новый уровень, поделиться с ними опытом, полученным тяжелым трудом. Легкой прогулки ждать не стоит, именно поэтому книга называется «Экстремальный Си. Параллелизм, ООП и продвинутые возможности». Мы не станем отвлекаться и сравнивать C с другими языками программирования.

Я попытался сделать текст максимально практичным, однако он все равно содержит большое количество фундаментального теоретического материала, имеющего реальное применение. Множество примеров, представленных здесь, помогут вам подготовиться к тому, с чем вам предстоит столкнуться в реальных проектах.

Возможность взяться за столь весомую тему — большая честь для меня. Это сложно выразить словами, и потому скажу лишь, что мне было очень приятно писать о том, что так близко моему сердцу. Это моя первая книга, и возможностью быть ее автором я обязан Эндрю Валдрону.

Заодно хочу передать привет и поблагодарить Йена Хью, редактора-консультанта, с которым мы бок о бок трудились над каждой главой, Алиакбара Аббаси за его замечания и предложения, а также Кишора Рита, Гаурава Гаваса, Веронику Пэйс и многих других людей, внесших ценный вклад в подготовку и издание этой книги.

Я предлагаю вам стать моими спутниками в этом длинном путешествии. Надеюсь, чтение книги изменит ваш кругозор, поможет вам увидеть C в новом свете и заодно сделает вас отличным программистом.

Для кого эта книга

Книга предназначена для читателей, уже имеющих минимальный уровень знаний в области разработки на C и C++. Основная аудитория — начинающие и middle-программисты на C/C++; именно они смогут извлечь максимальную пользу из прочитанного материала, применяя полученные навыки и знания. Надеюсь, книга поможет им ускорить карьерный рост и стать старшими разработчиками. Кроме того, прочитав ее, они смогут претендовать на большое количество вакансий с высокими требованиями и, как правило, хорошей зарплатой. Те или иные темы могут пригодиться и опытным программистам на C/C++, хотя я ожидаю, что эти люди в целом знакомы с изложенным здесь материалом и могут почерпнуть для себя лишь некоторые полезные детали.

Еще одна категория читателей, которым может пригодиться эта книга, — студенты и научные сотрудники. Возможно, вы получаете высшее образование или учитесь в аспирантуре в любой научной или технической области, такой как информатика, разработка ПО, искусственный интеллект, *Интернет вещей* (Internet of Things, IoT), астрономия, физика частиц и космология, либо занимаетесь исследованиями в этих сферах. Книга позволит повысить уровень ваших знаний о C/C++ и Unix-подобных операционных системах, а также поможет отточить соответствующие навыки программирования. Представленный материал пригодится инженерам и ученым, работающим над сложными, многопоточными или даже многопроцессными системами для удаленного управления устройствами, моделирования, обработки больших объемов данных, машинного/глубокого обучения и т. д.

Структура издания

Книга состоит из семи условных частей, каждая из которых посвящена определенным аспектам программирования на C. В первой части рассматривается создание проекта на C, во второй обсуждается память, в третьей — объектная ориентированность, а в четвертой речь в основном идет о системах Unix и их связях с языком C. В пятой части мы поговорим о конкурентности, в шестой — о межпроцессном взаимодействии, а в седьмой, заключительной части речь пойдет о тестировании и сопровождении кода. Ниже приводится краткое описание каждой из 23 глав книги.

Глава 1 «Основные возможности языка» посвящена основным возможностям C, которые определяют то, как мы используем данный язык. В число главных тем входят определение макросов с помощью препроцессора и директив, указатели на переменные и функции, механизмы вызова функций, а также структуры.

Глава 2 «Компиляция и компоновка» содержит описание сборки проектов на C. Во всех подробностях будут рассмотрены как процесс компиляции в целом, так и отдельные его элементы.

Глава 3 «Объектные файлы» посвящена результатам компиляции проекта на C. Вы познакомитесь с объектными файлами, заглянете внутрь этих файлов и посмотрите, какую информацию из них можно извлечь.

Глава 4 «Структура памяти процесса» исследует внутреннее устройство памяти процесса. Вы узнаете, из каких сегментов состоит память и чем статическая память отличается от динамической.

Глава 5 «Стек и куча» содержит информацию о таких сегментах, как стек и куча. Мы поговорим о переменных, которые в них хранятся, и об управлении их жизненным циклом в C. Вы научитесь передовым практикам работы с кучей.

Глава 6 «ООП и инкапсуляция» — первая из четырех глав, относящихся к объектной ориентированности в C. Мы пройдемся по теории, стоящей за ООП, и будет дано определение важным терминам, которые часто встречаются в технической литературе.

Глава 7 «Композиция и агрегация» посвящена композиции и ее специальной разновидности — агрегации. Мы обсудим их отличия, которые проиллюстрируем с помощью примеров.

Глава 8 «Наследование и полиморфизм» исследует один из самых важных аспектов *объектно-ориентированного программирования (ООП)* — наследование. Я покажу, как происходит наследование между двумя классами и как это можно реализовать в C. Вдобавок мы поговорим еще об одной обширной теме — полиморфизме.

Глава 9 «Абстракция данных и ООП в C++» является заключительной для третьей части книги и отведена теме абстракции. Вы познакомитесь с абстрактными типами

данных и узнаете, как они реализованы в C. Мы обсудим C++ и рассмотрим объектно-ориентированные концепции на примере этого языка.

При обсуждении языка C нельзя не упомянуть о Unix. Глава 10 «История и архитектура Unix» объясняет, почему эти две технологии так тесно связаны между собой и как данный симбиоз способствует их живучести. Мы также рассмотрим архитектуру операционной системы Unix и узнаем, как программы используют предоставляемые ею возможности.

Глава 11 «Системные вызовы и ядра» посвящена пространству ядра в архитектуре Unix. Мы подробно обсудим системные вызовы и рассмотрим, как их можно создавать в Linux. Вдобавок поговорим о разных видах ядер и увидим принцип работы модулей ядра Linux на примере простого модуля.

Глава 12 «Последние нововведения в C» представляет последнюю версию стандарта C под названием C18. Вы увидите, чем она отличается от предыдущей версии, C11. Я также продемонстрирую ряд новых возможностей, которые появились с момента выхода C99.

Глава 13 «Конкурентность» открывает пятую часть и посвящена конкурентности. Мы в основном обсудим среды конкурентного выполнения и их различные свойства, такие как чередование. Я объясню, почему эти системы недетерминистические и каким образом данная особенность может вызывать проблемы конкурентности, такие как состояние гонки.

Глава 14 «Синхронизация» побуждает продолжить наше обсуждение сред конкурентного выполнения и обращает внимание на разные проблемы, которые в них встречаются, включая состояние гонки, конкуренцию за данные и взаимную блокировку. Вы познакомитесь с методиками, позволяющими преодолеть эти проблемы, такими как семафоры, мьютексы и условные переменные.

Глава 15 «Многопоточное выполнение» демонстрирует одновременное выполнение нескольких потоков и способы управления ими. Я также приведу реальные примеры с проблемами конкурентности в C, перечисленные в предыдущей главе.

Глава 16 «Синхронизация потоков» описывает методы синхронизации нескольких потоков. Среди представленных здесь тем можно выделить семафоры, мьютексы и условные переменные.

Глава 17 «Процессы» представляет способы создания или порождения новых процессов. Мы также обсудим пассивные и активные методики разделения состояния между разными процессами и рассмотрим проблемы с конкурентностью, затронутые в главе 14, используя реальные примеры на языке C.

Глава 18 «Синхронизация процессов» в основном посвящена имеющимся механизмам для синхронизации разных процессов, находящихся на одном и том же компьютере, включая межпроцессные семафоры, мьютексы и условные переменные.

Глава 19 «Локальные сокеты и IPC» основное внимание уделяет пассивным методам *межпроцессного взаимодействия* (interprocess communication, IPC). Основной акцент делается на взаимодействии процессов, находящихся на одном компьютере. Вы также познакомитесь с программированием сокетов и научитесь создавать каналы между процессами, принадлежащими разным сетевым узлам.

Глава 20 «Программирование сокетов» представляет сетевое программирование и содержит примеры кода, которые проиллюстрируют разные типы сокетов, включая сокеты домена Unix, а также TCP- и UDP-сокеты, основанные на поточных и датаграммных каналах.

Глава 21 «Интеграция с другими языками» демонстрирует, как библиотеку на C, собранную в виде динамического объектного файла, можно загружать и использовать в программах, написанных на C++, Java, Python и Golang.

Глава 22 «Модульное тестирование и отладка» посвящена тестам разных видов, но мы сосредоточимся на модульном тестировании в C. Вы познакомитесь с библиотеками CMocka и Google Test, предназначенными для написания наборов тестов в C. Применительно к отладке мы пройдемся по различным инструментам, которые позволяют отлаживать разного рода программные ошибки.

Глава 23 «Системы сборки», заключительная, представляет системы сборки, такие как Make, Ninja и Bazel, и один генератор сборочных скриптов, CMake.

Условия, при соблюдении которых книга будет максимально полезной

Как уже объяснялось, от читателя требуется определенный уровень знаний и навыков в области программирования.

- Общее понимание архитектуры компьютера. Вы должны иметь представление о памяти, центральном процессоре, периферийных устройствах и их характеристиках и понимать, как компьютерные программы взаимодействуют с этими элементами.
- Владение основами программирования. Вы должны знать, что такое алгоритм, как проследить за его выполнением, что собой представляет исходный код и как в двоичной системе выполняются арифметические операции.
- Навыки работы с *терминалом* и основными *утилитами командной строки* в Unix-подобных операционных системах, таких как Linux или macOS.
- Понимание таких тем, как условные выражения, разные виды циклов, структуры или классы минимум в одном языке программирования, указатели в C или C++, функции и т. д.

- Знание основ ООП. Требование необязательное, поскольку ООП подробно рассматривается в книге, однако знание этой темы поможет вам лучше понять материал, изложенный в главах, посвященных *объектной ориентированности*.

Кроме того, настоятельно рекомендуется скачать репозиторий с кодом и самостоятельно выполнять команды, которые приводятся в листингах командной оболочки. Пожалуйста, используйте компьютер под управлением Linux, macOS или другой операционной системы, совместимой с POSIX.

Скачивание файлов с примерами кода

Вы можете скачать файлы с кодом, выполнив следующие шаги.

1. Войдите или зарегистрируйтесь на сайте http://www.packt.com.
2. Выберите вкладку Support (Поддержка).
3. Щелкните на ссылке Code Downloads (Загрузка кода).
4. Введите английское название книги (Extreme C) в поле Search (Искать) и следуйте инструкциям, которые появятся на экране.

Скачав файл, не забудьте его распаковать, используя последнюю версию одного из следующих инструментов:

- WinRAR/7-Zip для Windows;
- Zipeg/iZip/UnRarX для Mac;
- 7-Zip/PeaZip для Linux.

Архив с кодом для этой книги также доступен на GitHub по адресу github.com/PacktPublishing/Extreme-C. Все обновления кода вносятся в существующий GitHub-репозиторий.

У нас есть богатая библиотека книг и видеороликов. Примеры кода для них можно найти на сайте github.com/PacktPublishing/.

Условные обозначения

В книге используются листинги кода и командной оболочки. Первые содержат либо код на языке C, либо псевдокод. Пример листинга кода показан ниже (листинг 17.1).

Листинг 17.1. Создание дочернего процесса с помощью API fork (ExtremeC_examples_chapter17_1.c)

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

```
int main(int argc, char** argv) {
  printf("This is the parent process with process ID: %d\n",
         getpid());
  printf("Before calling fork() ...\n");
  pid_t ret = fork();
  if (ret) {
    printf("The child process is spawned with PID: %d\n", ret);
  } else {
    printf("This is the child process with PID: %d\n", getpid());
  }
  printf("Type CTRL+C to exit ...\n");
  while (1);
  return 0;
}
```

Как видите, приведенный выше код можно найти в файле `ExtremeC_examples_chapter17_1.c`, который находится в архиве с примерами для этой книги, в каталоге главы 17. Архив с кодом доступен по ссылке github.com/PacktPublishing/Extreme-C.

Если листинг не связан ни с каким файлом, он содержит псевдокод или код на языке C, который не вошел в архив. Вот пример.

Листинг 13.1. Простое задание с пятью инструкциями

```
Task P {
    1. num = 5
    2. num++
    3. num = num - 2
    4. x = 10
    5. num = num + x
}
```

Иногда некоторые строки в листингах выделены жирным шрифтом. Они обычно обсуждаются перед листингом или после него, а выделяются для того, чтобы вам было легче их найти.

Листинги командной оболочки используются для иллюстрации вывода консольных команд, запускаемых в терминале. Сами команды, как правило, напечатаны жирным шрифтом, а их вывод — обычным. Вот пример.

Терминал 17.6. Чтение из объекта разделяемой памяти, созданного в примере 17.4, и его удаление

```
$ ls /dev/shm
shm0
$ gcc ExtremeC_examples_chapter17_5.c -lrt -o ex17_5.out
$ ./ex17_5.out
Shared memory is opened with fd: 3
The contents of the shared memory object: ABC

$ ls /dev/shm
$
```

Команды начинаются либо с $, либо с #. В первом случае команду следует выполнять от имени обычного пользователя, а во втором — от имени администратора.

Консольные команды обычно выполняют в каталоге соответствующей главы, который находится в архиве с кодом. Если нужно перейти в определенный рабочий каталог, то я вам об этом сообщу.

Курсивом выделены новые термины или слова, на которых нужно акцентировать внимание. Шрифт без засечек используется для ссылок и названий каталогов, а также для элементов интерфейса. Например: «Выберите раздел System info (Системная информация) на панели Administration (Администрирование)». Названия файлов оформляются моноширинным шрифтом.

Предупреждения и важные замечания оформлены так.

Советы и приемы оформлены таким образом.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

1 Основные возможности языка

Эта книга поможет вам получить как базовые, так и углубленные знания, необходимые для разработки и сопровождения реальных приложений на C. Как правило, для написания успешных программ одного лишь синтаксиса языка программирования недостаточно — и это особенно актуально для C, в сравнении с большинством других языков. И потому мы рассмотрим все концепции, с помощью которых вы сможете создавать замечательное ПО, — от простых утилит до сложных многопроцессорных систем.

Глава 1 в основном посвящена конкретным возможностям C, которые, как вы сами увидите, будут чрезвычайно полезными при написании программ. Вы будете применять их в ситуациях, регулярно встречающихся в разработке ПО. О программировании на C написано множество замечательных книг и практических руководств, подробно освещающих почти все аспекты синтаксиса этого языка, но, прежде чем идти дальше, будет полезно обсудить некоторые из его ключевых особенностей.

В число этих особенностей входят директивы препроцессора, указатели на переменные, функции и структуры. Конечно, все это можно встретить и в более современных языках программирования, а аналогичные концепции доступны в Java, C#, Python и т. д. Например, *ссылки* в Java можно считать аналогом указателей на переменные в C. Эти возможности и связанные с ними концепции настолько фундаментальны, что ни один программный компонент не смог бы работать без них, даже если бы его можно было запустить! Даже простейшая программа типа Hello world нуждается в загрузке целого ряда динамических библиотек, что, в свою очередь, требует использования *указателей на функции*!

Светофор, компьютерная система вашего автомобиля, микроволновая печь на вашей кухне, операционная система вашего смартфона или, наверное, любого другого устройства, о котором вы обычно даже не задумываетесь, — все это содержит программные компоненты, написанные на языке C.

Появление C оказало огромное влияние на нашу жизнь, и без него наш мир выглядел бы совсем иначе.

Эта глава посвящена основным возможностям и механизмам, необходимым для написания кода на C. Будут подробно рассмотрены следующие элементы языка.

- *Директивы препроцессора, макросы и условная компиляция*. Наличие препроцессора — одна из особенностей C, которая не характерна для большинства других языков программирования. Препроцессор дает множество преимуществ, и мы обсудим некоторые интересные сценарии его применения, включая *макросы* и *условные директивы*.
- *Указатели на переменные* рассматриваются в соответствующем разделе. Чтобы лучше их понять, мы также обсудим примеры их неправильного использования.
- *Функции*. В соответствующем разделе подробно описывается все, что с ними связано. Мы не станем ограничиваться одним лишь синтаксисом. На самом деле синтаксис — самое простое! Мы будем рассматривать функции в качестве составных элементов для написания *процедурного* кода. Мы также обсудим *механизм вызова функций* и то, как они получают свои аргументы от вызывающего их кода.
- *Указатели на функции*. Это, несомненно, один из важнейших аспектов C. Указатели могут ссылаться не только на переменные, но и на функции. Возможность сохранять указатели на существующую логику крайне важна для разработки алгоритмов, и именно поэтому данной теме посвящен отдельный раздел. Указатели на функции имеют широкий спектр применения, от загрузки динамических библиотек до *полиморфизма*. Они будут повсеместно встречаться в следующих нескольких главах.
- *Структуры* в C имеют простой синтаксис и выражают простую идею, но это основные составляющие элементы для написания *хорошо организованного* и более *объектно-ориентированного* кода. Их важность в сочетании с указателями на функции невозможно переоценить! В последнем разделе данной главы мы еще раз пройдемся по всем аспектам структур в C, о которых вам нужно знать, и рассмотрим присущие им нюансы.

Основные возможности C и сопутствующие концепции играют ключевую роль в экосистеме Unix, благодаря чему этот язык, несмотря на почтенный возраст и строгий синтаксис, является важной и влиятельной технологией. О том, как C и Unix связаны между собой, мы поговорим в следующих главах. А пока сосредоточимся на директивах препроцессора.

Прежде чем продолжим, имейте в виду: вы уже должны быть знакомы с C. В текущей главе в основном представлены обычные примеры, но без знания синтаксиса языка вы не сможете двигаться дальше. Ниже перечислены темы, в которых необходимо ориентироваться любому читателю данной книги.

- Общее понимание архитектуры компьютера. Вы должны иметь представление о памяти, центральном процессоре, периферийных устрой-

ствах и их характеристиках и понимать, как компьютерные программы взаимодействуют с этими элементами.

- Знание основ программирования. Вы должны знать, что такое алгоритм, как проследить за его выполнением, что собой представляет исходный код и как работают арифметические операции в двоичной системе.
- Навыки работы с терминалом и основными утилитами командной строки в Unix-подобных операционных системах, таких как Linux или macOS.
- Владение такими темами, как условные выражения, разные виды циклов, структуры или классы минимум в одном языке программирования, указатели в C или C++, функции и т. д.
- Понимание основ ООП. Требование необязательное, поскольку ООП подробно рассматривается в этой книге, однако знание данной темы поможет вам лучше понять материал, изложенный в главах третьей части, посвященной объектной ориентированности.

Директивы препроцессора

Препроцессор — важная часть C. Мы подробно рассмотрим его в главе 2, но пока будем считать, что это механизм, который позволяет вам генерировать и модифицировать свой исходный код до его передачи компилятору. Это значит, процесс компиляции в C имеет по меньшей мере один дополнительный этап по сравнению с другими языками программирования. В них исходный код сразу попадает в компилятор, но в C и C++ он должен сначала пройти через препроцессор.

Этот дополнительный этап делает C и (C++) уникальным языком программирования, поскольку программист может фактически изменять свой исходный код перед передачей его компилятору. В большинстве высокоуровневых языков программирования такой возможности нет.

Задача препроцессора — заменить специальные директивы подходящим кодом на C и подготовить итоговые исходники к компиляции.

Управлять препроцессором в C и влиять на его поведение можно с помощью набора *директив*. Они представляют собой строчки кода, начинающиеся символом # как в заголовочных, так и в исходных файлах. Эти строчки имеют смысл только для препроцессора, но не для компилятора. C поддерживает различные директивы, но часть из них играют ключевую роль, особенно те, которые используются в определении макросов и условной компиляции.

В следующем подразделе я объясню, что такое макросы, и приведу различные примеры их использования. Кроме того, мы проанализируем их достоинства и недостатки.

Макросы

Макросы в C окружены ореолом таинственности. Одни говорят, что они делают исходный код слишком сложным и малопонятным, а другие уверены, что из-за них возникают проблемы при отладке приложений. Возможно, вы и сами встречали подобные слухи. Но правдивы ли они и если да, то в какой степени? Являются ли макросы злом, которого следует избегать? Или же они имеют определенные преимущества, которые могли бы пригодиться в вашем проекте?

В действительности макросы можно найти в любом известном проекте на C. Чтобы в этом убедиться, скачайте какое-нибудь популярное приложение наподобие HTTP-сервера Apache и поищите в его исходниках `#define` с помощью утилиты `grep`. Вы найдете длинный список файлов, в которых определен данный макрос. Макросы — неотъемлемая часть жизни разработчика на C. Даже если вы не используете их сами, они, скорее всего, попадутся вам в чужом коде. Поэтому вы должны знать, что они собой представляют и как с ними работать.

Утилита grep — стандартная утилита командной строки в Unix-подобных операционных системах, предназначенная для поиска шаблонных выражений в потоках символов. С ее помощью можно искать текст и шаблоны в содержимом всех файлов в заданном каталоге.

Макросы можно применять различными способами. Ниже перечислено несколько примеров:

- определение константы;
- использование вместо обычной функции на C;
- разворачивание цикла;
- предотвращение дублирования заголовков;
- генерация кода;
- условная компиляция.

Макросы применяются во множестве других ситуаций, однако ниже мы сосредоточимся на тех из них, которые перечислены выше.

Определение макроса

Для определения макросов используется директива `#define`. Каждый макрос имеет имя и (иногда) список параметров. У него также есть *значение*, которое подставляется вместо его имени на этапе работы препроцессора под названием

«развертывание макросов». С помощью директивы `#undef` макрос можно сделать *неопределенным*. Начнем с простого примера (листинг 1.1).

Листинг 1.1. Определение макроса (ExtremeC_examples_chapter1_1.c)

```
#define ABC 5

int main(int argc, char** argv) {
  int x = 2;
  int y = ABC;

  int z = x + y;
  return 0;
}
```

В приведенном выше листинге `ABC` — не переменная с целочисленным значением и не целочисленная константа. На самом деле это макрос с именем `ABC`, значение которого равно `5`. После его развертывания итоговый код, который передается компилятору, выглядит примерно так (листинг 1.2).

Листинг 1.2. Код, сгенерированный в результате развертывания макроса из примера 1.1

```
int main(int argc, char** argv) {
  int x = 2;
  int y = 5;
  int z = x + y;
  return 0;
}
```

Код в листинге 1.2 имеет корректный с точки зрения C синтаксис, понятный компилятору. Препроцессор развернул макрос, подставив его значение туда, где было указано его имя, и вдобавок убрал комментарий в начальной строчке.

Теперь рассмотрим еще один пример (листинг 1.3).

Листинг 1.3. Определение функционального макроса (ExtremeC_examples_chapter1_2.c)

```
#define ADD(a, b) a + b

int main(int argc, char** argv) {
  int x = 2;
  int y = 3;
  int z = ADD(x, y);
  return 0;
}
```

В приведенном выше коде, как и в примере 1.1, `ADD` не является функцией. Это всего лишь *функциональный макрос*, который принимает аргументы. После обработки препроцессором итоговый код будет выглядеть следующим образом (листинг 1.4).

Листинг 1.4. Пример 1.2 после обработки препроцессором и развертывания макроса

```
int main(int argc, char** argv) {
  int x = 2;
  int y = 3
  int z = x + y;
  return 0;
}
```

Как видите, произошло следующее развертывание: аргумент `x`, который использовался в качестве параметра `a`, был заменен всеми экземплярами `a` в значении макроса. То же самое произошло с параметром `b` и соответствующим ему аргументом `y`. Затем была произведена заключительная замена, и в обработанном препроцессором коде мы получили `x + y` вместо `ADD(a, b)`.

Поскольку функциональные макросы могут принимать аргументы, с их помощью можно имитировать функции C. Иными словами, вы можете вынести часто используемую логику в функциональный макрос. Таким образом, препроцессор подставит вместо макроса часто применяемую логику и вам не нужно будет создавать новую функцию на языке C. Мы обсудим это более подробно и сравним оба подхода.

Макросы существуют только перед этапом компиляции. То есть компилятор теоретически ничего о них не знает. Это очень важный момент, о котором необходимо помнить, если вы собираетесь использовать макросы вместо функций. О функциях компилятору известно все, поскольку они являются частью грамматики языка C и хранятся в *синтаксическом дереве*. А макрос — просто директива, которую понимает только препроцессор.

Макросы позволяют *генерировать* код перед компиляцией. В других языках программирования, таких как Java, для этого применяются специальные *генераторы кода*. Я приведу примеры того, как это делается в контексте макросов.

Вопреки распространенному заблуждению, современные компиляторы C знают о директивах и анализируют исходный код еще до его обработки препроцессором. Взгляните на следующий пример (листинг 1.5).

Листинг 1.5. Определение макроса, которое вызывает ошибку «необъявленный идентификатор» (example.c)

```
#include <stdio.h>

#define CODE \
printf("%d\n", i);

int main(int argc, char** argv) {
  CODE
  return 0;
}
```

Если скомпилировать приведенный выше код с помощью `clang` в macOS, то получится следующий вывод (терминал 1.1).

Терминал 1.1. Вывод компилятора ссылается на определение макроса

```
$ clang example.c
code.c:7:3: error: use of undeclared identifier 'i'

CODE
^
code.c:4:16: note: expanded from macro 'CODE'
printf("%d\n", i);
               ^
1 error generated.
$
```

Как видите, компилятор сгенерировал сообщение об ошибке, в котором указана строчка с объявлением макроса.

Стоит отметить: большинство современных компиляторов позволяют просматривать результаты работы препроцессора непосредственно перед компиляцией. Например, задействуя `gcc` или `clang`, можно указать параметр `-E`, чтобы вывести обработанный препроцессором код. Пример использования параметра `-E` продемонстрирован в терминале 1.2. Обратите внимание: это лишь часть вывода.

Терминал 1.2. Код example.c после обработки препроцессором

```
$ clang -E example.c
# 1 "sample.c"# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 361 "<built-in>" 3
...
# 412 "/Library/Developer/CommandLineTools/SDKs/MacOSX10.14.sdk/usr/include/
stdio.h" 2 3 4
# 2 "sample.c" 2
...
int main(int argc, char** argv) {
  printf("%d\n", i);
  return 0;
}
$
```

Мы подошли к важной концепции. *Единица трансляции* (или *единица компиляции*) — код на языке C, который прошел через препроцессор и готов к компиляции.

В единице трансляции все директивы заменены подключенными файлами или развернутыми макросами, благодаря чему получается один длинный блок кода на C.

Итак, вы уже познакомились с макросами. Теперь рассмотрим более сложные примеры, которые продемонстрируют всю эффективность и опасность данного механизма. Экстремальное программирование позволяет мастерски обращаться с опасными и тонкими концепциями, и это, как мне кажется, и есть суть языка C.

Ниже показан интересный пример. Обратите внимание на последовательное применение макросов в цикле (листинг 1.6).

Листинг 1.6. Использование макросов для генерации цикла (ExtremeC_examples_chapter1_3.c)

```
#include <stdio.h>

#define PRINT(a) printf("%d\n", a);
#define LOOP(v, s, e) for (int v = s; v <= e; v++) {
#define ENDLOOP }

int main(int argc, char** argv) {
  LOOP(counter, 1, 10)
    PRINT(counter)
  ENDLOOP
  return 0;
}
```

Как видите, код внутри функции `main` нельзя назвать корректным с точки зрения C! Но после обработки препроцессором мы получаем обычный исходный код, который компилируется без проблем. Ниже показан результат работы препроцессора (листинг 1.7).

Листинг 1.7. Код из примера 1.3 после обработки препроцессором

```
...
... содержимое stdio.h …
...
int main(int argc, char** argv) {
  for (int counter = 1; counter <= 10; counter++) {
    printf("%d\n", counter);
  }
  return 0;
}
```

В функции `main` листинга 1.6 для написания алгоритма использовался набор инструкций, непохожих на синтаксис C. Затем после работы препроцессора мы получили листинг 1.7 с полностью рабочим и корректным кодом. Это одна из важных областей применения макросов: создание новых *предметно-ориентированных языков* (domain specific language, DSL) и применение их для написания кода.

Языки DSL могут быть весьма полезными в разных частях проекта; например, они активно применяются в фреймворках для тестирования, таких как Google Test framework (gtest), где с их помощью оформляются определения, ожидания и тестовые сценарии.

Следует отметить: в итоговом коде, прошедшем через препроцессор, нет никаких директив. Это значит, что вместо директивы `#include` в листинге 1.6 было подставлено содержимое файла, на который она ссылалась. Именно поэтому перед функцией `main` в листинге 1.7 можно видеть содержимое заголовочного файла `stdio.h` (которое мы заменили многоточием).

Рассмотрим следующий пример с двумя новыми операторами для работы с параметрами макроса: `#` и `##` (листинг 1.8).

Листинг 1.8. Использование операторов # и ## в макросе (ExtremeC_examples_chapter1_4.c)

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

#define CMD(NAME) \
  char NAME ## _cmd[256] = ""; \
  strcpy(NAME ## _cmd, #NAME);

int main(int argc, char** argv) {

  CMD(copy)
  CMD(paste)
  CMD(cut)

  char cmd[256];
  scanf("%s", cmd);

  if (strcmp(cmd, copy_cmd) == 0) {
    // ...
  }
  if (strcmp(cmd, paste_cmd) == 0) {
    // ...
  }
  if (strcmp(cmd, cut_cmd) == 0) {
    // ...
  }

  return 0;
}
```

При развертывании макроса оператор `#` переводит параметр в строковую форму, заключенную в кавычки. Например, в приведенном выше листинге он превращает параметр `NAME` в `"copy"`.

Оператор ## работает иначе. Он соединяет параметры с другими элементами в определении макроса — обычно в целях формирования имен переменных. Ниже показан исходный код примера 1.4 после обработки препроцессором:

Листинг 1.9. Пример 1.4 после обработки препроцессором

```
...
... содержимое stdio.h ...
...
... содержимое string.h ...
...
int main(int argc, char** argv) {

  char copy_cmd[256] = ""; strcpy(copy_cmd, "copy");
  char paste_cmd[256] = ""; strcpy(paste_cmd, "paste");
  char cut_cmd[256] = ""; strcpy(cut_cmd, "cut");

  char cmd[256];
  scanf("%s", cmd);

  if (strcmp(cmd, copy_cmd) == 0) {

  }
  if (strcmp(cmd, paste_cmd) == 0) {

  }
  if (strcmp(cmd, cut_cmd) == 0) {
  }

  return 0;
}
```

Чтобы понять, как операторы # и ## применяются к аргументам макроса, сравните код до и после работы препроцессора. Обратите внимание: в итоговом коде определение каждого макроса развертывается в одну строчку.

Длинные макросы рекомендуется разбивать на несколько строчек, однако не забывайте использовать обратную косую черту (\), чтобы препроцессор знал, что определение продолжается на следующей строчке. Обратите внимание: \ не заменяется символом новой строки. Это просто говорит о том, что следующая строчка — продолжение определения того же макроса.

Теперь обсудим разные типы макросов. Далее речь пойдет о *вариативных макросах*, которые могут принимать переменное число аргументов.

Вариативные макросы

Следующий пример посвящен вариативным макросам с переменным количеством входящих аргументов. Иногда вариативный макрос принимает два аргумента, иногда четыре, а иногда семь. Это может быть очень полезным, если вы не знаете заранее, сколько аргументов принимает ваш макрос в той или иной ситуации. Ниже показан простой пример (листинг 1.10).

Листинг 1.10. Определение и использование вариативного макроса (ExtremeC_examples_chapter1_5.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define VERSION "2.3.4"

#define LOG_ERROR(format, ...) \
  fprintf(stderr, format, __VA_ARGS__)

int main(int argc, char** argv) {

  if (argc < 3) {
    LOG_ERROR("Invalid number of arguments for version %s\n.", VERSION);
    exit(1);
  }

  if (strcmp(argv[1], "-n") != 0) {
    LOG_ERROR("%s is a wrong param at index %d for version %s.", argv[1], 1,
    VERSION);
    exit(1);
  }

  // ...

  return 0;
}
```

В этом листинге можно заметить новый идентификатор: `__VA_ARGS__`. Препроцессор подставляет вместо него все оставшиеся входящие аргументы, которые еще не были присвоены никаким параметрам.

Взгляните на второй экземпляр `LOG_ERROR`. Согласно определению макроса входящие аргументы `argv[1]`, `1` и `VERSION` *не* присваиваются ни одному из параметров. Поэтому при развертывании макроса они будут использованы вместо `__VA_ARGS__`.

Стоит отметить: функция `fprintf` производит запись в *файловый дескриптор*. В примере 1.5 роль этого дескриптора играет `stderr` — *поток ошибок* процесса.

Вдобавок обратите внимание на точку с запятой после каждого экземпляра LOG_ERROR. Их использование связано с тем, что они не входят в состав макроса, поэтому программист *должен* добавлять их самостоятельно, чтобы после работы препроцессора код был синтаксически корректным.

Ниже показан итоговый код, который выводит препроцессор (листинг 1.11).

Листинг 1.11. Пример 1.5 после обработки препроцессором

```
...
... содержимое stdio.h ...
...
... содержимое stdlib.h ...
...
... содержимое string.h ...
...
int main(int argc, char** argv) {

  if (argc < 3) {
    fprintf(stderr, "Invalid number of arguments for version %s\n.", "2.3.4");
    exit(1);
  }

  if (strcmp(argv[1], "-n") != 0) {
    fprintf(stderr, "%s is a wrong param at index %d for version %s.",
    argv[1], 1, "2.3.4");
    exit(1);
  }

  // ...

  return 0;
}
```

Ниже показан более сложный и довольно распространенный пример использования вариативных макросов с попыткой имитации цикла. Прежде чем в C++ появился цикл foreach, фреймворк *boost* предоставлял (и все еще предоставляет) его аналог на основе ряда макросов.

По следующей ссылке находится определение макроса BOOST_FOREACH (в самом конце заголовочного файла): https://www.boost.org/doc/libs/1_35_0/boost/foreach.hpp. Это функциональный макрос, который используется для итерации коллекций в boost.

В примере 1.6, показанном ниже, определен простой цикл, который и в подметки не годится циклу foreach из boost. Тем не менее это хорошая демонстрация того, как с помощью вариативных макросов можно повторять разные инструкции (листинг 1.12).

Листинг 1.12. Имитация цикла с помощью вариативных макросов (ExtremeC_examples_chapter1_6.c)

```
#include <stdio.h>

#define LOOP_3(X, ...) \
  printf("%s\n", #X);

#define LOOP_2(X, ...) \
  printf("%s\n", #X); \
  LOOP_3(__VA_ARGS__)

#define LOOP_1(X, ...) \
  printf("%s\n", #X); \
  LOOP_2(__VA_ARGS__)

#define LOOP(...) \
  LOOP_1(__VA_ARGS__)

int main(int argc, char** argv) {

  LOOP(copy paste cut)
  LOOP(copy, paste, cut)
  LOOP(copy, paste, cut, select)

  return 0;
}
```

Чтобы объяснить, как работает этот пример, нужно сначала взглянуть на итоговый код, прошедший обработку препроцессора. Так вам будет легче понять, что же на самом деле произошло (листинг 1.13).

Листинг 1.13. Пример 1.6 после обработки препроцессором

```
...
... содержимое stdio.h ...
...
int main(int argc, char** argv) {

  printf("%s\n", "copy paste cut"); printf("%s\n", "");
  printf("%s\n", "");
  printf("%s\n", "copy"); printf("%s\n", "paste"); printf("%s\n", "cut");
  printf("%s\n", "copy"); printf("%s\n", "paste"); printf("%s\n", "cut");

  return 0;
}
```

Если внимательно присмотреться к преобразованному коду, то можно заметить, что макрос `LOOP` был развернут не в цикл наподобие `for` или `while`, а в повторяющиеся инструкции `printf`. Причина этого очевидна и связана с тем, что препроцессор не занимается написанием замысловатого кода на C. Он всего лишь заменяет макросы теми инструкциями, которые мы сами предоставили.

Имитировать цикл с помощью макроса можно только одним способом: разместить отдельные повторяющиеся инструкции одна за другой. Это значит, что простой цикл с 1000 итерациями превратится в 1000 инструкций в коде на C и в результате у нас не будет никакого реального цикла.

Использование представленного выше метода увеличивает размер двоичного файла, что можно считать недостатком. Размещение инструкций друг за другом вместо того, чтобы выполнять их в цикле, называется *развертыванием цикла* (loop unrolling) и хорошо подходит в определенных ситуациях: например, когда нужен приемлемый уровень производительности независимо от того, сколько ресурсов доступно в той или иной среде. Учитывая все вышесказанное, развертывание цикла можно считать компромиссом между размером двоичного файла и производительностью. Мы еще вернемся к этому далее.

Нужно сделать еще одно замечание относительно предыдущего примера. Как видите, разные способы использования макроса `LOOP` в функции `main` дали различные результаты. В первом случае мы передали `copy paste cut` без запятых между словами. Препроцессор воспринял это выражение как одно входящее значение, и потому имитация цикла состояла всего из одной итерации.

Во втором случае было передано значение `copy, paste, cut`, но уже с запятыми между словами. Препроцессор решил, что это три отдельных аргумента, и потому имитация цикла имела три итерации. Наглядная демонстрация приведена в терминале 1.3.

В третьем случае из четырех переданных слов: `copy, paste, cut, select` — было обработано только три. После преобразования получился точно такой же код, как и во втором сценарии. Это объясняется тем фактом, что наши циклические макросы рассчитаны на списки не длиннее трех элементов. Все идущее после третьего значения игнорировалось.

Обратите внимание: это не вызывает никаких ошибок компиляции, поскольку итоговый сгенерированный код на C получается полностью корректным. Тем не менее наши макросы поддерживают ограниченное количество элементов.

Терминал 1.3. Компиляция и вывод примера 1.6

```
$ gcc ExtremeC_examples_chapter1_6.c
$ ./a.out
co py paste cut


copy
paste
cut
$
```

Преимущества и недостатки макросов

Для начала немного поговорим о проектировании программного обеспечения. Определение макросов и их совместное использование — своеобразное, порой увлекательное искусство! Прежде чем приступать к написанию макросов, вы пытаетесь представить, как будет выглядеть итоговый код. Этот механизм позволяет легко копировать и модифицировать ваш код, что располагает к злоупотреблению. Чрезмерное применение макросов может стать большой проблемой не только для вас, но и для ваших коллег. В чем же причина?

Макросы имеют важную особенность: перед компиляцией вместо них подставляются другие строчки, в результате чего вы получаете один длинный блок кода, лишенный всякой модульности. Конечно, модульность остается в вашем замысле и, наверное, в самом макросе, но в скомпилированных двоичных файлах ее нет. Вот тут-то макросы и начинают создавать архитектурные проблемы.

При проектировании ПО мы пытаемся упаковать похожие алгоритмы и концепции в несколько аккуратных *модулей*, которые можно использовать повторно, но макросы стремятся сделать все линейным и плоским. Поэтому если вы применяете их в качестве логических элементов архитектуры своего приложения, то информация о них может быть утеряна после работы препроцессора и не попасть в итоговые единицы трансляции. В связи с этим архитекторы и проектировщики ПО руководствуются следующим правилом.

> Если макрос можно оформить в виде функции языка C, то делайте выбор в пользу функции!

Макросы считаются плохим решением и с точки зрения отладки. Разработчики постоянно используют ошибки компиляции (наряду с *журнальными записями* и иногда с *предупреждениями компиляции*) для поиска участков кода с неправильным синтаксисом. Предупреждения и ошибки компиляции помогают анализировать допущенные оплошности, и генерируются они компилятором.

Старые компиляторы языка C ничего не знали о макросах и обращались с компилируемым исходником (единицей трансляции) как с длинным, линейным, плоским блоком. Поэтому разработчик видел одно (оригинальный код на C с макросами), а компилятор — совершенно другое (преобразованный код без макросов). В результате вывод компилятора было сложно понять.

Есть надежда, что эта проблема уже не настолько серьезная. Популярные современные компиляторы C, такие как `gcc` и `clang`, знают больше о работе препроцессора и пытаются использовать в своих сообщениях тот исходный код, который видит разработчик. В остальном, если применять директивы `#include`, проблема остается, поскольку основное содержимое единицы трансляции известно лишь после

подключения всех файлов. В заключение можно отметить, что проблема с отладкой не столь серьезная по сравнению со сложностями при проектировании ПО, которые мы описали несколькими абзацами выше.

Помните, о чем мы говорили при разборе примера 1.6? Речь шла о компромиссе между размером двоичного файла и производительностью программы. Более общая разновидность этого компромисса заключается в выборе между одним большим исполняемым файлом и множеством мелких. Оба варианта предоставляют одни и те же возможности, но первый может быть более производительным.

Количество двоичных файлов, используемых в проекте (особенно если он большой) более или менее прямо пропорционально уровню *модульности* и усилиям, потраченным на проектирование. Например, если проект состоит из 60 библиотек (динамических или статических), вызываемых из одной программы, то это значит, что план разработки подразумевал разбиение зависимостей на разные библиотеки с их последующим использованием в одном главном исполняемом файле.

Иными словами, когда проект разрабатывается в соответствии с общепринятыми принципами проектирования ПО, он обычно состоит из множества легковесных двоичных файлов с минимально допустимыми размерами, а не из одного огромного исполняемого файла.

В ходе проектирования мы не используем линейную структуру; каждый программный компонент получает подходящее место в развитой иерархии. И это по определению будет иметь отрицательное (хотя в большинстве случаев незначительное) влияние на производительность.

Из этого следует, что пример 1.6 на самом деле иллюстрирует баланс между архитектурой и производительностью. Если вам нужна высокая производительность, то иногда приходится жертвовать стройной архитектурой и размещать компоненты линейно. Например, вы можете *развернуть* свои циклы.

С другой стороны, обеспечение высокой производительности начинается с выбора подходящих алгоритмов для решения задач, определенных на этапе проектирования. Затем обычно идет *оптимизация* или *улучшение производительности*. На данном этапе производительность повышается за счет выполнения вычислений линейным и последовательным образом, без принудительных переходов на разные участки кода. Для этого можно использовать другие, производительные (и обычно более сложные) алгоритмы или модифицировать уже имеющиеся. Здесь может возникнуть конфликт с философией архитектуры. Как уже упоминалось ранее, в ходе проектирования мы пытаемся организовать компоненты в виде иерархии и избавиться от линейности, но центральный процессор рассчитан на то, что все инструкции уже загружены в линейном порядке и готовы к обработке. Поэтому данный компромисс необходимо продумать и сбалансировать для каждой отдельной задачи.

Обсудим развертывание циклов более подробно. Эта методика в основном применяется в разработке для встраиваемых систем и особенно в средах с ограниченными

вычислительными ресурсами. Она представляет собой замену циклов линейными инструкциями, что улучшает производительность и избавляет от накладных расходов на циклическое выполнение.

Именно это мы и сделали в примере 1.6; мы имитировали цикл с помощью макросов, получив тем самым линейный набор инструкций. В каком-то смысле можно сказать, что макросы позволяют улучшить производительность во встраиваемых системах и средах, в которых даже незначительное изменение способа выполнения инструкций существенно повышает скорость работы. Более того, макросы могут сделать код более понятным и избавить нас от повторяющихся блоков кода.

Что касается приведенной ранее цитаты (утверждающей, что макросы следует заменять соответствующими функциями на языке C), то она относится к архитектуре, и в некоторых контекстах ее можно игнорировать. Например, если улучшение производительности является ключевым требованием, то линейный набор инструкций может оказаться необходимым.

Еще одна область применения макросов — генерация кода. Макросы позволяют внедрять в проект языки DSL. Microsoft MFC, Qt, Linux Kernel и wxWidgets — лишь несколько примеров из тысяч библиотек, которые определяют собственные языки DSL на основе макросов. Это в основном касается проектов на C++, однако они используют эту возможность языка C с целью организовать свои API.

В заключение нужно отметить: макросы в C могут быть положительным фактором, при условии, что их влияние на преобразованный препроцессором код хорошо изучено. Если вы участвуете в командной разработке проекта, то всегда интересуйтесь тем, как ваши коллеги применяют макросы, и не забывайте сообщать о своих решениях на этот счет.

Условная компиляция

Условная компиляция — еще одна уникальная особенность C. Она позволяет препроцессору генерировать разный исходный код в зависимости от тех или иных обстоятельств. Несмотря на название, компилятор не выполняет никаких условных действий — просто код, который передает ему препроцессор, может зависеть от определенных условий. Они проверяются препроцессором во время подготовки кода. В условной компиляции могут участвовать разные директивы. Их список приведен ниже:

- `#ifdef`;
- `#ifndef`;
- `#else`;
- `#elif`;
- `#endif`.

Простейший сценарий их использования показан в примере 1.7 (листинг 1.14).

Листинг 1.14. Пример условной компиляции (ExtremeC_examples_chapter1_7.c)

```
#define CONDITION

int main(int argc, char** argv) {
#ifdef CONDITION
  int i = 0;
  i++;
#endif
  int j= 0;
  return 0;
}
```

Обрабатывая приведенный выше код, препроцессор обнаруживает определение макроса `CONDITION` и делает пометку о том, что он определен. Обратите внимание: для этого макроса не указано никакого значения, и это совершенно корректно. Затем препроцессор опускается ниже и находит инструкцию `#ifdef`. Поскольку макрос `CONDITION` уже определен, все строчки между `#ifdef` и `#endif` попадают в итоговый исходный код.

Преобразованный результат показан в листинге 1.15.

Листинг 1.15. Пример 1.7 после обработки препроцессором

```
int main(int argc, char** argv) {
  int i = 0;
  i++;

  int j= 0;
  return 0;
}
```

Если бы макрос не был определен, то у директив `#if-#endif` не было бы никакой замены. Следовательно, преобразованный код выглядел бы приблизительно так (листинг 1.16).

Листинг 1.16. Пример 1.7 после обработки препроцессором в случае, если макрос CONDITION не определен

```
int main(int argc, char** argv) {



  int j= 0;
  return 0;
}
```

Обратите внимание на пустые строчки в листингах 1.15 и 1.16, оставшиеся после того, как препроцессор заменил участок `#ifdef-#endif` его вычисленным значением.

Макросы можно определять, передавая команде компиляции параметры -D. Например, в предыдущем случае макрос CONDITION можно было бы определить так:

```
$ gcc -DCONDITION -E main.c
```

Благодаря этой замечательной возможности вы можете определять свои макросы за пределами исходных файлов. Она особенно полезна, когда один и тот же исходник необходимо скомпилировать для разных архитектур, таких как Linux или macOS, с разными библиотеками и определениями макросов по умолчанию.

Директива `#ifndef` часто используется для *предотвращения дублирования заголовков*. Она не позволяет препроцессору подключить один и тот же заголовок дважды. Можно с уверенностью сказать, что она содержится в начале почти всех заголовочных файлов в практически любом проекте на C или C++.

Пример предотвращения дублирования заголовков приведен в листинге 1.17. Представьте, что это содержимое заголовочного файла и оно может быть включено в единицу компиляции два раза. Обратите внимание: пример 1.8 — лишь отдельный заголовочный файл, не предназначенный для компиляции.

Листинг 1.17. Пример защиты от дублирования заголовков (ExtremeC_examples_chapter1_8.h)

```
#ifndef EXAMPLE_1_8_H
#define EXAMPLE_1_8_H

void say_hello();
int read_age();

#endif
```

Как видите, определения всех переменных и функций находятся между `#ifndef` и `#endif`, что защищает их от повторного включения. Ниже я объясню, как это работает.

Когда заголовок подключается в первый раз, макрос `EXAMPLE_1_8_H` все еще не определен, поэтому препроцессор заходит в блок `#ifndef-#endif`. Следующая инструкция определяет макрос `EXAMPLE_1_8_H`, и препроцессор копирует в преобразованный код все, что находится выше директивы `#endif`. Когда происходит второе включение, макрос `EXAMPLE_1_8_H` уже определен, поэтому препроцессор пропускает все

содержимое блока `#ifndef-#endif` и переходит к следующей за `#endif` инструкции, если таковая имеется.

Между между `#ifndef` и `#endif` обычно размещают все содержимое заголовочного файла, а снаружи остаются лишь комментарии.

В заключение следует сказать, что для защиты от *двойного включения* вместо пары `#ifndef-#endif` можно использовать одну директиву `#pragma once`. Ее единственной особенностью является то, что, несмотря на поддержку в почти всех препроцессорах, она не входит в стандарт C. Поэтому если ваш код должен быть *переносимым*, то от нее лучше отказаться.

В листинге 1.18 показано применение `#pragma once` вместо директив `#ifndef-#endif`.

Листинг 1.18. Использование директивы #pragma once в примере 1.8

```
#pragma once

void say_hello();
int read_age();
```

На этом обсуждение директив препроцессора можно считать завершенным. Я продемонстрировал некоторые их интересные особенности и способы применения. Следующий раздел посвящен еще одной важной возможности языка C — указателям на переменные.

Указатели на переменные

Указатели на переменные (или просто указатели) — одна из самых фундаментальных концепций в языке C. В большинстве высокоуровневых языков программирования они недоступны напрямую. Вместо них применяются аналогичные механизмы, такие как *ссылки* в Java. Стоит отметить, что указатели уникальны в том смысле, что адреса, на которые они указывают, могут использоваться напрямую аппаратным обеспечением, чего нельзя сказать об их высокоуровневых аналогах.

Глубокое понимание указателей и того, как они работают, — неотъемлемая черта любого квалифицированного программиста на C. Это один из ключевых аспектов управления памятью, и, несмотря на свой простой синтаксис, указатели в случае некорректного использования могут привести к катастрофическим последствиям. Темы, связанные с управлением памятью, рассматриваются в главах 4 и 5, а здесь мы сосредоточимся на указателях. Если вы уверенно ориентируетесь в основных

терминах и концепциях, имеющих отношение к указателям, то можете пропустить этот раздел.

Синтаксис

В основе любого вида указателей лежит простая идея; это всего лишь обычная переменная, которая хранит *адрес памяти*. Первое, что приходит на ум при упоминании указателей, — это символ *, применяемый для их определения в C. Это показано в примере 1.9. В листинге 1.19 показано, как объявить и использовать указатель на переменную.

Листинг 1.19. Пример объявления и использования указателей в C (ExtremeC_examples_chapter1_9.c)

```
int main(int argc, char** argv) {
  int var = 100;
  int* ptr = 0;
  ptr = &var;
  *ptr = 200;
  return 0;
}
```

В этом примере есть все, что необходимо знать о синтаксисе указателей. В первой строчке объявляется и помещается на вершину *стека* переменная `var`. О сегменте стека мы поговорим в главе 4. Во второй строчке объявляется указатель `ptr`, начальное значение которого равно нулю. Указатели, имеющие значение 0, называют *нулевыми*. Пока указатель `ptr` равен нулю, он считается нулевым. Если во время объявления указателю *не* присваивается действительный адрес, то его необходимо *обнулить*.

Как видите, в листинге 1.19 не подключаются никакие заголовочные файлы. Указатели — часть языка C, и для их использования не нужно ничего подключать. На самом деле программы на C могут обходиться без каких-либо заголовочных файлов.

Все эти объявления будут корректными с точки зрения C:

```
int* ptr = 0;

int * ptr = 0;

int *ptr = 0;
```

В третьей строке в функции `main` находится оператор `&`, который называют *оператором указания* или *унарным оператором*. Он возвращает адрес переменной,

следующий за ним. Он нужен для получения адресов переменных, иначе мы не сможем правильно инициализировать наши указатели.

В той же строке возвращаемый адрес присваивается указателю `ptr`, который при этом перестает быть нулевым. В четвертой строке перед указателем находится еще один оператор, обозначенный символом `*`. Это *оператор разыменования*. Он открывает непрямой доступ к ячейке памяти, на которую указывает `ptr`. Иными словами, он позволяет считывать и модифицировать переменную `var` через указатель, который на нее ссылается. Эта строка эквивалентна выражению `var = 200;`.

Нулевой указатель не содержит действительного адреса памяти. Вы *должны* избегать его разыменования, поскольку оно приводит к *неопределенному поведению* (что обычно заканчивается сбоем программы).

Заканчивая разбор предыдущего примера, отмечу, что при написании кода нам обычно доступен макрос `NULL` со значением `0`, который можно использовать для обнуления указателей в момент их объявления (листинг 1.20). Этому макросу следует отдавать предпочтение, поскольку он помогает отличать обычные переменные от указателей.

Листинг 1.20. Обнуление указателя с помощью макроса NULL

```
char* ptr = NULL;
```

В C++ применяются точно такие же указатели, как и в C, и их тоже нужно обнулять, присваивая им `0` или `NULL`. Но в C++11 появилось новое ключевое слово для инициализации указателей. Это не макрос, как `NULL`, и не целое число, как `0`. Речь идет о ключевом слове `nullptr`, которое позволяет обнулять указатели и проверять, являются ли они нулевыми. Пример его использования в C++11 показан в листинге 1.21.

Листинг 1.21. Обнуление указателя с помощью nullptr в C++11

```
char* ptr = nullptr;
```

Никогда не забывайте, что указатели нужно инициализировать во время объявления. Если вы не хотите с самого начала хранить в них действительный адрес памяти, то делайте их нулевыми, присваивая им 0 или NULL. В противном случае может возникнуть фатальная ошибка!

Большинство современных компиляторов сами обнуляют любые неинициализированные указатели. Это значит, все указатели до инициализации равны `0`. Однако не думайте, будто это оправдывает объявление указателей без надлежащей инициализации. Помните, что вы пишете свой код для разных архитектур, старых

и новых, и в устаревших системах это может привести к проблемам. Кроме того, большинство *профилировщиков памяти*, обнаружив неинициализированные указатели, выводят предупреждения и сообщения об ошибках. О профилировщиках мы подробно поговорим в главах 4 и 5.

Арифметические операции с указателями на переменные

Память проще всего представить себе в виде очень длинного одномерного байтового массива. Если следовать этому сравнению, то перемещаться по памяти можно только вперед и назад; других направлений не предусмотрено. То же самое касается указателей, ссылающихся на разные байты в памяти. Инкрементируя и декрементируя указатель, вы перемещаетесь вперед и назад соответственно. Это единственные арифметические операции, которые он поддерживает.

Аналогия, что арифметические операции с указателями подобны перемещениям по байтовому массиву, позволяет ввести новое понятие: *длина арифметического шага*. Оно нужно в связи с тем, что инкрементация указателя на 1 может привести к перемещению в памяти вперед больше чем на 1 байт. Длина арифметического шага есть у каждого указателя; она определяет, на сколько байтов он будет перемещаться при добавлении или вычитании значения 1. Эта величина определяется *типом данных* указателя.

На каждой платформе все указатели, хранящиеся в памяти, должны занимать ячейки определенного размера, то есть содержать одинаковое количество байтов. Но это вовсе не означает, что все они имеют одну и ту же длину арифметического шага. Как уже упоминалось выше, данная величина зависит от типа данных указателя.

Например, указатели типа `int` и `char` имеют одинаковый размер, но длина их арифметического шага отличается: `int*` обычно имеет четырехбайтный шаг, а `char*` — однобайтный. Следовательно, при инкрементации целочисленного указателя мы перемещаемся в памяти на четыре байта вперед (добавляем четыре байта к текущему адресу), а инкрементация указателя типа `char` приводит к перемещению вперед на 1 байт. В примере 1.10 демонстрируется длина арифметического шага для двух указателей с двумя разными типами данных (листинг 1.22).

Листинг 1.22. Длина арифметического шага для двух указателей (ExtremeC_examples_chapter1_10.c)

```
#include <stdio.h>

int main(int argc, char** argv) {
  int var = 1;
```

```
  int* int_ptr = NULL; // обнуляем указатель
  int_ptr = &var;

  char* char_ptr = NULL;
  char_ptr = (char*)&var;

  printf("Before arithmetic: int_ptr: %u, char_ptr: %u\n",
         (unsigned int)int_ptr, (unsigned int)char_ptr);

  int_ptr++;      // арифметический шаг обычно равен 4 байтам
  char_ptr++;     // арифметический шаг равен 1 байту

  printf("After arithmetic: int_ptr: %u, char_ptr: %u\n",
         (unsigned int)int_ptr, (unsigned int)char_ptr);

  return 0;
}
```

В терминале 1.4 показан вывод программы из примера 1.10. Имейте в виду, что при каждом запуске на разных платформах или даже на одном компьютере могут выводиться разные адреса, поэтому ваш вывод, скорее всего, будет отличаться.

Терминал 1.4. Вывод программы из примера 1.10 после первого запуска

```
$ gcc ExtremeC_examples_chapter1_10.c
$ ./a.out
Before arithmetic: int_ptr: 3932338348, char_ptr: 3932338348
After arithmetic:  int_ptr: 3932338352, char_ptr: 3932338349
$
```

Если сравнить адреса до и после арифметических операций, то становится очевидно, что длина шага для целочисленного указателя равна 4 байтам, а для указателя типа `char` — одному. При следующем запуске указатели, вероятно, будут содержать какие-то другие адреса, но длина их арифметического шага останется прежней (терминал 1.5).

Терминал 1.5. Вывод программы из примера 1.10 после второго запуска

```
$ ./a.out
Before arithmetic: int_ptr: 4009638060, char_ptr: 4009638060
After arithmetic:  int_ptr: 4009638064, char_ptr: 4009638061
$
```

Теперь, обсудив длину арифметического шага, мы можем рассмотреть классический образец использования арифметики указателей: *перебор* адресов на участке памяти. Примеры 1.11 и 1.12 должны выводить все элементы целочисленного массива; в первом применен тривиальный подход без использования указателей, а во втором задействованы арифметические операции, рассмотренные выше.

В листинге 1.23 представлен код примера 1.11.

Листинг 1.23. Перебор элементов массива без арифметики указателей (ExtremeC_examples_chapter1_11.c)

```
#include <stdio.h>

#define SIZE 5

int main(int argc, char** argv) {
  int arr[SIZE];
  arr[0] = 9;
  arr[1] = 22;
  arr[2] = 30;
  arr[3] = 23;
  arr[4] = 18;

  for (int i = 0; i < SIZE; i++) {
    printf("%d\n", arr[i]);
  }

  return 0;
}
```

Код в листинге 1.23 должен быть вам знаком. Обращение к определенному индексу массива и чтения его содержимого здесь выполняется с помощью *счетчика цикла*. Но если вместо индексов (целых чисел между [и]) вы хотите использовать указатели, то вам понадобится другой подход. В листинге 1.24 показано, как указатели позволяют перебирать элементы в пределах массива.

Листинг 1.24. Перебор элементов массива с помощью арифметики указателей (ExtremeC_examples_chapter1_12.c)

```
#include <stdio.h>

#define SIZE 5

int main(int argc, char** argv) {
  int arr[SIZE];
  arr[0] = 9;
  arr[1] = 22;
  arr[2] = 30;
  arr[3] = 23;
  arr[4] = 18;

  int* ptr = &arr[0];

  for (;;) {
    printf("%d\n", *ptr);
    if (ptr == &arr[SIZE - 1]) {
```

```
      break;
    }
    ptr++;
  }

  return 0;
}
```

Второй способ, продемонстрированный в листинге 1.24, основан на бесконечном цикле, который прерывается, когда адрес указателя `ptr` равен адресу последнего элемента массива. Мы знаем, что массив — это набор смежных переменных, размещенных в памяти. И потому, инкрементируя и декрементируя указатель, мы можем перемещаться по массиву вперед и назад и таким образом обращаться к разным элементам.

Листинг 1.24 наглядно демонстрирует: указатель `ptr` имеет тип данных `int*`. Это вызвано тем фактом, что он должен иметь возможность указывать на каждый отдельный элемент массива. Обратите внимание: все элементы массива имеют один и тот же тип, `int`, вследствие чего их размеры должны совпадать. Следовательно, инкрементация указателя `ptr` смещает его к следующему элементу. Как видите, перед началом цикла `for` указателю `ptr` был присвоен адрес первого элемента, и в каждой итерации он перемещается вперед по участку памяти, который принадлежит массиву. Это самый что ни на есть классический пример использования арифметики указателей.

Следует отметить, что в C массив на самом деле является указателем на свой первый элемент. Поэтому в нашем примере переменная `arr` имеет тип данных `int*`. Следовательно, вместо:

```
int* ptr = &arr[0];
```

можно было бы написать:

```
int* ptr = arr;
```

Обобщенные указатели

Указатели типа `void*` называют *обобщенными*. Как и любые другие указатели, они могут ссылаться на любой адрес, но их фактический тип данных неизвестен, поэтому мы не знаем длину их арифметического шага. Обобщенные указатели обычно используются для хранения содержимого других указателей без запоминания их типов. В связи с этим их нельзя разыменовывать и с ними невозможно проводить арифметические операции, поскольку мы не знаем их тип данных. В примере 1.13 показана неудачная попытка разыменования обобщенного указателя (листинг 1.25).

Листинг 1.25. Разыменование обобщенного указателя вызывает ошибку компиляции (ExtremeC_examples_chapter1_13.c)

```
#include <stdio.h>

int main(int argc, char** argv) {
  int var = 9;
  int* ptr = &var;
  void* gptr = ptr;
  printf("%d\n", *gptr);

  return 0;
}
```

Попробовав скомпилировать приведенный выше код с помощью `gcc` в Linux, вы получите следующее сообщение об ошибке (терминал 1.6).

Терминал 1.6. Компиляция примера 1.13 в Linux

```
$ gcc ExtremeC_examples_chapter1_13.c
In function 'main':
warning: dereferencing 'void *' pointer
  printf("%d\n", *gptr);
                 ^~~~
error: invalid use of void expression
  printf("%d\n", *gptr);
$
```

И если попытаться скомпилировать этот код с помощью `clang` в macOS, то получится другое сообщение об ошибке, которое сигнализирует о той же проблеме (терминал 1.7).

Терминал 1.7. Компиляция примера 1.13 в macOS

```
$ clang ExtremeC_examples_chapter1_13.c
error: argument type 'void' is incomplete
  printf("%d\n", *gptr);
                 ^
1 error generated.
$
```

Как видите, оба компилятора запрещают разыменование обобщенных указателей. На самом деле эта операция не имеет смысла! Для чего же могут понадобиться такие указатели? Что ж, они отлично подходят для определения *обобщенных функций*, которые могут принимать в качестве аргументов широкий спектр указателей. Это подробно продемонстрировано в примере 1.14 (листинг 1.26).

Листинг 1.26. Пример обобщенной функции (ExtremeC_examples_chapter1_14.c)

```
#include <stdio.h>

void print_bytes(void* data, size_t length) {
  char delim = ' ';
  unsigned char* ptr = data;

  for (size_t i = 0; i < length; i++) {
    printf("%c 0x%x", delim, *ptr);
    delim = ',';
    ptr++;
  }
  printf("\n");
}

int main(int argc, char** argv) {
  int a = 9;
  double b = 18.9;

  print_bytes(&a, sizeof(int));
  print_bytes(&b, sizeof(double));

  return 0;
}
```

В приведенном выше листинге функция `print_bytes` принимает адрес в виде указателя `void*` и целое число, обозначающее длину. С помощью этих аргументов она выводит все байты, начиная с заданного адреса и вплоть до заданной длины. Функция принимает обобщенный указатель, благодаря чему пользователь может передать все, что ему вздумается. Имейте в виду: присваивание значений обобщенному *(пустому)* указателю *не* требует явного приведения типов. Именно поэтому я не указывал типы при передаче адресов `a` и `b`.

Внутри функции `print_bytes` используется указатель типа `unsigned char`, позволяющий перемещаться по памяти. Никаких других прямых арифметических операций с параметром `data` совершать нельзя. Как вы уже, наверное, знаете, длина шагов `char*` и `unsigned char*` равна 1 байту. Это делает данный тип указателей наиболее подходящим для побайтового перебора адресов в заданном диапазоне и их обработки байт за байтом.

В заключение стоит отметить, что `size_t` — это стандартный беззнаковый тип данных, который в языке C обычно используется для хранения размеров.

Функция size_t описана в подразделе 6.5.3.4 стандарта ISO/ICE 9899:TC3. Это знаменитая спецификация C99, пересмотренная в 2007 году. На сегодняшний день она является основой для всех реализаций C. Текст ISO/ICE 9899:TC3 (2007) находится по ссылке www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf.

Размер указателей

Если поискать в Google фразу «размер указателей в C», то обнаружится отсутствие однозначной информации по этому поводу. Вы найдете множество вариантов, но правда в том, что размер указателя не является понятием языка программирования как такового, а зависит от конкретной архитектуры. Язык C мало знает о подобных подробностях, относящихся к аппаратному обеспечению; он пытается предоставить универсальный способ работы с указателями и другими аспектами программирования. Именно поэтому язык C является стандартом; в нем описываются только сами указатели и их арифметика.

Под архитектурой имеется в виду оборудование, которое используется в компьютерной системе. Подробнее об этом — в главе 2.

Функцию `sizeof` всегда можно использовать для получения размера указателя. Достаточно проверить результат `sizeof(char*)` в вашей текущей архитектуре. В 32- и 64-битных архитектурах указатели, как правило, занимают соответственно 4 и 8 байт, однако в некоторых системах могут иметь другой размер. Помните: код, который вы пишете, *не должен* зависеть от конкретного размера указателя и не должен делать об этом никаких предположений. В противном случае у вас возникнут проблемы при его переносе на другие архитектуры.

Висячие указатели

Неправильное применение указателей вызывает множество хорошо известных проблем. Один из самых скверных примеров — висячие указатели. Указатель обычно содержит адрес участка памяти, выделенного для переменной. Чтение или изменение адреса, по которому не зарегистрировано никакой переменной, считается большой оплошностью и может привести к сбою программы или *ошибке сегментации*. Последнее — крайне неприятная проблема, с которой, наверное, сталкивался любой разработчик на C/C++. Она обычно возникает при неправильном использовании указателей. Вы обращаетесь к участку памяти, к которому у вас нет доступа. Раньше там находилась ваша переменная, но к моменту обращения она уже не существует.

Попробуем воспроизвести эту ситуацию в примере 1.15 (листинг 1.27).

Листинг 1.27. Попытка спровоцировать ошибку сегментации (ExtremeC_examples_chapter1_15.c)

```
#include <stdio.h>

int* create_an_integer(int default_value) {
  int var = default_value;
```

```
  return &var;
}

int main() {
  int* ptr = NULL;
  ptr = create_an_integer(10);
  printf("%d\n", *ptr);
  return 0;
}
```

В этом примере используется функция `create_an_integer`, которая создает целое число. Она объявляет переменную типа `int` со значением по умолчанию и возвращает ее адрес вызывающему коду. Функция `main` получает адрес созданного числа, `var`, и сохраняет его в указатель `ptr`. Затем указатель разыменовывается, а значение, хранящееся в `var`, выводится на экран.

Но не все так просто. Если скомпилировать этот код с помощью `gcc` на компьютере под управлением Linux, то он сгенерирует предупреждение, хотя сама компиляция завершится успешно и вы получите готовый исполняемый файл (терминал 1.8).

Терминал 1.8. Компиляция примера 1.15 в Linux

```
$ gcc ExtremeC_examples_chapter1_15.c
In function 'f':
warning: function returns address of local variable [-Wreturn-local-addr]
    return &var;
           ^~~
$
```

Это важное предупреждение, которое программист может легко упустить. Мы еще обсудим данную ситуацию позже, в главе 5. Посмотрим, что произойдет, если запустить полученный исполняемый файл.

При выполнении примера 1.15 возникает ошибка сегментации, что приводит к немедленному аварийному завершению программы (терминал 1.9).

Терминал 1.9. Ошибка сегментации при выполнении примера 1.15

```
$ ./a.out
Segmentation fault (core dumped)
$
```

Так в чем же проблема? Указатель `ptr` — висячий, поскольку участок памяти, на который он ссылается (принадлежавший переменной `var`), уже освобожден.

Переменная `var` локальная и освобождается сразу после завершения функции, в которой была объявлена, `create_an_integer`. Но вместе с тем эта функция возвращает

ее адрес, который затем присваивается указателю `ptr` внутри `main`. В результате `ptr` становится висячим указателем, ссылаясь на недействительный участок памяти, а попытка его разыменования приводит к серьезной проблеме и аварийному завершению программы.

Если вернуться чуть назад, то можно увидеть, что компилятор явно предупреждает нас об этой проблеме.

Он говорит о *возвращении адреса локальной переменной*, которая освобождается после выхода из функции. Смышленый компилятор! Относитесь серьезно к его предупреждениям, и вам не придется иметь дело с жуткими ошибками.

Но как бы мы переделали этот пример? Конечно, используя *кучу*. Эту разновидность памяти мы подробно рассмотрим в главах 4 и 5, а пока исправим наш код с помощью *выделения кучи* и продемонстрируем преимущества данного подхода по сравнению с применением *стека*.

В примере 1.16 показано, как выделяются переменные в куче (листинг 1.28). Это позволяет передавать адреса между функциями без каких-либо проблем.

Листинг 1.28. Исправленная версия примера 1.15 с помощью кучи (ExtremeC_examples_chapter1_16.c)

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

int* create_an_integer(int default_value) {
  int* var_ptr = (int*)malloc(sizeof(int));
  *var_ptr = default_value;
  return var_ptr;
}

int main() {
  int* ptr = NULL;
  ptr = create_an_integer(10);
  printf("%d\n", *ptr);
  free(ptr);
  return 0;
}
```

Как видите, мы подключили новый заголовочный файл, `stdlib.h`, и задействовали две новые функции: `malloc` и `free`. Простое объяснение звучит так: переменная, которая создается внутри функции `create_an_integer`, больше не является локальной. Она выделяется в куче, и ее время жизни не ограничено функцией, в которой была объявлена. Следовательно, к ней может обратиться вызывающая (внешняя) функция. Указатели, ссылающиеся на эту переменную, больше не висячие, и их можно разыменовывать (при условии, что переменная не была освобождена и все

еще существует). В конце своего жизненного цикла переменная освобождается путем вызова функции `free`. Обратите внимание: любое содержимое кучи, которое больше не требуется, нужно обязательно освобождать.

В этом разделе мы обсудили все основные аспекты указателей на переменные. Далее мы поговорим о функциях и их принципе работы в C.

Общая информация о функциях

C — *процедурный* язык программирования. Функции в нем ведут себя подобно процедурам, играя роль основных элементов, из которых состоит программа. Поэтому необходимо разобраться в том, что они собой представляют и что происходит, когда вы входите в функцию и выходите из нее. В целом функции (или процедуры) можно считать обычными переменными, которые вместо значений хранят алгоритмы. Объединяя переменные и функции в новые типы, мы получаем возможность хранить значения и связанные с ними алгоритмы в виде одной сущности. Данный подход применяется в *объектно-ориентированном программировании*, и мы рассмотрим его в третьей части нашей книги. В этом же разделе мы хотим исследовать функции и обсудить их характеристики в языке C.

Анатомия функции

В этом разделе мы собрали воедино все, что касается функций в C. Если материал кажется вам знакомым, то можете перелистнуть страницы.

Функция — это логический блок с именем и списками принимаемых параметров и возвращаемых результатов. В C, как и во многих других языках программирования, на которые он повлиял, функции возвращают лишь одно значение. В объектно-ориентированных языках, таких как C++ и Java, функции обычно называются *методами* и, в отличие от C, могут генерировать *исключения*. Выполнение логики функции с помощью ее имени называется вызовом. Корректный вызов должен передать функции все аргументы, которые ей необходимы, и ждать, пока она не завершится. Обратите внимание: в C все функции *блокирующие*. Это значит, вызывающий код ждет, пока они не закончат работу, и только потом собирает возвращенные результаты.

Помимо блокирующих функций, существуют и *неблокирующие*. Вызывающий код может продолжать выполняться, не дожидаясь их завершения. В таком случае обычно используется механизм *обратных вызовов*, которые срабатывают, когда вызванная функция завершает работу. Неблокирующие функции также иногда называют *асинхронными*. Поскольку в C они не существуют, для их реализации применяются многопоточные решения. Мы обсудим эти концепции более подробно в пятой части книги.

Интересно, что в наши дни интерес к неблокирующим функциям продолжает расти. Существует целая область под названием *«событийно-ориентированное программирование»*, и неблокирующие функции являются ее ключевым элементом, превосходя по частоте использования блокирующие аналоги.

В событийно-ориентированном программировании вызовы самих функций происходят внутри *цикла событий*, а обратные вызовы срабатывают при возникновении некоего события. Такой подход к программированию популяризуют библиотеки `libuv` и `libev`, которые позволяют проектировать ПО вокруг одного или нескольких событийных циклов.

Роль функций в архитектуре приложений

Функции — фундаментальные составные элементы процедурной разработки. Благодаря повсеместной поддержке в языках программирования они оказывают огромное влияние на то, как мы пишем свой код. Функции позволяют хранить логику в сущностях наподобие переменных и вызывать ее там и тогда, где и когда это нужно. Таким образом, мы можем написать один блок кода и многократно его использовать в разных местах.

Помимо прочего, функции позволяют изолировать друг от друга разные фрагменты логики. Иными словами, они вводят слой абстракции между различными логическими компонентами. Представьте, что у вас есть функция `avg`, которая вычисляет среднее значение для элементов входящего массива и вызывается из функции `main`. Считается, что логика внутри `avg` изолирована от логики внутри `main`.

Следовательно, если вы хотите поменять логику `avg`, то вам не нужно редактировать функцию `main`. Дело в том, что она полагается только на имя и доступность `avg`. Это было большим достижением, по крайней мере по тем временам, когда для написания и выполнения программ приходилось использовать перфокарты!

Этот подход до сих пор применяется в проектировании библиотек на C и даже в языках программирования более высокого уровня, таких как C++ и Java.

Управление стеком

Если взглянуть на участки памяти процессов, выполняющихся в Unix-подобной операционной системе, то можно заметить, что все они имеют похожую структуру. Мы подробно обсудим ее в главе 4, а пока познакомимся с одним из ее *сегментов* — стеком. Это участок памяти, который по умолчанию выделяется для всех локальных переменных, массивов и структур. Поэтому, когда вы объявляете локальную переменную в функции, она всегда создается в стеке — а если точнее, на его вершине.

Обратите внимание на название *«стек»* (stack — «стопка»). Этот сегмент ведет себя как набор элементов, которые складываются один поверх другого. Переменные и массивы всегда создаются на его вершине, а та переменная, которая находится в самом верху, удаляется первой. Мы еще вернемся к данной аналогии чуть ниже.

Стек используется еще и для вызова функций. Перед началом выполнения новой функции ее адрес возврата и все передаваемые ей аргументы собираются в *стековый фрейм* и кладутся на вершину стека. Когда эта функция завершается, фрейм извлекается из стека, после чего начинают выполняться инструкции, находящиеся по адресу возврата (это обычно приводит к продолжению работы вызывающей функции).

Все локальные переменные, объявленные в теле функции, помещаются на вершину стека. Как следствие, все они освобождаются по завершении функции. Именно поэтому мы называем их *локальными переменными* и можем быть уверены в том, что одна функция не имеет доступа к переменным другой. Этот механизм также объясняет, почему переменные не объявляются до входа в функцию и после выхода из нее.

Иметь понимание стека и того, как он работает, совершенно необходимо для написания корректного и осмысленного кода. Это также предотвращает распространенные ошибки, связанные с памятью. Стоит отметить, что в стеке нельзя создавать переменные любых размеров. Это ограниченная область памяти, которую можно заполнить до конца и получить ошибку *переполнения стека*. Обычно это происходит при вызове слишком большого количества функций, заполняющих все пространство стека стековыми фреймами. Подобная ситуация часто наблюдается во время работы с рекурсивными функциями, которые вызывают сами себя без каких-либо условий выхода или лимитов.

Передача по значению и передача по ссылке

В большинстве книг по программированию есть раздел, посвященный передаче аргументов в функцию: по значению или по ссылке. К счастью или к сожалению, в языке C доступен только первый вариант.

В C нет никаких ссылок, что делает передачу по ссылке невозможной. Все, что передается функции, копируется в ее локальные переменные и перестает быть доступным, когда она завершается.

Несмотря на многие примеры вызовов с передачей аргументов по ссылке, спешу вас уверить: в C это не более чем иллюзия. В оставшейся части данного подраздела я попытаюсь ее развеять и убедить вас в том, что во всех этих примерах аргументы передаются по значению. Это показано в листинге 1.29.

Листинг 1.29. Пример вызова функции с передачей по значению (ExtremeC_examples_chapter1_17.c)

```
#include <stdio.h>

void func(int a) {
  a = 5;
}

int main(int argc, char** argv) {
  int x = 3;
  printf("Before function call: %d\n", x);
  func(x);
  printf("After function call: %d\n", x);
  return 0;
}
```

Несложно предвидеть вывод этой программы. Переменная `x` не поменяется, поскольку передается по значению. Терминал 1.10 демонстрирует вывод примера 1.17 и подтверждает то, о чем мы и так догадывались.

Терминал 1.10. Вывод примера 1.17

```
$ gcc ExtremeC_examples_chapter1_17.c
$ ./a.out
Before function call: 3
After function call: 3
$
```

Как показывает пример 1.18 (листинг 1.30), в языке C не существует передачи по ссылке.

Листинг 1.30. Пример того, что передача по указателю отличается от передачи по ссылке (ExtremeC_examples_chapter1_18.c)

```
#include <stdio.h>

void func(int* a) {
  int b = 9;
  *a = 5;
  a = &b;
}

int main(int argc, char** argv) {
  int x = 3;
  int* xptr = &x;
  printf("Value before call: %d\n", x);
  printf("Pointer before function call: %p\n", (void*)xptr);
  func(xptr);
```

```
  printf("Value after call: %d\n", x);
  printf("Pointer after function call: %p\n", (void*)xptr);
  return 0;
}
```

Мы получим следующий вывод (терминал 1.11).

Терминал 1.11. Вывод примера 1.18

```
$ gcc ExtremeC_examples_chapter1_18.c
$ ./a.out
The value before the call: 3
Pointer before function call: 0x7ffee99a88ec
The value after the call: 5
Pointer after function call: 0x7ffee99a88ec
$
```

Как видите, содержимое указателя не поменялось после вызова функции. Это значит, указатель передается по значению. Разыменование данного указателя внутри функции `func` позволяет обратиться к переменной, на которую он ссылается. Но вывод показывает, что изменение указателя внутри функции не меняет соответствующий аргумент в вызывающем коде. В языке C во время вызова все аргументы передаются по значению, а разыменовывание указателей позволяет менять переменные вызывающей функции.

Кроме того, стоит отметить, что в приведенном выше примере мы передаем не сами переменные, а их указатели. Обычно в качестве аргументов рекомендуется использовать указатели, а не крупные объекты, и несложно догадаться почему. Копирование восьмибайтного указателя куда более эффективно, чем копирование большого объекта, занимающего сотни байтов.

Удивительно, но в нашем примере эта эффективность себя не проявила! Причиной тому факт, что тип `int` занимает всего 4 байта, поэтому копируется быстрее, чем его восьмибайтный указатель. Но со структурами и массивами все иначе, ведь они копируются последовательно, байт за байтом, поэтому их лучше передавать по указателю.

Итак, мы рассмотрели некоторые аспекты функций в C. Теперь поговорим об их указателях.

Указатели на функции

Указатели на функции — еще одна супервозможность языка программирования C. Предыдущие два раздела были посвящены функциям и указателям на переменные. Здесь мы объединим обе темы и обсудим более интересную концепцию: указатели на функции.

Эти указатели можно использовать множеством способов, но один из самых важных заключается в разбиении одной большой программы на несколько мелких частей с последующей их загрузкой в отдельный компактный исполняемый файл. Эта методика лежит в основе *модуляризации* и проектирования ПО в целом. Указатели на функции используются в реализации полиморфизма в C++, позволяя расширять существующую логику. В данном разделе мы рассмотрим принцип их работы и подготовим вас к более сложным темам, которые будут представлены в следующих главах.

Указатель на переменную содержит ее адрес. Точно так же указатель на функцию содержит ее адрес, позволяя вызывать ее опосредованным образом. Начнем обсуждение этой темы с примера 1.19 (листинг 1.31).

Листинг 1.31. Вызов разных функций с помощью одного и того же указателя (ExtremeC_examples_chapter1_19.c)

```
#include <stdio.h>

int sum(int a, int b) {
  return a + b;
}

int subtract(int a, int b) {
  return a - b;
}

int main() {
  int (*func_ptr)(int, int);
  func_ptr = NULL;

  func_ptr = &sum;
  int result = func_ptr(5, 4);
  printf("Sum: %d\n", result);

  func_ptr = &subtract;
  result = func_ptr(5, 4);
  printf("Subtract: %d\n", result);

  return 0;
}
```

В данном листинге `func_ptr` — указатель, который может ссылаться только на определенный вид функций, соответствующих его сигнатуре. Это значит, функции, адрес которых он содержит, должны принимать два целочисленных аргумента и возвращать целочисленный результат.

Как видите, мы определили две функции, `sum` и `subtract`, которые отвечают сигнатуре указателя `func_ptr`. Мы используем `func_ptr`, чтобы вызвать каждую из этих

функций по отдельности с одними и теми же аргументами, и затем сравниваем результаты. В терминале 1.12 показан вывод данного примера.

Терминал 1.12. Вывод примера 1.19

```
$ gcc ExtremeC_examples_chapter1_19.c
$ ./a.out
Sum: 9
Subtract: 1
$
```

Как видно в примере 1.19, мы можем вызывать разные функции с одинаковыми аргументами, используя один указатель, и это важно. Если вы знакомы с объектно-ориентированным программированием, то на ум сразу приходят *виртуальные функции* и *полиморфизм*. На самом деле это единственный способ реализации полиморфизма в C с имитацией виртуальных функций C++. Мы поговорим об ООП в третьей части книги.

Указатели на функции, как и любые другие, необходимо как следует инициализировать. Если указатель не инициализируется непосредственно во время объявления, то его обязательно необходимо обнулить. Обнуление указателей на функции продемонстрировано в следующем примере. Это довольно похоже на то, как мы обнуляли указатели на перемены.

Для указателей на функции обычно рекомендуется определить новый *псевдоним типа*. В примере 1.20 показано, как это делается (листинг 1.32).

Листинг 1.32. Вызов разных функций с помощью одного указателя (ExtremeC_examples_chapter1_20.c)

```
#include <stdio.h>

typedef int bool_t;
typedef bool_t (*less_than_func_t)(int, int);

bool_t less_than(int a, int b) {
  return a < b ? 1 : 0;
}

bool_t less_than_modular(int a, int b) {
  return (a % 5) < (b % 5) ? 1 : 0;
}

int main(int argc, char** argv) {
  less_than_func_t func_ptr = NULL;

  func_ptr = &less_than;
  bool_t result = func_ptr(3, 7);
  printf("%d\n", result);
```

```
  func_ptr = &less_than_modular;
  result = func_ptr(3, 7);
  printf("%d\n", result);

  return 0;
}
```

Ключевое слово `typedef` позволяет определить псевдоним для уже существующего типа. В приведенном выше примере используются два таких псевдонима: `bool_t` для типа `int` и `less_than_func_t` для указателей на функции типа `bool_t (*)(int, int)`. Благодаря этому код более понятен и можно выбрать более короткое имя для длинного и сложного типа. В C к именам новых типов принято добавлять `_t`; это соглашение об именовании соблюдается во многих других стандартных псевдонимах, таких как `size_t` и `time_t`.

Структуры

Структуры — одна из основополагающих концепций проектирования в C. В наши дни их аналоги можно встретить в практически любом современном языке программирования.

Структуры следует рассматривать в контексте истории развития информатики, начиная с тех далеких дней, когда C был единственным языком, который их поддерживал. Появление структур было огромным шагом навстречу *инкапсуляции* и одной из многочисленных мер по отказу от прямого использования машинных инструкций. Принципы, на которых основано человеческое мышление, не претерпели существенных изменений за последние тысячелетия; одним из них попрежнему является инкапсуляция.

Но лишь с появлением C мы получили инструмент (в данном случае язык программирования), который соответствовал нашей картине мира и мог хранить и обрабатывать понятные нам логические элементы. Наконец-то у нас появился язык, соответствующий нашим мыслям и идеям, и все это благодаря структурам. Структуры в C были далеки от идеала, если сравнивать их с механизмами инкапсуляции современных языков, но их оказалось достаточно для построения платформы, на основе которой впоследствии были созданы наши самые лучшие инструменты.

Зачем нужны структуры

Вы, наверное, знаете, что у каждого языка программирования есть набор *примитивных типов данных*. С их помощью можно создавать собственные структуры данных и писать на их основе свои алгоритмы. Эти типы — часть языка, и их нельзя изменить или удалить. Например, из C невозможно убрать такие примитивные типы, как `int` и `double`.

Но если вы хотите определить собственные, *пользовательские типы данных*, не предусмотренные в самом языке, то вам помогут структуры. Это типы, которые создает пользователь, и они не входят в язык.

Обратите внимание: ключевое слово `typedef` не имеет никакого отношения к пользовательским типам. Оно лишь создает псевдоним для типов, которые уже существуют. Если вашей программе требуются совершенно новые типы, то вы должны задействовать структуры.

У структур есть аналоги в других языках программирования; например, в C++ и Java это классы, а в Perl — пакеты. Там они называются *создателями типов*.

Зачем нужны пользовательские типы

Так зачем же создавать в программе новые типы? Ответ на этот вопрос раскрывает принципы, стоящие за проектированием ПО, и методы, которые мы используем в повседневной разработке. Сами того не замечая, мы создаем новые типы у себя в уме, когда анализируем что-либо.

Мы не воспринимаем окружающий нас мир в виде чисел и символов. Наш мозг научился группировать связанные между собой атрибуты в отдельные объекты. Более подробно об этом мы поговорим в главе 6. Но, отвечая на первоначальный вопрос, новые типы нужны для анализа задач на более высоком логическом уровне, приближенном к человеческому мышлению.

Здесь необходимо упомянуть о понятии *«бизнес-логика»*. Это набор всех сущностей и нормативно-правовых норм, которые встречаются в той или иной области. Например, бизнес-логика банковской системы состоит из таких концепций, как «клиент», «счет», «баланс», «деньги», «наличные», «платеж» и др. Все вместе они делают возможными и осмысленными, скажем, операции по снятию денег со счета.

Представьте, что вам нужно объяснить логику какой-нибудь банковской операции с помощью одних лишь чисел и символов. Это почти невозможно. Но даже если программист умудрится это сделать, полученный результат будет практически бессмысленным с точки зрения *бизнес-аналитика*. В реальных средах разработки с четко определенной бизнес-логикой программисты и бизнес-аналитики работают рука об руку. Поэтому им нужны общие терминология, глоссарий, типы, операции, нормы, логика и т. д.

В наши дни язык программирования, который не позволяет расширять свою *систему типов*, можно считать мертвым. Наверное, поэтому многие считают язык C таковым, ведь создание новых типов в нем, в отличие от более высокоуровневых аналогов, таких как C++ или Java, требует определенных усилий. Действительно, создать хорошую систему типов в C не так-то просто, но все, что для этого требуется, в нем уже есть.

Даже сегодня существует множество причин, чтобы выбрать C в качестве главного языка в своем проекте и смириться с тем, что для создания и поддержки хорошей системы типов потребуются значительные усилия. Многие компании так и делают.

Несмотря на то что новые типы постоянно требуются в ходе анализа ПО, центральные процессоры их не понимают. Аппаратное обеспечение пытается ограничиваться примитивными типами и быстрыми вычислениями, поскольку создано именно для этого. Как следствие, если ваша программа написана на высокоуровневом языке, то должна быть преобразована в набор процессорных инструкций, что может потребовать дополнительного времени и ресурсов.

К счастью, язык C в этом смысле не далеко ушел от логики уровня процессора, и его систему типов можно легко преобразовать. Вы, наверное, слышали, что C — низкоуровневый язык программирования, близкий к машинным кодам. Это одна из причин, почему некоторые компании до сих пор стремятся писать и поддерживать свои основные фреймворки на C.

Принцип работы структур

Структуры инкапсулируют связанные между собой значения в одном общем типе. В качестве простейшего примера мы можем сгруппировать переменные `red`, `green` и `blue` в новый отдельный тип данных под названием `color_t`. Этот новый тип может использоваться для представления цвета в формате RGB во многих программах, таких как графические редакторы. Соответствующую структуру в C можно определить следующим образом (листинг 1.33).

Листинг 1.33. Структура в C, представляющая цвет в формате RGB

```
struct color_t {
  int red;
  int green;
  int blue;
};
```

Как уже упоминалось, структуры выполняют инкапсуляцию — один из основополагающих принципов проектирования ПО. Он заключается в группировании и изоляции связанных между собой полей в рамках нового типа. Затем этот тип можно использовать для определения нужных нам переменных. Мы рассмотрим данную тему во всех подробностях в ходе обсуждения объектно-ориентированного проектирования в главе 6.

Обратите внимание: в именах новых типов данных используется суффикс _t.

Размещение структур в памяти

Обычно программистам на C необходимо знать, как именно выглядит структурная переменная в памяти. В некоторых архитектурах неэффективное размещение структур в памяти может привести к ухудшению производительности. Не забывайте, что наш код превращается в процессорные инструкции. Значения хранятся в памяти, и процессор должен иметь возможность их быстро считывать и записывать. Если программист знает, как устроена память его программы, то ему будет легче понять, как работает центральный процессор, и адаптировать свой код, чтобы получить лучшие результаты.

В примере 1.21 мы создаем новый структурный тип, `sample_t`, и объявляем одну структурную переменную, `var`. Затем заполняем поля этой переменной некоторыми значениями и выводим ее размер и байты, которые она хранит. Таким образом, мы сможем продемонстрировать то, как она представлена в памяти (листинг 1.34).

Листинг 1.34. Вывод количества байтов, выделенных для структурной переменной (ExtremeC_examples_chapter1_21.c)

```
#include <stdio.h>

struct sample_t {
  char first;
  char second;
  char third;
  short fourth;
};

void print_size(struct sample_t* var) {
  printf("Size: %lu bytes\n", sizeof(*var));
}

void print_bytes(struct sample_t* var) {
  unsigned char* ptr = (unsigned char*)var;
  for (int i = 0; i < sizeof(*var); i++, ptr++) {
    printf("%d ", (unsigned int)*ptr);
  }
  printf("\n");
}

int main(int argc, char** argv) {
  struct sample_t var;
  var.first = 'A';
  var.second = 'B';
  var.third = 'C';
  var.fourth = 765;
  print_size(&var);
  print_bytes(&var);
  return 0;
}
```

Стремление понять, как тот или иной компонент размещен в памяти, в основном присуще программистам на C/C++ и редко встречается среди тех, кто пишет на более высокоуровневых языках программирования. Например, в Java и Python программисты обычно не так хорошо понимают всю подноготную управления памятью; с другой стороны, указанные языки не предоставляют особо подробных сведений на этот счет.

Как можно видеть в листинге 1.34, перед объявлением структурной переменной в C необходимо указывать ключевое слово `struct`. В приведенном выше примере это продемонстрировано с помощью выражения `struct sample_t var`, в котором `struct` находится перед структурным типом. Вы, наверное, уже знаете, что для доступа к полям структурной переменной используется символ `.` (точка) или `->` (стрелка), если речь идет о структурном указателе.

Если вы не хотите набирать `struct` при определении каждого нового структурного типа и объявлении каждой структурной переменной, то можете создать для своих структур псевдонимы. Используйте для этого `typedef`, как показано ниже:

```
typedef struct {
  char first;
  char second;
  char third;
  short fourth;
} sample_t;
```

Теперь вы можете объявить переменную, не прибегая к использованию ключевого слова `struct`:

```
sample_t var;
```

В терминале 1.13 показан вывод примера 1.21 после его компиляции и запуска на компьютере под управлением macOS. Обратите внимание: генерируемые числа могут варьироваться в зависимости от системы.

Терминал 1.13. Вывод примера 1.21

```
$ clang ExtremeC_examples_chapter1_21.c
$ ./a.out
Size: 6 bytes
65 66 67 0 253 2
$
```

Как видите, выражение `sizeof(sample_t)` вернуло 6 байт. По своему размещению в памяти структурная переменная очень похожа на массив. В нем все элементы размещаются в памяти последовательно; то же самое относится к структурной переменной и ее полям. Разница лишь в том, что в массиве все элементы имеют один и тот же тип и, следовательно, одинаковый размер, чего нельзя сказать о структурных переменных. Каждое поле может иметь свой тип и размер. Мы можем легко

вычислить, какой объем памяти занимает массив, но размер структурной переменной зависит от нескольких факторов и не является столь очевидным.

На первый взгляд в определении размера структурной переменной нет ничего сложного. В нашем предыдущем примере структура состоит из четырех полей: трех — типа `char` и одного — типа `short`. Если предположить, что `sizeof(char)` равно 1 байту, а `sizeof(short)` — двум, то после простого подсчета можно утверждать, что тип `sample_t` должен занимать в памяти 5 байт. Но в нашем выводе видно: `sizeof(sample_t)` равно 6 байтам. На один больше! Откуда взялась эта разница?

Опять же, если взглянуть на содержимое структурной переменной `var` в памяти, можно увидеть, что оно выглядит не так, как можно было бы ожидать (а именно `65 66 67 253 2`).

Чтобы прояснить ситуацию и объяснить, почему размер структурной переменной не равен 5 байтам, следует ввести понятие *выравнивания данных в памяти*. Прежде чем совершать какие-либо вычисления, процессор должен сначала загрузить из памяти подходящие значения, а затем сохранить полученный результат обратно в память. Вычисления сами по себе происходят невероятно быстро, но доступ к памяти выполняется сравнительно медленно. Понимание того, как процессор взаимодействует с памятью, может пригодиться при оптимизации и отладке программы.

Обычно при каждом обращении к памяти процессор считывает определенное количество байтов. Эту величину принято называть *машинным словом*. Таким образом, память поделена на слова, каждое из которых является атомарной единицей чтения и записи. Количество байтов в машинном слове зависит от архитектуры. Например, в большинстве 64-битных компьютеров слово занимает 32 бита, или 4 байта. Касательно выравнивания принято считать, что переменная выровнена в памяти, если ее первый байт совпадает с началом машинного слова. Так процессор может оптимизировать обращения к памяти, необходимые для загрузки ее значения.

Вернемся к примеру 1.21. Каждое из первых трех полей, `first`, `second` и `third`, занимает 1 байт и находится в первом машинном слове структуры; все они могут быть прочитаны за одно обращение к памяти. А вот четвертое поле, `fourth`, занимает 2 байта. Если забыть о выравнивании данных в памяти, то его начальный байт должен стать последним байтом первого слова.

В подобном случае для загрузки значения данного поля из памяти процессору пришлось бы выполнить два обращения к памяти и заодно сместить определенные биты. Именно поэтому мы видим дополнительный ноль после байта `67`. Этот нулевой байт был добавлен с целью завершить текущее слово и начать новое с четвертого поля. Следовательно, мы можем сказать, что первое машинное слово было дополнено одним нулевым байтом. Компилятор использует *дополнительные байты* для выравнивания значений в памяти.

Выравнивание можно отключить. В терминологии языка C невыровненные структуры называются *упакованными*, и их использование может привести к двоичной несовместимости и ухудшению производительности. Определить упакованную структуру довольно легко. Мы покажем, как это делается, в примере 1.22; от 1.21 он отличается только тем, что структура `sample_t` в нем упакована. Обратите внимание: похожий код здесь заменен многоточием (листинг 1.35).

Листинг 1.35. Объявление упакованной структуры (ExtremeC_examples_chapter1_22.c)

```
#include <stdio.h>

struct __attribute__((__packed__)) sample_t {
  char first;
  char second;
  char third;
  short fourth;
} ;

void print_size(struct sample_t* var) {
  // ...
}

void print_bytes(struct sample_t* var) {
  // ...
}

int main(int argc, char** argv) {
  // ...
}
```

В терминале 1.14 показано, как этот код компилируется с помощью `clang` и запускается в macOS.

Терминал 1.14. Вывод примера 1.22

```
$ clang ExtremeC_examples_chapter1_22.c
$ ./a.out
Size: 5 bytes
65 66 67 253 2
$
```

Как видите, здесь выводится именно тот размер, который мы ожидали увидеть в примере 1.21. Итоговое размещение структуры в памяти тоже соответствует нашим ожиданиям. Упакованные структуры обычно применяются в средах, в которых память является дефицитным ресурсом; при этом они могут существенно ухудшить производительность в большинстве архитектур. Только новые процессоры могут читать невыровненные значения, занимающие несколько машинных слов, не расходуя лишние ресурсы. Выравнивание данных в памяти по умолчанию включено.

Вложенные структуры

Как я уже объяснил в предыдущих разделах, типы данных в C можно разделить на две основные категории. Одни являются примитивными и встроенными в язык, а другие определяются программистами с помощью ключевого слова `struct`. Первые называются PDT (primitive data types — примитивные типы данных), а вторые — UDT (user data types — пользовательские типы данных).

До сих пор UDT (структуры) в наших примерах состояли только из PDT. Но в этом подразделе мы покажем примеры структур, которые состоят из других структур. Такие UDT называются *сложными типами данных* и являются результатом вложенности пользовательских типов.

Начнем с примера 1.23 (листинг 1.36).

Листинг 1.36. Объявление вложенных структур (ExtremeC_examples_chapter1_23.c)

```
typedef struct {
  int x;
  int y;
} point_t;

typedef struct {
  point_t center;
  int radius;
} circle_t;

typedef struct {
  point_t start;
  point_t end;
} line_t;
```

В этом листинге у нас есть три структуры: `point_t`, `circle_t` и `line_t`. Первая является простым пользовательским типом и состоит только из примитивных полей. Но поля двух других структур имеют тип `point_t`, что делает их сложными пользовательскими типами.

Размер сложных и простых структур вычисляется одним и тем же способом: путем сложения размеров всех полей. Конечно, не стоит забывать о выравнивании, которое может повлиять на размер сложной структуры. Так, если `sizeof(int)` равно 4 байтам, то `sizeof(point_t)` будет равно 8. Точно так же `sizeof(circle_t)` и `sizeof(line_t)` равны 12 и 16 байтам соответственно.

Структурные переменные часто называют объектами. Они выступают прямым аналогом объектов в объектно-ориентированном программировании, и вы увидите, что, помимо значений, они могут инкапсулировать и функции. Поэтому такое название вполне обоснованно.

Указатели на структуры

Указатели могут ссылаться не только на PDT, но и на UDT. Работает это точно так же: указатель содержит адрес памяти и с ним можно выполнять арифметические операции, как мы делали это с указателями на примитивные типы. Длина арифметического шага в случае с UDT эквивалентна размеру структуры. Если вы плохо ориентируетесь в указателях и арифметических операциях, которые они поддерживают, то, пожалуйста, прочитайте соответствующий раздел на с. 44.

Необходимо понимать: структурный указатель ссылается на адрес первого поля структурной переменной. В примере 1.23 указатель типа `point_t` хранил адрес своего первого поля, `x`. То же самое касается типа `circle_t`. Указатель на `circle_t` хранил адрес своего первого поля, `center`; но, поскольку это поле в действительности являлось объектом `point_t`, данный указатель фактически ссылался на первое поле этого объекта, `x`. Таким образом, у нас может быть три указателя с адресом одной и той же ячейки памяти. Это показано в листинге 1.37.

Листинг 1.37. Три разных указателя трех разных типов ссылаются на один и тот же байт памяти (ExtremeC_examples_chapter1_24.c)

```
#include <stdio.h>

typedef struct {
  int x;
  int y;
} point_t;

typedef struct {
  point_t center;
  int radius;
} circle_t;

int main(int argc, char** argv) {
  circle_t c;

  circle_t* p1 = &c;
  point_t*  p2 = (point_t*)&c;
  int*      p3 = (int*)&c;

  printf("p1: %p\n", (void*)p1);
  printf("p2: %p\n", (void*)p2);
  printf("p3: %p\n", (void*)p3);

  return 0;
}
```

Получится следующий вывод (терминал 1.15).

Как видите, все указатели ссылаются на один и тот же байт, но их типы отличаются. Такой подход обычно используется для расширения структур, которые берутся

из внешних библиотек (то есть для добавления к ним новых полей). К тому же именно так мы реализуем *наследование* в C. Мы вернемся к этому в главе 8.

Терминал 1.15. Вывод примера 1.24

```
$ clang ExtremeC_examples_chapter1_24.c
$ ./a.out
p1: 0x7ffee846c8e0
p2: 0x7ffee846c8e0
p3: 0x7ffee846c8e0
$
```

Это был последний раздел данной главы. Далее мы займемся процессом компиляции и покажем, как правильно скомпилировать и скомпоновать проект, написанный на языке C.

Резюме

В текущей главе мы рассмотрели некоторые важные особенности языка программирования C. Мы попытались копнуть глубже и показать архитектурные аспекты этих возможностей и стоящие за ними концепции. Конечно, их эффективное использование требует более глубокого и разностороннего понимания. Ниже указаны темы, которые мы обсудили.

- На поведение препроцессора C можно влиять с помощью различных директив. Благодаря этому мы можем генерировать нужный исходный код.
- Макросы и механизм их развертывания позволяет генерировать код на языке C до передачи единицы трансляции компилятору.
- Условные директивы позволяют модифицировать код, который проходит через препроцессор, с учетом определенных условий. В результате мы можем создать разный код для разных ситуаций.
- Мы также рассмотрели указатели на переменные и их использование в C.
- Вы познакомились с обобщенными указателями и узнали, как написать функцию, которая принимает указатели любых типов.
- Мы обсудили некоторые проблемы, такие как ошибки сегментации и висячие указатели, чтобы увидеть несколько катастрофических ситуаций, которые могут возникнуть при неправильном использовании указателей.
- Затем мы поговорили о функциях и их синтаксисе.
- Мы исследовали архитектурные аспекты функций и то, какую роль они играют в хорошо спроектированных процедурных программах на C.

- Был также объяснен механизм вызова функций и то, как стековые фреймы применяются для передачи аргументов.
- Мы рассмотрели указатели на функции. Их гибкий синтаксис позволяет сохранять логику в сущностях, подобных переменным, и использовать ее в дальнейшем. На самом деле это фундаментальный механизм, с помощью которого загружается и выполняется любая современная программа.
- Структуры в сочетании с указателями на функции привели к появлению в C инкапсуляции. Более подробно об этом — в третьей части.
- Мы попытались выяснить архитектурные аспекты структур и их влияние на процесс проектирования программ в C.
- Вдобавок мы обсудили размещение в памяти структурных переменных и то, как это можно оптимизировать для максимально эффективного использования процессора.
- Помимо этого, были затронуты вложенные структуры. Мы заглянули внутрь сложных структурных переменных и увидели, как они должны размещаться в памяти.
- Заключительный раздел этой главы был посвящен указателям на структуры.

Глава 2 станет нашим первым шагом на пути к созданию проекта на C. В ней мы обсудим процесс компиляции и механизм компоновки. Чтобы перейти к следующим главам, необходимо внимательно прочитать изложенный в ней материал.

2 Компиляция и компоновка

В программировании все начинается с исходного кода. На самом деле *исходный код*, который еще иногда называют *кодовой базой*, обычно состоит из целого ряда текстовых файлов. Каждый такой файл содержит текстовые инструкции, написанные на языке программирования.

Мы уже знаем, что центральный процессор не умеет выполнять текстовые инструкции. Сначала их нужно скомпилировать (или транслировать) в машинные коды, выполнение которых обеспечивает работу программы.

В данной главе мы пошагово разберем весь процесс превращения исходного кода на языке C в готовый продукт компиляции. Это очень глубокая тема, и потому я разбил ее на пять разделов.

1. *Стандартный процесс компиляции в C*. Здесь мы узнаем, как обычно происходит компиляция в C, из каких этапов она состоит и какова их роль в создании готового продукта из исходного кода.
2. *Препроцессор*. Здесь мы во всех подробностях обсудим препроцессор, который обрабатывает код перед компиляцией.
3. *Компилятор*. Здесь будет рассказано, как компиляторы создают *промежуточное представление* исходного кода и затем транслируют его в язык ассемблера.
4. *Ассемблеры*. Вслед за компиляторами перейдем к *ассемблерам*, которые играют важную роль в трансляции ассемблерных инструкций, полученных из компилятора, в машинные коды.
5. *Компоновщику* посвящен заключительный раздел. Так называется механизм сборки, который создает готовые продукты из проектов на C. На данном этапе могут возникать специфические ошибки, и их предотвращение и исправление потребует от вас глубокого понимания этого инструмента. Мы также обсудим разные виды файлов, которые можно получить в результате компоновки одного и того же проекта, и будет дано несколько советов о дизассемблировании объектных файлов и анализе их содержимого. Более того, мы затронем *декорирование имен* в C++ и увидим, как оно помогает избежать определенных эффектов на этапе компоновки в проектах на C++.

Материал этой главы в основном предназначен для Unix-подобных операционных систем, но мы обсудим и особенности других ОС, таких как Microsoft Windows.

В первом разделе нам нужно объяснить процесс компиляция в языке C. Вы обязательно должны понимать, как в результате исходный код превращается в исполняемые и библиотечные файлы. Чтобы продолжать чтение этой и последующих глав, необходимо знать, на каких концепциях основан данный процесс и из каких этапов он состоит. Обратите внимание: продукты компиляции проекта на C подробно обсуждаются в главе 3.

Процесс компиляции

Компиляция файлов, написанных на языке C, обычно длится лишь несколько секунд, но за это время исходный код успевает пройти процесс обработки с участием четырех отдельных компонентов, каждый из которых имеет определенное назначение:

- препроцессор;
- компилятор;
- ассемблер;
- компоновщик.

Каждый из них принимает определенный ввод от предыдущего компонента и генерирует определенный вывод для следующего. Этот процесс продолжается, пока последний компонент не сгенерирует итоговый *продукт*.

Исходный код превращается в продукт только при успешном прохождении всех необходимых этапов. Это значит, что даже небольшой сбой в одном из компонентов может привести к ошибкам *компиляции* или *компоновки*, сообщения о которых вы увидите на экране.

Чтобы получить некоторые промежуточные продукты, такие как *переносимые объектные файлы*, достаточно успешно пройти первые три этапа. Последний, *компоновка*, обычно используется для создания более крупных продуктов; например, *исполняемый файл* формируется путем слияния нескольких объектных. Поэтому в результате сборки исходных файлов на языке C может получиться один или несколько объектных файлов, включая переносимые, исполняемые и *разделяемые*.

На сегодня у языка C есть много разных компиляторов. Одни из них свободные, с открытым исходным кодом, другие являются коммерческими, проприетарными решениями. Некоторые компиляторы предназначены строго для одной платформы, в то время как существуют и кросс-платформенные, хотя следует отметить, что почти на любой платформе есть по крайней мере один компилятор для C.

Полный список доступных компиляторов можно найти в «Википедии»: https://en.wikipedia.org/wiki/List_of_compilers#C_compilers.

Прежде чем рассказывать о том, какие систему и компилятор мы будем использовать в этой главе, сначала я уделю некоторое внимание термину *«платформа»* и объясню, что мы понимаем под ним.

Платформа — сочетание операционной системы и оборудования (или архитектуры), самая важная часть которого — *набор инструкций* центрального процессора. Операционная система играет роль программного компонента платформы, а аппаратный компонент определяется архитектурой. Например, мы можем иметь дело с ОС Ubuntu на ARM-плате или с Microsoft Windows на компьютере с 64-битным процессором AMD.

Кросс-платформенное программное обеспечение может работать на разных платформах. Но при этом необходимо понимать, что *кросс-платформенность* отличается от *переносимости*. Кросс-платформенное ПО обычно предоставляет разные двоичные (итоговые объектные) файлы и установщики для каждой среды, тогда как переносимые программы везде используют одни и те же исполняемые и установочные файлы.

Некоторые компиляторы для C, такие как `gcc` и `clang`, являются кросс-платформенными — они умеют генерировать код для разных платформ. А вот Java создает переносимый байт-код.

В контексте C/C++ переносимость означает, что мы можем скомпилировать свой исходный код для разных платформ, не нуждаясь во внесении каких-либо (даже малейших) изменений. Но мы вовсе не имеем в виду, что переносимыми будут итоговые объектные файлы.

В статье на «Википедии», упоминаемой выше, приводится несметное количество компиляторов для C. К счастью, все они поддерживают стандартный процесс компиляции, с которым вы познакомитесь в текущей главе.

Несмотря на столь богатый выбор, в этой главе мы должны остановиться на каком-то одном варианте. В качестве компилятора по умолчанию мы будем использовать `gcc` 7.3.0. Выбор пал на него ввиду доступности в большинстве операционных систем; к тому же ему посвящено много онлайн-ресурсов.

Нам также нужно определиться с платформой по умолчанию. В данной главе мы будем использовать операционную систему Ubuntu 18.04 и 64-битный процессор AMD в качестве архитектуры.

В этой главе время от времени будут упоминаться другие компиляторы, операционные системы и архитектуры — для сравнения разных платформ и инструментов. В таких случаях заранее указывается, о какой платформе или компиляторе идет речь.

В следующих разделах я пошагово опишу процесс компиляции. Вначале я покажу небольшой пример компиляции и компоновки исходных текстов в проекте C. В ходе этого вы познакомитесь с новыми терминами и концепциями, относящимися к сборке программ. Только потом каждый компонент будет рассмотрен в отдельном разделе. Там вы найдете подробную информацию с акцентом на внутренние концепции и процессы.

Сборка проекта на языке C

В данном подразделе демонстрируется, как собрать проект на C. Пример, с которым мы будем работать, состоит из нескольких исходных файлов, что характерно почти для всех проектов на этом языке. Но прежде, чем переходить к сборке, сначала рассмотрим структуру типичного проекта на C.

Заголовочные и исходные файлы

Любой проект на C содержит исходный код (или кодовую базу) и другие документы, которые описывают разрабатываемое приложение и используемые стандарты. Код C обычно хранится в файлах двух типов:

- *в заголовочных файлах*, как правило, имеющих расширение `.h`;
- *в исходных файлах* с расширением `.c`.

Для краткости заголовочные файлы в этой главе называются заголовками, а исходные файлы — исходниками.

Заголовочный файл обычно содержит перечисления, макросы и определения типов, а также *объявления* функций, глобальных переменных и структур. В языке C объявление и определение некоторых элементов программирования, таких как функции, переменные и структуры, могут находиться в разных файлах.

В C++ используется тот же принцип, но в других языках программирования, таких как Java, элементы определяются там, где объявлены. Эта замечательная черта C и C++ позволяет отделить объявление от определения, но вместе с тем может сделать исходный код более сложным.

Объявления принято хранить в заголовочных файлах, а соответствующие определения — в исходных. Это особенно относится к функциям.

Объявления функций настоятельно рекомендуется размещать в заголовках, а их определения — в соответствующих исходниках. И хотя это не является обязательным требованием, такой подход к проектированию позволит вам хранить определения функций вне заголовочных файлов.

Определения структур и объявления можно хранить и в разных файлах, но это делается в особых случаях. Соответствующий пример будет рассмотрен в главе 8, в которой мы обсудим *наследование* классов.

К заголовочным файлам можно подключать только другие заголовки, но не исходники. К исходным файлам можно подключать только заголовки. Подключение одних заголовочных файлов к другим считается дурным тоном. Если вы так делаете, то это обычно говорит о серьезной проблеме в архитектуре вашего проекта.

Чтобы лучше понять эту тему, рассмотрим пример. В листинге 2.1 показан код с объявлением функции `average`, состоящий из ее *возвращаемого типа* и *сигнатуры*. Сигнатура — просто имя функции со списком ее входных параметров.

Листинг 2.1. Объявление функции average

```
double average(int*, int);
```

Здесь мы видим сигнатуру функции, которая называется `average` и принимает указатель на массив целых чисел и второй целочисленный аргумент, обозначающий количество элементов в массиве. Согласно этому объявлению, функция возвращает значение типа `double`. Обратите внимание: возвращаемый тип входит в состав объявления, но редко считается частью сигнатуры функции.

Как видно в листинге 2.1, объявление функции заканчивается точкой с запятой и у него нет тела, заключенного в фигурные скобки. Стоит также отметить, что у параметров нет имен; это корректный синтаксис, но только в объявлениях, а не в определениях. Тем не менее параметры рекомендуется именовать даже при объявлении.

Объявление функции показывает, как ее задействовать, а определение содержит ее реализацию. Чтобы применить функцию, пользователю не обязательно знать имена ее параметров, поэтому в объявлении их можно опустить.

В листинге 2.2 представлено определение функции `average`, которую мы объявили ранее. Здесь вы можете видеть код, из которого состоит логика функции; он всегда находится в теле, заключенном в фигурные скобки.

Листинг 2.2. Определение функции average

```
double average(int* array, int length) {
  if (length <= 0) {
    return 0;
  }
  double sum = 0.0;
  for (int i = 0; i < length; i++) {
    sum += array[i];
  }
  return sum / length;
}
```

Мы уже об этом говорили, но я подчеркну еще раз: объявление функции хранится в заголовочном файле, а определение (или тело) — в исходном. Нарушать данное правило можно лишь в редких случаях. Кроме того, чтобы иметь доступ к объявлению, исходник должен подключить заголовочный файл. Именно так это работает в C и C++.

Если вы не до конца понимаете, зачем это нужно, то не волнуйтесь: по ходу чтения ясности прибавится.

Наличие у объявления нескольких определений в единице трансляции приведет к ошибке компиляции. Это касается всех функций, структур и глобальных переменных. Следовательно, вы не можете предоставить два определения для одной и той же функции.

Чтобы продолжить наше обсуждение, обратимся к первому примеру в этой главе, который должен продемонстрировать, как правильно скомпилировать проект на C/C++, состоящий из нескольких исходных файлов.

Пример исходных файлов

Пример 2.1 состоит из трех файлов: одного заголовка и двух исходников. Все они находятся в одном каталоге. Представленный в примере код пытается вычислить среднее значение для массива из пяти элементов.

Заголовочный файл играет роль моста, связующего два исходных файла и позволяющего разделить код на две части, которые собираются вместе. Без заголовка код нельзя разделить на два файла, не нарушая описанное выше правило (одни исходники не должны подключаться к другим). Показанный здесь заголовочный файл содержит все, что необходимо одному исходнику для использования функциональности другого.

В заголовочном файле находится объявление всего одной функции, `avg`, необходимой для работы программы. Один из исходных файлов содержит определение этой

функции, а другой — функцию `main`, которая является точкой входа в программу. Без `main` нельзя получить исполняемый двоичный файл для запуска программы. Эта функция интерпретируется компиляторами как место, с которого нужно начинать выполнение.

Теперь перейдем к содержимому этих файлов. В листинге 2.3 показан заголовок с перечислением и объявлением функции `avg`.

Листинг 2.3. Заголовочный файл из примера 2.1 (ExtremeC_examples_chapter2_1.h)

```
#ifndef EXTREMEC_EXAMPLES_CHAPTER_2_1_H
#define EXTREMEC_EXAMPLES_CHAPTER_2_1_Htypedef enum {
  NONE,
  NORMAL,
  SQUARED
} average_type_t;

// объявление функции
double avg(int*, int, average_type_t);

#endif
```

Здесь мы можем видеть перечисление — набор именованных целочисленных констант. В языке C у перечислений не может быть отдельных объявлений и определений: они должны объявляться и определяться в одном и том же месте.

Помимо перечисления, в этом листинге можно видеть *предварительное объявление* функции `avg`. Предварительным называется объявление, которое находится перед соответствующим определением. Здесь также применяется *предотвращение дублирования*, которое не дает компилятору подключить заголовочный файл дважды.

В листинге 2.4 показан исходный файл с определением функции `avg`.

Листинг 2.4. Исходный файл с определением функции avg (ExtremeC_examples_chapter2_1.c)

```
#include "ExtremeC_examples_chapter2_1.h"

double avg(int* array, int length, average_type_t type) {
  if (length <= 0 || type == NONE) {
    return 0;
  }
  double sum = 0.0;
  for (int i = 0; i < length; i++) {
    if (type == NORMAL) {
      sum += array[i];
```

```
    } else if (type == SQUARED) {
      sum += array[i] * array[i];
    }
  }
  return sum / length;
}
```

Вы уже могли заметить, что имя файла заканчивается на `.c`. Исходный файл подключает ранее представленный заголовок, поскольку перед использованием перечисления `average_type_t` и функции `avg` ему нужны их объявления. Применение нового типа (в данном случае перечисления `average_type_t`) без его предварительного объявления приводит к ошибке компиляции.

Взгляните на листинг 2.5 со вторым исходным файлом, который содержит функцию `main`.

Листинг 2.5. Главная функция в примере 2.1 (ExtremeC_examples_chapter2_1_main.c)

```
#include <stdio.h>

#include "ExtremeC_examples_chapter2_1.h"

int main(int argc, char** argv) {
  // объявление массива
  int array[5];

  // заполнение массива
  array[0] = 10;
  array[1] = 3;
  array[2] = 5;
  array[3] = -8;
  array[4] = 9;

  // вычисление среднего значения с помощью функции avg
  double average = avg(array, 5, NORMAL);
  printf("The average: %f\n", average);

  average = avg(array, 5, SQUARED);
  printf("The squared average: %f\n", average);

  return 0;
}
```

В любом проекте на языке C функция `main` служит точкой входа в программу. В приведенном выше листинге она объявляет и заполняет целочисленный массив, а затем вычисляет для него два разных средних значения. Обратите внимание на то, как `main` вызывает `avg`.

Сборка примера

Файлы из примера 2.1, с которыми вы познакомились выше, нужно собрать, чтобы получить итоговый исполняемый файл, который можно будет запустить в качестве программы. Сборка проекта на C/C++ требует компиляции его кодовой базы в *переносимые объектные файлы* (которые еще называют *промежуточными*) и затем объединения их в конечные продукты, такие как *статические библиотеки* или *исполняемые файлы*.

В других языках программирования сборка проходит аналогичным образом, только промежуточные и конечные продукты будут иметь другие имена и, скорее всего, другие форматы. Например, в Java промежуточными продуктами выступают class-файлы с *байт-кодом*, а конечными — JAR- или WAR-файлы.

Для компиляции представленных здесь исходников мы не будем использовать интегрированную среду разработки (Integrated Development Environment, IDE). Вместо этого мы применим компилятор напрямую, без вспомогательного ПО. Описанные здесь шаги ничем не отличаются от тех, которые фоново выполняет IDE, компилируя набор исходных файлов.

Прежде чем продолжать, необходимо отметить два важных правила, о которых нужно помнить.

Правило 1: компилируются только исходные файлы. Заголовки не должны содержать ничего, кроме объявлений. Поэтому для сборки примера 2.1 нам нужно всего два исходных файла: `ExtremeC_examples_chapter2_1.c` и `ExtremeC_examples_chapter2_1_main.c`.

Правило 2: каждый исходный файл компилируется по отдельности. В контексте примера 2.1 это означает, что компилятор следует запустить два раза, по одному для каждого исходника.

В одной команде компилятора можно указать сразу два исходных файла, но я не делаю этого в наших примерах и вам не рекомендую.

Таким образом, если проект состоит из 100 исходных файлов, то нам придется скомпилировать каждый из них отдельно; то есть компилятор нужно будет запустить 100 раз! Вы можете подумать, что это слишком много, но именно так следует компилировать проекты на языках C и C++. Поверьте, вам будут встречаться проекты, в которых для получения одного исполняемого файла необходимо скомпилировать несколько тысяч исходников!

Если заголовочный файл содержит код, подлежащий компиляции, то его не нужно компилировать напрямую. Вместо этого мы компилируем исходный файл, к которому он подключен. Таким образом, данный код будет скомпилирован как часть исходника.

К исходному файлу не подключаются другие исходники, поэтому он всегда компилируется отдельно. Помните: согласно общепринятым рекомендациям, исходные файлы в C/C++ подключать нельзя.

Теперь сосредоточимся на этапах, из которых состоит процесс сборки проекта на C. Сначала выполняется предобработка, и о ней мы поговорим в следующем подразделе.

Этап 1: предобработка

Это первый этап компиляции. К исходному файлу подключается ряд заголовков. Но перед компиляцией препроцессор собирает их содержимое в единый блок кода. Иными словами, после предобработки мы получаем один фрагмент кода на языке C, сформированный путем копирования заголовочных файлов в исходные.

На данном этапе должны быть выполнены и другие *директивы препроцессора*. Код, прошедший такую обработку, называется *единицей трансляции* (или *единицей компиляции*). Единица трансляции — это отдельный логический блок кода на C, сгенерированный препроцессором и готовый к компиляции.

В единице трансляции не остается ни одной директивы препроцессора. Напомню, что все такие директивы в C (и C++) начинаются с # — например, #include и #define.

Единицу трансляции можно сохранить без дальнейшей компиляции. В случае с `gcc` для этого достаточно указать параметр `-E` (регистр учитывается). В некоторых редких случаях, особенно в кросс-платформенной разработке, анализ данной единицы может пригодиться при исправлении необычных проблем.

В терминале 2.1 показана единица трансляции для файла `ExtremeC_examples_chapter2_1.c`, сгенерированная компилятором `gcc` на нашей платформе по умолчанию.

Терминал 2.1. Единица трансляции, полученная в ходе компиляции ExtremeC_examples_chapter2_1.c

```
$ gcc -E ExtremeC_examples_chapter2_1.c
# 1 "ExtremeC_examples_chapter2_1.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 31 "<command-line>"
```

```
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 32 "<command-line>" 2
# 1 "ExtremeC_examples_chapter2_1.c"

# 1 "ExtremeC_examples_chapter2_1.h" 1

typedef enum {
  NONE,
  NORMAL,
  SQUARED
} average_type_t;


double avg(int*, int, average_type_t);
# 5 "ExtremeC_examples_chapter2_1.c" 2

double avg(int* array, int length, average_type_t type) {
  if (length <= 0 || type == NONE) {
    return 0;
  }
  double sum = 0;
  for (int i = 0; i < length; i++) {
  if (type == NORMAL) {
      sum += array[i];
    } else if (type == SQUARED) {
      sum += array[i] * array[i];
    }
  }
  return sum / length;
}
$
```

Как видите, все объявления из заголовочного файла были скопированы в единицу трансляции, а все комментарии — удалены.

Единица трансляции для файла `ExtremeC_examples_chapter2_1_main.c` получилась очень большой из-за включения заголовочного файла `stdio.h`.

Все объявления в этом заголовочном файле и любые заголовки, которые к нему подключены, рекурсивно копируются в единицу трансляции. Чтобы вы понимали, насколько большой она может оказаться для файла `ExtremeC_examples_chapter2_1_main.c`, отмечу, что на нашей платформе по умолчанию она занимает 836 строк кода на C!

Параметр -E также поддерживается компилятором clang.

Итак, первый этап завершен. На вход поступает исходный файл, а на выходе получается соответствующая единица трансляции.

Этап 2: компиляция в ассемблерный код

После получения единицы трансляции можно переходить ко второму этапу — *компиляции*. На вход подается единица компиляции, полученная на предыдущем этапе, а на выходе получается соответствующий *ассемблерный код*. Он все еще может быть прочитан человеком, но уже зависит от аппаратной архитектуры и приближен к оборудованию. Для превращения в машинные инструкции его нужно продолжать обрабатывать.

Вы всегда можете остановиться после второго этапа и сохранить полученный ассемблерный код, передав компилятору `gcc` параметр `-S` (большая буква `S`). Итоговый файл будет иметь то же имя, что и исходник, только с расширением `.s`.

В терминале 2.2 показан ассемблерный код исходного файла `ExtremeC_examples_chapter2_1.c`. Обратите внимание: здесь опущены некоторые его части.

Терминал 2.2. Ассемблерный код, сгенерированный в ходе компиляции файла ExtremeC_examples_chapter2_1.c

```
$ gcc -S ExtremeC_examples_chapter2_1.c
$ cat ExtremeC_examples_chapter2_1.s
    .file   "ExtremeC_examples_chapter2_1.c"
    .text
    .globl  avg
    .type   avg, @function
avg:
.LFB0:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    movq    %rdi, -24(%rbp)
    movl    %esi, -28(%rbp)
    movl    %edx, -32(%rbp)
    cmpl    $0, -28(%rbp)
    jle .L2
    cmpl    $0, -32(%rbp)
    jne .L3
.L2:
    pxor    %xmm0, %xmm0
    jmp .L4
.L3:
    ...
.L8:
    ...
.L6:
    ...
```

```
.L7:
    ...
.L5:
  ...
.L4:
  ...
.LFE0:
    .size   avg, .-avg
    .ident  "GCC: (Ubuntu 7.3.0-16ubuntu3) 7.3.0"
    .section    .note.GNU-stack,"",@progbits
$
```

В рамках данного этапа компилятор анализирует единицу трансляции и превращает ее в ассемблерный код, рассчитанный на *целевую архитектуру*. Под таковой понимается аппаратное обеспечение или центральный процессор, на котором будет выполняться скомпилированная программа.

В терминале 2.2 был показан ассемблерный код, сгенерированный для 64-битного процессора AMD компилятором `gcc`, запущенным на компьютере той же архитектуры. В терминале 2.3 вы можете видеть ассемблерный код для 32-битного процессора ARM, полученный на платформе Intel x86-64 с помощью компилятора `gcc`. В обоих случаях использовался один и тот же исходный код на языке C.

Терминал 2.3. Ассемблерный код, полученный в ходе компиляции ExtremeC_examples_chapter2_1.c для 32-битной архитектуры ARM

```
$ cat ExtremeC_examples_chapter2_1.s
    .arch armv5t
    .fpu softvfp
    .eabi_attribute 20, 1
    .eabi_attribute 21, 1
    .eabi_attribute 23, 3
    .eabi_attribute 24, 1
    .eabi_attribute 25, 1
    .eabi_attribute 26, 2
    .eabi_attribute 30, 6
    .eabi_attribute 34, 0
    .eabi_attribute 18, 4
    .file "ExtremeC_examples_chapter2_1.s"
    .global    __aeabi_i2d
    .global    __aeabi_dadd
    .global    __aeabi_ddiv
    .text
    .align      2
    .global    avg
    .syntax unified
    .arm
    .type  avg, %function
```

```
avg:
    @ args = 0, pretend = 0, frame = 32
    @ frame_needed = 1, uses_anonymous_args = 0
    push    {r4, fp, lr}
    add    fp, sp, #8
    sub    sp, sp, #36
    str    r0, [fp, #-32]
    str    r1, [fp, #-36]
    str    r2, [fp, #-40]
    ldr    r3, [fp, #-36]
    cmp    r3, #0
    ble    .L2
    ldr    r3, [fp, #-40]
    cmp    r3, #0
    bne    .L3
.L2:
    ...
.L3:
    ...
.L8:
    ...
.L6:
    ...
.L7:
    ...
.L5:
    ...
.L4:
    mov    r0, r3
    mov    r1, r4
    sub    sp, fp, #8
    @ sp needed
    pop    {r4, fp, pc}
    .size avg, .-avg
    .ident      "GCC: (Ubuntu/Linaro 5.4.0-6ubuntu1~16.04.9) 5.4.020160609"
    .section   .note.GNU-stack,"",%progbits
$
```

Как можно видеть в терминалах 2.2 и 2.3, для двух архитектур был сгенерирован разный ассемблерный код. И это несмотря на тот факт, что мы использовали один и тот же исходный файл на языке C. Во втором случае мы применили компилятор `arm-linux-gnueabi-gcc`, запущенный на компьютере с архитектурой Intel x64-86 и под управлением Ubuntu 16.04.

Целевой называется архитектура, для которой компилируется исходник и на которой будет выполняться программа. Архитектура сборки используется для компиляции исходника и может отличаться от целевой. Например, исходный код на C можно скомпилировать для 64-битного процессора AMD, используя 32-битный компьютер ARM.

Получение ассемблерного кода из исходного — важнейший этап процесса компиляции.

Это вызвано тем, что ассемблерный код крайне близок к языку, который понимает центральный процессор. Благодаря такой важной роли компилятор — одна из самых важных и изученных областей информатики.

Этап 3: компиляция в машинные инструкции

Цель — сгенерировать инструкции машинного уровня (или *машинный код*) на основе ассемблерного кода, созданного компилятором на предыдущем этапе. У каждой архитектуры есть свой *ассемблер*, который может преобразовать собственный ассемблерный код в машинные инструкции.

Файл с машинными инструкциями, который мы сгенерируем в этом подразделе, называется *объектным файлом*. Как вы знаете, у проекта на языке C может быть несколько продуктов компиляции, но здесь мы основное внимание уделим переносимым объектным файлам. Это, вне сомнения, самый важный промежуточный продукт, который можно получить в процессе сборки.

Переносимые объектные файлы иногда называют промежуточными.

Цель этапа — сгенерировать переносимый объектный файл из ассемблерного кода, созданного компилятором. Любой другой созданный нами продукт будет основан на объектных файлах, сгенерированных ассемблером на данном этапе.

Об этих и других продуктах мы поговорим в следующих разделах текущей главы.

Двоичный файл и объектный файл содержат машинные инструкции и являются синонимами. Хотя в других контекстах понятие «двоичный файл» может иметь другое значение — например, когда речь идет о двоичных и текстовых файлах.

В большинстве Unix-подобных операционных систем есть утилита под названием `as`, с помощью которой ассемблерный код можно превратить в переносимый объектный файл.

Однако эти объектные файлы не являются исполняемыми; они содержат только те машинные инструкции, которые были сгенерированы для единицы трансляции. Таким образом, в объектном файле находятся машинные инструкции

лишь для соответствующих функций и предварительно выделенных глобальных переменных.

В терминале 2.4 показано, как с помощью утилиты `as` превратить файл `ExtremeC_examples_chapter2_1_main.s` в переносимый объектный файл.

Терминал 2.4. Создание объектного файла из ассемблерного кода одного из исходников в примере 2.1

```
$ as ExtremeC_examples_chapter2_1.s -o ExtremeC_examples_chapter2_1.o
$
```

Как можно видеть в приведенной выше команде, имя выходного объектного файла указывается с помощью параметра `-o`. Переносимые объектные файлы обычно имеют расширение `.o` (или `.obj` в Microsoft Windows), поэтому указанное нами имя тоже заканчивается на `.o`.

Содержимое объектного файла, будь то `.o` или `.obj`, нельзя представить в текстовом виде, и потому вам не удастся его прочитать. В связи с этим принято говорить, что объектный файл имеет *двоичное содержимое*.

Несмотря на то что ассемблер можно использовать напрямую, как в терминале 2.4, делать так не рекомендуется. Пусть лучше компилятор запустит утилиту `as` и сгенерирует для нас переносимый объектный файл.

В этой книге термины «объектный файл» и «переносимый объектный файл» иногда используются в качестве синонимов. Однако не все объектные файлы — переносимые. В некоторых контекстах могут подразумеваться разделяемые объектные файлы.

Почти любой компилятор языка C, если передать ему параметр `-c`, может сгенерировать соответствующий объектный файл непосредственно из исходника. Иными словами, параметр `-c` объединяет три предыдущих шага.

В терминале 2.5 показан пример того, как с помощью параметра `-c` скомпилировать `ExtremeC_examples_chapter2_1.c` и сгенерировать соответствующий объектный файл.

Терминал 2.5. Компиляция одного из исходников в примере 2.1 с последующей генерацией переносимого объектного файла

```
$ gcc -c ExtremeC_examples_chapter2_1.c
$
```

Представленная выше команда объединяет все три этапа, которые мы успели рассмотреть: предобработку, компиляцию в ассемблерный код и компиляцию

в машинные инструкции. Это значит, в результате выполнения данной команды будет сгенерирован переносимый объектный файл с тем же именем, что и у исходника (только с расширением `.o`).

Обратите внимание: под компиляцией зачастую имеют в виду первые три этапа, а не только второй. Иногда этот термин используется в качестве синонима «сборки» и подразумевает все четыре этапа. Например, выражение «процесс сборки C» мы можем заменить на «процесс компиляции C».

Генерация машинных инструкций — последний этап компиляции исходного файла. Иными словами, компиляцию можно считать завершенной, когда получен соответствующий переносимый объектный файл. Дальше можно переходить к компиляции других исходников.

В примере 2.1 нужно скомпилировать два исходных файла. Для этого выполним следующие команды, чтобы получить соответствующие объектные файлы (терминал 2.6).

Терминал 2.6. Генерация переносимых объектных файлов из исходников примера 2.1

```
$ gcc -c ExtremeC_examples_chapter2_1.c -o impl.o
$ gcc -c ExtremeC_examples_chapter2_1_main.c -o main.o
$
```

Как видите, мы указали новые имена для наших объектных файлов, воспользовавшись параметром `-o`. В итоге после компиляции они будут называться `impl.o` и `main.o`.

Пришло время напомнить, что эти переносимые объектные файлы не являются исполняемыми. Если конечным продуктом компиляции проекта должен быть исполняемый файл, то нам нужно взять все (или по крайней мере некоторые) переносимые объектные файлы, уже сгенерированные нами, и скомпоновать их в единое целое.

Этап 4: компоновка

Как мы уже знаем, пример 2.1 содержит функцию `main`, поэтому должен быть собран в исполняемый файл. Но пока у нас есть только два переносимых объектных файла. Следовательно, на данном этапе их необходимо объединить, чтобы создать еще один объектный файл, на сей раз исполняемый. Это делается путем *компоновки*.

Но сначала поговорим о том, как добавить поддержку новой архитектуры или оборудования в имеющуюся Unix-подобную систему.

Поддержка новых архитектур

Мы уже знаем, что в рамках одной архитектуры выпускается ряд процессоров, каждый из которых может выполнять определенный набор инструкций.

Наборы инструкций разрабатываются изготовителями процессоров, такими как компании Intel и ARM. Кроме того, эти компании создают для своей архитектуры специальную разновидность ассемблера.

Программу можно собрать для новой архитектуры, если выполнены два условия.

1. Известна версия ассемблера.
2. Доступна утилита (или программа) от соответствующего производителя, позволяющая скомпилировать ассемблерный код в машинные инструкции.

Если оба условия выполняются, то мы можем сгенерировать машинные инструкции из исходного кода на C. И только после этого их можно будет сохранить в *объектных файлах подходящего формата*, например *ELF* или *Mach-O*.

Определившись с версией ассемблера, утилитой для компиляции и форматом объектных файлов, вы можете создать на их основе другие инструменты, которые разработчики могут использовать для программирования на C. Но вы вряд ли заметите существование этих инструментов, поскольку вместо вас с ними напрямую обычно взаимодействует только компилятор C.

Двумя инструментами, без которых нельзя обойтись при работе с новой архитектурой, являются:

- компилятор языка C;
- компоновщик.

Это основополагающие компоненты для поддержки новой архитектуры. В сочетании с операционной системой и аппаратным обеспечением они составляют новую платформу.

Необходимо понимать, что Unix-подобные системы имеют модульную структуру. Написав несколько основополагающих модулей, таких как ассемблер, компилятор и компоновщик, вы можете создавать на их основе другие компоненты и не успеете оглянуться, как вся система уже будет работать на новой архитектуре.

Подробное описание этапа

Из всего вышесказанного мы уже знаем: для работы платформам, аналогичным Unix-подобным операционным системам, нужны ранее упомянутые инструменты, такие как ассемблер и компоновщик (помните, что их можно использовать отдельно от компилятора).

Компоновщик по умолчанию в Unix-подобных системах — `ld`. В терминале 2.7 показано, как создать исполняемый файл из переносимых объектных файлов, которые были сгенерированы в предыдущих подразделах, когда мы работали с примером 2.1, используя утилиту `ld` напрямую. Но, как вы сами увидите, применять компоновщик вручную не так-то просто.

Терминал 2.7. Попытка скомпоновать объектные файлы путем ручного использования утилиты ld

```
$ ld impl.o main.o
ld: warning: cannot find entry symbol _start; defaulting to
00000000004000e8
main.o: In function 'main':
ExtremeC_examples_chapter3_1_main.c:(.text+0x7a):
undefined reference to 'printf'
ExtremeC_examples_chapter3_1_main.c:(.text+0xb7):
undefined reference to 'printf'
ExtremeC_examples_chapter3_1_main.c:(.text+0xd0):
undefined reference to '__stack_chk_fail'
$
```

Наша команда завершилась неудачно, сгенерировав сообщения об ошибках. Если присмотреться, то можно заметить, что на трех участках сегмента Text утилита `ld` обнаружила три вызова *неопределенных* функций (или *ссылок* на функции).

Два из этих вызовов относятся к функции `printf`, которую мы использовали внутри `main`. Но третий, `__stack_chk_fail`, мы не делали. Он исходит из другого места, но откуда именно? Его сделал дополнительный код, который компилятор поместил в переносимые объектные файлы. Данная функция принадлежит Linux и в объектных файлах, сгенерированных на других платформах, может отсутствовать. Но, что бы это ни было и что бы оно ни делало, компоновщик, похоже, не может найти его определение в предоставленных объектных файлах.

Как уже говорилось выше, компоновщик по умолчанию, `ld`, сгенерировал эти ошибки, поскольку не сумел найти определения этих функций. И здесь нет ничего неожиданного, ведь мы сами не определили `printf` и `__stack_chk_fail` в примере 2.1.

Это значит, нам нужно предоставить компоновщику какие-то другие объектные файлы (не обязательно переносимые), содержащие определения функций `printf` и `__stack_chk_fail`.

Теперь вам должно быть понятно, почему использовать утилиту `ld` напрямую может быть очень сложно. В частности, чтобы выполнить компоновку и сгенерировать рабочий исполняемый файл, необходимо указать дополнительные объектные файлы и параметры.

К счастью, в Unix-подобных системах большинство известных компиляторов сами умеют передавать утилите `ld` подходящие параметры и указывать дополнительные объектные файлы. Поэтому нам не нужно использовать ее.

Теперь рассмотрим намного более простой способ создания итогового исполняемого файла. В терминале 2.8 показано, как скомпоновать объектные файлы из примера 2.1 с помощью `gcc`.

Терминал 2.8. Использование gcc для компоновки объектных файлов

```
$ gcc impl.o main.o
$ ./a.out
The average: 3.800000
The squared average: 55.800000
$
```

После выполнения этих команд можно выдохнуть с облегчением, поскольку мы наконец собрали пример 2.1 и запустили итоговый исполняемый файл!

Сборка проекта состоит из компиляции исходников с последующей их компоновкой и, возможно, добавлением других библиотек. В результате получаются итоговые продукты.

На минуту остановимся и обсудим наши недавние действия. На протяжении последних нескольких подразделов мы успешно собрали пример 2.1, скомпилировав его исходники в переносимые объектные файлы и затем скомпоновав их в итоговую исполняемую программу.

Подобный процесс характерен для любой кодовой базы на C/C++. Разница лишь в количестве исходных файлов в вашем проекте, которые нужно скомпилировать.

На каждом этапе сборки используется отдельный компонент. В оставшихся разделах текущей главы представлена наиболее важная информация, касающаяся каждого компонента в данном процессе.

Для начала уделим внимание препроцессору.

Препроцессор

В самом начале книги, в главе 1, мы вскользь затронули концепцию *препроцессора*. В частности, мы поговорили о макросах, условной компиляции и предотвращении дублирования заголовков.

Как вы помните, мы назвали препроцессор неотъемлемой частью C. Это уникальная возможность, которая редко встречается в других языках программирования.

Если не вдаваться в подробности, то препроцессор позволяет модифицировать исходный код до его компиляции. Вместе с тем с его помощью можно разделять исходный код, например вынести объявления в заголовочные файлы и затем подключать их к разным исходникам.

Необходимо помнить: препроцессор не имеет никакого представления о языке C и потому не в состоянии найти какие-либо синтаксические ошибки. Вместо этого он выполняет относительно простые операции, которые в основном заключаются в замене текста. Так, представьте, что у вас есть текстовый файл `sample.c` следующего содержания (листинг 2.6).

Листинг 2.6. Исходный файл с неким текстом

```
#include <stdio.h>
#define file 1000

Hello, this is just a simple text file but ending with .c extension!
This is not a C file for sure!
But we can preprocess it!
```

Попробуем пропустить этот код через препроцессор, используя `gcc`. Обратите внимание: некоторые части терминала 2.9 были опущены. Это вызвано тем, что подключение файла `stdio.h` делает единицу трансляции слишком большой.

Терминал 2.9. Пример кода на C из листинга 2.6, прошедший предобработку

```
$ gcc -E sample.c
# 1 "sample.c"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 341 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
# 1 "sample.c" 2
# 1 "/usr/include/stdio.h" 1 3 4
# 64 "/usr/include/stdio.h" 3 4
# 1 "/usr/include/_stdio.h" 1 3 4
# 68 "/usr/include/_stdio.h" 3 4
# 1 "/usr/include/sys/cdefs.h" 1 3 4
# 587 "/usr/include/sys/cdefs.h" 3 4
# 1 "/usr/include/sys/_symbol_aliasing.h" 1 3 4
# 588 "/usr/include/sys/cdefs.h" 2 3 4
# 653 "/usr/include/sys/cdefs.h" 3 4
...
...
extern int __vsnprintf_chk (char * restrict, size_t, int, size_t,
       const char * restrict, va_list);
# 412 "/usr/include/stdio.h" 2 3 4
```

```
# 2 "sample.c" 2



Hello, this is just a simple text 1000 but ending with .c extension!
This is not a C 1000 for sure!
But we can preprocess it!
$
```

Как видите, содержимое `stdio.h` было вставлено перед текстом.

Если присмотреться, то можно заметить еще одну интересную подстановку. Все вхождения `file` в тексте были заменены числом `1000`.

Этот пример показывает, как именно работает препроцессор: выполняет простые задачи — копирует содержимое другого файла или развертывает макрос, заменяя текст. Препроцессор ничего не знает о языке C; прежде чем выполнять какие-либо дальнейшие действия, он должен разобрать входящий файл с помощью синтаксического анализатора. То есть использует анализатор для поиска директив в исходном коде.

Синтаксический анализатор — это, как правило, программа, которая разбирает входные данные и извлекает из них определенные элементы для дальнейшего анализа и обработки. Чтобы разбить входные данные на более мелкие и полезные элементы, анализатор должен понимать их структуру.

Ввиду разницы в грамматике препроцессор и компилятор языка C используют разные синтаксические анализаторы. Благодаря этому анализатор препроцессора можно применять не только для предобработки файлов на языке C.

Такие возможности препроцессора, как подключение файлов и развертывание макросов, позволяют не только собирать программы на C. С их помощью можно обрабатывать любые текстовые файлы.

Отличный источник информации о препроцессоре `gcc` — официальный документ *The GNU C Preprocessor Internals* (http://www.chiark.greenend.org.uk/doc/cpp-4.3-doc/cppinternals.html). В нем описывается принцип работы препроцессора GNU C, который применяется в компиляторе `gcc` для предварительной обработки исходных файлов.

Проследовав по приведенной выше ссылке, вы сможете узнать, как препроцессор анализирует директивы и создает *синтаксическое дерево*. Там же описаны различные алгоритмы развертывания макросов. Эта тема выходит за рамки данной главы, но если вы хотите написать собственный препроцессор для какого-то внутреннего

языка программирования или просто обработки неких текстовых файлов, то этот документ послужит отличной отправной точкой.

В большинстве Unix-подобных операционных систем есть утилита под названием *cpp* (расшифровывается как C *Pre-Processor*, а не как C *Plus Plus*!). Она входит в пакет разработки для языка C, который поставляется вместе с любой разновидностью Unix. Утилита позволяет предварительно обрабатывать файлы на C, что, собственно, и делают в фоновом режиме такие компиляторы, как `gcc`. Имея исходный файл, вы можете применить к нему `cpp` примерно так (терминал 2.10).

Терминал 2.10. Использование утилиты cpp для предварительной обработки исходного кода

```
$ cpp ExtremeC_examples_chapter2_1.c
# 1 "ExtremeC_examples_chapter2_1.c"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 340 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
...
...
# 5 "ExtremeC_examples_chapter2_1.c" 2

double avg(int* array, int length, average_type_t type) {
  if (length <= 0 || type == NONE) {
    return 0;
  }
  double sum = 0;
  for (int i = 0; i < length; i++) {
    if (type == NORMAL) {
      sum += array[i];
    } else if (type == SQUARED) {
      sum += array[i] * array[i];
    }
  }
  return sum / length;
}
$
```

В качестве заключительного примечания к этому разделу следует отметить, что файлы с расширением `.i`, передаваемые компилятору, не проходят этап предобработки, поскольку предполагается, что они уже были обработаны препроцессором. Таким образом, они сразу начинают компилироваться.

Если принудительно пропустить файл с расширением `.i` через препроцессор, то будет выведено предупреждение. Обратите внимание: терминал 2.11 был сгенерирован компилятором `clang`.

Терминал 2.11. Передача компилятору clang уже обработанного файла с расширением .i

```
$ clang -E ExtremeC_examples_chapter2_1.c  > ex2_1.i
$ clang -E ex2_1.i
clang: warning: ex2_1.i: previously preprocessed input
[-Wunused-command-line-argument]
$
```

Как видите, `clang` предупреждает нас о том, что данный файл уже прошел предварительную обработку.

В следующем разделе мы отдельно поговорим о роли компилятора в процессе сборки.

Компилятор

Как уже говорилось выше, компилятор принимает на вход единицу трансляции, подготовленную препроцессором, и генерирует соответствующие инструкции ассемблера. После компиляции множественных исходников на языке C начинается преобразование сгенерированного ассемблерного кода в переносимые объектные файлы, которые затем объединяются (возможно, с помощью других объектных файлов) в библиотеку или исполняемую программу; при этом применяются такие инструменты, как ассемблер и компоновщик, входящие в состав платформы.

В качестве примера многочисленных инструментов для разработки на C, доступных в Unix, мы обсудили `as` и `ld`. Эти утилиты используются в основном для создания объектных файлов, совместимых с платформой. Они существуют отдельно от `gcc` или какого-либо другого компилятора; то есть разрабатываются вне проекта `gcc` (мы взяли `gcc` за пример) и должны быть доступны на любой платформе даже при отсутствии компилятора. Компилятор всего лишь использует их в процессе сборки, они в него не встроены.

Это связано с тем, что платформа более осведомлена о том, какой набор инструкций поддерживает ее процессор и какие форматы/ограничения характерны для ее операционной системы. Компилятор обычно ничего об этом не знает, разве что должен как-то *оптимизировать* единицу трансляции. Из вышесказанного следует, что важнейшая задача, которую выполняет `gcc`, — преобразование единицы трансляции в инструкции ассемблера. Это то, что мы называем компиляцией.

Одна из сложностей компиляции кода на C состоит в получении корректных ассемблерных инструкций, совместимых с целевой архитектурой. Утилита `gcc` позволяет компилировать один и тот же исходник для разных архитектур, таких как ARM, Intel x86, AMD и др. Как уже отмечалось ранее, у каждой архитектуры есть свой набор инструкций процессора и полная ответственность за генерацию

корректного ассемблерного кода для конкретной архитектуры лежит на компиляторе `gcc` (или другом).

Чтобы преодолеть указанные трудности, `gcc` (или любой другой компилятор) разделяет данную задачу на два этапа. Вначале он анализирует единицу трансляции и переводит ее в переносимую структуру данных под названием *«дерево абстрактного синтаксиса»* (abstract syntax tree, AST); данная структура не имеет прямого отношения к языку C. Затем на ее основе генерируется подходящий ассемблерный код для целевой архитектуры. Первый этап не зависит от конкретной платформы и набора поддерживаемых инструкций. Все, что относится к определенной архитектуре, происходит на втором этапе. За первый этап отвечает подкомпонент *интерфейс компилятора* (compiler frontend), а за второй — *кодогенератор* (compiler backend).

В следующих подразделах мы подробно обсудим оба этапа. Начнем с AST.

Дерево абстрактного синтаксиса

Как я уже объяснил выше, интерфейс компилятора должен проанализировать единицу трансляции и создать промежуточную структуру данных. Эта структура создается путем разбора исходного кода в соответствии с *грамматикой* языка C и сохранения результата в виде дерева, *не* привязанного к конкретной архитектуре. Итоговую структуру данных обычно называют AST.

AST можно сгенерировать для любого языка программирования, а не только для C, поэтому данная структура должна быть достаточно абстрактной и независимой от синтаксиса C.

Благодаря такому подходу интерфейс компилятора может поддерживать другие языки. Именно поэтому утилиты `gcc` и `clang` входят в состав проектов *GNU Compiler Collection (GCC)* и *Low-Level Virtual Machine (LLVM)* соответственно, объединяющих в себе целый ряд компиляторов для множества языков, помимо C и C++, таких как Java, Fortran и т. д.

Получив дерево абстрактного синтаксиса, кодогенератор может приступать к его оптимизации и последующему созданию ассемблерного кода для целевой архитектуры. Чтобы лучше понять, как это работает, рассмотрим реальный экземпляр AST. Возьмем для примера следующий исходный код на C (листинг 2.7).

Листинг 2.7. Простой код на C, из которого будет сгенерировано дерево абстрактного синтаксиса (ExtremeC_examples_chapter2_2.c)

```
int main() {
  int var1 = 1;
  double var2 = 2.5;
```

```
  int var3 = var1 + var2;
  return 0;
}
```

Дальше воспользуемся компилятором `clang`, чтобы вывести AST на основе этого кода. Результат можно видеть на рис. 2.1.

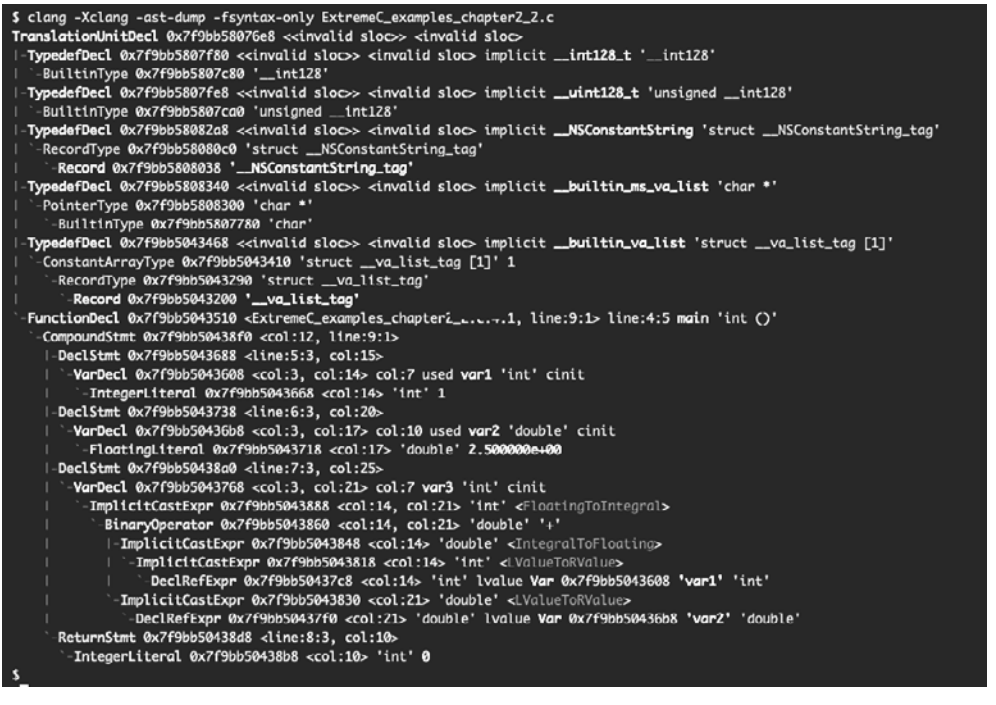

Рис. 2.1. AST на основе примера 2.2

Мы уже использовали компилятор `clang` в разных примерах, теперь познакомимся с ним поближе. Это интерфейс компилятора для языка C, разработанный организацией LLVM Developer Group для кодогенератора `llvm`. В качестве абстрактной структуры данных, которая передается между интерфейсом компилятора и кодогенератором, проект *LLVM Compiler Infrastructure Project* использует *промежуточное представление* (intermediate representation, LLVM IR). Компилятор LLVM славится тем, что может выводить IR для дальнейшего исследования. Древовидный вывод, показанный выше, — промежуточное представление, сгенерированное из исходного кода примера 2.2.

Это всего лишь основы AST. Мы не станем подробно разбирать приведенный здесь вывод, поскольку у каждого компилятора есть своя реализация этих деревьев. Чтобы тщательно рассмотреть эту тему, понадобилось бы еще несколько глав.

Но если внимательно посмотреть на рис. 2.1, то можно заметить строчку, начинающуюся с `-FunctionDecl`. Она представляет функцию `main`. Выше находится метаинформация о единице трансляции, которая передается компилятору.

Вслед за `FunctionDecl` идут элементы (или *узлы*) дерева для объявлений, бинарных операций, инструкции `return` и даже для выражений неявного приведения типов. Просматривая AST, можно найти и узнать много интересного и нового!

Еще одно преимущество генерации AST для исходного кода состоит в том, что вы можете переставить местами инструкции, удалить неиспользуемые ветки и заменить некоторые блоки в целях повышения производительности, не меняя при этом логику программы. Как уже отмечалось ранее, данный процесс называется оптимизацией и его в той или иной степени (в зависимости от конфигурации) выполняет любой компилятор языка C.

Следующим компонентом, который мы обсудим подробно, будет ассемблер.

Ассемблер

Ранее мы уже говорили о том, что для создания объектных файлов с подходящими машинными инструкциями платформа должна предоставлять ассемблер. В Unix-подобных операционных системах ассемблер можно использовать с помощью специальной утилиты. В этом разделе вы узнаете, что именно ассемблер может сохранить в объектный файл.

Если установить на том же компьютере другую Unix-подобную операционную систему, то версия ассемблера может измениться. Это очень важно, поскольку вы можете получить другие объектные файлы, хотя аппаратное обеспечение и соответствующие машинные инструкции останутся прежними!

Объектные файлы, сгенерированные в Linux для 64-битной архитектуры AMD, могут отличаться от результатов компиляции той же программы в другой ОС, такой как FreeBSD или macOS, и на том же оборудовании. Это значит, что несмотря на одинаковые машинные инструкции, объектные файлы не могут совпадать ввиду различных форматов, используемых в разных системах.

Иными словами, каждая ОС поддерживает свой двоичный формат *(формат объектных файлов)* для хранения инструкций машинного уровня. Поэтому содержимое объектного файла определяется двумя факторами: архитектурой (аппаратным обеспечением) и операционной системой. Эту комбинацию мы обычно называем платформой.

Подводя итоги, отмечу: объектные файлы и, следовательно, ассемблер, который их сгенерировал, зависят от платформы. В Linux используется формат *ELF*

(Executable and Linking Format — формат исполняемых и компонуемых файлов). Как можно догадаться по названию, он применяется для всех исполняемых/объектных файлов и разделяемых библиотек. Проще говоря, в Linux ассемблер создает ELF-файлы. В главе 3 мы подробно обсудим объектные файлы и их форматы.

Теперь поговорим о компоненте под названием *«компоновщик»*. Я продемонстрирую и объясню, как он генерирует конечный результат в проекте на C.

Компоновщик

Сборка проекта на языке C начинается с компиляции всех его исходников в соответствующие переносимые объектные файлы. Данный этап необходим для создания конечных продуктов, но одного его недостаточно. Остается сделать еще один шаг. Но прежде, чем продолжать, рассмотрим то, какие *продукты* (иногда их называют *артефактами*) можно сгенерировать в проекте на языке C.

Из проекта на C/C++ можно получить следующие продукты:

- ряд исполняемых файлов, которые в большинстве Unix-подобных систем имеют расширение `.out`. В Microsoft Windows, как правило, используется расширение `.exe`;
- ряд статических библиотек с расширением `.a` (в большинстве Unix-подобных систем) или `.lib` (в Microsoft Windows);
- ряд динамических библиотек или разделяемых объектных файлов. В Unix-подобных операционных системах они обычно имеют расширение `.so`, а в macOS и Microsoft Windows — `.dylib` и `.dll` соответственно.

Переносимые объектные файлы не входят в число этих продуктов, и потому я не упомянул о них в данном списке. Они представляют собой промежуточные ресурсы, которые используются на этапе компоновки для получения перечисленных выше продуктов; далее они не нужны. Вся ответственность за создание конечных продуктов из переносимых объектных файлов ложится на компоновщик.

Последнее, но важное замечание относится к терминам, которые мы здесь используем: все три продукта называются *объектными файлами*. Поэтому объектные файлы, сгенерированные ассемблером в качестве промежуточных ресурсов, лучше называть *переносимыми*.

Теперь вкратце опишу каждый из конечных продуктов. Более подробно об этом мы поговорим в следующей главе, которая полностью посвящена объектным файлам.

Исполняемый объектный файл можно запустить как *процесс*. Он обычно содержит значительную часть возможностей, предоставляемых проектом. Он должен иметь точку входа, с которой начинается выполнение машинных инструкций. Точкой входа в программу на языке C служит функция `main`, но в самом исполняемом объектном файле эту роль играет другая функция, зависящая от конкретной платформы; выполнив ряд системных подготовительных инструкций (добавленных на этапе компоновки), она в итоге вызовет `main`.

Статическая библиотека — не что иное, как архив с несколькими переносимыми объектными файлами внутри. Поэтому для ее создания используется не сам компоновщик, а стандартная системная программа архивации. В Unix-подобных системах это `ar`.

Статические библиотеки обычно подключаются к другим исполняемым файлам и становятся их частью. Это самый простой и удобный способ инкапсулировать программную логику и использовать ее в дальнейшем. В операционной системе существует огромное количество статических библиотек, каждая из которых содержит код доступа к определенным системным функциям.

Разделяемые объектные файлы создаются непосредственно компоновщиком и имеют более сложную структуру по сравнению с обычным архивом. Их также используют по-другому; в частности, перед применением их нужно сначала загрузить в активный процесс на этапе выполнения.

Для сравнения, статические библиотеки становятся частью итогового исполняемого файла на *этапе компоновки*. Кроме того, один и тот же разделяемый объектный файл могут загрузить и использовать сразу несколько разных процессов. В следующей главе вы увидите, как такие файлы можно загружать и применять *во время выполнения* программы на языке C.

Теперь пришло время поговорить о самом этапе компоновки. Я расскажу, с помощью каких его элементов создаются конечные продукты, особенно исполняемые файлы.

Принцип работы компоновщика

В этом разделе мы объясним, как работает компоновщик и что на самом деле понимают под компоновкой. Представьте, что вам нужно собрать проект на языке C, состоящий из пяти исходных файлов, и получить итоговую исполняемую программу. В процессе сборки вы скомпилируете все исходники и получите пять переносимых объектных файлов. Далее для создания исполняемого файла вам понадобится компоновщик.

Если подытожить вышесказанное, то компоновщик объединяет все переносимые объектные файлы вдобавок к указанным статическим библиотекам, чтобы получить готовую программу. Однако этот процесс нельзя назвать тривиальным.

Содержимое объектных файлов, объединяемых в рабочую программу, имеет несколько аспектов, на которые следует обратить внимание. Чтобы понять, как работает компоновщик, нам нужно знать, что он делает с переносимыми объектными файлами, а для этого мы должны выяснить их внутреннее устройство.

Если не вдаваться в подробности, то объектный файл содержит инструкции машинного уровня, эквивалентные единице трансляции. Однако они хранятся не в произвольном порядке, а сгруппированы в так называемые *символы*.

На самом же деле в объектном файле хранится много элементов, но символы — компонент, который объясняет принцип работы компоновщика и то, как из нескольких объектных файлов получается один. Чтобы разобраться в этом, обратимся к примеру 2.3. Я попытаюсь показать процесс компиляции и сохранения функций в соответствующие переносимые объектные файлы. Взгляните на следующий код, состоящий из двух функций (листинг 2.8).

Листинг 2.8. Код с определениями двух функций (ExtremeC_examples_chapter2_3.c)

```
int average(int a, int b) {
  return (a + b) / 2;
}

int sum(int* numbers, int count) {
  int sum = 0;
  for (int i = 0; i < count; i++) {
    sum += numbers[i];
  }
  return sum;
}
```

Для начала, чтобы получить объектный файл, этот код нужно скомпилировать. После выполнения команды, приведенной в терминале 2.12, будет создан файл `target.o`. Мы выполняем компиляцию на нашей платформе по умолчанию.

Терминал 2.12. Компиляция исходного файла из примера 2.3

```
$ gcc -c ExtremeC_examples_chapter2_3.c -o target.o
$
```

Дальше заглянем внутрь `target.o` с помощью утилиты `nm`. Она выводит символы, хранящиеся в объектном файле (терминал 2.13).

Терминал 2.13. Использование утилиты nm для просмотра символов, определенных в переносимом объектном файле

```
$ nm target.o
0000000000000000 T average
000000000000001d T sum
$
```

В этом терминале мы видим символы, определенные в объектном файле. Их имена в точности совпадают с именами функций из листинга 2.8.

Аналогичным образом можно использовать утилиту `readelf` — она выводит *таблицу символов* объектного файла, содержащую все символы, определенные в объектном файле, с некоторыми подробностями (терминал 2.14).

Терминал 2.14. Применение утилиты readelf для просмотра таблицы символов в переносимом объектном файле

```
$ readelf -s target.o

Symbol table '.symtab' contains 10 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS
ExtremeC_examples_chapter
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    2
     4: 0000000000000000     0 SECTION LOCAL  DEFAULT    3
     5: 0000000000000000     0 SECTION LOCAL  DEFAULT    5
     6: 0000000000000000     0 SECTION LOCAL  DEFAULT    6
     7: 0000000000000000     0 SECTION LOCAL  DEFAULT    4
     8: 0000000000000000    29 FUNC    GLOBAL DEFAULT    1 average
     9: 000000000000001d    69 FUNC    GLOBAL DEFAULT    1 sum
$
```

Как видите, в таблице символов, которую выводит `readelf`, есть две функции. Остальные символы ссылаются на различные разделы объектного файла. Кое-какие из них мы обсудим в этой и следующей главах.

Если вы хотите вывести под каждым символом дизассемблированные машинные инструкции, то можете использовать утилиту `objdump` (терминал 2.15).

Терминал 2.15. Применение утилиты objdump для просмотра инструкций символов, определенных в переносимом объектном файле

```
$ objdump -d target.o

target.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <average>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   89 7d fc                mov    %edi,-0x4(%rbp)
```

```
   7:   89 75 f8                mov    %esi,-0x8(%rbp)
   a:   8b 55 fc                mov    -0x4(%rbp),%edx
   d:   8b 45 f8                mov    -0x8(%rbp),%eax
  10:   01 d0                   add    %edx,%eax
  12:   89 c2                   mov    %eax,%edx
  14:   c1 ea 1f                shr    $0x1f,%edx
  17:   01 d0                   add    %edx,%eax
  19:   d1 f8                   sar    %eax
  1b:   5d                      pop    %rbp
  1c:   c3                      retq

000000000000001d <sum>:
  1d:   55                      push   %rbp
  1e:   48 89 e5                mov    %rsp,%rbp
  21:   48 89 7d e8             mov    %rdi,-0x18(%rbp)
  25:   89 75 e4                mov    %esi,-0x1c(%rbp)
  28:   c7 45 f8 00 00 00 00    movl   $0x0,-0x8(%rbp)
  2f:   c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
  36:   eb 1d                   jmp    55 <sum+0x38>
  38:   8b 45 fc                mov    -0x4(%rbp),%eax
  3b:   48 98                   cltq
  3d:   48 8d 14 85 00 00 00    lea    0x0(,%rax,4),%rdx
  44:   00
  45:   48 8b 45 e8             mov    -0x18(%rbp),%rax
  49:   48 01 d0                add    %rdx,%rax
  4c:   8b 00                   mov    (%rax),%eax
  4e:   01 45 f8                add    %eax,-0x8(%rbp)
  51:   83 45 fc 01             addl   $0x1,-0x4(%rbp)
  55:   8b 45 fc                mov    -0x4(%rbp),%eax
  58:   3b 45 e4                cmp    -0x1c(%rbp),%eax
  5b:   7c db                   jl     38 <sum+0x1b>
  5d:   8b 45 f8                mov    -0x8(%rbp),%eax
  60:   5d                      pop    %rbp
  61:   c3                      retq
$
```

Судя по этому выводу, каждый символ типа `FUNC` соответствует функции, которая была определена в исходном коде. Таким образом, мы видим: каждый переносимый объектный файл содержит только часть символов функций, необходимых для сборки полноценной исполняемой программы.

Но вернемся к основной теме данного раздела. Компоновщик собирает все символы из разных переносимых объектных файлов в один большой объектный файл, формируя тем самым исполняемую программу. Чтобы продемонстрировать, как это происходит в реальных ситуациях, мы должны воспользоваться другим примером, функции в котором разбросаны по нескольким исходникам. Так вы сможете убедиться в том, что для создания программы компоновщик ищет символы в заданных переносимых объектных файлах.

Пример 2.4 состоит из четырех файлов: трех исходников и одного заголовка. В заголовочном файле объявлено две функции, определения которых находятся в разных исходных файлах. Третий исходник содержит функцию `main`.

Функции, представленные в примере 2.4, крайне просты, и после компиляции каждая из них будет состоять всего из нескольких машинных инструкций. Кроме того, здесь не подключается ни один стандартный заголовок языка C. Это сделано для того, чтобы сделать компактной единицу трансляции каждого исходного файла.

В листинге 2.9 показан заголовочный файл.

Листинг 2.9. Объявление функций в примере 2.4 (ExtremeC_examples_chapter2_4_decls.h)

```
#ifndef EXTREMEC_EXAMPLES_CHAPTER_2_4_DECLS_H
#define EXTREMEC_EXAMPLES_CHAPTER_2_4_DECLS_H

int add(int, int);
int multiply(int, int);

#endif
```

Глядя на этот код, можно заметить выражения для предотвращения *дублирования заголовков*. Кроме того, обе функции имеют похожие сигнатуры. Каждая из них принимает на вход два целых числа, а возвращает одно.

Как уже говорилось ранее, определения этих функций находятся в разных исходных файлах. Первый файл выглядит следующим образом (листинг 2.10).

Листинг 2.10. Определение функции add (ExtremeC_examples_chapter2_4_add.c)

```
int add(int a, int b) {
  return a + b;
}
```

Здесь явно видно, что данный исходный файл не подключает никаких заголовков. Но в нем определена функция с точно такой же сигнатурой, которую мы объявили в заголовочном файле.

В листинге 2.11 можно видеть, что второй исходный файл похож на первый. Он содержит определение функции `multiply`.

Листинг 2.11. Определение функции multiply (ExtremeC_examples_chapter2_4_multiply.c)

```
int multiply(int a, int b) {
  return a * b;
}
```

Теперь можно перейти к третьему исходнику, в котором находится функция `main` (листинг 2.12).

Листинг 2.12. Главная функция в примере 2.4 (ExtremeC_examples_chapter2_4_main.c)

```
#include "ExtremeC_examples_chapter2_4_decls.h"

int main(int argc, char** argv) {
  int x = add(4, 5);
  int y = multiply(9, x);
  return 0;
}
```

Чтобы получить объявления предыдущих двух функций, к третьему исходному файлу нужно подключить заголовок. В противном случае мы получим ошибку компиляции и не сможем использовать функции `add` и `multiply` просто потому, что они не объявлены.

Кроме того, функция `main` ничего не знает об определениях `add` и `multiply`. Поэтому встает важный вопрос: каким образом она их находит, если в ней не указаны никакие другие исходные файлы? Обратите внимание: файл, показанный в листинге 2.12, содержит только один заголовок, поэтому никак не связан с двумя другими исходниками.

Чтобы ответить на этот вопрос, нужно понимать, как работает компоновка. Компоновщик собирает вместе все необходимые определения из разных объектных файлов; таким образом, код, написанный в функции `main`, может обращаться к коду других функций.

Для компиляции файла, который использует внешнюю функцию, достаточно одного объявления. Но чтобы ваша программа могла работать, компоновщик должен поместить определения в итоговый исполняемый файл.

Теперь пришло время скомпилировать пример 2.4 и продемонстрировать на практике все вышесказанное. Создадим соответствующие переносимые объектные файлы с помощью команд, показанных в терминале 2.16.

Терминал 2.16. Компиляция всех исходников примера 2.4 в соответствующие переносимые объектные файлы

```
$ gcc -c ExtremeC_examples_chapter2_4_add.c -o add.o
$ gcc -c ExtremeC_examples_chapter2_4_multiply.c -o multiply.o
$ gcc -c ExtremeC_examples_chapter2_4_main.c -o main.o
$
```

Дальше нам нужно взглянуть на таблицу символов каждого переносимого объектного файла (терминал 2.17).

Терминал 2.17. Вывод символов, определенных в файле add.o

```
$ nm add.o
0000000000000000 T add
$
```

Как видите, здесь определен символ `add`. Следующий объектный файл выглядит так (терминал 2.18).

Терминал 2.18. Вывод символов, определенных в файле multiply.o

```
$ nm multiply.o
0000000000000000 T multiply
$
```

Точно так же внутри файла `multiply.o` определен символ `multiply`. И последний объектный файл показан в терминале 2.19.

Терминал 2.19. Вывод символов, определенных в файле main.o

```
$ nm main.o
                 U add
                 U _GLOBAL_OFFSET_TABLE_
0000000000000000 T main
                 U multiply
$
```

Несмотря на то что третий исходный файл, показанный в листинге 2.12, содержит только функцию `main`, в его объектном файле можно видеть еще два символа: `add` и `multiply`. Но, в отличие от `main`, у них нет адресов, и они помечены как `U` (unresolved — «неразрешенная»). Это значит, компилятор распознал их в единице трансляции, однако не смог найти их определения. И, как уже объяснялось ранее, именно этого следовало ожидать.

Исходный файл с функцией `main`, представленный в листинге 2.12, не должен ничего знать о функциях, которые определены в других единицах трансляции, но тот факт, что он зависит от объявлений `add` и `multiply`, должен быть как-то отражен в соответствующем переносимом объектном файле.

Итак, подытожим: мы получили три объектных файла, в одном из которых есть неразрешенные символы. Таким образом, перед нами стоит вполне понятная задача: предоставить компоновщику символы, которые можно найти в других объектных файлах. Получив все необходимые символы, компоновщик сможет продолжить

их объединение в итоговый рабочий исполняемый файл. Если компоновщику не удастся найти определения неразрешенных символов, то он прервет работу и проинформирует нас об *ошибке компоновки*.

Дальше нужно скомпоновать полученные объектные файлы. Выполним для этого следующую команду (терминал 2.20).

Терминал 2.20. Компоновка всех объектных файлов

```
$ gcc add.o multiply.o main.o
$
```

Обратите внимание: если передать `gcc` список объектных файлов без каких-либо параметров, то будет выполнен этап компоновки. На самом деле компилятор в фоне передает эти файлы компоновщику, прибавляя к ним другие статические библиотеки и объектные файлы, необходимые на текущей платформе.

Посмотрим, что произойдет, если компоновщику не удастся найти подходящие определения. Укажем для этого только два промежуточных объектных файла: `main.o` и `add.o` (терминал 2.21).

Терминал 2.21. Компоновка лишь двух объектных файлов: main.o и add.o

```
$ gcc add.o main.o
main.o: In function 'main':
ExtremeC_examples_chapter2_4_main.c:(.text+0x2c): undefined
reference to 'multiply'
collect2: error: ld returned 1 exit status
$
```

Как видите, компоновщику не удалось найти в предоставленных объектных файлах символ `multiply`.

Двигаемся дальше. Укажем два других объектных файла: `main.o` и `multiply.o` (терминал 2.22).

Терминал 2.22. Компоновка лишь двух объектных файлов: main.o и multiply.o

```
$ gcc main.o multiply.o
main.o: In function 'main':
ExtremeC_examples_chapter2_4_main.c:(.text+0x1a): undefined
reference to 'add'
collect2: error: ld returned 1 exit status
$
```

Как и ожидалось, возникла та же проблема. Причина в том, что символ `add` не был найден в двух объектных файлах, которые мы предоставили.

И в завершение укажем оставшееся сочетание: `add.o` и `multiply.o`. Мы ожидаем, что эта команда будет работать, поскольку ни в одном из этих файлов нет неразрешенных символов. Результат показан в терминале 2.23.

Терминал 2.23. Компоновка лишь двух объектных файлов: add.o и multiply.o

```
$ gcc add.o multiply.o
/usr/lib/gcc/x86_64-linux-gnu/7/../../../x86_64-linux-gnu/Scrt1.o:
In function '_start':
(.text+0x20): undefined reference to 'main'
collect2: error: ld returned 1 exit status
$
```

Как видите, компоновщик снова выдал ошибку! Причина указана в выводе: ни в одном из объектных файлов не нашлось символа `main`, который необходим для создания исполняемой программы. Компоновщику нужна точка входа, роль которой в стандарте C играет функция `main`.

Обратите внимание на то, где находилась ссылка на символ `main`, — это очень важно. Компоновщик обнаружил ее в функции `_start`, которая находится в файле `/usr/lib/gcc/x86_64-Linux-gnu/7/../../../x86_64-Linux-gnu/Scrt1.o`.

Файл `Scrt1.o`, по всей видимости, переносимый объектный файл, который мы не создавали. На самом деле это один из стандартных объектных файлов языка C, скомпилированных для Linux в рамках пакета `gcc`; если компоновщик их не подключит, то программа не сможет запуститься.

Как вы только что могли убедиться, с вашим исходным кодом происходит много разных вещей, которые могут привести к конфликтам. Кроме того, есть целый ряд других объектных файлов, которые необходимо скомпоновать с вашей программой, чтобы сделать ее исполняемой.

Компоновщик можно обмануть!

Чтобы сделать наше обсуждение еще интересней, рассмотрим редкие случаи, когда этап компоновки проходит без сюрпризов, но итоговый результат получается не таким, как ожидалось. Пример подобной ситуации приводится в данном подразделе.

Пример 2.5 основан на том, что компоновщик находит некорректное определение и сохраняет его в итоговый исполняемый объектный файл.

У нас есть два исходных файла, один из которых содержит определение функции с тем же именем, которое используется в функции `main`, но с другой сигнатурой. Содержимое этих файлов приведено в листингах 2.13 и 2.14.

Листинг 2.13. Определение функции add из примера 2.5 (ExtremeC_examples_chapter2_5_add.c)

```
int add(int a, int b, int c, int d) {
  return a + b + c + d;
}
```

Листинг 2.14. Функция main из примера 2.5 (ExtremeC_examples_chapter2_5_main.c)

```
#include <stdio.h>

int add(int, int);

int main(int argc, char** argv) {
  int x = add(5, 6);
  printf("Result: %d\n", x);
  return 0;
}
```

Как видите, внутри `main` используется другая версия функции `add` с другой сигнатурой; она принимает два целых числа, а не четыре, как указано в первом исходном файле (см. листинг 2.13).

Такие функции называются *перегруженными*. Можно не сомневаться: при компиляции и компоновке этих исходников что-то пойдет не так. Интересно, удастся ли нам успешно собрать данный пример?

Скомпилируем и скомпонуем переносимые объектные файлы, воспользовавшись командами, приведенными в терминале 2.24.

Терминал 2.24. Сборка примера 2.5

```
$ gcc -c ExtremeC_examples_chapter2_5_add.c -o add.o
$ gcc -c ExtremeC_examples_chapter2_5_main.c -o main.o
$ gcc add.o main.o -o ex2_5.out
$
```

Как видите, этап компоновки прошел успешно и мы получили итоговый исполняемый файл! Это явно говорит о том, что определенные символы могут обмануть компоновщик. Теперь взглянем на вывод, который возвращает эта программа (терминал 2.25).

Терминал 2.25. Два запуска примера 2.5 со странными результатами!

```
$ ./ex2_5.out
Result: -1885535197
$ ./ex2_5.out
Result: 1679625283
$
```

Мы получили неправильный вывод, и к тому же он меняется с каждым запуском! Этот пример показывает, что передача компоновщику не того символа может вызвать проблемы. Символы — всего лишь имена, которые не несут в себе никакой информации о сигнатурах соответствующих функций. Аргументы функции — не более чем абстракция языка C; в ассемблерном коде и машинных инструкциях они фактически не существуют.

Продолжим наше исследование. Рассмотрим ассемблерный код функции `add` в другом примере. Здесь мы имеем две функции с теми же сигнатурами, которые использовались в примере 2.5.

Будем исходить из того, что пример 2.6 содержит два исходных файла. В листингах 2.15 и 2.16 показаны их коды.

Листинг 2.15. Первое определение функции add в примере 2.6 (ExtremeC_examples_chapter2_6_add_1.c)

```
int add(int a, int b, int c, int d) {
  return a + b + c + d;
}
```

Листинг 2.16. Второе определение функции add в примере 2.6 (ExtremeC_examples_chapter2_6_add_2.c)

```
int add(int a, int b) {
  return a + b;
}
```

Как и прежде, начнем с компиляции обоих исходных файлов (терминал 2.26).

Терминал 2.26. Компиляция исходников в примере 2.6 в соответствующие объектные файлы

```
$ gcc -c ExtremeC_examples_chapter2_6_add_1.c -o add_1.o
$ gcc -c ExtremeC_examples_chapter2_6_add_2.c -o add_2.o
$
```

Теперь нужно взглянуть на ассемблерный код символа `add` в обоих файлах. Начнем с `add_1.o` (терминал 2.27).

Терминал 2.27. Использование objdump для просмотра ассемблерного кода символа add в файле add_1.o

```
$ objdump -d add_1.o

add_1.o:     file format elf64-x86-64


Disassembly of section .text:
```

```
0000000000000000 <add>:
   0:   55                push  %rbp
   1:   48 89 e5          mov   %rsp,%rbp
   4:   89 7d fc          mov   %edi,-0x4(%rbp)
   7:   89 75 f8          mov   %esi,-0x8(%rbp)
   a:   89 55 f4          mov   %edx,-0xc(%rbp)
   d:   89 4d f0          mov   %ecx,-0x10(%rbp)
  10:   8b 55 fc          mov   -0x4(%rbp),%edx
  13:   8b 45 f8          mov   -0x8(%rbp),%eax
  16:   01 c2             add   %eax,%edx
  18:   8b 45 f4          mov   -0xc(%rbp),%eax
  1b:   01 c2             add   %eax,%edx
  1d:   8b 45 f0          mov   -0x10(%rbp),%eax
  20:   01 d0             add   %edx,%eax
  22:   5d                pop   %rbp
  23:   c3
$
```

В терминале 2.28 показан ассемблерный код символа `add` из другого объектного файла, `add_2.o`.

Терминал 2.28. Использование objdump для просмотра ассемблерного кода символа add в файле add_2.o

```
$ objdump -d add_2.o

add_2.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <add>:
   0:   55                push  %rbp
   1:   48 89 e5          mov   %rsp,%rbp
   4:   89 7d fc          mov   %edi,-0x4(%rbp)
   7:   89 75 f8          mov   %esi,-0x8(%rbp)
   a:   8b 55 fc          mov   -0x4(%rbp),%edx
   d:   8b 45 f8          mov   -0x8(%rbp),%eax
  10:   01 d0             add   %edx,%eax
  12:   5d                pop   %rbp
  13:   c3                retq
$
```

Когда происходит вызов, на вершине стека создается новый *фрейм*, который содержит как входные аргументы функции, так и ее адрес возврата. Подробней о механизме вызова функций можно почитать в главах 4 и 5.

В терминалах 2.27 и 2.28 наглядно показан процесс извлечения аргументов со стекового фрейма. В ассемблерном коде файла `add_1.o` (см. терминал 2.27) есть строчки, показанные в листинге 2.17.

Листинг 2.17. Ассемблерные инструкции для копирования аргументов первой функции add из стекового фрейма в регистры

```
4:  89 7d fc            mov   %edi,-0x4(%rbp)
7:  89 75 f8            mov   %esi,-0x8(%rbp)
a:  89 55 f4            mov   %edx,-0xc(%rbp)
d:  89 4d f0            mov   %ecx,-0x10(%rbp)
```

Эти инструкции копируют четыре значения из адресов памяти, на которые указывает регистр `%rbp`, в локальные регистры.

Регистры — участки центрального процессора, к которым можно быстро обращаться. Прежде чем выполнять вычисления, значения лучше переместить из основной памяти в регистры — это обеспечит высокую эффективность. Регистр %rbp всегда указывает на текущий фрейм стека с аргументами, передаваемыми функции.

Ассемблерный код второго объектного файла выглядит очень похоже. Единственное его отличие в том, что операция копирования выполняется не четыре, а два раза (листинг 2.18).

Листинг 2.18. Ассемблерные инструкции для копирования аргументов второй функции add из стекового фрейма в регистры

```
4:   89 7d fc        mov   %edi,-0x4(%rbp)
7:   89 75 f8        mov   %esi,-0x8(%rbp)
```

Эти инструкции копируют два значения просто потому, что функция принимает два аргумента. Именно поэтому вывод примера 2.5 получился таким странным. Функция `main` размещает в стековом фрейме всего два значения, но в определении функции `add` указано четыре входных параметра. Таким образом, чтобы прочитать недостающие аргументы, некорректное определение `add`, скорее всего, выходит за стековый фрейм. В итоге операции суммирования передаются неправильные значения.

Чтобы этого избежать, имена символов функций можно менять в зависимости от входных типов. Данный подход называется *декорированием имен* (name mangling), и его в основном применяют в языке C++, поскольку тот поддерживает *перегрузку функций*. Мы еще вернемся к этому в следующем подразделе.

Декорирование имен в C++

Чтобы продемонстрировать декорирование имен в C++, попробуем собрать пример 2.6 с помощью компилятора этого языка, GNU C++ (`g++`).

После этого можно будет воспользоваться утилитой `readelf`, чтобы вывести таблицу символов каждого сгенерированного нами объектного файла. Таким образом, мы

сможем увидеть, как компилятор C++ изменил имена символов функций с учетом типов их входных параметров.

Как уже отмечалось ранее, процессы компиляции в C и C++ очень похожи. Поэтому на выходе можно ожидать переносимые объектные файлы. Рассмотрим оба объектных файла, созданных в результате компиляции примера 2.6 (терминал 2.29).

Терминал 2.29. Использование утилиты readelf для просмотра таблицы символов в объектных файлах, сгенерированных компилятором C++

```
$ g++ -c ExtremeC_examples_chapter2_6_add_1.o
$ g++ -c ExtremeC_examples_chapter2_6_add_2.o
$ readelf -s ExtremeC_examples_chapter2_6_add_1.o

Symbol table '.symtab' contains 9 entries:
  Num:    Value          Size Type    Bind   Vis      Ndx  Name
   0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT   UND
   1: 0000000000000000     0 FILE    LOCAL  DEFAULT   ABS  ExtremeC_
examples_chapter
   2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
   3: 0000000000000000     0 SECTION LOCAL  DEFAULT    2
   4: 0000000000000000     0 SECTION LOCAL  DEFAULT    3
   5: 0000000000000000     0 SECTION LOCAL  DEFAULT    5
   6: 0000000000000000     0 SECTION LOCAL  DEFAULT    6
   7: 0000000000000000     0 SECTION LOCAL  DEFAULT    4
   8: 0000000000000000    36 FUNC    GLOBAL DEFAULT    1   _Z3addiiii
$ readelf -s ExtremeC_examples_chapter2_6_add_2.o

Symbol table '.symtab' contains 9 entries:
  Num:    Value          Size Type    Bind   Vis      Ndx  Name
   0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT   UND
   1: 0000000000000000     0 FILE    LOCAL  DEFAULT   ABS  ExtremeC_
examples_chapter
   2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
   3: 0000000000000000     0 SECTION LOCAL  DEFAULT    2
   4: 0000000000000000     0 SECTION LOCAL  DEFAULT    3
   5: 0000000000000000     0 SECTION LOCAL  DEFAULT    5
   6: 0000000000000000     0 SECTION LOCAL  DEFAULT    6
   7: 0000000000000000     0 SECTION LOCAL  DEFAULT    4
   8: 0000000000000000    20 FUNC    GLOBAL DEFAULT    1   _Z3addii
$
```

Как видите, символы разных перегруженных версий функции `add` имеют разные имена. Версия, принимающая четыре целых числа, называется `_Z3addiiii`, а ее аналог с двумя целочисленными аргументами имеет имя `_Z3addii`.

Каждая буква `i` в имени символа обозначает один целочисленный входной параметр.

Итак, символы имеют разные имена, и если их перепутать, то компоновщик не сможет найти определение неверно указанного символа и вернет ошибку.

Декорирование имен обеспечивает поддержку перегрузки функций в C++ и помогает избежать проблем, с которыми мы столкнулись в предыдущем разделе.

Резюме

В данной главе мы рассмотрели основополагающие этапы и компоненты, необходимые для сборки проекта на языке C. Код недостаточно просто написать — его нужно еще и собрать. В этой главе вы:

- познакомились с процессом компиляции кода на языке C и его различными этапами. Мы обсудили каждый этап и узнали, что он принимает на вход и какой результат он возвращает;
- узнали определение термина *«платформа»* и то, как разные ассемблеры могут генерировать разные машинные инструкции для одной и той же программы на языке C;
- подробно изучили каждый этап и компонент;
- изучили разделение компилятора на интерфейс и кодогенератор, за счет которого GCC и LLVM поддерживают много разных языков;
- рассмотрели форматы объектных файлов и их зависимости от платформы;
- узнали, что объектные файлы зависят от платформы и должны иметь подходящий формат;
- рассмотрели принцип работы компоновщика, использование символов для поиска недостающих определений в ходе формирования конечного продукта. Изучили разные виды продуктов, которые можно получить из проекта на C, и узнали, почему переносимые (или промежуточные) объектные файлы не причисляют к продуктам компиляции;
- узнали, как обмануть компоновщик, если предоставить ему неправильное определение (см. пример 2.5);
- рассмотрели декорирование имен в C++ и как с его помощью можно избежать проблем, которые мы видели в примере 2.5.

В следующей главе мы продолжим наше обсуждение объектных файлов и их внутренней структуры.

3 Объектные файлы

В этой главе подробно обсуждаются различные продукты компиляции проектов на языках C/C++, включая переносимые, исполняемые и разделяемые объектные файлы, а также статические библиотеки. Следует отметить, что переносимые объектные файлы считаются промежуточными ресурсами; это ингредиенты для другого рода продуктов, которые являются конечными.

Учитывая современные реалии языка C, нам стоит продолжить наше обсуждение разных видов объектных файлов и их внутренней структуры. В большинстве книг, посвященных C, рассматривается только синтаксис и сам язык; но, чтобы стать успешным программистом, вам понадобятся более глубокие знания.

Создание ПО состоит не только из разработки и программирования. На самом деле это более обширный процесс, который включает в себя написание кода, компиляцию, оптимизацию, генерацию подходящих продуктов и последующие этапы, необходимые для их сопровождения на целевой платформе.

Вы должны ориентироваться в этих промежуточных этапах на уровне, который позволит вам решать любые потенциальные проблемы. Особенно это касается разработки встраиваемых систем, которые могут иметь сложные и необычные аппаратные архитектуры и наборы инструкций.

Эта глава состоит из шести разделов.

1. *Двоичный интерфейс приложений*. Здесь мы поговорим об ABI (application binary interface) и его значимости.
2. *Форматы объектных файлов*. Этот раздел посвящен разным форматам объектных файлов — как актуальным, так и устаревшим. Вы также познакомитесь с самым распространенным форматом объектных файлов в Unix-подобных системах — ELF.
3. *Переносимые объектные файлы*. Здесь будут рассмотрены переносимые объектные файлы и самые первые продукты компиляции проекта на языке C. Мы заглянем внутрь этих файлов (на примере ELF) и посмотрим, что они хранят.
4. *Исполняемые объектные файлы*. В этом разделе мы поговорим об исполняемых объектных файлах и объясним, как они создаются из разных переносимых

объектных файлов. Вы увидите различия в их внутренней структуре на примере формата ELF.

5. *Статические библиотеки*. Этот раздел посвящен статическим библиотекам и их созданию. Будет показано, как написать программу, которая использует готовую статическую библиотеку.
6. *Динамические библиотеки*. Здесь речь пойдет о разделяемых объектных файлах. Я продемонстрирую процесс их создания из разных переносимых объектных файлов и покажу, как их использовать в программе. Мы также затронем их внутреннюю структуру на примере формата ELF.

Материал, приведенный в этой главе, в основном относится к Unix-подобным системам, но мы обсудим особенности и других ОС, таких как Microsoft Windows.

Чтение дальнейшего материала потребует понимания основных концепций и этапов сборки проекта на C. Вы должны знать, что такое единица трансляции и чем компиляция отличается от компоновки. Прежде чем продолжать, пожалуйста, прочитайте предыдущую главу.

Начнем эту главу с обсуждения ABI.

Двоичный интерфейс приложений

Как вы уже, наверное, знаете, любые библиотеки и фреймворки, независимо от используемых в них технологий и языка написания, предоставляют определенный набор возможностей, известный как *программный интерфейс приложения* (Application Programming Interface, API). Если библиотеку нужно применять в другом коде, то это следует делать через ее API. Подчеркиваю, что API — это публичный интерфейс библиотеки, которого должно быть достаточно для ее полноценного использования; все остальное считается «черным ящиком», внутрь которого нельзя заглянуть.

Но представьте, что со временем API библиотеки претерпел некоторые изменения. Чтобы использовать ее последнюю версию, мы должны адаптировать свой код к новому API, иначе у нас ничего не получится. В принципе, мы можем оставить поддержку определенной (возможно, старой) версии библиотеки и игнорировать дальнейшие обновления, но давайте исходить из того, что мы хотим перейти на самую последнюю версию.

Говоря простым языком, API — своего рода соглашение (или стандарт), принятое двумя программными компонентами, которые обслуживают или используют друг друга. Концепция ABI имеет похожее назначение, только на другом уровне. Если API обеспечивает совместимость двух программных компонентов с точки зрения их функционального взаимодействия, то ABI гарантирует, что две программы

и их соответствующие объектные файлы совместимы на уровне машинных инструкций.

Например, программа не может использовать динамические или статические библиотеки с другим ABI. Что еще хуже, исполняемый файл (который, в сущности, является объектным) нельзя запустить в операционной системе, поддерживающей другой ABI. Целый ряд важнейших аспектов разработки, таких как *динамическая компоновка*, *загрузка исполняемого файла* и *соглашение о вызове функций*, должен быть реализован в строгом соответствии с согласованным ABI.

ABI обычно содержит следующую информацию:

- набор инструкций целевой архитектуры, включая инструкции процессора, структуры памяти, порядок следования байтов, регистры и т. д.;
- существующие типы данных, их размеры и правила выравнивания;
- соглашение о вызове функций, включая такие подробности, как структура *стекового фрейма* и порядок размещения аргументов в стеке;
- механизм *системных вызовов* в Unix-подобных ОС;
- используемые форматы *объектных файлов*, включая *переносимые*, *исполняемые* и *разделяемые*. Более подробном об этом — в следующем разделе;
- если речь идет об объектных файлах, сгенерированных компилятором C++, то в состав ABI также входит метод декорирования имен и структура *виртуальной таблицы*.

В Unix-подобных операционных системах, таких как Linux и BSD, самым распространенным стандартом ABI является *System V*. В рамках System V ABI по умолчанию используется *формат объектных файлов ELF* (Executable and Linking Format — формат выполнения и компоновки).

Описание System V ABI для 64-битной архитектуры AMD доступно на https://www.uclibc.org/docs/psABI-x86_64.pdf. Можете пролистать содержание этого документа, чтобы увидеть, какие области он охватывает.

В следующем разделе мы обсудим форматы объектных файлов — в частности, ELF.

Форматы объектных файлов

Как уже объяснялось в главе 2, на каждой платформе для хранения машинных инструкций используется свой формат объектных файлов. Обратите внимание: речь здесь идет о структуре самих файлов, а не о наборе инструкций, которые поддерживает архитектура. Вы уже знаете, что это два отдельных аспекта ABI: формат объектных файлов и набор инструкций архитектуры.

В этом разделе будет проведен краткий обзор некоторых из наиболее известных форматов объектных файлов. Для начала посмотрим, какие форматы используются в разных операционных системах.

- *ELF* применяется в Linux и многих других Unix-подобных ОС.
- *Mach-O* используется в системах OS X (macOS и iOS).
- *PE* применяется в Microsoft Windows.

Если обратиться к истории, то можно сказать, что все существующие на сегодняшний день форматы объектных файлов происходят от старого формата `a.out`, который был разработан для ранних версий Unix.

Название расшифровывается как *assembler output* (вывод ассемблера). Несмотря на то что этот формат уже устарел, его название до сих пор используется по умолчанию в качестве имени исполняемых файлов, которые генерирует большинство компоновщиков. Вы, наверное, помните файлы `a.out` по ряду примеров, приведенных в первой главе.

Но вскоре на смену `a.out` пришел другой формат, *COFF* (Common Object File Format — общий формат объектных файлов), ставший впоследствии основой для ELF — формата, который применяется в большинстве Unix-подобных операционных систем. В рамках выпуска OX/X компания Apple создала свою замену `a.out` под названием Mach-O. В Windows в качестве формата объектных файлов используется *PE* (Portable Execution — переносимый исполняемый файл), основанный на COFF.

Более развернутую историю форматов объектных файлов можно найти на странице https://en.wikipedia.org/wiki/COFF#History. Знание исторического контекста той или иной технологии поможет лучше понять, как она развивалась и какими характеристиками обладает (или обладала в прошлом).

Как видите, все современные форматы объектных файлов в целом имеют общих предков: `a.out` и затем COFF.

ELF — стандартный формат объектных файлов для Linux и большинства Unix-подобных операционных систем. На самом деле он входит в состав System V ABI и активно применяется в большинстве систем семейства Unix. На сегодня он является наиболее распространенным форматом объектных файлов и среди прочего используется в таких ОС, как:

- Linux;
- FreeBSD;
- NetBSD;
- Solaris.

Это значит, один и тот же объектный файл ELF, созданный для одной операционной системы, можно запускать в другой (при условии, что используется та же архитектура). У ELF, как и у любого другого *формата файлов*, есть структура. Ее краткий обзор приводится в следующих разделах.

Более подробную информацию о формате ELF и его структуре можно найти по адресу https://www.uclibc.org/docs/psABI-x86_64.pdf. Обратите внимание: в этом документе идет речь о System V ABI для 64-битной архитектуры AMD (amd64).

HTML-версия доступна по ссылке http://www.sco.com/developers/gabi/2003-12-17/ch4.intro.html.

В следующих разделах мы поговорим о промежуточных и конечных продуктах компиляции проекта на языке C. Начнем с переносимых объектных файлов.

Переносимые объектные файлы

Как мы уже объясняли в предыдущей главе, переносимые объектные файлы создаются на этапе компиляции в машинный код в ходе сборки проекта. Они считаются промежуточными и используются в качестве ингредиентов для создания дальнейших и конечных продуктов. В связи с этим мы уделим им дополнительное внимание и изучим их содержимое.

В переносимом объектном файле, полученном из скомпилированной единицы трансляции, можно найти следующие элементы:

- инструкции машинного уровня, сгенерированные из функций, найденных в единице трансляции (код);
- значения инициализированных глобальных переменных, объявленных в единице трансляции (данные);
- *таблицу символов*, содержащую все символы, которые были объявлены и на которые ссылается единица трансляции.

Это ключевые элементы, которые можно встретить в любом переносимом объектном файле. Конечно, то, как они структурированы, зависит от конкретного формата, но с помощью подходящих инструментов их можно легко извлечь. Вскоре я покажу, как это делается, на примере файла переносимого объектного файла в формате ELF.

Но прежде, чем переходить к примеру, поговорим о том, почему эти объектные файлы называются *переносимыми*. Что означает данный термин? Он связан с процессом, в ходе которого компоновщик формирует более крупный объектный файл — исполняемый или разделяемый.

О содержимом исполняемых файлов мы поговорим в следующем разделе, а пока достаточно сказать, что они представляют собой совокупность всех переносимых объектных файлов, которые входят в их состав. Теперь уделим внимание машинным инструкциям.

Все инструкции машинного уровня, найденные в отдельных переносимых объектных файлах, группируются. Для этого необходимо сделать так, чтобы их было легко *перемещать* или *переносить*. Это достигается за счет того, что каждая инструкция получает свой адрес только после компоновки. Вот почему мы называем эти объектные файлы переносимыми. Покажу это на реальном примере.

Пример 3.1 состоит из двух исходных файлов: в первом находятся определения двух функций, `max` и `max_3`, а во втором — функция `main`, которая использует объявления `max` и `max_3`. Содержимое первого исходника показано в листинге 3.1.

Листинг 3.1. Исходный файл с определениями двух функций (ExtremeC_examples_chapter3_1_funcs.c)

```
int max(int a, int b) {
  return a > b ? a : b;
}

int max_3(int a, int b, int c) {
  int temp = max(a, b);
  return c > temp ? c : temp;
}
```

Второй файл представлен в листинге 3.2.

Листинг 3.2. Функция main, использующая уже объявленные функции. Определения находятся в отдельном исходном файле (ExtremeC_examples_chapter3_1.c)

```
int max(int, int);
int max_3(int, int, int);

int a = 5;
int b = 10;

int main(int argc, char** argv) {
  int m1 = max(a, b);
  int m2 = max_3(5, 8, -1);
  return 0;
}
```

Теперь создадим из приведенных выше исходников переносимые объектные файлы. Так мы сможем исследовать их содержимое и продемонстрировать то, о чем мы говорили ранее. Обратите внимание: компиляция выполняется в Linux, поэтому конечный результат должен быть в формате ELF (терминал 3.1).

Терминал 3.1. Компиляция исходников в соответствующие переносимые объектные файлы

```
$ gcc -c ExtremeC_examples_chapter3_1_funcs.c -o funcs.o
$ gcc -c ExtremeC_examples_chapter3_1.c -o main.o
$
```

Мы получили два переносимых объектных файла в формате ELF: `funcs.o` и `main.o`. Элементы, которые мы обсуждали ранее, разделены по разным секциям. Чтобы узнать, какие секции присутствуют в переносимом объектном файле, можно воспользоваться утилитой `readelf` (терминал 3.2).

Терминал 3.2. Содержимое объектного файла funcs.o в формате ELF

```
$ readelf -hSl funcs.o
[7/7]
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:                              REL (Relocatable file)
  Machine:                           Advanced Micro Devices X86-64
...
  Number of section headers:         12
  Section header string table index: 11

Section Headers:
  [Nr] Name              Type             Address           Offset
    Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
    0000000000000000  0000000000000000           0     0     0
  [ 1] .text             PROGBITS         0000000000000000  00000040
    0000000000000045  0000000000000000  AX       0     0     1
...
  [ 3] .data             PROGBITS         0000000000000000  00000085
    0000000000000000  0000000000000000  WA       0     0     1
  [ 4] .bss              NOBITS           0000000000000000  00000085
    0000000000000000  0000000000000000  WA       0     0     1
...
  [ 9] .symtab           SYMTAB           0000000000000000  00000110
    00000000000000f0  0000000000000018          10     8     8
  [10] .strtab           STRTAB           0000000000000000  00000200
    0000000000000030  0000000000000000           0     0     1
  [11] .shstrtab         STRTAB           0000000000000000  00000278
    0000000000000059  0000000000000000           0     0     1
...
$
```

Как видите, данный переносимый объектный файл состоит из 11 секций. Принадлежащие к уже знакомым нам элементам секции выделены жирным шрифтом. Секция `.text` содержит все машинные инструкции для единицы трансляции. В секциях `.data` и `.bss` находятся соответственно значения для инициализированных глобальных переменных и количество байтов, необходимых для неинициализированных глобальных переменных. Секция `.symtab` хранит таблицу символов.

Обратите внимание: оба наших объектных файла состоят из одних и тех же секций, но имеют разное содержимое. Поэтому мы не станем показывать аналогичный вывод для `main.o`.

Как уже упоминалось ранее, в одной из секций ELF-файла находится таблица символов. Мы уже подробно рассматривали эту таблицу и ее записи в предыдущей главе. Сейчас же поговорим о том, как с ее помощью компоновщик генерирует исполняемые и разделяемые объектные файлы. Но сначала обращу ваше внимание на один аспект, который еще не затрагивался. Он будет касаться того, почему эти объектные файлы называются переносимыми.

Выведем таблицу символов для `funcs.o`. В предыдущей главе мы делали это с помощью `objdump`, но теперь воспользуемся утилитой `readelf` (терминал 3.3).

Терминал 3.3. Таблица символов объектного файла funcs.o

```
$ readelf -s funcs.o

Symbol table '.symtab' contains 10 entries:
  Num:    Value          Size Type    Bind   Vis      Ndx Name
    0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
...
    6: 0000000000000000     0 SECTION LOCAL  DEFAULT    7
    7: 0000000000000000     0 SECTION LOCAL  DEFAULT    5
    8: 0000000000000000    22 FUNC    GLOBAL DEFAULT    1 max
    9: 0000000000000016    47 FUNC    GLOBAL DEFAULT    1 max_3
$
```

Как можно видеть в столбце `Value`, функциям `max` и `max_3` присвоены адреса соответственно `0` и `22` (`16` в шестнадцатеричной системе). Это значит, что инструкции, относящиеся к данным символам, являются смежными, а их адреса начинаются с 0. Эти символы и соответствующие им машинные инструкции готовы к перемещению на другие участки итогового исполняемого файла. Взглянем на таблицу символов файла `main.o` (терминал 3.4).

Символы, относящиеся к глобальным переменным `a` и `b`, а также символ функции `main` размещены по адресам, которые не похожи на итоговые. Это признак переносимого объектного файла. Как я уже говорил, символы в переносимых объектных файлах не обладают итоговыми, абсолютными адресами; данная информация определяется на этапе компоновки.

Терминал 3.4. Таблица символов объектного файла main.o

```
$ readelf -s main.o

Symbol table '.symtab' contains 14 entries:
  Num:    Value          Size Type    Bind   Vis      Ndx Name
    0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
...
    8: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    3 a
    9: 0000000000000004     4 OBJECT  GLOBAL DEFAULT    3 b
   10: 0000000000000000    69 FUNC    GLOBAL DEFAULT    1 main
   11: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND _GLOBAL_OFFSET_TABLE_
   12: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND max
   13: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND max_3
$
```

В следующем разделе мы создадим из данных файлов исполняемую программу. Вы увидите, как изменится таблица символов.

Исполняемые объектные файлы

Теперь пришло время поговорить об исполняемых объектных файлах. Вы уже должны знать, что это один из конечных продуктов компиляции проекта на языке C. Они содержат те же элементы, что и их переносимые аналоги: машинные инструкции, значения для инициализированных глобальных переменных и таблицу символов. Отличается только расположение этих элементов. Наше обсуждение будет вестись в контексте ELF-файлов, поскольку их легко сгенерировать и изучить их внутреннюю структуру.

Чтобы получить исполняемый объектный файл в формате ELF, вернемся к примеру 3.1. В предыдущем разделе мы сгенерировали переносимые объектные файлы из двух исходников. Теперь скомпонуем их в исполняемую программу.

Как уже объяснялось в предыдущей главе, для этого нужно выполнить команду, показанную в терминале 3.5.

Терминал 3.5. Компоновка скомпилированных объектных файлов из примера 3.1

```
$ gcc funcs.o main.o -o ex3_1.out
$
```

В предыдущем разделе мы говорили о секциях, которые присутствуют в ELF-файлах. Следует сказать, что исполняемые объектные файлы этого формата состоят из большего количества секций, но содержат и ряд *сегментов*. То же самое относится и к разделяемым объектным файлам, но об этом чуть позже. Каждый сегмент состоит из определенного количества секций (от нуля и больше), сгруппированных по своему содержимому.

Например, все секции с машинными инструкциями попадают в один сегмент. В главе 4 вы увидите, что эти сегменты аккуратно ложатся на статические *сегменты памяти* активного процесса.

Взглянем на содержимое исполняемого файла, чтобы понять, о чем идет речь. Выводить секции и сегменты объектного файла в формате ELF, в том числе и исполняемого, можно с помощью уже знакомой нам команды (терминал 3.6).

Терминал 3.6. Содержимое исполняемого объектного файла ex3_1.out в формате ELF

```
$ readelf -hSl ex3_1.out
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:                              DYN (Shared object file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x4f0
  Start of program headers:          64 (bytes into file)
  Start of section headers:          6576 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         9
  Size of section headers:           64 (bytes)
  Number of section headers:         28
  Section header string table index: 27

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        0000000000000238  00000238
   000000000000001c  0000000000000000   A       0     0      1
  [ 2] .note.ABI-tag     NOTE            0000000000000254  00000254
   0000000000000020  0000000000000000   A       0     0      4
  [ 3] .note.gnu.build-i NOTE            0000000000000274  00000274
   0000000000000024  0000000000000000   A       0     0      4
...
  [26] .strtab           STRTAB          0000000000000000  00001678
   0000000000000239  0000000000000000           0     0      1
  [27] .shstrtab         STRTAB          0000000000000000  000018b1
   00000000000000f9  0000000000000000           0     0      1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
```

```
  C (compressed), x (unknown), o (OS specific), E (exclude),
  l (large), p (processor specific)

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
         FileSiz            MemSiz              Flags   Align
  PHDR           0x0000000000000040 0x0000000000000040 0x0000000000000040
         0x00000000000001f8 0x00000000000001f8  R       0x8
  INTERP         0x0000000000000238 0x0000000000000238 0x0000000000000238
         0x000000000000001c 0x000000000000001c  R       0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
...
  GNU_EH_FRAME   0x0000000000000714 0x0000000000000714 0x0000000000000714
         0x000000000000004c 0x000000000000004c  R       0x4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
         0x0000000000000000 0x0000000000000000  RW
0x10
  GNU_RELRO      0x0000000000000df0 0x0000000000200df0 0x0000000000200df0
         0x0000000000000210 0x0000000000000210  R       0x1

Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr
.gnu.version .gnu.version_r .rela.dyn .init .plt .plt.got .text .fi ni .rodata
.eh_frame_hdr .eh_frame
   03     .init_array .fi ni_array .dynamic .got .data .bss
   04     .dynamic
   05     .note.ABI-tag .note.gnu.build-id
   06     .eh_frame_hdr
   07
   08     .init_array .fini_array .dynamic .got
$
```

Этот вывод имеет несколько нюансов.

- С точки зрения формата ELF этот объектный файл является разделяемым. Иными словами, в ELF исполняемый объектный файл отличается от разделяемого только наличием определенных сегментов наподобие `INTERP`. Этот сегмент (который на самом деле ссылается на секцию `.interp`) используется загрузчиком для запуска и выполнения исполняемого файла.
- Четыре сегмента выделены жирным шрифтом. Первый, `INTERP`, уже рассмотрен в предыдущем пункте. Второй, `TEXT`, содержит все секции с машинными инструкциями. В третьем, `DATA`, находятся все значения, которые будут использоваться для инициализации глобальных переменных и других ранних структур. Четвертый сегмент ссылается на секцию с информацией, которая относится к *динамической компоновке*, такой как местоположение разделяемых объектных файлов, загружаемых во время выполнения.

- Если сравнивать с переносимым разделяемым объектным файлом, здесь есть больше секций, которые, вероятно, содержат данные, необходимые для загрузки и выполнения программы.

Как уже объяснялось в предыдущем разделе, в таблице символов переносимого объектного файла нет никаких абсолютных итоговых адресов. Это связано с тем, что секции, содержащие машинные инструкции, еще не скомпонованы.

Если копнуть чуть глубже, то процесс компоновки заключается в извлечении всех однотипных секций из заданных переносимых объектных файлов и объединении их в более крупные секции с последующим сохранением в конечный исполняемый или разделяемый объектный файл. Таким образом, только после данного этапа символам можно придать итоговый вид и присвоить адреса, которые больше не будут меняться. В исполняемом объектном файле абсолютными являются обычные адреса, а в разделяемом — *относительные*. Более подробно поговорим об этом в разделе, посвященном динамическим библиотекам.

Рассмотрим таблицу символов исполняемого файла `ex3_1.out`. Таблица содержит много записей, поэтому в терминале 3.7 она сокращена.

Терминал 3.7. Таблица символов исполняемого объектного файла ex3_1.out

```
$ readelf -s ex3_1.out
Symbol table '.dynsym' contains 6 entries:
  Num:    Value          Size Type    Bind   Vis      Ndx Name
    0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
...
    5: 0000000000000000     0 FUNC    WEAK   DEFAULT  UND
__cxa_finalize@GLIBC_2.2.5 (2)

Symbol table '.symtab' contains 66 entries:
  Num:    Value          Size Type    Bind   Vis      Ndx Name
    0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
...
   45: 0000000000201000     0 NOTYPE  WEAK   DEFAULT   22 data_start
   46: 0000000000000610    47 FUNC    GLOBAL DEFAULT   13 max_3
   47: 0000000000201014     4 OBJECT  GLOBAL DEFAULT   22 b
   48: 0000000000201018     0 NOTYPE  GLOBAL DEFAULT   22 _edata
   49: 0000000000000704     0 FUNC    GLOBAL DEFAULT   14 _fini
   50: 00000000000005fa    22 FUNC    GLOBAL DEFAULT   13 max
   51: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND
__libc_start_main@@GLIBC_
...
   64: 0000000000000000     0 FUNC    WEAK   DEFAULT  UND
__cxa_finalize@@GLIBC_2.2
   65: 00000000000004b8     0 FUNC    GLOBAL DEFAULT   10 _init
$
```

Как видите, исполняемый объектный файл содержит две разные таблицы символов. Первая, `.dynsym`, включает символы, которые должны быть разрешены во время загрузки программы, а во второй, `.symtab`, находится все то же, что и в динамической таблице, плюс все разрешенные символы.

Разрешенные символы имеют абсолютные адреса, полученные на этапе компоновки. Адреса функций `max` и `max_3` выделены жирным шрифтом.

На этом я завершаю краткий обзор исполняемых объектных файлов. В следующем разделе речь пойдет о статических библиотеках.

Статические библиотеки

Как уже объяснялось, статические библиотеки — один из возможных продуктов компиляции проекта на языке C. В этом разделе мы поговорим о том, что они собой представляют, как их создают и используют. Затем плавно перейдем к обсуждению динамических библиотек.

Статическая библиотека в Unix — обычный архив с переносимыми объектными файлами. Как правило, она компонуется с другими объектными файлами в целях получения итоговой исполняемой программы.

Обратите внимание: сами по себе статические библиотеки не являются объектными файлами. Они скорее служат их контейнерами. Иными словами, это не ELF-файлы в Linux и не Mach-O-файлы в macOS. Это просто архивы, созданные Unix-утилитой `ar`.

Перед компоновкой из статической библиотеки извлекаются переносимые объектные файлы. Затем компоновщик ищет в них неопределенные символы и пытается их разрешить.

Создадим статическую библиотеку для проекта на C/C++ с несколькими исходными файлами. Вначале нужно создать переносимые объектные файлы. Скомпилировав все исходники, вы можете воспользоваться архиватором `ar`, чтобы создать архив со статической библиотекой.

В системах семейства Unix применительно к статическим библиотекам действует общепринятое соглашение об именовании. Имя файла должно начинаться с `lib` и иметь расширение `.a`. В разных операционных системах могут действовать разные правила; например, в Microsoft Windows статические библиотеки имеют расширение `.lib`.

Представьте гипотетический проект на языке C с исходными файлами вида `aa.c`, `bb.c` и вплоть до `zz.c`. Чтобы сгенерировать из них переносимые объектные файлы,

их нужно скомпилировать примерно так, как показано в терминале 3.8. Напомню, что процесс компиляции был рассмотрен во всех подробностях в предыдущей главе.

Терминал 3.8. Компиляция множества исходников в соответствующие переносимые объектные файлы

```
$ gcc -c aa.c -o aa.o
$ gcc -c bb.c -o bb.o
.
.
.
$ gcc -c zz.c -o zz.o
$
```

В результате выполнения этих команд мы получим нужные нам переносимые объектные файлы. Обратите внимание: в большом проекте с тысячами исходников это может занять много времени. Конечно, сборку можно существенно ускорить за счет мощного компьютера и параллельной компиляции.

Для создания статической библиотеки достаточно выполнить команду, показанную в терминале 3.9.

Терминал 3.9. Примерно так выглядит создание статической библиотеки из множества переносимых объектных файлов

```
$ ar crs libexample.a aa.o bb.o ... zz.o
$
```

В итоге получится архив `libexample.a` со всеми переносимыми объектными файлами, созданными ранее. Я не стану объяснять назначение параметра `crs`, который передается утилите `ar`; вы можете почитать о нем на странице https://stackoverflow.com/questions/29714300/what-does-the-rcs-option-in-ar-do.

Архив, который создает команда ar, может быть и несжатым. Это всего лишь способ объединения множества файлов в один. Данная команда — инструмент общего назначения, и с ее помощью можно создавать архивы из файлов любого рода.

Итак, мы знаем, как создается статическая библиотека. Попробуем сделать это сами.

В примере 3.2 представлен проект на языке C с определениями геометрических функций, которые нужно объединить в библиотеку и сделать доступными для

других приложений. Здесь мы имеем дело с тремя исходниками и одним заголовком.

Из этих трех исходных файлов нужно создать статическую библиотеку под названием `libgeometry.a`. В сочетании с заголовочным файлом ее можно будет использовать для написания другой программы, которая будет применять определенные в библиотеке геометрические функции.

Содержимое исходников и заголовка показано в листингах ниже. В первом файле, `ExtremeC_examples_chapter3_2_geometry.h`, находятся все объявления, которые нужно экспортировать из нашей геометрической библиотеки. Эти объявления будут использоваться нашим будущим приложением.

Все команды для создания объектных файлов, которые здесь приводятся, проверялись в Linux. Если вы хотите выполнить их в другой операционной системе, то, возможно, придется их немного откорректировать.

Следует отметить, что будущее приложение *должно* зависеть только от объявлений, но не от определений. Поэтому сначала рассмотрим объявления нашей геометрической библиотеки (листинг 3.3).

Листинг 3.3. Заголовочный файл в примере 3.2 (ExtremeC_examples_chapter3_2_geometry.h)

```
#ifndef EXTREME_C_EXAMPLES_CHAPTER_3_2_H
#define EXTREME_C_EXAMPLES_CHAPTER_3_2_H

#define PI 3.14159265359

typedef struct {
  double x;
  double y;
} cartesian_pos_2d_t;

typedef struct {
  double length;
  // в градусах
  double theta;
} polar_pos_2d_t;

typedef struct {
  double x;
  double y;
  double z;
} cartesian_pos_3d_t;
```

```
typedef struct {
  double length;
  // в градусах
  double theta;
  // в градусах
  double phi;
} polar_pos_3d_t;

double to_radian(double deg);
double to_degree(double rad);

double cos_deg(double deg);
double acos_deg(double deg);

double sin_deg(double deg);
double asin_deg(double deg);

cartesian_pos_2d_t convert_to_2d_cartesian_pos(
        const polar_pos_2d_t* polar_pos);
polar_pos_2d_t convert_to_2d_polar_pos(
        const cartesian_pos_2d_t* cartesian_pos);

cartesian_pos_3d_t convert_to_3d_cartesian_pos(
        const polar_pos_3d_t* polar_pos);
polar_pos_3d_t convert_to_3d_polar_pos(
        const cartesian_pos_3d_t* cartesian_pos);

#endif
```

Второй файл является исходным и содержит определения тригонометрических функций, которые соответствуют первым шести объявлениям в представленном выше заголовке (листинг 3.4).

Листинг 3.4. Исходный файл с определениями тригонометрических функций (ExtremeC_examples_chapter3_2_trigon.c)

```
#include <math.h>

// нам нужно подключить заголовочный файл,
// поскольку мы хотим использовать макрос PI
#include "ExtremeC_examples_chapter3_2_geometry.h"

double to_radian(double deg) {
  return (PI * deg) / 180;
}

double to_degree(double rad) {
  return (180 * rad) / PI;
}
```

```
double cos_deg(double deg) {
  return cos(to_radian(deg));
}

double acos_deg(double deg) {
  return acos(to_radian(deg));
}

double sin_deg(double deg) {
  return sin(to_radian(deg));
}

double asin_deg(double deg) {
  return asin(to_radian(deg));
}
```

Обратите внимание: заголовочный файл можно было бы и не включать, не содержи он такие объявления, как `PI` или `to_degree`, которые используются в наших исходниках.

Еще один исходный файл содержит определения всех двумерных геометрических функций (листинг 3.5).

Листинг 3.5. Исходный файл с определениями двумерных функций (ExtremeC_examples_chapter3_2_2d.c)

```
#include <math.h>

// нам нужно подключить заголовочный файл,
// поскольку мы хотим использовать типы polar_pos_2d_t,
// cartesian_pos_2d_t и т. д., а также тригонометрические
// функции, реализованные в другом исходнике
#include "ExtremeC_examples_chapter3_2_geometry.h"

cartesian_pos_2d_t convert_to_2d_cartesian_pos(
        const polar_pos_2d_t* polar_pos) {
  cartesian_pos_2d_t result;
  result.x = polar_pos->length * cos_deg(polar_pos->theta);
  result.y = polar_pos->length * sin_deg(polar_pos->theta);
  return result;
}

polar_pos_2d_t convert_to_2d_polar_pos(
        const cartesian_pos_2d_t* cartesian_pos) {
  polar_pos_2d_t result;
  result.length = sqrt(cartesian_pos->x * cartesian_pos->x +
    cartesian_pos->y * cartesian_pos->y);
  result.theta =
      to_degree(atan(cartesian_pos->y / cartesian_pos->x));
  return result;
}
```

Последний, четвертый файл содержит определения трехмерных геометрических функций (листинг 3.6).

Листинг 3.6. Исходный файл с определениями трехмерных функций (ExtremeC_examples_chapter3_2_3d.c)

```
#include <math.h>

// нам нужно подключить заголовочный файл,
// поскольку мы хотим использовать типы polar_pos_3d_t,
// cartesian_pos_3d_t и т. д., а также тригонометрические
// функции, реализованные в другом исходнике
#include "ExtremeC_examples_chapter3_2_geometry.h"

cartesian_pos_3d_t convert_to_3d_cartesian_pos(
        const polar_pos_3d_t* polar_pos) {
  cartesian_pos_3d_t result;
  result.x = polar_pos->length *
    sin_deg(polar_pos->theta) * cos_deg(polar_pos->phi);
  result.y = polar_pos->length *
    sin_deg(polar_pos->theta) * sin_deg(polar_pos->phi);
  result.z = polar_pos->length * cos_deg(polar_pos->theta);
  return result;
}

polar_pos_3d_t convert_to_3d_polar_pos(
        const cartesian_pos_3d_t* cartesian_pos) {
  polar_pos_3d_t result;
  result.length = sqrt(cartesian_pos->x * cartesian_pos->x +
    cartesian_pos->y * cartesian_pos->y +
    cartesian_pos->z * cartesian_pos->z);
  result.theta =
      to_degree(acos(cartesian_pos->z / result.length));
  result.phi =
      to_degree(atan(cartesian_pos->y / cartesian_pos->x));
  return result;
}
```

Теперь создадим статическую библиотеку. Для этого прежде всего нужно скомпилировать приведенные выше исходники в соответствующие переносимые объектные файлы. Следует отметить, что мы не можем скомпоновать результаты компиляции для получения исполняемой программы, поскольку в данном проекте нет функции `main`. Поэтому переносимые объектные файлы можно либо оставить как есть, либо объединить их в статическую библиотеку. Есть и третий вариант: создать из них разделяемый объектный файл, но об этом мы поговорим в следующем разделе.

Здесь же мы попробуем их архивировать, чтобы получить статическую библиотеку. Для выполнения компиляции в Linux используются команды, показанные в терминале 3.10.

Терминал 3.10. Компиляция исходников в соответствующие переносимые объектные файлы

```
$ gcc -c ExtremeC_examples_chapter3_2_trigon.c -o trigon.o
$ gcc -c ExtremeC_examples_chapter3_2_2d.c -o 2d.o
$ gcc -c ExtremeC_examples_chapter3_2_3d.c -o 3d.o
$
```

Для архивации этих файлов в статическую библиотеку нужно выполнить команды, приведенные в терминале 3.11.

Терминал 3.11. Создание статической библиотеки из переносимых объектных файлов

```
$ ar crs libgeometry.a trigon.o 2d.o 3d.o
$ mkdir -p /opt/geometry
$ mv libgeometry.a /opt/geometry
$
```

Как видите, у нас получился файл `libgeometry.a`. Мы переместили его в каталог /opt/geometry, чтобы любая другая программа могли его легко найти. Опять же, если передать команде `ar` параметр `t`, можно просмотреть содержимое архива (терминал 3.12).

Терминал 3.12. Вывод содержимого статической библиотеки

```
$ ar t /opt/geometry/libgeometry.a
trigon.o
2d.o
3d.o
$
```

В этом терминале видно, что статическая библиотека, как и ожидалось, состоит из трех переносимых объектных файлов.

Итак, мы создали статическую библиотеку для примера 3.2 с геометрическими функциями. Теперь попробуем воспользоваться ею в новом приложении. При работе со статическими библиотеками в языке C необходим доступ к предоставляемым ими объявлениям. Они известны как *публичный интерфейс* библиотеки (чаще можно встретить название *API*).

Объявления нужны на этапе компиляции, ведь компилятору нужно знать о существовании типов, сигнатур функций и т. д. Для этого используются заголовочные файлы. Другая информация, например размеры типов и адреса функций, понадобится на более поздних этапах, таких как компоновка и загрузка.

Как уже отмечалось ранее, API в языке C (интерфейсы, предоставляемые библиотекой) обычно представлены в виде набора заголовков. Поэтому для написания новой программы, которая использует наши геометрические функции, достаточно иметь заголовочный файл из примера 3.2 и статическую библиотеку `libgeometry.a`.

Для использования статической библиотеки нужно написать еще один исходный файл, который будет подключать ее API и вызывать ее функции. Создадим новый пример, 3.3, код которого показан в листинге 3.7.

Листинг 3.7. Главная функция, использующая некоторые геометрические операции (ExtremeC_examples_chapter3_3.c)

```
#include <stdio.h>

#include "ExtremeC_examples_chapter3_2_geometry.h"

int main(int argc, char** argv) {
  cartesian_pos_2d_t cartesian_pos;
  cartesian_pos.x = 100;
  cartesian_pos.y = 200;
  polar_pos_2d_t polar_pos =
      convert_to_2d_polar_pos(&cartesian_pos);
  printf("Polar Position: Length: %f, Theta: %f (deg)\n",
    polar_pos.length, polar_pos.theta);
  return 0;
}
```

Этот код подключает заголовочный файл из примера 3.2, поскольку ему нужны объявления функций, которые он будет использовать.

Теперь данный исходник нужно скомпилировать, чтобы получить соответствующий переносимый объектный файл для системы Linux (терминал 3.13).

Терминал 3.13. Компиляция примера 3.3

```
$ gcc -c ExtremeC_examples_chapter3_3.c -o main.o
$
```

Теперь нам необходимо скомпоновать результат со статической библиотекой, которую мы создали в примере 3.2. В данном случае предполагается, что файл `libgeometry.a` находится в каталоге /opt/geometry, как было показано в терминале 3.11. Следующая команда завершит сборку, выполнив компоновку и создав исполняемый объектный файл `ex3_3.out` (терминал 3.14).

Терминал 3.14. Компоновка со статической библиотекой, созданной в примере 3.2

```
$ gcc main.o -L/opt/geometry -lgeometry -lm -o ex3_3.out
$
```

Чтобы понять, как работает эта команда, рассмотрим каждый отдельный параметр, который в ней указан.

- Параметр `-L/opt/geometry` сообщает компилятору `gcc`, что каталог /opt/geometry входит в число тех мест, в которых можно найти статические и разде-

ляемые библиотеки. По умолчанию компоновщик ищет библиотечные файлы в традиционных каталогах, таких как /usr/lib или /usr/local/lib. Если не указать параметр `-L`, то компоновщик выполнит поиск только по этим стандартным путям.

- Параметр `-lgeometry` говорит компилятору `gcc`, что ему нужно искать файл `libgeometry.a` или `libgeometry.so`. Расширение принадлежит разделяемым библиотекам, о которых мы поговорим в следующем разделе. Обратите внимание на то, по какому принципу выбирается имя. Если, к примеру, передать параметр `-lxyz`, то компоновщик будет искать в заданных и стандартных каталогах файл `libxyz.a` или `libxyz.so`. В случае неудавшегося поиска компоновщик остановится и сгенерирует ошибку.
- Параметр `-lm` сообщает компилятору `gcc`, что нужно искать еще одну библиотеку с именем `libm.a` или `libm.so`. Она хранит определения математических функций, доступных в *glibc*, — в частности, `cos`, `sin` и `acos`. Обратите внимание: мы собираем пример 3.3 на компьютере под управлением Linux, поэтому в качестве реализации стандартной библиотеки C используется glibc. В macOS и, возможно, в других операционных системах данный параметр можно опустить.
- Параметр `-o ex3_3.out` говорит компилятору `gcc`, что итоговый исполняемый файл должен называться `ex3_3.out`.

Если все пройдет гладко, то после выполнения представленной выше команды у вас получится исполняемый двоичный файл, который содержит все переносимые объектные файлы, найденные в `libgeometry.a`, плюс `main.o`.

Стоит отметить, что после компоновки программа не будет зависеть от наличия статических библиотек, поскольку все их содержимое *встраивается* в ее исполняемый файл. Иными словами, полученная программа является самостоятельной и для ее запуска не требуется присутствие статической библиотеки.

Однако исполняемые файлы, полученные путем компоновки большого количества статических библиотек, обычно отличаются огромными размерами. Чем больше статических библиотек и переносимых объектных файлов у них внутри, тем крупнее программа. Иногда итоговый размер может достигать сотен мегабайт или даже нескольких гигабайтов.

Разработчику приходится выбирать между размером двоичного файла и количеством его зависимостей. Ваша программа может быть компактной, но при этом использовать разделяемые библиотеки. Это значит, что конечный исполняемый файл не является самодостаточным и не сможет работать, если внешние разделяемые библиотеки отсутствуют или их не удается найти. Более подробно об этом поговорим в следующих разделах.

Итак, было показано, что собой представляют статические библиотеки и как их следует создавать и использовать. Кроме того, вы увидели, как сторонняя программа может взаимодействовать с доступным API, и узнали, как ее скомпоновать

с существующей статической библиотекой. В следующем разделе речь пойдет о динамических библиотеках. Будет продемонстрировано, как из исходников примера 3.2 создать разделяемый объектный файл (динамическую библиотеку) вместо статической библиотеки.

Динамические библиотеки

Динамические (они же разделяемые) библиотеки — еще один продукт компиляции с возможностью повторного использования. Как можно догадаться по названию, от статических библиотек они отличаются тем, что не входят в состав итогового исполняемого файла. Вместе этого их необходимо загружать и подключать во время запуска процесса.

Поскольку статические библиотеки — часть исполняемой программы, компоновщик помещает в итоговый исполняемый файл все, что удается найти в их переносимых файлах. Иными словами, компоновщик распознает неопределенные символы и необходимые определения и пытается найти их в заданных переносимых объектных файлах, после чего помещает их все в готовую программу.

Конечный продукт создается только после того, как будут найдены все неопределенные символы. То есть мы обнаруживаем все зависимости и находим соответствующий код на этапе компоновки. Если же говорить о динамических библиотеках, неопределенные символы могут оставаться и после работы компоновщика; их поиск начинается в момент, когда программа готовится к загрузке и выполнению.

Таким образом, если у нас есть неопределенные динамические символы, то этап компоновки необходимо видоизменить. Во время загрузки исполняемого файла и его подготовки к выполнению в виде процесса используется *динамический компоновщик*, или просто *загрузчик*.

Поскольку неопределенные динамические символы отсутствуют в самом исполняемом файле, их нужно искать в другом месте. Они должны загружаться из разделяемых объектных файлов — близких родственников статических библиотек. В большинстве Unix-подобных систем разделяемые объектные файлы имеют расширение `.so`, а в macOS — `.dylib`.

Перед самим запуском процесса разделяемый объектный файл загружается в участок памяти, доступный этому процессу. Данная процедура выполняется динамическим компоновщиком (или загрузчиком), который загружает и выполняет программу.

В разделе, посвященном исполняемым объектным файлам, уже говорилось о том, что исполняемые и разделяемые объектные файлы формата ELF состоят из сегментов, каждый из которых может содержать произвольное количество секций (от нуля и больше). Эти разновидности ELF-файлов имеют два основных различия.

Во-первых, символы имеют относительные и абсолютные адреса, что позволяет загружать их в рамках сразу нескольких процессов.

Это значит, в каждом процессе инструкция имеет свой адрес, но расстояние между двумя инструкциями остается неизменным. Иными словами, адреса фиксируются относительно смещения. Причиной тому факт, что переносимые объектные файлы *позиционно независимы*. Мы поговорим об этом несколько позже.

Например, если две инструкции процесса находятся по адресам 100 и 200, то в другом процессе у них могут быть адреса 140 и 240, а еще в одном — 323 и 423. Расстояние между адресами является абсолютным, но сами адреса меняются. Эти две инструкции всегда остаются на расстоянии 100 адресов друг от друга.

Во-вторых, в разделяемых объектных файлах, в отличие от исполняемых, нет сегментов для загрузки программы. Это фактически означает, что разделяемые объектные файлы нельзя выполнять.

Прежде чем углубляться в подробности обращения разных процессов к разделяемым объектным файлам, следует показать, как эти файлы создаются и используются. Создадим динамические библиотеки из примера 3.2, с которым мы работали в предыдущем разделе.

Как вы помните, мы создали статическую библиотеку с геометрическими функциями. В этом разделе мы скомпилируем тот же исходный код, чтобы получить из него разделяемый объектный файл. В терминале 3.15 показаны команды для компиляции трех исходников в соответствующие переносимые объектные файлы. Единственное отличие от примера 3.2 — параметр `-fPIC`, который передается компилятору `gcc`.

Терминал 3.15. Компиляция исходников из примера 3.2 в соответствующие позиционно независимые переносимые объектные файлы

```
$ gcc -c ExtremeC_examples_chapter3_2_2d.c -fPIC -o 2d.o
$ gcc -c ExtremeC_examples_chapter3_2_3d.c -fPIC -o 3d.o
$ gcc -c ExtremeC_examples_chapter3_2_trigon.c -fPIC -o trigon.o
$
```

Глядя на эти команды, можно заметить дополнительный параметр, `-fPIC`, который мы указали при компиляции исходников. Он *обязателен*, если вам нужно создать динамическую библиотеку из набора переносимых объектных файлов. *PIC* расшифровывается как *position independent code* (позиционно независимый код). Как я уже объяснял, позиционная независимость означает, что инструкции внутри переносимого объектного файла имеют не фиксированные, а относительные адреса; таким образом, в разных процессах они могут находиться на разных участках памяти. Причиной тому — наш способ использования динамических библиотек.

Нет никакой гарантии, что динамический компоновщик будет загружать разделяемый объектный файл по одному и тому же адресу для разных процессов. На самом деле загрузчик отображает разделяемый объектный файл в память, и диапазоны адресов у этих отображений могут различаться. Если бы адреса инструкций были абсолютными, то мы не смогли бы загружать одну и ту же динамическую библиотеку сразу в несколько участков памяти, принадлежащих разным процессам.

Более подробную информацию о том, как работает динамическая загрузка программ и разделяемых объектных файлов, можно найти по следующим ссылкам:

- https://software.intel.com/sites/default/files/m/a/1/e/dsohowto.pdf;
- https://www.technovelty.org/linux/plt-and-got-the-key-to-code-sharing-and-dynamic-libraries.html.

Чтобы создать динамическую библиотеку, нам снова нужно будет воспользоваться компилятором (в нашем случае это `gcc`). В отличие от статической библиотеки, которая является обычным архивом, разделяемый объектный файл остается объектным. Поэтому его нужно создать с помощью того же компоновщика (такого как `ld`), который мы использовали для генерации переносимых объектных файлов.

Вы уже знаете, что в большинстве Unix-подобных систем для этого предусмотрена утилита `ld`. Но по причинам, которые были изложены в предыдущей главе, я настоятельно рекомендую не использовать ее напрямую.

В терминале 3.16 показана команда, которая позволяет создать динамическую библиотеку из набора переносимых объектных файлов, скомпилированных с параметром `-fPIC`.

Терминал 3.16. Создание динамической библиотеки из переносимых объектных файлов

```
$ gcc -shared 2d.o 3d.o trigon.o -o libgeometry.so
$ mkdir -p /opt/geometry
$ mv libgeometry.so /opt/geometry
$
```

В первой команде мы передали параметр `-shared`, чтобы компилятор создал разделяемый объектный файл из переносимых. В результате получилась библиотека под названием `libgeometry.so`. Мы переместили ее в каталог /opt/geometry, чтобы другие программы, которые хотят ее использовать, могли легко к ней обращаться. Дальше нужно снова скомпилировать и скомпоновать пример 3.3.

Ранее мы компоновали пример 3.3 с созданной нами статической библиотекой `libgeometry.a`. Здесь мы повторим этот процесс, однако на сей раз выполним компоновку с разделяемым объектным файлом `libgeometry.so`.

На первый взгляд, всё (особенно команды) выглядит без изменений, но это не так. Мы скомпонуем пример 3.3 с `libgeometry.so` вместо `libgeometry.a`; более того, динамическая библиотека не встраивается в итоговый исполняемый файл, а загружается во время его запуска (терминал 3.17). Прежде чем приступать к повторной компоновке примера 3.3, не забудьте убрать файл статической библиотеки, `libgeometry.a`, из каталога /opt/geometry.

Терминал 3.17. Компоновка примера 3.3 с собранным нами разделяемым объектным файлом

```
$ rm -fv /opt/geometry/libgeometry.a
$ gcc -c ExtremeC_examples_chapter3_3.c -o main.o
$ gcc main.o -L/opt/geometry-lgeometry -lm -o ex3_3.out
$
```

Как уже объяснялось ранее, параметр `-lgeometry` заставляет компилятор найти библиотеку, статическую или разделяемую, и скомпоновать ее с остальными объектными файлами. Поскольку статическую библиотеку мы удалили, компилятор остановит свой выбор на динамической. Но даже если заданная библиотека существует в двух вариантах, при компоновке программы `gcc` отдает предпочтение разделяемому объектному файлу.

При попытке запустить файл `ex3_3.out` вы, скорее всего, получите ошибку, показанную в терминале 3.18.

Терминал 3.18. Попытка запустить пример 3.3

```
$ ./ex3_3.out
./ex3_3.out: error while loading shared libraries: libgeometry.so:
cannot open shared object file: No such file or directory
$
```

Эта ошибка нам еще не попадалась, поскольку до сих пор мы использовали статическую компоновку со статической библиотекой. Но теперь мы пытаемся запустить программу, у которой есть *динамические зависимости*, поэтому нам необходимо предоставить ей соответствующие библиотечные файлы. Но сначала разберемся, что на самом деле произошло и почему мы получили данное сообщение.

Исполняемый файл `ex3_3.out` зависит от библиотеки `libgeometry.so`, внутри которой находятся некоторые из его зависимостей. Следует отметить, что с файлом `libgeometry.a` все иначе. После компоновки со статической библиотекой исполняемый файл получается самодостаточным, поскольку в него копируется все ее содержимое; таким образом, он больше не зависит от ее существования.

Описанное не относится к разделяемым объектным файлам. Мы получили ошибку, так как загрузчику программы (динамическому компоновщику) не удалось

найти `libgeometry.so` по своим стандартным поисковым путям. Поэтому нам нужно добавить к ним каталог /opt/geometry, в котором находится файл `libgeometry.so`. Для этого нужно обновить переменную среды `LD_LIBRARY_PATH` так, чтобы она указывала на текущий каталог.

Загрузчик проверит значение этой переменной среды и выполнит поиск необходимых разделяемых библиотек по заданному пути. Следует отметить, что в одной переменной можно указать несколько путей (используя `:` в качестве разделителя) (терминал 3.19).

Терминал 3.19. Запуск примера 3.3 с указанием LD_LIBRARY_PATH

```
$ export LD_LIBRARY_PATH=/opt/geometry
$ ./ex3_3.out
Polar Position: Length: 223.606798, Theta: 63.434949 (deg)
$
```

На сей раз программа успешно запустилась! Это значит, ее загрузчик нашел разделяемый объектный файл, а динамический компоновщик загрузил из данного файла все необходимые символы.

Обратите внимание: в терминале 3.19 для изменения `LD_LIBRARY_PATH` использовалась команда `export`. Но переменные среды часто указывают вместе с запуском программы. Это показано в терминале 3.20. В обоих случаях результат будет идентичный.

Терминал 3.20. Запуск примера 3.3 с указанием LD_LIBRARY_PATH в рамках самой команды

```
$ LD_LIBRARY_PATH=/opt/geometry ./ex3_3.out
Polar Position: Length: 223.606798, Theta: 63.434949 (deg)
$
```

Компонуя программу с несколькими разделяемыми объектными файлами, как в данном примере, мы сообщаем системе о том, что в ходе ее запуска нужно найти и загрузить ряд динамических библиотек. Поэтому при запуске программы загрузчик в первую очередь автоматически ищет эти разделяемые объектные файлы и отображает нужные символы в подходящие адреса памяти, доступные процессу. Только после этого процессор начинает выполнять код.

Ручная загрузка разделяемых библиотек

Разделяемые объектные файлы можно загружать и использовать иначе, не задействуя загрузчик (динамический компоновщик), который делает это *автоматически*. Идея в том, что программист загружает динамическую библиотеку *вручную* непосредственно перед использованием ее символов (функций). У этого подхода

есть свои варианты применения, о которых мы поговорим после рассмотрения приведенных здесь примеров.

В примере 3.4 показана отложенная (или ручная) загрузка разделяемого объектного файла уже после этапа компоновки. Исходники позаимствованы из примера 3.3, только библиотека `libgeometry.so` загружается вручную внутри самой программы.

Прежде чем продолжать, необходимо заново сгенерировать `libgeometry.so`, используя немного другой метод, иначе пример 3.4 не будет работать. В Linux для этого нужно выполнить команду, показанную в терминале 3.21.

Терминал 3.21. Компоновка libgeometry.so со стандартной математической библиотекой

```
$ gcc -shared 2d.o 3d.o trigon.o -lm -o libgeometry.so
$
```

Обратите внимание на новый параметр, `-lm`, который позволит выполнить компоновку разделяемого объектного файла со стандартной математической библиотекой `libm.so`. Мы используем его в связи с тем, что в ходе ручной загрузки файла `libgeometry.so` его зависимости должны загружаться автоматически. В противном случае мы получим сообщения об отсутствии таких символов, как `cos` или `sqrt`, которые нужны самой библиотеке `libgeometry.so`. Обратите внимание: стандартная математическая библиотека не компонуется с итоговым исполняемым файлом; она будет разрешена автоматически во время загрузки `libgeometry.so`.

Скомпоновав разделяемый объектный файл, мы можем приступить к примеру 3.4 (листинг 3.8).

Листинг 3.8. Ручная загрузка геометрической библиотеки в примере 3.4 (ExtremeC_examples_chapter3_4.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

#include "ExtremeC_examples_chapter3_2_geometry.h"

polar_pos_2d_t (*func_ptr)(cartesian_pos_2d_t*);

int main(int argc, char** argv) {

  void* handle = dlopen ("/opt/geometry/libgeometry.so", RTLD_LAZY);
  if (!handle) {
    fprintf(stderr, "%s\n", dlerror());
    exit(1);
  }
```

```
  func_ptr = dlsym(handle, "convert_to_2d_polar_pos");
  if (!func_ptr)  {
    fprintf(stderr, "%s\n", dlerror());
    exit(1);
  }

  cartesian_pos_2d_t cartesian_pos;
  cartesian_pos.x = 100;
  cartesian_pos.y = 200;
  polar_pos_2d_t polar_pos = func_ptr(&cartesian_pos);
  printf("Polar Position: Length: %f, Theta: %f (deg)\n",
    polar_pos.length, polar_pos.theta);
  return 0;
}
```

Глядя на приведенный выше код, можно заметить, каким образом мы использовали функции `dlopen` и `dlsym`, чтобы загрузить разделяемый объектный файл и выполнить в нем поиск символа `convert_to_2d_polar_pos`. Функция `dlsym` возвращает указатель, с помощью которого можно вызвать нужную функцию.

Стоит отметить, что поиск разделяемого объектного файла выполняется в каталоге /opt/geometry и в случае неудачи выводится сообщение об ошибке. Обратите внимание: в macOS динамические библиотеки имеют расширение `.dylib`, поэтому если вы работаете в данной ОС, то приведенный выше код нужно подправить.

Показанные в терминале 3.22 команды компилируют листинг 3.8 и запускают исполняемый файл.

Терминал 3.22. Запуск примера 3.4

```
$ gcc ExtremeC_examples_chapter3_4.c -ldl -o ex3_4.out
$ ./ex3_4.out
Polar Position: Length: 223.606798, Theta: 63.434949 (deg)
$
```

Как видите, мы не скомпоновали программу с `libgeometry.so`, поскольку нам хотелось загружать эту библиотеку вручную, возникни такая необходимость. Данный метод часто называют *отложенной загрузкой* разделяемых объектных файлов. Несмотря на название, он может оказаться крайне полезным.

Один из примеров тому — ситуация, когда у вас есть разные реализации или версии одной и той же библиотеки. Гибкость отложенной загрузки позволяет вам выбрать нужные разделяемые объектные файлы, исходя из логики программы, и сделать это в подходящий момент. Для сравнения: традиционный способ подразумевает автоматическую разгрузку во время запуска программы, что дает вам меньше контроля за происходящим.

Резюме

Эта глава в основном посвящена различным типам объектных файлов, которые являются продуктами компиляции проектов на C/C++. Мы рассмотрели следующие темы:

- API и ABI и их отличия;
- разные форматы объектных файлов и краткую историю их появления. Все они произошли от общего предка, но со временем их пути разделились, что в итоге привело к появлению имеющихся на сегодня форматов;
- переносимые объектные файлы и их внутреннюю структуру в контексте формата ELF;
- исполняемые объектные файлы и их отличия от переносимых объектных файлов. Мы также провели краткий обзор исполняемой программы в формате ELF;
- статические и динамические таблицы символов, чтение их содержимого с помощью инструментов командной строки;
- статическую и динамическую компоновку, поиск по разным таблицам символов при создании итогового двоичного файла или запуске программы;
- статические библиотечные файлы. Мы обсудили тот факт, что они фактически являются архивами с рядом переносимых объектных файлов;
- разделяемые объектные файлы (динамические библиотеки). Мы рассмотрели, как их можно собрать из нескольких переносимых объектных файлов;
- позиционно независимый код. Вы узнали, почему переносимые объектные файлы, участвующие в создании разделяемой библиотеки, должны быть позиционно независимыми.

В следующей главе мы рассмотрим еще один ключевой аспект программирования на C/C++: структуру памяти процесса. Вы познакомитесь с различными сегментами памяти и узнаете, как писать код, который корректно работает с ними.

4 Структура памяти процесса

В этой главе мы поговорим о памяти и ее структуре внутри процесса. Управление памятью — крайне важная тема для любого программиста на языке C, и, чтобы применять рекомендуемые методики, необходимо иметь общее представление о ее структуре. На самом деле это касается не только C. Использование многих языков программирования, таких как C++ или Java, возможно при наличии базового понимания устройства и принципа работы памяти; в противном случае вы столкнетесь с серьезными проблемами, которые будет непросто выявить и исправить.

Вероятно, вы знаете, что в C управление памятью полностью ручное. Более того, вся ответственность за выделение областей памяти и их освобождение после того, как они больше не нужны, ложится на программиста.

В высокоуровневых языках программирования, таких как Java или C#, управление памятью происходит иначе и частично выполняется внутренней платформой языка — например, *Java Virtual Machine (JVM)* в случае с Java. В таких языках программист занимается лишь выделением памяти, но не ее освобождением. Ресурсы освобождаются автоматически с помощью компонента под названием *«сборщик мусора»*.

Поскольку сборщиков мусора в C и C++ нет, о концепциях и проблемах, относящихся к управлению памятью, необходимо поговорить отдельно. Вот почему упомянутым темам посвящены эта и следующая главы, в которых мы будем обсуждать общие вопросы работы с памятью в C/C++.

В этой главе мы займемся следующим:

- рассмотрим структуру памяти типичного процесса. Это поможет нам исследовать анатомию процесса и то, как он взаимодействует с памятью;
- обсудим статическую и динамическую схемы размещения в памяти;
- познакомимся с сегментами, которые можно встретить в вышеупомянутых схемах. Вы увидите, что некоторые из них находятся в исполняемом объектном файле, а остальные создаются во время загрузки процесса;

- рассмотрим инструменты и команды, которые помогут нам обнаруживать сегменты и просматривать их содержимое — как в объектном файле, так и глубоко внутри активного процесса.

В рамках этой главы мы познакомимся с двумя сегментами: *стеком* и *кучей*. Они являются частью динамической схемы размещения в памяти процесса, и в них происходят все операции по выделению и освобождению ресурсов. Более подробно об этих сегментах мы поговорим в главе 5, поскольку именно с ними программисты взаимодействуют чаще всего.

Для начала обсудим *внутреннее устройство памяти процесса*. Так вы сможете получить общее представление о том, на какие сегменты делится память активного процесса и для чего предназначен каждый сегмент.

Внутреннее устройство памяти процесса

При каждом запуске исполняемого файла операционная система создает новый процесс. Процесс — активная запущенная программа, которая загружена в память и имеет уникальный *идентификатор* (process identifier, PID). ОС полностью контролирует создание и загрузку новых процессов.

Процесс перестает быть активным либо в результате нормального завершения, либо при получении сигнала наподобие `SIGTERM`, `SIGINT` или `SIGKILL`, который в итоге заставляет его прекратить работу. Сигналы `SIGTERM` и `SIGINT` можно игнорировать, но `SIGKILL` останавливает процесс немедленно и принудительно.

Краткое описание упомянутых выше сигналов:

- SIGTERM — сигнал, запрашивающий завершение; дает возможность процессу подготовиться к выходу;
- SIGINT — сигнал прерывания, который обычно отправляется активным процессам путем нажатия Ctrl+C;
- SIGKILL — сигнал немедленного завершения; принудительно закрывает процесс, не давая ему возможности подготовиться.

Когда операционная система создает процесс, одно из первых действий, которые она совершает, — выделяет участок памяти с заранее определенной внутренней структурой. Та выглядит более или менее похоже в разных ОС, особенно в тех, которые принадлежат к семейству Unix.

В этой главе мы исследуем данную структуру и познакомимся с рядом важных и полезных терминов, которые с ней связаны.

Память типичного процесса делится на несколько частей, которые называются *сегментами*. Каждый из них представляет собой область памяти с определенной задачей, предназначенную для хранения данных конкретного типа. Ниже приведен список сегментов, из которых состоит память активного процесса:

- сегмент неинициализированных данных или BSS (block started by symbol — блок, начинающийся с символа);
- сегмент данных;
- текстовый сегмент или сегмент кода;
- сегмент стека;
- сегмент кучи.

В следующих разделах мы исследуем каждый из этих сегментов по отдельности и узнаем, какую роль они играют в выполнении программы. А стек и кучу во всех подробностях обсудим в главе 5. Но прежде, чем углубляться в подробности перечисленных выше сегментов, сначала познакомимся с инструментами, которые помогут нам исследовать память.

Исследование структуры памяти

Unix-подобные операционные системы предоставляют набор инструментов для исследования сегментов памяти процесса. В этом разделе вы узнаете, что одни из этих сегментов находятся в исполняемом файле, а другие создаются динамически на этапе выполнения, при создании процесса.

Как вы уже должны знать из предыдущих глав, исполняемый объектный файл и процесс — две разные вещи, поэтому неудивительно, что для их исследования применяются разные инструменты.

Благодаря содержанию главы 3 вы уже знаете, что исполняемый объектный файл содержит машинные инструкции и за его создание отвечает компилятор. Но процесс — активная программа, созданная путем запуска исполняемого объектного файла; она занимает участок основной памяти, а центральный процессор постоянно извлекает и выполняет ее инструкции.

Процесс — динамическая сущность, выполняемая внутри операционной системы, в то время как исполняемый объектный файл — просто набор данных с подготовленной начальной структурой, и на ее основе создается будущий процесс. Действительно, некоторые сегменты в структуре памяти активного процесса берутся непосредственно из исполняемого файла, а остальные создаются динамически

во время его загрузки. Первые называют *статической схемой размещения в памяти*, а вторые — *динамической*.

Статическая и динамическая схемы включают определенный набор сегментов. Содержимое статической заранее записывается в исполняемый объектный файл во время компиляции исходного кода. А динамическая схема формируется инструкциями процесса, которые выделяют память для переменных и массивов и изменяют ее в соответствии с логикой программы.

Учитывая все вышесказанное, содержимое статической памяти можно предсказать по исходному коду или скомпилированному объектному файлу. Но с динамической схемой все сложнее, поскольку ее можно узнать, только запустив программу. Кроме того, при каждом выполнении одной и той же программы содержимое динамической памяти может меняться. Иными словами, динамическое содержимое любого процесса уникально, и исследовать его нужно, пока он работает.

Начнем с рассмотрения статической схемы размещения в памяти.

Исследование статической схемы размещения в памяти

Инструменты, которые используются для исследования статической памяти, обычно рассчитаны на объектные файлы. Для начала рассмотрим пример 4.1; это минимальная программа на языке C без какой-либо логики и без переменных (листинг 4.1).

Листинг 4.1. Минимальная программа на C (ExtremeC_examples_chapter4_1.c)

```
int main(int argc, char** argv) {
  return 0;
}
```

Сначала нам нужно скомпилировать эту программу. В Linux мы используем `gcc` (терминал 4.1).

Терминал 4.1. Компиляция примера 4.1 в Linux с помощью gcc

```
$ gcc ExtremeC_examples_chapter4_1.c -o ex4_1-linux.out
$
```

После успешной компиляции и компоновки мы получили итоговый исполняемый объектный файл с именем `ex4_1-linux.out`. Он содержит предустановленную статическую схему размещения в памяти, принятую в операционной системе Linux; эта схема будет частью всех процессов, основанных на данном исполняемом файле.

Первым инструментом, с которым вы познакомитесь, будет команда `size`. С ее помощью можно вывести статическую схему размещения в памяти исполняемого объектного файла.

В терминале 4.2 показано, как команда `size` позволяет просмотреть различные сегменты, из которых состоит статическая память.

Терминал 4.2. Использование команды size для просмотра статических сегментов файла ex4_1-linux.out

```
$ size ex4_1-linux.out
   text    data     bss     dec     hex    filename
   1099     544       8    1651     673    ex4_1-linux.out
$
```

Как видите, в состав статической схемы размещения входят сегменты Text, Data и BSS. Их размеры приводятся в байтах.

Теперь скомпилируем тот же код из примера 4.1, но в другой операционной системе. Мы выбрали macOS и компилятор `clang` (терминал 4.3).

Терминал 4.3. Компиляция примера 4.1 с помощью clang в macOS

```
$ clang ExtremeC_examples_chapter4_1.c -o ex4_1-macos.out
$
```

Операционная система macOS, как и Linux, совместима с POSIX, поэтому тоже должна содержать утилиту `size` (поскольку та входит в состав POSIX). Таким образом, чтобы просмотреть статические сегменты файла `ex4_1-macos.out`, можно задействовать ту же команду (терминал 4.4).

Терминал 4.4. Использование команды size для просмотра статических сегментов файла ex4_1-macos.out

```
$ size ex4_1-macos.out
__TEXT __DATA  __OBJC  others       dec          hex
4096   0       0       4294971392   4294975488   100002000
$ size -m ex4_1-macos.out
Segment __PAGEZERO: 4294967296
Segment __TEXT: 4096
    Section __text: 22
    Section __unwind_info: 72
    total 94
Segment __LINKEDIT: 4096
total 4294975488
$
```

В этом терминале мы дважды воспользовались утилитой `size`; второй ее запуск дал нам больше информации о найденных сегментах памяти. Вы могли заметить, что в macOS, как и в Linux, присутствуют сегменты Text и Data, однако нет BSS. Дело в том, что в macOS он тоже существует, просто команда `size` его не показывает. Сегмент BSS содержит неинициализированные глобальные переменные, поэтому нет никакой необходимости выделять для него место в объектном файле — достаточно лишь знать, сколько байтов он занимает.

В приведенных выше терминалах есть интересная деталь. В Linux размер сегмента Text равен 1099 байтам, а в macOS — 4 Кбайт. Вдобавок можно заметить, что сегмент Data для минимальной программы на C в Linux занимает некое место, хотя в macOS он пустой. Очевидно, что на разных платформах низкоуровневые аспекты работы с памятью реализованы по-разному.

Несмотря на эту небольшую разницу между Linux и macOS, мы видим, что сегменты Text, Data и BSS содержат в статической схеме размещения. Далее я последовательно объясню, для чего используется каждый из этих сегментов. В следующих разделах мы обсудим их по отдельности и увидим, как на них влияют малейшие изменения в коде.

Сегмент BSS

Начнем с сегмента BSS. Его название расшифровывается как *block started by symbol* (блок, начинающийся с символа). С давних пор так обозначают области памяти, зарезервированные для неинициализированных машинных слов. Сегмент BSS фактически предназначен для хранения либо неинициализированных, либо обнуленных глобальных переменных.

Расширим пример 4.1, добавив в него несколько неинициализированных глобальных переменных, и посмотрим, как это повлияет на сегмент BSS. В листинге 4.2 показан пример 4.2.

Листинг 4.2. Минимальная программа на C с несколькими неинициализированными и обнуленными глобальными переменными (ExtremeC_examples_chapter4_2.c)

```
int global_var1;
int global_var2;
int global_var3 = 0;

int main(int argc, char** argv) {
  return 0;
}
```

Переменные `global_var1` и `global_var2` — целочисленные глобальные переменные, не прошедшие инициализацию. Снова воспользуемся командой `size`, чтобы узнать, как изменился итоговый исполняемый объектный файл в Linux по сравнению с примером 4.1 (терминал 4.5).

Терминал 4.5. Использование команды size для просмотра статических сегментов файла ex4_2-linux.out

```
$ gcc ExtremeC_examples_chapter4_2.c -o ex4_2-linux.out
$ size ex4_2-linux.out
   text    data     bss     dec     hex    filename
   1099     544      16    1659    67b    ex4_2-linux.out
$
```

Если сравнить этот результат с аналогичным выводом в примере 4.1, то можно заметить изменение размера сегмента BSS. То есть объявление *не*инициализированных или обнуленных глобальных переменных увеличивает сегмент BSS. Эти особые глобальные переменные — часть статической схемы размещения, и место для них выделяется во время загрузки процесса; кроме того, они не покидают память, пока процесс не завершит работу. Иными словами, имеют статический жизненный цикл.

С архитектурной точки зрения в алгоритмах обычно лучше использовать локальные переменные. Слишком большое количество глобальных переменных может увеличить размер двоичного файла. Кроме того, хранение деликатных данных в глобальной области видимости может отрицательно сказаться на безопасности. Проблемы с конкурентностью, особенно такие, как состояние гонки, а также загрязнение пространства имен и отсутствие информации о владельце, — это лишь некоторые осложнения, которые может вызвать применение глобальных переменных.

Скомпилируем пример 4.2 в macOS и посмотрим на вывод команды `size` (терминал 4.6).

Терминал 4.6. Использование команды size для просмотра статических сегментов файла ex4_2-macos.out

```
$ clang ExtremeC_examples_chapter4_2.c -o ex4_2-macos.out
$ size ex4_2-macos.out
__TEXT __DATA  __OBJC  others       dec          hex
4096   4096    0       4294971392   4294979584   100003000
$ size -m ex4_2-macos.out
Segment __PAGEZERO: 4294967296
Segment __TEXT: 4096
    Section __text: 22
    Section __unwind_info: 72
    total 94
Segment __DATA: 4096
    Section __common: 12
    total 12
Segment __LINKEDIT: 4096
total 4294979584
$
```

И снова результат отличается от того, который мы видели в Linux. В этой ОС при отсутствии глобальных переменных в сегменте BSS было выделено 8 байт. В примере 4.2 мы добавили три неинициализированные глобальные переменные общим размером 12 байт, и после этого компилятор C в Linux расширил сегмент BSS еще на 8 байт. В macOS сегмент BSS отсутствует в выводе команды `size`, но размер сегмента `data` был увеличен с 0 до 4 байт (что в macOS является стандартным размером страницы памяти). Это значит, что компилятор `clang` выделил для `data` в схеме размещения новую страницу памяти. Опять-таки это просто свидетельствует о том, насколько разнятся низкоуровневые аспекты схемы размещения в памяти на разных платформах.

Неважно, сколько байтов памяти нужно выделить программе. Аллокатор считает память в страницах; программа получает то количество страниц, которое покрывает ее нужды. Подробности об аллокаторе памяти в Linux можно найти по ссылке https://www.kernel.org/doc/gorman/html/understand/understand009.html.

В терминале 4.6 внутри сегмента `_DATA` есть секция `__common` размером 12 байт; на самом деле так обозначается сегмент BSS, которого нет в выводе `size`. Эта секция содержит три неинициализированные глобальные целочисленные переменные общим размером 12 байт (по 4 байта каждая). Стоит также отметить, что неинициализированные глобальные переменные по умолчанию *обнуляются*. Лучшего значения, чем 0, для них и не придумаешь.

Теперь поговорим о следующем сегменте в статической схеме размещения — Data.

Сегмент Data

Чтобы продемонстрировать, какого рода содержимое находится в этом сегменте, объявим больше глобальных переменных, однако на сей раз инициализируем их с помощью ненулевых значений. В листинге 4.3, основанном на примере 4.2, добавлены две новые инициализированные глобальные переменные.

Листинг 4.3. Минимальная программа на C с инициализированными и неинициализированными глобальными переменными (ExtremeC_examples_chapter4_3.c)

```
int global_var1;
int global_var2;
int global_var3 = 0;

double global_var4 = 4.5;
char global_var5 = 'A';

int main(int argc, char** argv) {
  return 0;
}
```

В терминале 4.7 показан вывод утилиты `size` для примера 4.3, полученный в Linux.

Терминал 4.7. Использование команды size для просмотра статических сегментов файла ex4_3-linux.out

```
$ gcc ExtremeC_examples_chapter4_3.c -o ex4_3-linux.out
$ size ex4_3-linux.out
   text    data     bss     dec     hex    filename
   1099     553      20    1672     688    ex4_3-linux.out
$
```

Мы уже знаем, что сегмент Data предназначен для хранения инициализированных глобальных переменных с ненулевыми значениями. Если сравнить вывод команды `size` в примерах 4.2 и 4.3, то в глаза сразу бросается то, что сегмент Data увеличился на 9 байт; это общий размер двух добавленных нами переменных (восьмибайтной типа `double` и однобайтной типа `char`).

Взглянем на изменения в macOS (терминал 4.8).

Терминал 4.8. Использование команды size для просмотра статических сегментов файла ex4_3-macos.out

```
$ clang ExtremeC_examples_chapter4_3.c -o ex4_3-macos.out
$ size ex4_3-macos.out
__TEXT __DATA  __OBJC  others       dec          hex
4096   4096       0    4294971392   4294979584   100003000
$ size -m ex4_3-macos.out
Segment __PAGEZERO: 4294967296
Segment __TEXT: 4096
    Section __text: 22
    Section __unwind_info: 72
    total 94
Segment __DATA: 4096
    Section __data: 9
    Section __common: 12
    total 21
Segment __LINKEDIT: 4096
total 4294979584
$
```

При первом запуске не видно никаких изменений, поскольку суммарный размер всех глобальных переменных все еще намного меньше 4 Кбайт. Но вслед за этим в сегменте `_DATA` появилась новая секция, `__data`. В памяти для нее было выделено 9 байт, что соответствует размеру инициализированных глобальных переменных, которые мы добавили. А размер неинициализированных глобальных переменных, как в примере 4.2 и в macOS, по-прежнему равен 12 байтам.

Следует обратить внимание и на то, что команда `size` показывает только размер сегментов, но не их содержимое. Чтобы просмотреть то, что хранится внутри сегментов объектного файла, каждая операционная система предоставляет свои

инструменты. Например, в Linux содержимое ELF-файла можно проанализировать с помощью команд `readelf` и `objdump`, которые также позволяют исследовать статическую схему размещения внутри объектных файлов. С кое-какими из этих инструментов мы познакомились в предыдущих двух главах.

Помимо глобальных, есть также статические переменные, объявленные внутри функций. При многократном вызове одной и той же функции эти переменные не меняют свои значения. В зависимости от платформы и наличия значения они могут храниться как в сегменте Data, так и в BSS. В листинге 4.4 показано, как объявить статические переменные внутри функции.

Листинг 4.4. Объявление двух статических переменных: инициализированной и неинициализированной

```
void func() {
  static int i;
  static int j = 1;
  ...
}
```

Как видите, переменные `i` и `j` — статические. Первая не инициализирована, а вторая имеет значение `1`. Неважно, сколько раз будет выполнена функция `func`, — эти переменные всегда будут хранить последние присвоенные им значения.

Поговорим о том, как все работает. На этапе выполнения эти переменные находятся либо в сегменте Data, либо в BSS (которые обладают статическим жизненным циклом), и функция `func` имеет к ним доступ. Поэтому их, в сущности, и называют *статическими*. Мы знаем, что переменная `j` хранится в сегменте Data, просто потому, что имеет начальное значение; в то же время переменная `i` не инициализирована, поэтому должна находиться в сегменте BSS.

Теперь познакомимся со второй командой для исследования содержимого сегмента BSS в Linux, `objdump`. С ее помощью можно выводить сегменты памяти объектных файлов. В macOS аналогичная команда называется `gobjdump`, но там ее нужно самостоятельно установить.

Попробуем проанализировать итоговый исполняемый файл, чтобы найти глобальные переменные, которые записываются в сегмент Data. Код примера 4.4 показан в листинге 4.5.

Листинг 4.5. Инициализированные глобальные переменные, которые должны быть записаны в сегмент Data (ExtremeC_examples_chapter4_4.c)

```
int     x = 33;           // 0x00000021
int     y = 0x12153467;
char z[6] = "ABCDE";

int main(int argc, char**argv) {
  return 0;
}
```

В приведенном выше коде нет ничего сложного. Он просто объявляет три глобальные переменные с некими начальными значениями. После компиляции нам нужно вывести содержимое сегмента Data, чтобы найти значения, которые в него записываются.

В терминале 4.9 показано, как скомпилировать исходный код и просмотреть сегмент Data с помощью команды `objdump`.

Терминал 4.9. Использование команды objdump для просмотра содержимого сегмента Data

```
$ gcc ExtremeC_examples_chapter4_4.c -o ex4_4.out
$ objdump -s -j .data ex4_4.out

a.out:     file format elf64-x86-64

Contents of section .data:
 601020 00000000 00000000 00000000 00000000  ................
 601030 21000000 67341512 41424344 4500      !....4..ABCDE.
$
```

Объясню, как следует читать этот вывод, особенно находящееся в секции `.data`. В первом столбце слева — адреса. В следующих четырех столбцах хранятся сами данные; как видите, каждый столбец занимает 4 байта. Поэтому размер всей строки равен 16 байтам. Последний столбец, расположенный справа, — представление байтов, содержащихся в предыдущих четырех столбцах, в кодировке ASCII. Точка означает, что символ нельзя показать в алфавитно-цифровом виде. Обратите внимание на параметры `-s` и `-j .data`: первый сообщает команде `objdump` о том, что нужно вывести все содержимое заданной секции, а второй указывает секцию `.data`.

Первая строчка занимает 16 байт и состоит из нулей. В ней нет никакой переменной, поэтому она нам неинтересна. Во второй строчке показано содержимое сегмента Data, начиная с адреса `0x601030`. Первые 4 байта — это значение переменной `x` в примере 4.4. Следующие 4 байта содержат значение переменной `y`. Последние 6 байт выделены для символов внутри массива `z`. Его содержимое отчетливо видно в последнем столбце.

Если внимательно посмотреть на терминал 4.9, то можно заметить, что шестнадцатеричное представление десятеричного числа 33, `0x00000021`, хранится в сегменте как `0x21000000`. То же самое относится к содержимому переменной `y`: мы записывали ее как `0x12153467`, но после сохранения она имеет вид `0x67341512`. Все выглядит так, будто порядок следования байтов поменялся на противоположный.

Это явление можно объяснить тем фактом, что в целом порядок следования байтов бывает двух типов: *от старшего к младшему* и *от младшего к старшему*. Если взять число `0x12153467`, то в первом случае оно не изменится, поскольку старший байт,

`0x12`, идет первым. Во втором случае это число будет выглядеть как `0x67341512`, поскольку первым идет младший байт, `0x67`.

Независимо от представления, в языке C мы всегда читаем корректное значение. Порядок следования байтов — характеристика центрального процессора, поэтому в разных архитектурах байты в итоговых объектных файлах могут размещаться в разном порядке. Это одна из причин, почему исполняемый файл нельзя запустить на платформе с другим порядком следования байтов.

Интересно, как этот вывод будет выглядеть в macOS? В терминале 4.10 показано, как просмотреть содержимое сегмента Data с помощью команды `gobjdump`.

Терминал 4.10. Использование команды gobjdump в macOS для просмотра содержимого сегмента Data

```
$ gcc ExtremeC_examples_chapter4_4.c -o ex4_4.out
$ gobjdump -s -j .data ex4_4.out

a.out:      file format mach-o-x86-64

Contents of section .data:
 100001000 21000000 67341512 41424344 4500      !...g4..ABCDE.
$
```

Здесь все следует читать точно так же, как и в Linux (см. вывод в терминале 4.9). Как видите, в macOS сегмент Data не содержит 16-байтных нулевых заголовков. Двоичный файл, очевидно, был скомпилирован для процессора с порядком следования байтов от младшего к старшему.

Напоследок нужно сказать, что для исследования содержимого объектных файлов можно использовать и другие инструменты — например, `readelf` в Linux и `dwarfdump` в macOS. Кроме того, двоичное содержимое объектного файла можно читать с помощью таких утилит, как `hexdump`.

В следующем подразделе мы обсудим сегмент Text и узнаем, как его просматривать с использованием `objdump`.

Сегмент Text

Как мы уже знаем из главы 2, в итоговый исполняемый файл записываются инструкции машинного уровня. Поскольку все машинные инструкции программы находятся в сегменте Text (или Code), он должен находиться в исполняемом объектном файле — а именно, в его статической схеме размещения. Процессор извлекает эти инструкции и выполняет их во время работы процесса.

Заглянем в сегмент Text реального исполняемого файла. В листинге 4.6 показан пример 4.5, который представляет собой всего лишь пустую функцию `main`.

Листинг 4.6. Минимальная программа на C (ExtremeC_examples_chapter4_5.c)

```
int main(int argc, char** argv) {
  return 0;
}
```

Здесь мы видим различные части итоговой исполняемой программы. Стоит отметить, что команда `objdump` доступна только в Linux; в других операционных системах для этого предусмотрены другие инструменты.

В терминале 4.11 демонстрируется использование команды `objdump` для извлечения содержимого разных секций, присутствующих в исполняемом объектном файле из примера 4.5. Обратите внимание: в следующем выводе показаны только те ассемблерные инструкции, которые относятся к функции `main`.

Терминал 4.11. Использование objdump для вывода содержимого секции, относящейся к функции main

```
$ gcc ExtremeC_examples_chapter4_5.c -o ex4_5.out
$ objdump -S ex4_5.out

ex4_5.out:     file format elf64-x86-64
Disassembly of section .init:

0000000000400390 <_init>:
... truncated.
.
.
Disassembly of section .plt:

00000000004003b0 <__libc_start_main@plt-0x10>:
... truncated

00000000004004d6 <main>:
  4004d6:   55                      push   %rbp
  4004d7:   48 89 e5                mov    %rsp,%rbp
  4004da:   b8 00 00 00 00          mov    $0x0,%eax
  4004df:   5d                      pop    %rbp
  4004e0:   c3                      retq
  4004e1:   66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
  4004e8:   00 00 00
  4004eb:   0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)

00000000004004f0 <__libc_csu_init>:
... truncated
.
.
.
0000000000400564 <_fini>:
... truncated
$
```

Здесь показано несколько секций с машинными инструкциями, включая `.text`, `.init` и `.plt`. Все вместе они делают возможными загрузку и выполнение программы. Каждая из этих секций — часть сегмента Text, который входит в статическую схему размещения в исполняемом объектном файле.

Наша программа в примере 4.5 содержит всего одну функцию, `main`, но в итоговом исполняемом файле мы видим десяток других функций.

В выводе, представленном в терминале 4.11, видно, что перед вызовом функции `main` выполняется другая логика. Как уже объяснялось в главе 2, в Linux эти функции обычно заимствуются из библиотеки `glibc` и используются для формирования готовой программы.

В следующем разделе мы начнем исследовать динамическую схему размещения процесса в памяти.

Исследование динамической схемы размещения в памяти

Динамическая схема размещения находится в памяти процесса и существует до тех пор, пока он не завершится. Процедурой запуска исполняемого объектного файла занимается программа под названием *«загрузчик»*. Он создает новый процесс и его начальную схему размещения в памяти, которая должна быть динамической. Для этого копируются сегменты, найденные в статической схеме размещения исполняемого объектного файла. Затем к ним добавляется два новых сегмента. И только после этого процесс может приступить к выполнению.

Если коротко, то в памяти активного процесса должно быть пять сегментов, три из которых копируются непосредственно из статической схемы размещения исполняемого объектного файла, а остальные два создаются с нуля и называются стеком и кучей. Последние являются динамическими сегментами и существуют только в период работы процесса. Это значит, вы не найдете никакого упоминания о них в исполняемом объектном файле.

В этом разделе наша главная задача — исследовать стек и кучу, а также познакомиться с инструментами и участками операционной системы, которые могут быть здесь полезными. Время от времени мы можем называть эти сегменты динамической схемой размещения в памяти процесса, игнорируя три других сегмента, которые копируются из объектного файла; в таком случае следует помнить, что динамическая память процесса состоит из всех пяти сегментов.

Стек — область памяти, в которой по умолчанию выделяется место для переменных. Она имеет ограниченный размер, поэтому большие объекты в ней хранить

нельзя. Для сравнения, куча — более крупная и гибкая область памяти, в которой могут поместиться большие объекты и массивы. Для работы с кучей нужен отдельный API, с которым мы познакомимся чуть позже.

Как вы помните, динамическая схема размещения — не то же самое, что *динамическое выделение памяти*. Не следует путать эти два понятия, поскольку они имеют разный смысл! Позже мы более подробно обсудим разные методы выделения памяти и отдельно остановимся на динамическом.

Пять сегментов, из которых состоит динамическая память, ссылаются на разные участки основной памяти, уже *выделенные* и доступные *только* соответствующему процессу. Все эти сегменты — динамические в том смысле, что во время выполнения их содержимое постоянно меняется; исключение составляет только сегмент Text, который является статическим и неизменяемым в буквальном смысле слова.

Исследование динамической памяти процесса требует отдельной процедуры. Прежде чем приступать, нужно запустить процесс: написать пример, который будет работать достаточно долго для того, чтобы мы могли получить доступ к его динамической памяти, используя специальные инструменты.

В следующем подразделе мы рассмотрим пример того, как исследовать структуру динамической памяти.

Отражение памяти

Начнем с простого примера. Код, представленный ниже, выполняется бесконечно долго. Таким образом, мы получим процесс, который никогда не завершается, что позволит нам изучить структуру его памяти. Конечно, закончив исследование, мы сможем от него избавиться с помощью команды `kill`. В листинге 4.7 показан код примера 4.6.

Листинг 4.7. Пример 4.6, который мы используем для исследования схемы динамической памяти (ExtremeC_examples_chapter4_6.c)

```
#include <unistd.h> // Needed for sleep function

int main(int argc, char** argv) {
  // Бесконечный цикл
  while (1) {
    sleep(1); // Засыпаем на 1 секунду
  };
  return 0;
}
```

Как видите, это обычный бесконечный цикл, благодаря которому процесс сможет работать сколь угодно долго. Таким образом, у нас будет время изучить его память. Сначала соберем данный код.

Заголовок unistd.h доступен только в Unix-подобных (или, если быть точным, в POSIX-совместимых) операционных системах. Это значит, в системе Microsoft Windows, которая несовместима с POSIX, вместо этого можно подключить заголовок windows.h.

В терминале 4.12 показано, как скомпилировать этот пример в Linux.

Терминал 4.12. Компиляция примера 4.6 в Linux

```
$ gcc ExtremeC_examples_chapter4_6.c -o ex4_6.out
$
```

Теперь запустим его. Чтобы командную строку можно было использовать в дальнейшем, запуск процесса выполняется в фоновом режиме (терминал 4.13).

Терминал 4.13. Запуск примера 4.6 в фоновом режиме

```
$ ./ ex4_6.out &
[1] 402
$
```

Итак, процесс выполняется в фоне. Если верить полученному выводу, его PID равен 402. В будущем мы воспользуемся этим значением в целях принудительного завершения программы. PID меняется при каждом запуске, поэтому у себя на компьютере вы, скорее всего, увидите другой номер. Обратите внимание: командная строка снова становится доступной сразу после фонового запуска процесса, и в ней можно будет выполнять дальнейшие команды.

Имея PID (process ID — идентификатор процесса), вы можете легко завершить процесс, используя команду kill. Например, если значение PID равно 402, то следующая команда будет работать в любой операционной системе семейства Unix: kill -9 402.

PID — идентификатор, который мы используем для исследования памяти процесса. Обычно система предоставляет собственные механизмы получения различных свойств активных процессов на основе их PID. Но здесь нас интересуют только подробности о динамической памяти, поэтому мы применим соответствующие инструменты, доступные в Linux.

На компьютере под управлением Linux сведения о процессе можно найти в файлах внутри каталога /proc. Он находится в специальной файловой системе под названием *procfs*, которая отличается от обычных ФС тем, что не предназначена собственно для хранения данных; это скорее иерархический интерфейс для получения информации о разных свойствах отдельных процессов или системы в целом.

Файловая система procfs есть не только в Linux. Она обычно является частью Unix-подобных операционных систем, хотя используют ее не все они. Например, в FreeBSD она применяется, а в macOS — нет.

Теперь с помощью procfs просмотрим структуру памяти активного процесса. Память процесса состоит из ряда *отражений*. Каждое из них представляет отдельную область памяти, отраженную в специальный файл или сегмент, являющийся частью процесса. Вскоре вы увидите, что у стека и кучи процесса есть собственные отражения.

Один из способов применения procfs — наблюдение за текущими отражениями процесса. Посмотрим, как это делается.

Мы знаем, что наш процесс имеет PID 402. С помощью команды `ls` можно вывести содержимое каталога /proc/402, как показано в терминале 4.14.

Терминал 4.14. Вывод содержимого /proc/402

```
$ ls -l /proc/402
total of 0
dr-xr-xr-x  2 root root 0 Jul 15 22:28 attr
-rw-r--r--  1 root root 0 Jul 15 22:28 autogroup
-r--------  1 root root 0 Jul 15 22:28 auxv
-r--r--r--  1 root root 0 Jul 15 22:28 cgroup
--w-------  1 root root 0 Jul 15 22:28 clear_refs
-r--r--r--  1 root root 0 Jul 15 22:28 cmdline
-rw-r--r--  1 root root 0 Jul 15 22:28 comm
-rw-r--r--  1 root root 0 Jul 15 22:28 coredump_filter
-r--r--r--  1 root root 0 Jul 15 22:28 cpuset
lrwxrwxrwx  1 root root 0 Jul 15 22:28 cwd -> /root/codes
-r--------  1 root root 0 Jul 15 22:28 environ
lrwxrwxrwx  1 root root 0 Jul 15 22:28 exe -> /root/codes/a.out
dr-x------  2 root root 0 Jul 15 22:28 fd
dr-x------  2 root root 0 Jul 15 22:28 fdinfo
-rw-r--r--  1 root root 0 Jul 15 22:28 gid_map
-r--------  1 root root 0 Jul 15 22:28 io
-r--r--r--  1 root root 0 Jul 15 22:28 limits
...
$
```

Как видите, внутри /proc/402 есть много файлов и каталогов, и все они соответствуют определенному свойству процесса. Чтобы получить список отражений, нужно

просмотреть содержимое файла `maps` в каталоге /proc/402. Выведем файл `/proc/402/maps` с помощью команды `cat` (терминал 4.15).

Терминал 4.15. Вывод содержимого /proc/402/maps

```
$ cat /proc/402/maps
00400000-00401000 r-xp 00000000 08:01 790655            .../
extreme_c/4.6/ex4_6.out
00600000-00601000 r--p 00000000 08:01 790655            .../
extreme_c/4.6/ex4_6.out
00601000-00602000 rw-p 00001000 08:01 790655            .../
extreme_c/4.6/ex4_6.out
7f4ee16cb000-7f4ee188a000 r-xp 00000000 08:01 787362    /lib/
x86_64-linux-gnu/libc-2.23.so
7f4ee188a000-7f4ee1a8a000 ---p 001bf000 08:01 787362    /lib/
x86_64-linux-gnu/libc-2.23.so
7f4ee1a8a000-7f4ee1a8e000 r--p 001bf000 08:01 787362    /lib/
x86_64-linux-gnu/libc-2.23.so
7f4ee1a8e000-7f4ee1a90000 rw-p 001c3000 08:01 787362    /lib/
x86_64-linux-gnu/libc-2.23.so
7f4ee1a90000-7f4ee1a94000 rw-p 00000000 00:00 0
7f4ee1a94000-7f4ee1aba000 r-xp 00000000 08:01 787342    /lib/
x86_64-linux-gnu/ld-2.23.so
7f4ee1cab000-7f4ee1cae000 rw-p 00000000 00:00 0
7f4ee1cb7000-7f4ee1cb9000 rw-p 00000000 00:00 0
7f4ee1cb9000-7f4ee1cba000 r--p 00025000 08:01 787342    /lib/
x86_64-linux-gnu/ld-2.23.so
7f4ee1cba000-7f4ee1cbb000 rw-p 00026000 08:01 787342    /lib/
x86_64-linux-gnu/ld-2.23.so
7f4ee1cbb000-7f4ee1cbc000 rw-p 00000000 00:00 0
7ffe94296000-7ffe942b7000 rw-p 00000000 00:00 0         [stack]
7ffe943a0000-7ffe943a2000 r--p 00000000 00:00 0         [vvar]
7ffe943a2000-7ffe943a4000 r-xp 00000000 00:00 0         [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0
[vsyscall]
$
```

Как видите, результат состоит из множества строк. Каждая из них представляет отражение памяти с диапазоном выделенных адресов (областью); она привязана к определенному файлу или сегменту в динамической схеме размещения процесса. У каждого отражения есть поля, разделенные одним или несколькими пробелами. Их описание приводится ниже (слева направо).

- *Диапазон адресов* — начальный и конечный адреса отраженного диапазона. Если данная область отражена в файл, то путь к нему указан рядом. Это элегантный способ отражения одного и того же загруженного разделяемого объектного файла в разных процессах. Я уже упоминал об этом в главе 3.
- *Разрешения* — этот столбец определяет, можно ли выполнять (`x`), читать (`r`) или изменять (`w`) содержимое. Область памяти также можно разделять (`s`)

с другими процессами или изолировать (p) ее в рамках процесса, которому она принадлежит.

- *Сдвиг* — если область отражена в файл, то это сдвиг относительно начала данного файла. Если не отражена, то данный столбец обычно содержит 0.
- *Устройство* — если область отражена в файл, то здесь указан номер устройства (в формате m:n), которое хранит данный файл. Это, к примеру, может быть номер жесткого диска, на котором находится разделяемый объектный файл.
- *inode* — файл, в который отражена область памяти, должен находиться в файловой системе и иметь в ней свой номер inode. Последний указывается в данном столбце. Файл *inode* — абстракция в файловых системах наподобие *ext4*, которые в основном используются в ОС семейства Unix. Номер inode может относиться как к файлам, так и к каталогам, и его используют для доступа к их содержимому.
- *Путь или описание* — если область отражена в файл, то здесь указывается путь к нему. В противном случае данный столбец может быть пустым или описывать назначение области. Например, метка `[stack]` говорит о том, что это стек.

Файл `maps` содержит еще больше полезной информации о динамической схеме размещения памяти процесса. Чтобы продемонстрировать это должным образом, обратимся к новому примеру.

Стек

Сначала поговорим о таком сегменте, как стек. Это ключевая часть динамической памяти любого процесса, предусмотренная почти во всех существующих архитектурах. В отражениях памяти она помечена как `[stack]`.

Стек и куча хранят динамические данные, которые постоянно меняются во время выполнения программы. Просмотреть динамическое содержимое этих сегментов не слишком просто; в большинстве случаев для этого необходим отладчик наподобие `gdb`, который читает байты памяти активного процесса.

Как уже отмечалось ранее, стек обычно имеет ограниченный размер и плохо подходит для хранения крупных объектов. Если в этом сегменте закончится свободное место, то процесс больше не сможет вызывать функции, поскольку стек активно используется механизмом вызова. В таких случаях процесс принудительно завершается операционной системой. *Переполнение стека* — широко известная ошибка, которая возникает при полном заполнении стека. Механизм вызова функций будет рассмотрен чуть ниже.

Как уже объяснялось, стек — область памяти, в которой по умолчанию выделяется место для переменных. Представьте, что вы объявили переменную внутри функции, как показано в листинге 4.8.

Листинг 4.8. Объявление локальной переменной, память для которой выделяется в стеке

```
void func() {
  // память, необходимая для следующей переменной, выделяется в стеке
  int a;
  ...
}
```

В данной функции при объявлении переменной не указано ничего, что позволило бы компилятору понять, в каком сегменте следует выделять память. В связи с этим компилятор использует стек по умолчанию. Это место, с которого начинается выделение.

Термин «стек» происходит от английского слова *stack* (стопка, штабель). Когда вы объявляете локальную переменную, она создается на вершине стека. При выходе из функции компилятор снимает со стека ее локальные переменные, в результате чего на вершину поднимаются значения внешней области видимости.

В абстрактном виде стек представляет собой структуру данных FILO (first in, last out — «первым пришел, последним вышел») или LIFO (last in, first out — «последним пришел, первым вышел»). Независимо от деталей реализации, каждая запись сохраняется на вершине стека, и постепенно на нее кладутся последующие записи. Запись нельзя достать, не убрав все остальные, которые находятся над ней.

В стеке хранятся не только переменные. При каждом вызове функции на стек кладется новая запись под названием *«стековый фрейм»*. Иначе вы бы не смогли вернуться в предыдущую функцию или вернуть результат вызывающему коду.

Наличие надежного механизма стека — неотъемлемая часть рабочей программы. Поскольку стек ограничен в размере, в нем рекомендуется объявлять небольшие переменные. Кроме того, не следует создавать слишком много стековых фреймов в результате бесконечной *рекурсии* или множества вызовов функций.

С другой стороны, стек — область памяти, в которой программист хранит данные и объявляет локальные переменные для своих алгоритмов. Операционная система, отвечающая за запуск программ, использует стек в целях размещения данных, необходимых ее внутренним механизмам, что является залогом успешного выполнения программы.

В этом смысле со стеком необходимо обращаться осторожно, поскольку его некорректное использование или повреждение его содержимого чревато прерыванием работы активного процесса или даже выводом его из строя. Для сравнения, с кучей имеет дело только программист. О ней мы поговорим в следующем подразделе.

Содержимое стека не так просто просматривать извне, используя только те инструменты, с которыми мы познакомились при исследовании статической схемы

размещения. Эта область памяти содержит приватные данные, которые могут требовать деликатного обращения. Кроме того, она доступна лишь одному процессу и другие программы не могут читать или изменять ее.

Таким образом, чтобы пройтись по стеку, нужно неким образом подключиться к процессу и взглянуть на память «его глазами». Для этого можно воспользоваться *отладчиком*. Он подключается к процессу, позволяя управлять им и исследовать содержимое его памяти. Мы будем применять данный подход для изучения стека в следующей главе. А пока закончим с обсуждением этого сегмента и перейдем к куче.

Куча

В примере 4.7 показано, как с помощью отражений памяти можно найти области, выделенные для сегмента кучи. Это довольно похоже на пример 4.6, но здесь, прежде чем входить в бесконечный цикл, мы выделяем в куче определенное количество байтов. Таким образом, мы снова пройдемся по отражениям памяти активного процесса и посмотрим, какие из них относятся к куче.

Код примера 4.7 показан в листинге 4.9.

Листинг 4.9. Пример 4.7 для исследования кучи (ExtremeC_examples_chapter4_7.c)

```
#include <unistd.h> // для функции sleep
#include <stdlib.h> // для функции malloc
#include <stdio.h> // для printf

int main(int argc, char** argv) {
  void* ptr = malloc(1024); // выделяем для кучи 1 Кбайт
  printf("Address: %p\n", ptr);
  fflush(stdout); // для принудительного вывода
  // бесконечный цикл
  while (1) {
    sleep(1); // засыпаем на 1 секунду
  };
  return 0;
}
```

В данном коде мы использовали функцию `malloc`. Это основной механизм выделения дополнительной памяти в сегменте кучи. Она принимает количество байтов, которые нужно выделить, и возвращает обобщенный указатель.

Напомню, что обобщенный (пустой) указатель содержит адрес памяти, но его нельзя *разыменовать* или использовать напрямую. Вначале его следует привести к определенному типу.

В примере 4.7 мы выделяем 1024 байта (или 1 Кбайт), выводим адрес указателя, полученный от `malloc`, и затем входим в цикл. Скомпилируем и запустим этот код (терминал 4.16).

Терминал 4.16. Компиляция и запуск примера 4.7

```
$ g++ ExtremeC_examples_chapter4_7.c -o ex4_7.out
$ ./ex4_7.out &
[1] 3451
Address: 0x19790010
$
```

Итак, запущенный в фоне процесс получил PID 3451. Теперь откроем его файл `maps` и посмотрим, какие области памяти у него отражаются (терминал 4.17).

Терминал 4.17. Вывод содержимого /proc/3451/maps

```
$ cat /proc/3451/maps
00400000-00401000 r-xp 00000000 00:2f 176521            .../
extreme_c/4.7/ex4_7.out
00600000-00601000 r--p 00000000 00:2f 176521            .../
extreme_c/4.7/ex4_7.out
00601000-00602000 rw-p 00001000 00:2f 176521            .../
extreme_c/4.7/ex4_7.out
01979000-0199a000 rw-p 00000000 00:00 0                 [heap]
7f7b32f12000-7f7b330d1000 r-xp 00000000 00:2f 30        /lib/
x86_64-linux-gnu/libc-2.23.so
7f7b330d1000-7f7b332d1000 ---p 001bf000 00:2f 30        /lib/
x86_64-linux-gnu/libc-2.23.so
7f7b332d1000-7f7b332d5000 r--p 001bf000 00:2f 30        /lib/
x86_64-linux-gnu/libc-2.23.so
7f7b332d5000-7f7b332d7000 rw-p 001c3000 00:2f 30        /lib/
x86_64-linux-gnu/libc-2.23.so
7f7b332d7000-7f7b332db000 rw-p 00000000 00:00 0
7f7b332db000-7f7b33301000 r-xp 00000000 00:2f 27        /lib/
x86_64-linux-gnu/ld-2.23.so
7f7b334f2000-7f7b334f5000 rw-p 00000000 00:00 0
7f7b334fe000-7f7b33500000 rw-p 00000000 00:00 0
7f7b33500000-7f7b33501000 r--p 00025000 00:2f 27        /lib/
x86_64-linux-gnu/ld-2.23.so
7f7b33501000-7f7b33502000 rw-p 00026000 00:2f 27        /lib/
x86_64-linux-gnu/ld-2.23.so
7f7b33502000-7f7b33503000 rw-p 00000000 00:00 0
7ffdd63c2000-7ffdd63e3000 rw-p 00000000 00:00 0         [stack]
7ffdd63e7000-7ffdd63ea000 r--p 00000000 00:00 0         [vvar]
7ffdd63ea000-7ffdd63ec000 r-xp 00000000 00:00 0         [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0
[vsyscall]
$
```

Если внимательно присмотреться к терминалу 4.17, то можно заметить новое отражение с меткой [heap], которое мы выделили. Эта область была добавлена ввиду использования функции malloc. Ее размер равен 0x21000 байт, или 132 Кбайт. Это значит, при выделении в коде 1 Кбайт процесс выделяет область размером 132 Кбайт.

Обычно это делается с целью предотвратить выделение памяти при дальнейшем использовании malloc. Дело в том, что выделение места в куче — затратная операция, которая расходует как память, так и время.

Возвращаясь к листингу 4.9, отмечу, что адрес, на который ссылается указатель ptr, тоже заслуживает внимания. Отражение кучи, показанное в терминале 4.17, занимает диапазон адресов с 0x01979000 по 0x0199a000, в который, очевидно, входит адрес 0x19790010, хранящийся в ptr и имеющий сдвиг размером 16 байт.

Размер 132 Кбайт — далеко не предел. Куча может достигать десятков гигабайтов. В ней обычно размещают постоянные, глобальные и очень большие объекты, такие как массивы и битовые потоки.

Как уже отмечалось, выделение и освобождение места в куче требует вызова специальных функций, которые предоставляются стандартом языка C. С локальными переменными, созданными на вершине стека, можно взаимодействовать напрямую, тогда как для доступа к содержимому кучи необходимо использовать указатели. Это одна из причин, почему понимание принципа работы указателей и умение ими пользоваться — ключевое требование для любого программиста на C. Рассмотрим пример 4.8, в котором показано, как с помощью указателей можно обратиться к сегменту кучи (листинг 4.10).

Листинг 4.10. Использование указателей для взаимодействия с кучей (ExtremeC_examples_chapter4_8.c)

```
#include <stdio.h>   // для функции printf
#include <stdlib.h>  // для функций malloc и free

void fill(char* ptr) {
  ptr[0] = 'H';
  ptr[1] = 'e';
  ptr[2] = 'l';
  ptr[3] = 'l';
  ptr[5] = 0;
}

int main(int argc, char** argv) {
  void* gptr = malloc(10 * sizeof(char));
  char* ptr = (char*)gptr;
  fill(ptr);
  printf("%s!\n", ptr);
  free(ptr);
  return 0;
}
```

Эта программа выделяет в куче 10 байт, используя функцию `malloc`. Та принимает количество байтов, которое нужно выделить, и возвращает обобщенный указатель на первый байт выделенного блока памяти.

Чтобы использовать возвращенный указатель, его нужно привести к подходящему типу. Поскольку мы выделяем память для хранения символов, указатель будет иметь тип `char`. Приведение типов выполняется с помощью функции `fill`.

Обратите внимание: переменные локальных указателей, `gptr` и `ptr`, выделяются в стеке. Им нужно место для хранения своих значений, и оно находится в стековой памяти. Но адреса, на которые они указывают, хранятся в куче. В этом нет ничего необычного. Локальные указатели выделяются в стеке, но области памяти, на которые они ссылаются, находятся внутри кучи. Более подробно об этом поговорим в главе 5.

Нужно сказать, что указатель `ptr` внутри функции `fill` тоже выделен в стеке, но находится в другой области видимости и отличается от одноименного указателя в функции `main`.

Когда речь идет о куче, за выделение памяти отвечает программа или, скорее, программист. Вдобавок программа должна сама освобождать память, которая ей больше не нужна. Наличие участка кучи, к которому *нельзя обратиться*, называется *утечкой памяти*. Это значит, у нас нет указателя для работы с данной областью.

Утечки памяти могут быть губительными для программы; если они постепенно накапливаются, то в конечном счете израсходуется вся доступная память, в результате чего процесс будет принудительно завершен. Именно поэтому наша программа вызывает функцию `free`, прежде чем вернуться в `main`. Данный вызов освободит выделенный блок кучи, после чего программа больше не должна использовать его адреса.

Более подробно о стеке и куче мы поговорим в следующей главе.

Резюме

В данной главе я в первую очередь пытался сделать краткий обзор структуры памяти процессов в Unix-подобных операционных системах. Было представлено много материала, поэтому напомню, какие темы мы обсудили:

- рассмотрели структуру динамической памяти активного процесса и статической памяти исполняемого объектного файла;
- увидели, что статическая схема размещения в памяти находится в исполняемом объектном файле и разбита на части, которые называются сегментами. Вы узнали, что в состав статической схемы размещения входят такие сегменты, как Text, Data и BSS;

- увидели, что сегмент Text (Code) используется для хранения инструкций машинного уровня, предназначенных для выполнения, когда на основе текущего исполняемого файла создается новый процесс;
- узнали, что сегмент BSS используется для хранения глобальных переменных, которые либо равны нолю, либо не инициализированы;
- выяснили, что сегмент Data используется для хранения инициализированных глобальных переменных;
- задействовали команды `size` и `objdump` в целях исследования внутренностей объектных файлов. Для поиска внутри объектного файла вышеупомянутых сегментов можно использовать инструменты наподобие `readelf`;
- исследовали динамическую схему размещения памяти процесса. Вы увидели, что в нее копируются все статические сегменты. Вместе с тем в динамической памяти появляется два новых сегмента: стек и куча;
- выяснили, что стек — область памяти, в которой по умолчанию выделяется место для переменных;
- узнали, что локальные переменные всегда создаются на вершине стека;
- выяснили, что механизм вызова функций основан на стеке и его принципе работы;
- увидели, что для выделения и освобождения места в сегментах кучи необходимо использовать специальный API (набор функций), входящий в стандартную библиотеку языка C;
- обсудили утечки памяти и показали, как они могут происходить в контексте сегмента кучи.

Следующая глава полностью посвящена стеку и куче. В ней будет дополнен базовый материал, который мы рассмотрели в этой главе. Вы увидите больше примеров и познакомитесь с новыми инструментами для исследования процессов; на этом мы завершим наше обсуждение работы с памятью в C.

5 Стек и куча

В предыдущей главе мы провели исследование структуры памяти активного процесса. Системное программирование без понимания устройства памяти и ее различных сегментов подобно проведению хирургической операции без знания анатомии человеческого тела. Мы познакомились с основными сведениями о сегментах памяти процесса, но в этой главе речь пойдет только о двух из них, которые используются чаще всего: о стеке и куче.

Куча и стек — основные сегменты, с которыми работает программист. Data, Text и BSS используются реже, и доступ к ним ограничен. Причиной тому факт, что данные сегменты генерируются компилятором и зачастую занимают небольшую долю в общем объеме памяти запущенного процесса. Это не значит, что они неважны; на самом деле они имеют прямое отношение к некоторым потенциальным проблемам. Но поскольку большую часть времени вы будете работать со стеком и кучей, именно в них будет возникать большинство неполадок.

В этой главе вы изучите:

- способы исследования стека и нужные для этого инструменты;
- устройство автоматического управления памятью в стеке;
- различные характеристики стека;
- рекомендации по использованию стека;
- приемы исследования кучи;
- способы выделения и освобождения блоков памяти в куче;
- рекомендации по использованию кучи;
- среды с ограниченными ресурсами и тонкую настройку памяти в высокопроизводительных средах.

Начнем наше исследование с подробного обсуждения стека.

Стек

Процесс может продолжать выполнение и без кучи, но без стека работать не будет. Это о многом говорит. Стек — главный аспект метаболизма процесса. Это продиктовано тем, как происходит вызов функций. В предыдущей главе я уже упоминал, что функцию можно вызвать, только используя стек. Без этого сегмента нельзя выполнить ни одну функцию, что делает невозможной работу программы в целом.

Учитывая все вышесказанное, стек и его содержимое тщательно спроектированы для обеспечения бесперебойной работы процесса. Поэтому беспорядок в стеке может нарушить и прервать выполнение программы. Выделение памяти в стеке происходит быстро и не требует применения никаких специальных функций. Более того, освобождение ресурсов и все действия по управлению памятью происходят автоматически. Все описанное звучит заманчиво и может подстегнуть вас к излишнему использованию стека.

К этому следует относиться серьезно. Применение стека имеет свои нюансы. Данный сегмент не очень большой, поэтому в нем нельзя хранить крупные объекты. К тому же некорректное использование его содержимого может нарушить выполнение и привести к сбою. Это продемонстрировано в следующем фрагменте кода (листинг 5.1).

Листинг 5.1. Переполнение буфера. Функция strcpy перезаписывает содержимое стека

```
#include <string.h>

int main(int argc, char** argv) {
  char str[10];
  strcpy(str, "akjsdhkhqiueryo34928739r27yeiwuyfiusdciuti7twe79ye");
  return 0;
}
```

Если запустить этот код, то программа, скорее всего, аварийно завершится. Дело в том, что функция `strcpy` переполняет содержимое стека. Как видно в листинге 5.1, массив `str` содержит десять символов, но `strcpy` записывает в него намного больше элементов. Вскоре вы сами увидите, что это фактически приводит к перезаписи ранее сохраненных переменных и стековых фреймов, в результате чего после выхода из функции `main` программа переходит не к той инструкции. В итоге дальнейшее выполнение становится невозможным.

Надеюсь, данный пример помог продемонстрировать хрупкость стека. Углубленному рассмотрению данного сегмента посвящен первый раздел текущей главы. Начнем с исследования его содержимого.

Исследование содержимого стека

Прежде чем продолжать изучение стека, нам нужно сначала научиться его читать и, возможно, даже изменять. Как утверждалось в предыдущей главе, стек — изолированная область памяти, читать и модифицировать которую имеет право только ее владелец. Чтобы работать со стеком, необходимо быть частью процесса, которому он принадлежит.

Здесь нам понадобится целый новый класс инструментов: *отладчики*. Это программа, которая подключается к стороннему процессу и позволяет его *отлаживать*. При этом программисты обычно занимаются отслеживанием и изменением различных сегментов памяти. Читать и модифицировать приватные блоки памяти можно только в ходе отладки. Отладчик также позволяет управлять порядком выполнения программных инструкций. Примеры того, как это делается, будут рассмотрены чуть позже.

Попробуем скомпилировать программу и подготовить ее к отладке. Я покажу, как использовать `gdb` (GNU debugger) для запуска программы и чтения ее стека. В примере 5.1 объявлен массив символов, выделенный на вершине стека и содержащий элементы, показанные в листинге 5.2.

Листинг 5.2. Объявление массива, выделенного на вершине стека (ExtremeC_examples_chapter5_1.c)

```
#include <stdio.h>

int main(int argc, char** argv) {
  char arr[4];
  arr[0] = 'A';
  arr[1] = 'B';
  arr[2] = 'C';
  arr[3] = 'D';
  return 0;
}
```

Это простая программа, в которой легко разобраться, но в ее памяти происходит нечто интересное. Прежде всего, память, необходимая для массива `arr`, выделяется в стеке, а не в куче просто потому, что мы не использовали функцию `malloc`. Помните: в стеке по умолчанию выделяются все переменные и массивы.

Чтобы выделить место в куче, необходимо воспользоваться функцией `malloc` или ее аналогом, таким как `calloc`. В противном случае память будет выделена в стеке, а точнее, на его вершине.

Чтобы программу можно было отлаживать, ее двоичный файл должен быть собран соответствующим образом. То есть нам нужно сообщить компилятору о том, что в исполняемом файле должны быть *отладочные символы*. Они будут

использоваться для поиска строчек кода, которые выполняются или приводят к сбою. Скомпилируем пример 5.1 и создадим исполняемый объектный файл с отладочными символами.

Для начала соберем наш код в среде Linux (терминал 5.1).

Терминал 5.1. Компиляция примера 5.1 с отладочным параметром -g

```
$ gcc -g ExtremeC_examples_chapter5_1.c -o ex5_1_dbg.out
$
```

Параметр `-g` говорит компилятору о том, что итоговый исполняемый объектный файл должен содержать отладочную информацию. Обратите внимание: он влияет на размер программы. В терминале 5.2 вы можете видеть разницу в размере между двумя исполняемыми файлами, второй из которых скомпилирован с параметром `-g`.

Терминал 5.2. Размер выходного исполняемого файла с параметром -g и без него

```
$ gcc ExtremeC_examples_chapter2_10.c -o ex5_1.out
$ ls -al ex5_1.out
-rwxrwxr-x 1 kamranamini kamranamini 8640 jul 24 13:55 ex5_1.out
$ gcc -g ExtremeC_examples_chapter2_10.c -o ex5_1_dbg.out
$ ls -al ex5_1.out
-rwxrwxr-x 1 kamranamini kamranamini 9864 jul 24 13:56 ex5_1_dbg.
out
$
```

Итак, у нас есть исполняемый файл с отладочными символами. Теперь мы можем запустить его с помощью отладчика. В нашем примере для отладки используется `gdb`. В терминале 5.3 показана команда для запуска отладчика.

Терминал 5.3. Запуск отладчика для примера 5.1

```
$ gdb ex5_1_dbg.out
```

В системах Linux gdb обычно является частью пакета build-essentials. В macOS его можно установить с помощью диспетчера пакетов brew, используя команду brew install gdb.

В результате запуска отладчика мы получим примерно такой вывод (терминал 5.4).

Терминал 5.4. Вывод отладчика после запуска

```
$ gdb ex5_1_dbg.out
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later http://gnu.org/
```

```
licenses/gpl.html
...
Reading symbols from ex5_1_dbg.out...done.
(gdb)
```

Как вы, наверное, заметили, я выполнил эту команду на компьютере под управлением Linux. Отладчик `gdb` предоставляет интерфейс командной строки, который позволяет выполнять отладочные команды. Введите команду `r` (или `run`), чтобы запустить исполняемый объектный файл, поданный на вход отладчику. В терминале 5.5 показано, как команда `run` выполняет программу.

Терминал 5.5. Вывод отладчика после выполнения команды run

```
...
Reading symbols from ex5_1_dbg.out...done.
(gdb) run
Starting program: .../extreme_c/5.1/ex5_1_dbg.out
[Inferior 1 (process 9742) exited normally]
(gdb)
```

В этом терминале после ввода команды `run` отладчик запускает процесс, подключается к нему и дает возможность программе выполнить свои инструкции вплоть до завершения. Он не прерывает выполнение, поскольку мы не указали *точку останова*. Она говорит `gdb` о том, что процесс нужно приостановить и ждать дальнейших инструкций. Таких точек может быть сколько угодно.

Создадим точку останова в функции `main`, используя команду `b` (или `break`). В результате `gdb` остановит выполнение, когда программа войдет в `main`. Это показано в терминале 5.6.

Терминал 5.6. Создание точки останова для функции main в gdb

```
(gdb) break main
Breakpoint 1 at 0x400555: file ExtremeC_examples_chapter5_1.c, line 4.
(gdb)
```

Запустим нашу программу еще раз. Это приведет к созданию нового процесса, к которому подключится отладчик. Результат показан в терминале 5.7.

Терминал 5.7. Повторное выполнение программы после создания точки останова

```
(gdb) r
Starting program: .../extreme_c/5.1/ex5_1_dbg.out

Breakpoint 1, main (argc=1, argv=0x7fffffffcbd8) at ExtremeC_
examples_chapter5_1.c:3
3       int main(int argc, char** argv) {
(gdb)
```

Как видите, выполнение остановилось на строчке 3, с которой начинается функция `main`. После этого отладчик ждет ввода следующей команды. Мы можем попросить `gdb` выполнить следующую строчку и снова остановиться. Иными словами, программу можно выполнить пошагово, строчка за строчкой. Таким образом, у вас будет достаточно времени для того, чтобы оглядеться и проверить значения переменных в памяти. На самом деле с помощью именно этого метода мы и будем исследовать стек и кучу.

В терминале 5.8 показано, как выполнить следующую строчку кода с помощью команды `n` (или `next`).

Терминал 5.8. Использование команды n (или next) для выполнения следующих строчек кода

```
(gdb) n
5        arr[0] = 'A';
(gdb) n
6        arr[1] = 'B';
(gdb) next
7        arr[2] = 'C';
(gdb) next
8        arr[3] = 'D';
(gdb) next
9        return 0;
(gdb)
```

Теперь, если ввести команду `print arr`, отладчик покажет содержимое массива в виде строки (терминал 5.9).

Терминал 5.9. Вывод содержимого массива с помощью отладчика gdb

```
(gdb) print arr
$1 = "ABCD"
(gdb)
```

Но вернемся к исходной теме. Отладчик `gdb` нам нужен для того, чтобы заглянуть внутрь стековой памяти. И теперь мы готовы это сделать. У нас есть приостановленный процесс со стеком, и с помощью командной строки `gdb` мы можем исследовать его память. Для начала выведем участок памяти, выделенный для массива `arr` (терминал 5.10).

Терминал 5.10. Вывод байтов памяти, начиная с массива arr

```
(gdb) x/4b arr
0x7fffffffcae0: 0x41    0x42    0x43    0x44
(gdb) x/8b arr
0x7fffffffcae0: 0x41    0x42    0x43    0x44    0xff    0x7f
0x00    0x00
(gdb)
```

Первая команда, `x/4b`, выводит 4 байта, хранящиеся на том участке, на который указывает `arr`. Напомню, что `arr` — просто указатель на первый элемент массива, поэтому с его помощью можно перемещаться по памяти.

Вторая команда, `x/8b`, выводит 8 байт, идущих после `arr`. Согласно примеру 5.1 (см. листинг 5.2), в массиве `arr` находятся значения `A`, `B`, `C` и `D`. Вы должны знать, что массив хранит не сами символы, а их ASCII-коды. Например, ASCII-код `A` равен `65` в десятеричной системе или `0x41` в шестнадцатеричной. В случае с `B` это `66` или `0x42`. Таким образом, `gdb` выводит те значения, которые мы сохранили в массив `arr`.

Но что насчет остальных 4 байт во второй команде? Они находятся в стеке и, вероятно, содержат данные последнего стекового фрейма, созданного во время вызова функции `main`.

Обратите внимание: по сравнению с другими сегментами стек заполняется в обратном порядке.

Другие области памяти заполняются, начиная с младшего адреса, но со стеком все не так.

Заполнение стека происходит от старших адресов к младшим. Отчасти это связано с историей развития современных компьютеров, а отчасти — с некоторыми возможностями сегмента стека (который ведет себя как одноименная структура данных).

Учитывая все вышесказанное, если прочитать стек от младших адресов к старшим, как в терминале 5.10, то мы получим уже сохраненные данные; попытка их изменить приведет к модификации стека, что чревато проблемами. Чуть позже я объясню, в чем заключается опасность и как это делать правильно.

Так почему же мы можем читать данные за пределами массива `arr`? Дело в том, что отладчик `gdb` выводит то количество байтов памяти, которое мы запросили. Команду `x` не заботят границы массива. Чтобы вывести диапазон, ей нужны лишь начальный адрес и количество байтов.

Если вы хотите изменить значения внутри стека, то вам нужно использовать команду `set`. Это позволит модифицировать существующие ячейки памяти. В данном случае ячейка представляет собой отдельный байт внутри массива `arr` (терминал 5.11).

Терминал 5.11. Изменение отдельного байта в массиве с помощью команды set

```
(gdb) x/4b arr
0x7fffffffcae0: 0x41    0x42    0x43    0x44
(gdb) set arr[1] = 'F'
(gdb) x/4b arr
0x7fffffffcae0: 0x41    0x46    0x43    0x44
(gdb) print arr
$2 = "AFCD"
(gdb)
```

Как видите, с помощью команды `set` нам удалось поменять второй элемент массива `arr` на `F`. Отладчик `gdb` также позволяет перезаписывать адреса, которые выходят за пределы вашего массива.

Внимательно рассмотрим следующий пример (терминал 5.12). Мы хотим изменить байт, адрес которого намного больше, чем у `arr`. То есть, как уже объяснялось, мы собираемся модифицировать данные, помещенные в стек ранее. Напомню, что стековая память заполняется в противоположном порядке по сравнению с другими сегментами.

Терминал 5.12. Изменение отдельного байта за пределами массива

```
(gdb) x/20x arr
0x7fffffffcae0: 0x41    0x42    0x43    0x44    0xff    0x7f
0x00    0x00
0x7fffffffcae8: 0x00    0x96    0xea    0x5d    0xf0    0x31
0xea    0x73
0x7fffffffcaf0: 0x90    0x05    0x40    0x00
(gdb) set *(0x7fffffffcaed) = 0xff
(gdb) x/20x arr
0x7fffffffcae0: 0x41    0x42    0x43    0x44    0xff    0x7f
0x00    0x00
0x7fffffffcae8: 0x00    0x96    0xea    0x5d    0xf0    0xff
0x00    0x00
0x7fffffffcaf0: 0x00    0x05    0x40м   0x00
(gdb)
```

Вот и все. Мы только что записали значение `0xff` по адресу `0x7fffffffcaed`, который находится вне массива `arr`. Вполне вероятно, что этот байт принадлежит стековому фрейму, сохраненному еще до входа в функцию `main`.

Что произойдет, если мы продолжим выполнение? Если была модифицирована важная часть стека, то можно ожидать сбоя; в крайнем случае это изменение будет обнаружено неким механизмом, который прервет работу программы. Команда `c` (или `continue`) возобновит выполнение процесса в `gdb`, и мы увидим следующее (терминал 5.13).

Терминал 5.13. Модификация важного байта в стеке приводит к принудительному завершению процесса

```
(gdb) c
Continuing.
*** stack smashing detected ***: .../extreme_c/5.1/ex5_1_dbg.out
terminated

Program received signal SIGABRT, Aborted.
0x00007ffff7a42428 in __GI_raise (sig=sig@entry=6) at ../sysdeps/Unix/sysv/
linux/raise.c:54
```

```
54      ../sysdeps/Unix/sysv/linux/raise.c: No such file or
directory.
(gdb)
```

Мы только что сломали стек! Изменение его адреса, который выделяли не вы, даже если речь идет об 1 байте, может быть очень опасным и обычно приводит к сбою или внезапному завершению.

Как уже говорилось ранее, в стеке выполняются самые важные процедуры, относящиеся к выполнению программы. Поэтому при записи в стековые переменные следует быть крайне осторожными. Значения нельзя записывать за рамками переменных и массивов, поскольку в стеке адреса растут в обратном направлении, что повышает вероятность модификации уже записанных байтов.

Если вы закончили отладку и хотите покинуть `gdb`, можете использовать команду `q` (или `quit`). Она должна позволить вам выйти из отладчика и вернуться в терминал.

Вдобавок следует отметить, что запись непроверенных значений в *буфер* (байтовый или символьный массив), выделенный на вершине стека (а не в куче), считается уязвимостью. Злоумышленник может тщательно подготовить массив байтов и «скормить» его программе, чтобы получить контроль за ее выполнением. Обычно это называют *эксплойтом* в результате *переполнения буфера*.

Эта уязвимость продемонстрирована в листинге 5.3.

Листинг 5.3. Программа, демонстрирующая переполнение буфера

```
int main(int argc, char** argv) {
  char str[10];
  strcpy(str, argv[1]);
  printf("Hello %s!\n", str);
}
```

Этот код не проверяет содержимое и размер ввода `argv[1]`, копируя его прямо в массив `arr`, выделенный на вершине стека.

Если вам повезет, то вы отделаетесь лишь сбоем в работе. Но иногда это может привести к вредоносной атаке.

Рекомендации по использованию стековой памяти

Теперь вы должны лучше себе представлять, что такое сегмент стека и как он работает. Обсудим рекомендуемые методики и нюансы, к которым необходимо относиться с осторожностью. Вы уже должны быть знакомы с понятием *«область видимости»*. Каждая переменная имеет свою область видимости, которая определяет ее время жизни. Это значит, что вместе с данной областью исчезают и все переменные, которые были в ней созданы.

Кроме того, только переменные, находящиеся в стеке, выделяются и освобождаются автоматически. Автоматическое управление памятью обусловлено природой стекового сегмента.

Любая переменная, которую вы объявляете в стеке, автоматически создается на его вершине. Выделение памяти происходит автоматически и может считаться началом жизни переменной. После этого поверх нее будет записано множество других переменных и стековых фреймов. Но пока она находится в стеке и над ней есть другие переменные, ее существование продолжается.

Однако рано или поздно программа должна завершиться, и перед ее выходом стек должен быть пустым. Поэтому в какой-то момент наша переменная будет извлечена из стека. Таким образом, освобождение ее памяти происходит автоматически и знаменует конец ее жизненного цикла. Вот почему мы говорим, что управление памятью стековых переменных происходит автоматически, без участия программиста.

Представьте, что вы объявили в функции `main` переменную, как показано в листинге 5.4.

Листинг 5.4. Объявление переменной на вершине стека

```
int main(int argc, char** argv) {
  int a;
  ...
  return 0;
}
```

Эта переменная будет оставаться в стеке, пока не завершится функция `main`. Иными словами, переменная существует, пока остается действительной ее область видимости (функция `main`). И, поскольку в этой функции происходит выполнение всей программы, время жизни данной переменной почти такое же, как у глобального значения, доступного на протяжении всей работы процесса.

Тем не менее глобальное значение всегда остается в памяти, даже когда завершаются главная функция и сама программа, в то время как наша переменная будет извлечена из стека. Обратите внимание на два участка кода, которые выполняются перед функцией `main` и после нее; они нужны для того, чтобы подготовить программы к выполнению и соответственно завершить работу. Кроме того, стоит отметить, что глобальные переменные выделяются в другом сегменте памяти, Data или BSS, который ведет себя не так, как стек.

Рассмотрим пример очень распространенной ошибки. Ее часто допускают программисты-любители при написании своих первых программ на языке C. Речь идет о возвращении адреса локальной переменной внутри функции.

В листинге 5.5 показан пример 5.2.

Листинг 5.5. Объявление переменной на вершине стека (ExtremeC_examples_chapter5_2.c)

```
int* get_integer() {
  int var = 10;
  return &var;
}

int main(int argc, char** argv) {
  int* ptr = get_integer();
  *ptr = 5;
  return 0;
}
```

Функция `get_integer` возвращает адрес локальной переменной `var`, объявленной в ее области видимости. Затем функция `main` пытается разыменовать указатель и обратиться к соответствующей области памяти. В терминале 5.14 показан вывод компилятора `gcc` в ходе сборки этого кода в системе Linux.

Терминал 5.14. Компиляция примера 5.2 в Linux

```
$ gcc ExtremeC_examples_chapter5_2.c -o ex5_2.out
ExtremeC_examples_chapter5_2.c: In function 'get_integer':
ExtremeC_examples_chapter5_2.c:3:11: warning: function returns
address of local variable [-Wreturn-local-addr]
   return &var;
          ^~~~
$
```

Как видите, мы получили предупреждение. Поскольку возвращение адреса локальной переменной — распространенная ошибка, компиляторы о ней уже знают и выводят понятное сообщение такого содержания: *функция возвращает адрес локальной переменной.*

А вот что произойдет при запуске программы (терминал 5.15).

Терминал 5.15. Выполнение примера 5.2 в Linux

```
$ ./ex5_2.out
Segmentation fault (core dumped)
$
```

Здесь видна ошибка сегментации, что можно интерпретировать как отказ программы. Обычно это происходит при доступе к некорректной области памяти, которая уже освободилась.

Некоторые предупреждения следует воспринимать как ошибки. Это, к примеру, касается предыдущего сообщения, поскольку подобный код обычно приводит к сбою программы. Если вы хотите, чтобы компилятор gcc считал ошибками любые предупреждения, то передайте ему параметр -Werror. То же самое можно сделать и для отдельных предупреждений; например, в предыдущем случае достаточно указать параметр -Werror=return-local-addr.

Если запустить программу с помощью отладчика `gdb`, то можно узнать больше подробностей о ее сбое. Но помните: ее нужно скомпилировать с параметром `-g`, иначе от `gdb` будет мало пользы.

Компиляция исходников с помощью параметра `-g` обязательна, если вы собираетесь отлаживать свою программу с помощью `gdb` или других инструментов, таких как `valgrind`. В терминале 5.16 показано, как скомпилировать пример 5.2 и запустить его в отладчике.

Терминал 5.16. Выполнение примера 5.2 в отладчике

```
$ gcc -g ExtremeC_examples_chapter5_2.c -o ex5_2_dbg.out
ExtremeC_examples_chapter5_2.c: In function 'get_integer':
ExtremeC_examples_chapter5_2.c:3:11: warning: function returns
address of local variable [-Wreturn-local-addr]
   return &var;
          ^~~~
$ gdb ex5_2_dbg.out
GNU gdb (Ubuntu 8.1-0ubuntu3) 8.1.0.20180409-git
...
Reading symbols from ex5_2_dbg.out...done.
(gdb) run
Starting program: .../extreme_c/5.2/ex5_2_dbg.out

Program received signal SIGSEGV, Segmentation fault.
0x00005555555546c4 in main (argc=1, argv=0x7fffffffdf88) at
ExtremeC_examples_chapter5_2.c:8
8    *ptr = 5;
(gdb) quit
$
```

Вывод отладчика `gdb` говорит о том, что источник сбоя находится в строчке 8 внутри функции `main` — именно там, где программа пытается выполнить запись по возвращенному адресу путем разыменования полученного указателя. Переменная `var` была локальной для функции `get_integer` и перестала существовать просто потому, что в строчке 8 мы уже вышли из `get_integer` и соответствующей области видимости. Таким образом, возвращенный указатель получился *висячим*.

Обычно указатели на переменные в текущей области видимости рекомендуется передавать другом функциям, но не наоборот, поскольку они существуют, пока

область видимости остается действительной. Дальнейшие вызовы кладут на стек новые значения, и текущая область видимости не исчезнет раньше них.

Следует отметить, что эта рекомендация не относится к программам с параллельным выполнением, поскольку в будущем одно из параллельных заданий может воспользоваться полученным указателем, содержащим адрес переменной в текущей области видимости, но к тому моменту данная область уже может не существовать.

В завершение этого раздела перечислю ключевые характеристики стека, с которыми вы познакомились:

- стековая память имеет ограниченный размер, поэтому не подходит для хранения крупных объектов;
- адреса в стеке увеличиваются в обратном порядке, поэтому, перемещаясь вперед по стеку, мы читаем уже сохраненные байты;
- управление памятью в стеке происходит автоматически. Это касается как выделения, так и освобождения места;
- каждая переменная в стеке обладает областью видимости, которая определяет ее время жизни. Это следует учитывать при проектировании логики программы. Вы не можете контролировать этот механизм;
- указатели должны ссылаться только на те стековые переменные, которые все еще находятся в области видимости;
- освобождение памяти стековых переменных происходит автоматически, прямо перед исчезновением области видимости, и вы не можете на это повлиять;
- указатели на переменные, существующие в текущей области видимости, можно передавать в другие функции в качестве аргументов, но только если вы уверены в том, что в момент использования этих указателей область видимости будет оставаться на месте. Данное правило может не распространяться на программы с параллельной логикой.

В следующем разделе мы поговорим о куче и ее различных возможностях.

Куча

Почти любой код, написанный на любом языке программирования, так или иначе использует кучу. Это вызвано тем, что она имеет уникальные преимущества, недоступные при работе со стеком.

Но у этого сегмента памяти есть и недостатки. Например, выделение места в нем происходит медленней, чем в стеке.

В этом разделе мы более подробно поговорим о самой куче и принципах, которыми необходимо руководствоваться при ее использовании.

Благодаря уникальным свойствам куча играет важную роль. Однако не все из них положительные. На самом деле некоторые из них следует отнести к рискам, смягчению которых нужно уделять отдельное внимание. Любой важный инструмент обладает хорошими и плохими качествами, и если вы хотите правильно его использовать, то вам нужно как следует изучить обе стороны.

Эти качества перечислены ниже. Попробуйте определить, какие из них полезные, а какие — рискованные.

1. *В куче никакие блоки памяти не выделяются автоматически.* Вместо этого для выделения каждого участка памяти программист должен использовать `malloc` или аналогичную функцию. На самом деле это можно считать преимуществом кучи по сравнению со стеком. Фреймы, содержащиеся в стеке, выделяются не самим программистом за счет вызова специальных функций, а автоматически.
2. *Куча большая.* В то время как стек имеет ограниченный размер и не подходит для размещения крупных объектов, куча позволяет хранить огромные объемы данных, достигающие десятков гигабайтов. По мере увеличения кучи аллокатору нужно запрашивать у операционной системы все больше страниц памяти, по которым распределяются блоки этого сегмента. Стоит отметить, что, в отличие от стека, куча выделяет адреса по направлению от младшего к старшему.
3. *Выделением и освобождением памяти в куче занимается программист.* Это значит, на программиста ложится вся ответственность за выделение памяти и последующее ее освобождение, когда она больше не нужна. Во многих современных языках освобождение блоков кучи выполняется автоматически параллельным компонентом под названием *«сборщик мусора»*. Но в C и C++ такого механизма нет, и потому освобождать блоки кучи нужно вручную. Это, несомненно, создает определенные риски, и программисты на C/C++ должны быть очень осторожными при работе с кучей. Если не освободить выделенный блок, то может произойти *утечка памяти*, которая в большинстве случаев является роковой.
4. *Переменные, выделенные в куче, не имеют никакой области видимости*, в отличие от стековых переменных. Такое свойство можно считать отрицательным, поскольку оно существенно усложняет управление памятью. Вы не знаете, когда нужно освободить переменную, поэтому для эффективного использования кучи необходимо выработать собственные понятия *области видимости* и *владельца*. Некоторые из способов, позволяющих это реализовать, рассматриваются в следующих разделах.

5. *К блокам кучи можно обращаться только с помощью указателей.* Иными словами, нет такого понятия, как «переменная кучи». Для навигации по куче используются указатели.
6. *Поскольку куча доступна только владеющему ей процессу, для ее исследования нужен отладчик.* К счастью, в языке C указатели работают одинаково как со стеком, так и с блоками кучи. Данная абстракция работает очень хорошо, и благодаря ей мы можем использовать одни и те же указатели для обращения к этим двум сегментам памяти. Таким образом, для исследования кучи и стека можно применять одни те же методы.

В следующем подразделе будет показано, как выделять и освобождать блоки памяти в куче.

Выделение и освобождение памяти в куче

Как уже отмечалось в предыдущем подразделе, место в куче необходимо выделять и освобождать вручную. Для этого программист должен использовать набор функций или API (функции для работы с памятью из стандартной библиотеки C).

Эти функции существуют, и их определения находятся в заголовке `stdlib.h`. Для получения блока памяти в куче используются функции `malloc`, `calloc` и `realloc`, а для освобождения памяти предусмотрена только одна функция: `free`. В примере 5.3 показано, как их применять.

Иногда в технической литературе кучу называют динамической памятью. Выделение динамической памяти — то же самое, что выделение кучи.

В листинге 5.6 показан исходный код примера 5.3. Он выделяет два блока кучи и затем выводит собственные отображения памяти.

Листинг 5.6. Пример 5.3, который выводит отражения памяти после выделения двух блоков в куче (ExtremeC_examples_chapter5_3.c)

```
#include <stdio.h>  // для функции printf
#include <stdlib.h> // для функций по работе с кучей из библиотеки C
void print_mem_maps() {
#ifdef __linux__
  FILE* fd = fopen("/proc/self/maps", "r");
  if (!fd) {
    printf("Could not open maps file.\n");
    exit(1);
  }
```

```
  char line[1024];
  while (!feof(fd)) {
    fgets(line, 1024, fd);
    printf("> %s", line);
  }
  fclose(fd);
#endif
}

int main(int argc, char** argv) {
  // выделяем 10 байт без инициализации
  char* ptr1 = (char*)malloc(10 * sizeof(char));
  printf("Address of ptr1: %p\n", (void*)&ptr1);
  printf("Memory allocated by malloc at %p: ", (void*)ptr1);
  for (int i = 0; i < 10; i++) {
    printf("0x%02x ", (unsigned char)ptr1[i]);
  }
  printf("\n");

  // выделяем 10 байт, каждый из которых обнулен
  char* ptr2 = (char*)calloc(10, sizeof(char));
  printf("Address of ptr2: %p\n", (void*)&ptr2);
  printf("Memory allocated by calloc at %p: ", (void*)ptr2);
  for (int i = 0; i < 10; i++) {
    printf("0x%02x ", (unsigned char)ptr2[i]);
  }
  printf("\n");

  print_mem_maps();

  free(ptr1);
  free(ptr2);

  return 0;
}
```

Приведенный выше код является кросс-платформенным и может быть скомпилирован в большинстве операционных систем семейства Unix. Но функция `print_mem_maps` работает только в Linux, поскольку макрос `__linux__` определен лишь в этой ОС. Таким образом, вы можете скомпилировать данный код в macOS, но вызов `print_mem_maps` не будет ничего делать.

Результаты выполнения этого примера в среде Linux показаны в терминале 5.17.

Терминал 5.17. Вывод примера 5.3 в Linux

```
$ gcc ExtremeC_examples_chapter5_3.c -o ex5_3.out
$ ./ex5_3.out
Address of ptr1: 0x7ffe0ad75c38
Memory allocated by malloc at 0x564c03977260: 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00
```

```
Address of ptr2: 0x7ffe0ad75c40
Memory allocated by calloc at 0x564c03977690: 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00
> 564c01978000-564c01979000 r-xp 00000000 08:01 5898436
/home/kamranamini/extreme_c/5.3/ex5_3.out
> 564c01b79000-564c01b7a000 r--p 00001000 08:01 5898436
/home/kamranamini/extreme_c/5.3/ex5_3.out
> 564c01b7a000-564c01b7b000 rw-p 00002000 08:01 5898436
/home/kamranamini/extreme_c/5.3/ex5_3.out
> 564c03977000-564c03998000 rw-p 00000000 00:00 0          [heap]
> 7f31978ec000-7f3197ad3000 r-xp 00000000 08:01 5247803    /lib/
x86_64-linux-gnu/libc-2.27.so
...
> 7f3197eef000-7f3197ef1000 rw-p 00000000 00:00 0
> 7f3197f04000-7f3197f05000 r--p 00027000 08:01 5247775    /lib/
x86_64-linux-gnu/ld-2.27.so
> 7f3197f05000-7f3197f06000 rw-p 00028000 08:01 5247775    /lib/
x86_64-linux-gnu/ld-2.27.so
> 7f3197f06000-7f3197f07000 rw-p 00000000 00:00 0
> 7ffe0ad57000-7ffe0ad78000 rw-p 00000000 00:00 0
[stack]
> 7ffe0adc2000-7ffe0adc5000 r--p 00000000 00:00 0          [vvar]
> 7ffe0adc5000-7ffe0adc7000 r-xp 00000000 00:00 0          [vdso]
> ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0
[vsyscall]
$
```

Здесь есть на что посмотреть. Программа выводит адреса указателей `ptr1` и `ptr2`. Если среди показанных выше отражений памяти найти сегмент стека, то можно заметить, что она начинается с `0x7ffe0ad57000` и заканчивается на `0x7ffe0ad78000`. В данном диапазоне находятся наши указатели.

Это значит, указатели выделены в стеке, но ссылаются на какой-то другой сегмент памяти — в данном случае на кучу. Использование стекового указателя для работы с блоком кучи — распространенная практика.

Имейте в виду: указатели `ptr1` и `ptr2` находятся в одной области видимости и освобождаются при завершении функции `main`, однако блоки памяти, полученные в куче, не входят ни в какую область видимости. Они будут существовать, пока их не освободят вручную. Вы можете видеть, что перед выходом из функции `main` оба блока освобождаются с помощью указателей, которые на них ссылаются, и функции `free`.

Относительно данного примера следует сделать еще одно замечание: адреса, возвращенные функциями `malloc` и `calloc`, находятся внутри сегмента кучи. В этом можно убедиться, если сравнить их с отражением памяти, помеченным как `[heap]`. Эта область находится в диапазоне от `0x564c03977000` до `0x564c03998000`, к которому принадлежат адреса указателей `ptr1` и `ptr2`: `0x564c03977260` и `0x564c03977260` соответственно.

Если вернуться к функциям выделения памяти в куче, то `calloc` расшифровывается как *clear and allocate* («очистить и выделить»), а `malloc` — *memory allocation* («выделение памяти»). То есть `calloc` очищает выделенный блок памяти, а `malloc` оставляет его инициализацию самой программе.

В C++ ключевые слова new и delete делают то же самое, что malloc и free соответственно. Кроме того, оператор new определяет размер выделяемого блока памяти по типу операнда и автоматически приводит к этому типу возвращаемый указатель.

Но, взглянув на содержимое этих блоков, можно заметить: оба состоят из нулевых байтов. Похоже, функция `malloc` тоже инициализировала блок памяти после его выделения. Но если верить ее описанию в спецификации языка C, то она не должна этого делать. Как же это объяснить? Чтобы разобраться, запустим тот же пример в среде macOS (терминал 5.18).

Терминал 5.18. Вывод примера 5.3 в macOS

```
$ clang ExtremeC_examples_chapter5_3.c -o ex5_3.out
$ ./ ex5_3.out
Address of ptr1: 0x7ffee66b2888
Memory allocated by malloc at 0x7fc628c00370: 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x80 0x00 0x00
Address of ptr2: 0x7ffee66b2878
Memory allocated by calloc at 0x7fc628c02740: 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00
$
```

Если присмотреться, то можно заметить, что в блоке памяти, который выделила функция `malloc`, находятся ненулевые байты, а блок, выделенный функцией `calloc`, состоит из одних нулей. Что же делать? Можно ли исходить из того, что `malloc` в Linux всегда выделяет обнуленные блоки?

Если вы собираетесь писать кросс-платформенную программу, то вам следует руководствоваться спецификацией языка C. В спецификации сказано: функция `malloc` не инициализирует выделяемый блок памяти.

Но даже если вы пишете свою программу только для Linux и вас не интересуют другие операционные системы, то имейте в виду: в будущем компиляторы могут изменить свое поведение. Поэтому, согласно спецификации C, мы должны всегда предполагать, что блок памяти, выделенный функцией `malloc`, является неинициализированным и в случае необходимости его нужно инициализировать вручную.

Обратите внимание: благодаря этому `malloc` обычно работает быстрее, чем `calloc`. На самом деле некоторые реализации `malloc` откладывают выделение блока памяти

до тех пор, пока к нему кто-то не обратится (для чтения или записи). Таким образом, ускоряется выделение памяти.

Если вы хотите инициализировать память, выделенную с помощью `malloc`, то можете использовать функцию `memset`. Как это делается, показано в листинге 5.7.

Листинг 5.7. Использование функции memset для инициализации блока памяти

```
#include <stdlib.h> // для malloc
#include <string.h> // для memset

int main(int argc, char** argv) {
  char* ptr = (char*)malloc(16 * sizeof(char));
  memset(ptr, 0, 16 * sizeof(char));    // заполняем нулями
  memset(ptr, 0xff, 16 * sizeof(char)); // заполняем байтами 0xff
  ...
  free(ptr);
  return 0;
}
```

Еще одно средство выделения памяти в куче — функция `realloc`. Мы не использовали ее в примере 5.3. Она перераспределяет память, изменяя размер уже выделенного блока. В листинге 5.8 показано, как это может происходить.

Листинг 5.8. Использование функции realloc для изменения размера уже выделенного блока

```
int main(int argc, char** argv) {
  char* ptr = (char*)malloc(16 * sizeof(char));
  ...
  ptr = (char*)realloc(32 * sizeof(char));
  ...
  free(ptr);

  return 0;
}
```

Функция `realloc` не изменяет содержимое выделенной памяти, а просто расширяет старый блок. Если это не удается сделать из-за *фрагментации*, то она находит другой блок подходящего размера и копирует в него данные из старого блока. В этом случае последний освобождается. Как видите, такое перераспределение может оказаться недешевой операцией, состоящей из множества этапов, и потому ее следует использовать с осторожностью.

Заканчивая рассматривать пример 5.3, остановимся на функции `free`. Она освобождает уже выделенный блок кучи, принимая его адрес в виде указателя. Как уже отмечалось ранее, когда блок кучи больше не нужен, его необходимо освободить. Если этого не сделать, то возникнет *утечка памяти*. В примере 5.4 я продемонстрирую, как искать утечки памяти с помощью утилиты `valgrind`.

Сначала создадим утечку (листинг 5.9).

Листинг 5.9. Создание утечки памяти из-за невыполнения освобождения выделенного блока при возвращении из главной функции

```
#include <stdlib.h> // для функций по работе с кучей

int main(int argc, char** argv) {
  char* ptr = (char*)malloc(16 * sizeof(char));
  return 0;
}
```

У данной программы есть утечка памяти, поскольку после ее завершения в куче остается неосвобожденный блок размером 16 байт. Это очень простой пример, но по мере увеличения объема исходного кода и появления новых компонентов утечки будет очень сложно (если вообще возможно) обнаружить невооруженным глазом.

Для поиска проблем с памятью в активном процессе применяются специальные профилировщики, наиболее известен из которых `valgrind`.

Чтобы проанализировать пример 5.4 с помощью `valgrind`, нам сначала нужно собрать его с параметром `-g`. Затем профилировщик можно использовать для запуска полученной программы. Во время работы исполняемого объектного файла `valgrind` записывает все операции выделения и освобождения памяти. После завершения или отказа программы он выводит записанную информацию и показывает, какой объем памяти не был освобожден. Таким образом, мы сможем узнать, сколько памяти утекло в ходе выполнения нашей программы.

В терминале 5.19 показано, как скомпилировать пример 5.4 и запустить его с помощью профилировщика `valgrind`.

Терминал 5.19. В выводе valgrind видно, что в результате выполнения примера 5.4 утекло 16 байт памяти

```
$ gcc -g ExtremeC_examples_chapter5_4.c -o ex5_4.out
$ valgrind ./ex5_4.out
==12022== Memcheck, a memory error detector
==12022== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==12022== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==12022== Command: ./ex5_4.out
==12022==
==12022==
==12022== HEAP SUMMARY:
==12022==     in use at exit: 16 bytes in 1 blocks
==12022==   total heap usage: 1 allocs, 0 frees, 16 bytes allocated
==12022==
==12022== LEAK SUMMARY:
==12022==    definitely lost: 16 bytes in 1 blocks
==12022==    indirectly lost: 0 bytes in 0 blocks
```

```
==12022==      possibly lost: 0 bytes in 0 blocks
==12022==    still reachable: 0 bytes in 0 blocks
==12022==         suppressed: 0 bytes in 0 blocks
==12022== Rerun with --leak-chck=full to see details of leaked memory
==12022==
==12022== For counts of detected and suppressed errors, rerun with: -v
==12022== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
$
```

В разделе `HEAP SUMMARY` указано, что наша программа выделила один блок, а освободила 0; в результате в момент выхода выделенными оставались 16 байт. Если опуститься чуть ниже к разделу `LEAK SUMMARY`, то можно увидеть, что эти 16 байт безвозвратно потеряны, то есть произошла утечка памяти!

На случай, если вы хотите узнать, в какой именно строчке был выделен этот потерянный блок, у `valgrind` предусмотрен специальный параметр. В терминале 5.20 показано, как с помощью профилировщика найти строчки кода, ответственные за выделение памяти.

Терминал 5.20. valgrind выводит строчки, ответственные за выделение памяти

```
$ gcc -g ExtremeC_examples_chapter5_4.c -o ex5_4.out
$ valgrind --leak-check=full ./ex5_4.out
==12144== Memcheck, a memory error detector
==12144== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==12144== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==12144== Command: ./ex5_4.out
==12144==
==12144==
==12144== HEAP SUMMARY:
==12144==     in use at exit: 16 bytes in 1 blocks
==12144==   total heap usage: 1 allocs, 0 frees, 16 bytes allocated
==12144==
==12144== 16 bytes in 1 blocks are definitely lost in loss record 1 of 1
==12144==    at 0x4C2FB0F: malloc (in /usr/lib/valgrind/
vgpreload_memcheck-amd64-linux.so)
==12144==    by 0x108662: main (ExtremeC_examples_chapter5_4.c:4)
==12144==
==12144== LEAK SUMMARY:
==12144==    definitely lost: 16 bytes in 1 blocks
==12144==    indirectly lost: 0 bytes in 0 blocks
==12144==    possibly lost: 0 bytes in 0 blocks
==12144==  still reachable: 0 bytes in 0 blocks
==12144==       suppressed: 0 bytes in 0 blocks
==12144==
==12144== For counts of detected and suppressed errors, rerun with : -v
==12144== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
$
```

Как видите, мы передали утилите `valgrind` параметр `--leak-check=full`, и теперь она показывает строчку кода, в которой был выделен потерянный блок кучи. Это строчка 4 — то есть вызов `malloc`. Таким образом, вы можете проследить за этим блоком и найти место, где его следует освободить.

Хорошо, отредактируем данный пример так, чтобы он освобождал выделенную память. Для этого перед выражением `return` достаточно добавить инструкцию `free(ptr)`, как показано в листинге 5.10.

Листинг 5.10. Освобождение выделенного блока памяти в примере 5.4

```
#include <stdlib.h> // для функций по работе с кучей

int main(int argc, char** argv) {
  char* ptr = (char*)malloc(16 * sizeof(char));
  free(ptr);
  return 0;
}
```

После внесения этого изменения единственный выделенный блок кучи будет освобожден. Снова соберем данный код и запустим его с помощью все того же профилировщика (терминал 5.21).

Терминал 5.21. Вывод valgrind после освобождения выделенного блока памяти

```
$ gcc -g ExtremeC_examples_chapter5_4.c -o ex5_4.out
$ valgrind --leak-check=full ./ex5_4.out
==12175== Memcheck, a memory error detector
==12175== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==12175== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==12175== Command: ./ex5_4.out
==12175==
==12175==
==12175== HEAP SUMMARY:
==12175==     in use at exit: 0 bytes in 0 blocks
==12175==   total heap usage: 1 allocs, 1 frees, 16 bytes allocated
==12175==
==12175== All heap blocks were freed -- no leaks are possible
==12175==
==12175== For counts of detected and suppressed errors, rerun with -v
==12175== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
$
```

Обратите внимание на сообщение `All Heap blocks were freed` (Все блоки кучи были освобождены). Оно фактически означает, что в программе не осталось утечек памяти. Программа, запущенная с помощью упомянутого профилировщика, может существенно замедлить свою работу (в 10–50 раз), но благодаря этому вы сможете легко выявить проблемы с памятью. Запускать код внутри `valgrind`, чтобы обнаружить утечки памяти на максимально раннем этапе, — рекомендованный подход.

Утечки памяти можно считать *техническим долгом*, если они вызваны плохой архитектурой, или *рисками*, если нам о них известно, но мы не знаем, что произойдет в случае их роста. Но, по моему мнению, к ним следует относиться как к *программным ошибкам*; в противном случае их поиск задним числом может отнять много времени. Именно так поступают команды разработчиков, пытаясь исправить их как можно быстрее.

Помимо `valgrind`, существуют другие профилировщики памяти. Среди самых известных можно выделить *LLVM ASAN* (Address Sanitizer) и *MemProf*. Профилировщики могут анализировать использование памяти и операции выделения с помощью различных методов. Часть из них перечислены ниже.

- Некоторые профилировщики могут служить изолированной средой для запуска программ и мониторинга их операций с памятью. Мы применяли этот метод для запуска примера 5.4 в изолированной среде `valgrind`. Он не требует перекомпиляции кода.
- Еще один метод заключается в использовании библиотек, которые предоставляют некоторые профилировщики в качестве оберток для системных вызовов, относящихся к управлению памятью. Таким образом, итоговый двоичный файл будет содержать всю логику, необходимую для профилирования. Профилировщик `valgrind` и ASAN можно скомпоновать с итоговым исполняемым объектным файлом в виде библиотек для профилирования памяти. Этот метод требует повторной компиляции и даже некоторой модификации исходного кода.
- Программы также могут *предварительно загружать* разные библиотеки с функциями, которые меняют поведение операций по выделению памяти из стандартной библиотеки C. Таким образом, вам не обязательно компилировать свой исходный код. Вы можете просто указать библиотеки таких профайлеров в переменной среды `LD_PRELOAD`, и они будут загружены вместо стандартных библиотек `libc`. Этот метод применяется в `MemProf`.

Подстановочная функция — это обертка вокруг стандартной функции. Она находится в динамической библиотеке, загружается перед оригиналом и перенаправляет ему поступающие вызовы. Предварительную загрузку динамических библиотек позволяет выполнять переменная среды LD_PRELOAD.

Принцип работы кучи

Как уже отмечалось, куча имеет несколько отличий от стека. Поэтому при работе с ней нужно руководствоваться отдельными правилами. Здесь мы поговорим об этих отличиях и узнаем, как следует обращаться с данным пространством памяти и чего не стоит делать.

Каждый блок памяти (или переменная) в стеке имеет область видимости, поэтому определить его время жизни не составляет труда. Каждый раз, когда мы покидаем область видимости, все ее переменные исчезают. Но с кучей все намного сложнее.

У блока кучи нет никакой области видимости, и потому его время жизни является неочевидным и должно быть определено вручную. Именно поэтому в современных языках программирования, таких как Java, применяется автоматическое освобождение или *сборка мусора* с поддержкой *поколений объектов*. Сама программа и библиотеки, которые она использует, не в состоянии определить время жизни кучи, вследствие чего вся ответственность за это ложится на программиста.

Когда дело доходит до принятия конкретных решений, особенно в этом случае, сложно предложить какой-то универсальный ответ. Любое мнение является предметом дискуссии и может вести к компромиссу.

Одна из лучших стратегий по преодолению сложностей, связанных с временем жизни кучи, состоит в определении *владельца* блока памяти, а не области видимости, которая вмещает в себя данный блок. Конечно, это нельзя назвать полноценным решением.

Владелец полностью ответственен за управление жизненным циклом блока кучи; он его изначально выделяет и затем, когда нужда в нем отпадает, освобождает.

Существует множество классических примеров применения этой стратегии. Она используется в большинстве известных библиотек для работы с кучей в языке C. В примере 5.5 показана очень простая реализация данного метода, позволяющая обеспечивать управление жизненным циклом объекта очереди. Итак, в листинге 5.11 продемонстрирована стратегия *владения*.

Листинг 5.11. Пример 5.5 демонстрирует стратегию владения в целях управления временем жизни кучи (ExtremeC_examples_chapter5_5.c)

```
#include <stdio.h> // для функции printf
#include <stdlib.h> // для функций по работе с кучей

#define QUEUE_MAX_SIZE 100

typedef struct {
  int front;
  int rear;
  double* arr;
} queue_t;

void init(queue_t* q) {
  q->front = q->rear = 0;
  // выделенными здесь блоками кучи владеет объект очереди
  q->arr = (double*)malloc(QUEUE_MAX_SIZE * sizeof(double));
}
```

```
void destroy(queue_t* q) {
  free(q->arr);
}

int size(queue_t* q) {
  return q->rear - q->front;
}

void enqueue(queue_t* q, double item) {
  q->arr[q->rear] = item;
  q->rear++;
}

double dequeue(queue_t* q) {
  double item = q->arr[q->front];
  q->front++;
  return item;
}

int main(int argc, char** argv) {
  // выделенными здесь блоками кучи владеет функция main
  queue_t* q = (queue_t*)malloc(sizeof(queue_t));

  // выделяем необходимую память для объекта очереди
  init(q);

  enqueue(q, 6.5);
  enqueue(q, 1.3);
  enqueue(q, 2.4);

  printf("%f\n", dequeue(q));
  printf("%f\n", dequeue(q));
  printf("%f\n", dequeue(q));

  // освобождаем ресурсы, полученные объектом очереди
  destroy(q);

  // освобождаем память, выделенную для объекта кучи, принадлежащего функции main
  free(q);
  return 0;
}
```

В этом примере используется два экземпляра владения для двух разных объектов. Первый относится к блоку кучи, на который ссылается указатель `arr` в структуре `queue_t`, принадлежащей объекту очереди. Пока этот объект существует, данный блок будет оставаться в памяти.

Второй экземпляр владения относится к блоку кучи, который функция `main` выделила в качестве заглушки для объекта очереди, `q` (и владельцем которого является). Очень важно отличать блоки кучи, принадлежащие объекту очереди

и функции `main`, поскольку освобождение одного не приводит к освобождению другого.

Чтобы продемонстрировать, как в приведенном выше коде может возникнуть утечка памяти, предположим, будто вы забыли вызвать функцию `destroy` для объекта очереди. Это непременно приведет к утечке, поскольку блок кучи, полученный внутри функции `init`, останется в памяти и будет освобожден.

Обратите внимание: факт владения блоком кучи со стороны объекта или функции следует отметить в комментариях. Код, которому этот блок не принадлежит, не должен его освобождать.

Кроме того, нужно сказать, что *повторное освобождение* одного и того же блока кучи приводит к повреждению памяти, и потому эту и любую другую проблему подобного рода следует исправлять сразу после обнаружения. В противном случае она может повлечь внезапные сбои программы.

Помимо стратегии владения, можно также использовать сборщик мусора — автоматический механизм, встроенный в программу, пытающийся освобождать блоки памяти, на которые не ссылается ни один указатель. В языке C одним из традиционных и широко известных инструментов этого типа является *консервативный сборщик мусора Бема — Демерса — Вайзера;* он предоставляет набор функций для выделения памяти, которые нужно вызывать вместо `malloc` и других стандартных операций языка C.

Больше информации о данном сборщике мусора можно найти здесь: http://www.hboehm.info/gc/.

Еще один подход к управлению временем жизни блоков кучи — использование объекта RAII (resource acquisition is initialization — «получение ресурса есть инициализация»). Это идиома объектно-ориентированного программирования, согласно которой время жизни ресурса (такого как выделенный блок кучи) можно привязать к времени жизни объекта. Иными словами, мы задействуем объект, который во время своего создания инициализирует ресурс, а во время своего уничтожения — освобождает его. К сожалению, данный метод нельзя реализовать в C, поскольку в этом языке программиста не уведомляют об уничтожении объектов. Но можно эффективно применять в C++ с помощью деструкторов. Объект RAII инициализирует ресурс в своем конструкторе, а тот содержит код, отвечающий за деинициализацию. Следует отметить, что в C++ деструктор вызывается автоматически, когда объект удаляется или выходит из области видимости.

Подводя итоги, перечислю факты и рекомендации, которые нужно учитывать при работе с кучей.

- Выделение памяти в куче отнимает определенные ресурсы. Разные функции выделения памяти имеют разные накладные расходы. Самой «дешевой» является функция `malloc`.
- Все блоки памяти, выделенные в пространстве кучи, должны быть освобождены либо сразу после того, как в них пропадает необходимость, либо перед завершением программы.
- Поскольку у блоков кучи нет области видимости, во избежание потенциальных утечек программа должна уметь управлять памятью.
- Как показывает практика, при выделении каждого блока кучи следует придерживаться выбранной стратегии управления памятью.
- Выбранную стратегию и предположения, на которых она основана, следует документировать на любом участке кода, где происходит доступ к блоку, чтобы в будущем программисты об этом знали.
- В определенных языках программирования, таких как C++, управлять ресурсами (такими как, к примеру, блоки кучи) можно с помощью объектов RAII.

До сих пор мы исходили из того, что имеем достаточно памяти для хранения крупных объектов и выполнения любого рода программ. Но в следующем разделе мы установим определенные ограничения на доступную память и обсудим среды с небольшим объемом памяти или с невозможностью ее расширения (ввиду таких факторов, как стоимость, время, производительность и т. д.). В подобных условиях доступную память нужно использовать максимально эффективно.

Управление памятью в средах с ограниченными ресурсами

Существуют среды, в которых память является ценным и зачастую ограниченным ресурсом. В ряде систем ключевым фактором выступает производительность, и программы, которые в них выполняются, должны работать быстро, вне зависимости от количества доступной памяти. Каждая среда требует применения определенных методик, позволяющих избежать нехватки памяти и снижения производительности. Прежде всего нужно определиться с тем, что такое среда с ограниченными ресурсами.

Ограниченность ресурсов еще не говорит о малом объеме памяти. *Ограничения* обычно касаются того, как программа может использовать данную память. Это могут быть жесткие лимиты, установленные вашим клиентом, определенная аппаратная конфигурация или отсутствие поддержки больших объемов памяти в вашей операционной системе (как, например, в MS-DOS).

Но даже при отсутствии подобных ограничений программист должен пытаться использовать как можно меньше памяти и делать это наиболее оптимальным образом. Потребление памяти — одно из ключевых *нефункциональных требований* к проекту, и потому его нужно тщательно отслеживать и оптимизировать.

В этом разделе вы сначала познакомитесь с методиками, которые используются в низкоуровневых средах в целях преодоления проблем с нехваткой памяти, а затем мы поговорим о приемах, распространенных в производительных средах и призванных увеличить скорость работы программ.

Среды с ограниченной памятью

В таких средах ограничением является память, и ваши алгоритмы должны уметь справляться с ее нехваткой. К данной категории обычно относятся встраиваемые системы с памятью размером десятки или сотни мегабайт. Существует несколько приемов по управлению памятью в подобных средах, однако ни один из них не сравнится по своей эффективности с хорошо оптимизированным алгоритмом. В этом случае обычно используются алгоритмы с низким потреблением памяти, которое обычно выливается в увеличение *времени работы*.

Поговорим об этом более подробно. Каждый алгоритм обладает *временно́й* сложностью и сложностью *памяти*. Первая описывает отношение между размером ввода и временем, необходимым для завершения алгоритма. Вторая описывает отношение между размером ввода и объемом памяти, который нужен алгоритму для выполнения работы. В математике эти отношения обычно обозначаются как *О большое*, но мы не станем на них останавливаться. Нас интересуют качественные характеристики, поэтому для обсуждения сред с ограниченной памятью можно обойтись и без математики.

В идеале алгоритм должен иметь низкую временну́ю сложность и низкую сложность памяти. Иными словами, быстрая работа и умеренное потребление памяти крайне желательны, но в реальности подобное сочетание встречается редко. Точно так же мы не ожидаем низкой производительности от алгоритма, которому нужно много памяти.

В большинстве случаев мы имеем дело с неким балансом между памятью и скоростью (то есть временем работы). Например, если один алгоритм сортировки работает быстрее другого, то ему обычно нужно больше памяти, хотя оба делают одно и то же.

При написании кода можно исходить из того, что он будет выполняться в системе с ограниченной памятью, даже если нам известно, что в конечной промышленной среде ресурсов будет более чем достаточно, — это хороший, но консервативный

подход. Такое предположение делается в целях снижения риска чрезмерного потребления памяти.

Обратите внимание: основой для этого предположения должен быть достаточно точный прогноз о среднем объеме доступной памяти в конечной системе. Алгоритмы, рассчитанные на среды с ограниченной памятью, более медленные по своей природе, и, чтобы не попасть в ловушку, их следует использовать с осторожностью.

Ниже мы рассмотрим кое-какие методики, которые помогут нам собирать расходованную память и уменьшить ее потребление в средах с ограниченными ресурсами.

Упакованные структуры

Один из самых простых способов уменьшить потребление памяти — использовать упакованные структуры. Они игнорируют выравнивание в памяти и применяют более компактную схему размещения своих полей.

На самом деле это компромиссное решение. Ввиду отказа от выравнивания чтение структуры занимает больше времени, что замедляет программу.

Это простой метод, но подходящий не для всех программ. Более подробно о нем можно почитать в разделе «Структуры» главы 1.

Сжатие

Это эффективная методика, особенно если программа работает с большим количеством текстовых данных, которые нужно хранить в памяти. Текстовые данные, по сравнению с двоичными, имеют высокую *степень сжатия*. Таким образом, программа может хранить текст в сжатом виде и экономить много памяти.

Но за все приходится платить: алгоритмы сжатия требуют интенсивных вычислений и сильно нагружают процессор, поэтому конечная производительность ухудшается. Данный метод идеально подходит для программ, которые хранят много текстовых данных, но нечасто их используют; в противном случае понадобится много операций сжатия/разжатия, что в конечном счете затруднит или сделает невозможным применение программы.

Внешнее хранилище данных

Использование внешнего хранилища данных в виде сетевого сервиса, облачной инфраструктуры или обычного жесткого диска — очень распространенный и действенный способ борьбы с нехваткой памяти. Обычно предполагается, что программа может выполняться в среде с ограниченными ресурсами, поэтому

существует много программ, которые используют данный подход, даже когда памяти в достатке.

Использование этой методики обычно предполагает, что оперативная память играет роль *кэша*. Еще одно предположение — все данные в память не поместятся и загружать их нужно по частям или *постранично*.

Эти алгоритмы не решают проблемы с нехваткой памяти как таковые, но пытаются решить проблемы с медленным внешним хранилищем данных. Внешние хранилища всегда очень медленные по сравнению с оперативной памятью. Поэтому алгоритмы должны балансировать между чтением из внутренней памяти и из внешнего хранилища. Данный подход используют все базы данных, включая PostgreSQL и Oracle.

В большинстве проектов подобные алгоритмы не имеет смысла изобретать и реализовывать с нуля, поскольку они довольно нетривиальные. Разработчики, стоящие за такими библиотеками, как SQLite, годами шлифуют свой код.

Если вам необходимо обращаться к внешнему хранилищу данных, такому как файл, БД или сетевой узел, сохраняя при этом умеренное использование памяти, то вы всегда можете подобрать подходящий инструмент.

Высокопроизводительные среды

Как уже объяснялось выше, быстрым алгоритмам свойственно повышенное потребление памяти. Поговорим об этом более подробно.

Наглядной иллюстрацией этого утверждения может быть использование кэша для увеличения производительности. Кэширование данных означает повышенный расход памяти, но взамен мы получаем повышенную скорость, если кэш используется должным образом

Но расширение памяти — не всегда оптимальный путь увеличения производительности. Существуют другие методы, которые имеют прямое или опосредованное отношение к памяти и могут существенно повлиять на скорость работы алгоритма. Но прежде, чем переходить к ним, сначала поговорим о кэшировании.

Кэширование

Термин *«кэширование»* объединяет в себе все похожие методики, применяемые во многих частях компьютерной системы и подразумевающие использование двух хранилищ данных с разными скоростями чтения/записи. Например, у центрального процессора есть ряд внутренних регистров, которые работают очень быстро на чтение и запись. Кроме того, процессор должен копировать данные из оперативной памяти, которая во много раз медленнее его регистров. Здесь нужен

механизм кэширования, иначе низкая скорость оперативной памяти станет доминирующей и нивелирует высокую производительность вычислений, свойственную процессору.

В качестве еще одного примера можно привести работу с базой данных. Файлы БД обычно хранятся на внешнем жестком диске, который на несколько порядков медленнее оперативной памяти. Очевидно, что здесь нужен механизм кэширования, иначе самая низкая скорость станет общим знаменателем и будет определять производительность всей системы.

Кэширование и связанные с ним нюансы заслуживают отдельной главы, поскольку перед их рассмотрением необходимо изучить абстрактные модели и соответствующие термины.

С помощью этих моделей можно предсказать, насколько хорошо проявит себя кэш и какого *повышения производительности* можно ожидать от его использования. Здесь мы попытаемся объяснить принцип работы кэширования простым и наглядным образом.

Представьте: у вас есть медленное хранилище, которое может содержать много элементов. И быстрое — с ограниченной вместимостью. Очевидно, здесь нужно искать компромисс. Хранилище с высокой скоростью, но малым размером можно назвать *кэшем*. Перед обработкой элементов их было бы разумно копировать из медленного хранилища в быстрое — просто потому, что оно быстрее.

Время от времени вам нужно будет обращаться к медленному хранилищу за новыми элементами. Очевидно, что элементы следует доставать не по одному, поскольку это было бы неэффективно. Вместо этого в быстрое хранилище лучше копировать целую группу элементов (назовем ее *бакетом*). Принято говорить, что элементы кэшируются в быстрое хранилище.

Теперь представьте: при обработке одного элемента вам нужно загрузить другой, и для этого приходится обращаться к медленному хранилищу. Первое, что приходит на ум, — поискать нужный элемент в недавно скопированном нами бакете, который уже находится в кэше.

Если поиск увенчался успехом, то нет нужды использовать медленное хранилище. Это так называемое *попадание*. При отсутствии элемента в кэше вам придется сходить в медленное хранилище и скопировать в кэш еще один бакет. Это называют *промахом*. Естественно, чем больше попаданий, тем выше производительность.

Этот принцип можно применить к кэшу центрального процессора и оперативной памяти. В кэше ЦПУ находятся последние инструкции и данные, прочитанные из более медленной оперативной памяти.

Далее обсудим дружественный к кэшированию код и понаблюдаем за тем, насколько быстрее его выполняет процессор.

Дружественный к кэшированию код

При выполнении инструкции процессору сначала нужно получить все необходимые данные. Они находятся в оперативной памяти по определенному адресу, который указан в инструкции.

Перед вычислением данные должны быть скопированы в регистры процессора. Однако он обычно копирует больше блоков, чем ожидалось, и помещает их в свой кэш.

В следующий раз, если ему понадобится значение, находящееся *недалеко* от предыдущего адреса, он сможет найти его в кэше и избежать обращения к оперативной памяти, что намного повысит скорость чтения. Как объяснялось в предыдущем пункте текста, это *попадание кэша*. Если значение не удается найти, то это *промах кэша*, в результате которого процессору придется считывать и копировать нужный адрес из оперативной памяти, что довольно медленно. В целом, чем выше частота попаданий, тем быстрее работает код.

Зачем процессор копирует соседние значения, находящиеся поблизости от искомого адреса? Это связано с *принципом локальности*. В компьютерных системах доступ к данным, находящимся в одном районе, обычно происходит чаще. Процессор следует данному принципу и загружает дополнительные данные из той же местности. Если алгоритм умеет извлекать пользу из такого поведения, то будет выполняться быстрее. Поэтому такие алгоритмы называют *дружественными к кэшированию*.

В примере 5.6 показана разница в производительности между двумя блоками кода, один из которых дружественен к кэшированию, а другой — нет (листинг 5.12).

Листинг 5.12. Пример 5.6 демонстрирует производительность обычного и дружественного к кэшу кода (ExtremeC_examples_chapter5_6.c)

```
#include <stdio.h>  // для функции printf
#include <stdlib.h> // для функций по работе с кучей
#include <string.h> // для функции strcmp

void fill(int* matrix, int rows, int columns) {
  int counter = 1;
  for (int i = 0; i < rows; i++) {
    for (int j = 0; j < columns; j++) {
      *(matrix + i * columns + j) = counter;
    }
    counter++;
  }
}

void print_matrix(int* matrix, int rows, int columns) {
  int counter = 1;
  printf("Matrix:\n");
  for (int i = 0; i < rows; i++) {
```

```
    for (int j = 0; j < columns; j++) {
      printf("%d ", *(matrix + i * columns + j));
    }
    printf("\n");
  }
}

void print_flat(int* matrix, int rows, int columns) {
  printf("Flat matrix: ");
  for (int i = 0; i < (rows * columns); i++) {
    printf("%d ", *(matrix + i));
  }
  printf("\n");
}

int friendly_sum(int* matrix, int rows, int columns) {
  int sum = 0;
  for (int i = 0; i < rows; i++) {
    for (int j = 0; j < columns; j++) {
      sum += *(matrix + i * columns + j);
    }
  }
  return sum;
}

int not_friendly_sum(int* matrix, int rows, int columns) {
  int sum = 0;
  for (int j = 0; j < columns; j++) {
    for (int i = 0; i < rows; i++) {
      sum += *(matrix + i * columns + j);
    }
  }
  return sum;
}

int main(int argc, char** argv) {

  if (argc < 4) {
    printf("Usage: %s [print|friendly-sum|not-friendly-sum] ");
    printf("[number-of-rows] [number-of-columns]\n", argv[0]);
    exit(1);
  }
  char* operation = argv[1];
  int rows = atol(argv[2]);
  int columns = atol(argv[3]);

  int* matrix = (int*)malloc(rows * columns * sizeof(int));
  fill(matrix, rows, columns);

  if (strcmp(operation, "print") == 0) {
    print_matrix(matrix, rows, columns);
```

```
    print_flat(matrix, rows, columns);
  }
  else if (strcmp(operation, "friendly-sum") == 0) {
    int sum = friendly_sum(matrix, rows, columns);
    printf("Friendly sum: %d\n", sum);
  }
  else if (strcmp(operation, "not-friendly-sum") == 0) {
    int sum = not_friendly_sum(matrix, rows, columns);
    printf("Not friendly sum: %d\n", sum);
  }
  else {
    printf("FATAL: Not supported operation!\n");
    exit(1);
  }

  free(matrix);
  return 0;
}
```

Эта программа вычисляет и выводит сумму всех элементов матрицы, однако имеет еще одну особенность.

Пользователь может изменять ее поведение путем передачи параметров. Допустим, мы хотим вывести матрицу размером 2×3, инициализированную алгоритмом из функции `fill`. Для этого пользователь должен указать параметр `print` и нужное количество строк и столбцов. В терминале 5.22 показано, как эти параметры передаются итоговому исполняемому файлу.

Терминал 5.22. Вывод примера 5.6 с матрицей 2×3

```
$ gcc ExtremeC_examples_chapter5_6.c -o ex5_6.out
$ ./ex5_6.out print 2 3
Matrix:
1 1 1
2 2 2
Flat matrix: 1 1 1 2 2 2
$
```

Данный вывод состоит из двух разных представлений матрицы: двумерного и *плоского*. Как видите, элементы матрицы *развертываются в памяти по строкам*. Это значит, они сохраняются строка за строкой. Поэтому если процессор копирует из строки один элемент, то все остальные элементы в данной строке тоже, скорее всего, копируются в кэш. И, как следствие, сложение лучше выполнять по строкам, а не *по столбцам*.

Если еще раз взглянуть на код, то можно заметить, что в функции `friendly_sum` сложение происходит построчно, а в `friendly_sum` — по столбцам. Посмотрим, сколько времени у них уйдет на сложение элементов матрицы с 20 000 строк и 20 000 столбцов. Разница налицо (терминал 5.23).

Терминал 5.23. Разница во времени выполнения алгоритмов сложения элементов матрицы с развертыванием по строкам и по столбцам

```
$ time ./ex5_6.out friendly-sum 20000 20000
Friendly sum: 1585447424

real   0m5.192s
user   0m3.142s
sys    0m1.765s

$ time ./ex5_6.out not-friendly-sum 20000 20000
Not friendly sum: 1585447424

real   0m15.372s
user   0m14.031s
sys   0m0.791s
$
```

Разница во времени составила около 10 секунд! Программа была скомпилирована в macOS с помощью `clang`. Такое отличие говорит о том, что одна и та же логика с одним и тем же объемом памяти может выполняться намного дольше, если в ней выбран не тот способ доступа к элементам матрицы! Этот пример наглядно демонстрирует преимущество кода, дружественного к кэшированию.

Утилита time доступна во всех операционных системах семейства Unix. С ее помощью можно измерить время между запуском и завершением программы.

Прежде чем переходить к следующему методу, следует сказать несколько слов о накладных расходах на выделение и освобождение памяти.

Накладные расходы на выделение и освобождение памяти

Здесь мы отдельно поговорим о цене, которую приходится платить за выделение и освобождение памяти в куче. Вас, наверное, удивит тот факт, что эти операции могут занимать довольно много времени и требовать существенных ресурсов, особенно если речь идет о выделении и освобождении множества блоков каждую секунду.

В отличие от стека, в котором выделение памяти происходит относительно быстро и не требует получения дополнительного диапазона адресов, куча должна сначала найти свободный блок памяти достаточного размера, и это может быть затратной операцией.

Для выделения и освобождения существует множество алгоритмов, и между этими операциями всегда приходится находить разумный баланс. Если вы хотите быстро выделять память, то для этого необходимо потреблять больше блоков. И наоборот,

если вы хотите использовать меньше памяти, то ваш алгоритм выделения будет работать медленней.

Помимо функций `malloc` и `free`, которые входят в стандартную библиотеку, в языке C можно использовать другие аллокаторы. В их число входят `ptmalloc`, `tcmalloc`, `Haord` и `dlmalloc`.

Как же решить эту скрытую проблему? Очень просто: реже выделять и освобождать память. В ряде программ, которым необходимо часто выделять место в куче, такое решение может показаться недостижимым. В подобных случаях в куче обычно выделяют один большой блок и пытаются работать с ним самостоятельно. В результате получается еще один слой логики для выделения/освобождения памяти (может быть, не такой сложный, как функции `malloc` и `free`), управляющий этим большим блоком.

Но есть другой метод, основанный на *пулах памяти*. Прежде чем завершать эту главу, мы кратко объясним, как он работает.

Пулы памяти

Как вы уже знаете, выделение и освобождение памяти — затратные операции. Чтобы сократить их количество и повысить производительность, можно использовать пул заранее выделенных блоков кучи фиксированного размера. Обычно каждый блок имеет идентификатор, который можно получить с помощью API, предназначенного для управления пулом. Позже, когда блок больше не нужен, его можно освободить. Поскольку объем выделенной памяти не меняется, это отличное решение для алгоритмов, стремящихся к предсказуемому поведению в средах с ограниченными ресурсами.

Подробное описание пулов памяти выходит за рамки данной книги; если вы хотите узнать больше, то в Интернете есть много материалов на эту тему.

Резюме

В этой главе мы в основном сосредоточились на стеке и куче и попытались выяснить, как их использовать. Мы также уделили некоторое внимание средам с ограниченной памятью и рассмотрели методики повышения производительности, такие как кэширование и пулы памяти.

В данной главе мы:

- обсудили инструменты и методики, которые используются для исследования сегментов стека и кучи;
- познакомились с отладчиками и использовали `gdb` в качестве основного инструмента для поиска проблем, связанных с памятью;

- поговорили о профилировщиках памяти и использовали `valgrind` для поиска утечек и висячих указателей на этапе выполнения программы;
- сравнили время жизни стековой переменной и блока кучи и выяснили, как их оценивать;
- убедились в том, что в стеке управление памятью происходит автоматически, а в куче — полностью вручную;
- прошлись по распространенным ошибкам, возникающим при работе с переменными стека;
- обсудили среды с ограниченными ресурсами и увидели, как в них можно оптимизировать память;
- поговорили о высокопроизводительных средах и о том, какие методики можно использовать для повышения производительности.

Следующие четыре главы посвящены объектной ориентированности. На первый взгляд может показаться, что это не имеет отношения к C, однако на самом деле на данном языке можно писать настоящий объектно-ориентированный код. Вы узнаете, как правильно проектировать и решать проблемы в стиле ООП, и получите рекомендации по написанию понятного и корректного кода на C.

В главе 6 рассматриваются инкапсуляция и основы объектно-ориентированного программирования. В ходе обсуждения этой темы будет предоставлен как теоретический материал, так и практические примеры.

6 ООП и инкапсуляция

Объектно-ориентированному программированию (ООП) посвящено множество отличных книг и статей. Но мне кажется, немногие из них рассматривают эту тему в контексте такого процедурного языка, как C! Действительно, разве это возможно? Можно ли писать объектно-ориентированные программы на языке, который на это не рассчитан? В частности, возможно ли написать на C программу в стиле ООП?

Если коротко, то да. Но сначала следует объяснить почему. Поставленные выше вопросы нужно разбить на части и подумать над тем, что на самом деле представляет собой ООП. Почему мы можем писать объектно-ориентированные программы на языке, который не поддерживает данный стиль? Это похоже на парадокс, но здесь нет ничего парадоксального. В текущей главе я попытаюсь объяснить, почему это возможно и как это следует делать.

Вы также можете задаться вопросом, какой смысл обсуждать и изучать ООП, если вы собираетесь задействовать C в качестве основного языка программирования? Почти все зрелые проекты на C, такие как открытые ядра, реализации сервисов наподобие HTTPD, Postfix, nfsd и ftpd, а также многочисленные библиотеки, такие как OpenSSL и OpenCV, написаны в объектно-ориентированном стиле. Это не означает, что C — объектно-ориентированный язык; просто подход, который использовался при организации внутренней структуры указанных проектов, основан на объектно-ориентированном образе мышления.

Я настоятельно рекомендую почитать данную главу и три следующие и узнать больше об ООП, поскольку это, во-первых, позволит вам мыслить и проектировать подобно авторам упомянутых выше библиотек, а во-вторых, это очень поможет вам читать исходный код таких проектов.

В синтаксисе C нет таких объектно-ориентированных концепций, как классы, наследование и виртуальные функции. Но их поддержку можно реализовать самостоятельно. На самом деле почти все компьютерные языки, которые когда-либо существовали, имели все необходимое для поддержки ООП — задолго до Smalltalk, C++ и Java. Дело в том, что любой язык программирования общего назначения должен предоставлять возможность расширения своих типов данных, а это уже первый шаг в направлении ООП.

Синтаксис C не может и *не должен* поддерживать объектно-ориентированные возможности; и не потому, что старый, а по нескольким очень весомым причинам, которые мы обсудим в данной главе. Проще говоря, на C можно писать в стиле ООП, просто это немного сложнее и требует чуть больше усилий.

ООП в C посвящено не так много книг и статей, и обычно их авторы пытаются создать *систему типов* для написания классов, реализации наследования, полиморфизма и т. д. В этих книгах поддержка ООП рассматривается как совокупность функций, макросов и препроцессора, с помощью которых на C можно писать объектно-ориентированные программы. В данной главе мы пойдем по другому пути. Вместо того чтобы создавать из C новую разновидность C++, мы поразмышляем о потенциале этого языка с точки зрения ООП.

Часто говорят, что ООП — еще одна парадигма программирования наряду с процедурной и функциональной. Но это верно лишь отчасти. ООП — скорее образ мышления и способ анализа проблем. Это взгляд на вселенную и иерархию объектов в ней. Это часть древнего, свойственного человеку подхода к постижению и исследованию физических и абстрактных вещей, которые нас окружают. Данный подход лежит в основе нашего понимания природы.

Мы всегда обдумывали любой вопрос с объектно-ориентированной точки зрения. ООП лишь берет этот взгляд на вещи, который свойственен нам всем, и применяет его к решению вычислительных задач. Таким образом, становится понятно, почему ООП — наиболее распространенная парадигма программирования, которая используется в разработке ПО.

В данной главе и трех последующих будет показано, что на языке C реализуемы любые концепции ООП — хотя это может быть довольно сложной задачей. Мы знаем: ООП имеет место в C, поскольку это уже было доказано на практике другими людьми (особенно что касается создания C++ поверх C), которые написали множество сложных и успешных программ на этом языке в объектно-ориентированном стиле.

Однако в этих главах вы *не* найдете упоминания конкретных библиотек или наборов макросов, с помощью которых можно объявлять классы, организовывать наследование и работать с другими концепциями ООП. Я также не стану навязывать никакие методологии или правила наподобие определенных соглашений об именовании. Мы просто будем использовать обычный язык C в целях реализации объектной ориентированности.

Причина, по которой я отвожу *целых четыре главы* теме ООП в C, такова: объектная ориентированность имеет обширную теоретическую основу и, чтобы продемонстрировать ее в полном объеме, нужно рассмотреть множество разных примеров. В этой главе будет представлен основной теоретический материал, стоящий за ООП, а в более практические аспекты мы погрузимся в следующих главах. Учитывая все вышесказанное, нам следует начать с теории, поскольку большинство умелых программистов на C, даже те, у кого за плечами многолетний опыт, незнакомы с концепциями ООП.

Эти четыре главы в совокупности охватят почти все, с чем можно столкнуться в ООП.

- Прежде всего будет дано определение самым фундаментальным терминам, которые используются в литературе в отношении ООП. Я расскажу, что такое классы, объекты, атрибуты, поведение, методы, предметные области и т. д. Они будут активно применяться в данных четырех главах. Эти концепции являются неотъемлемой частью общепринятой терминологии и ключом к пониманию других аспектов объектной ориентированности.
- Первый раздел этой главы посвящен терминологии, но не целиком; мы также подробно обсудим истоки идеи ООП, философию, лежащую в ее основе, и природу объектно-ориентированного образа мышления.
- Второй раздел посвящен языку C и тому, почему он не является и не может быть объектно-ориентированным. Это важный вопрос, который заслуживает хорошего ответа. К данной теме мы вернемся во время исследования системы Unix и ее тесных отношений с C в главе 10.
- В третьем разделе мы поговорим об *инкапсуляции* — одной из основополагающих концепций ООП. Если коротко, то это механизм, который позволяет создавать и использовать объекты. Тот факт, что вы можете размещать переменные и методы внутри объектов, — прямое следствие инкапсуляции. Мы тщательно рассмотрим данное явление и разберем несколько примеров.
- Затем мы перейдем к принципу *сокрытия*, который в определенном смысле является побочным (хотя и очень важным) эффектом поддержки инкапсуляции. Без него мы бы не могли изолировать и разделять программные модули, что фактически сделало бы невозможным предоставление клиентам API, не зависящих от реализации. Это последнее, о чем пойдет речь в данной главе.

Как уже упоминалось ранее, эта тема будет изложена в четырех главах. Например, в главе 7 мы поговорим о *композиции*, а в главах 8 и 9 — об *агрегации*, *наследовании*, *полиморфизме* и *абстрагировании*.

Начнем же мы с теории, стоящей за ООП. Посмотрим, как вычленить объектную модель из нашего процесса мышления и воплотить ее в программном компоненте.

Объектно-ориентированное мышление

Как уже говорилось во вступительной части данной главы, объектно-ориентированное мышление — способ анализа и разбиения на части окружающего нас мира. Когда вы смотрите на вазу, стоящую на столе, вы понимаете, что это два отдельных объекта, и для этого вам не нужно долго думать.

Вы интуитивно осознаете, что между этими предметами есть граница. Вы знаете, что изменение цвета вазы не повлияет на цвет стола.

Эти наблюдения говорят о том, что мы воспринимаем окружающий мир в объектно-ориентированном ключе. Иными словами, мы просто создаем у себя в голове отражение объектно-ориентированной реальности. Это часто можно видеть в компьютерных играх, программах для 3D-моделирования и инженерном ПО, где мы имеем дело со множеством взаимодействующих объектов.

Суть ООП состоит в применении объектно-ориентированного мышления к проектированию и разработке программного обеспечения. Мы воспринимаем окружающий мир объектно-ориентированным образом, поэтому парадигма ООП стала наиболее востребованной при написании программ.

Конечно, существуют проблемы, которые сложно решить с помощью этого подхода и для анализа и устранения которых было бы проще использовать другую парадигму. Но они считаются относительно редкими.

В следующем подразделе вы узнаете больше о переходе от объектно-ориентированного мышления к написанию объектно-ориентированного кода.

Как мы мыслим

Сложно найти программу, в которой нет ни следа объектно-ориентированного мышления, даже если она написана на C или на другом языке без поддержки ООП. Написание объектно-ориентированного кода естественно для нас. На это указывают даже имена переменных. Взгляните на следующий пример (листинг 6.1). В нем объявлены переменные, необходимые для хранения информации о десяти студентах.

Листинг 6.1. Четыре массива, названные по одному и тому же принципу с использованием префикса student_ и предназначенные для хранения информации о десяти студентах

```
  char*  student_first_names[10];
  char*  student_surnames[10];
   int   student_ages[10];
double   student_marks[10];
```

Объявления в данном листинге показывают, как переменные можно объединять по какому-то общему признаку (в данном случае это *«студент»*) с помощью их имен. Если этого не сделать, то наш объектно-ориентированный разум запутается в бессвязных именах. Представьте, что наш код выглядит так (листинг 6.2).

Листинг 6.2. Четыре массива с произвольными именами, предназначенные для хранения информации о десяти студентах

```
  char*  aaa[10];
  char*  bbb[10];
   int   ccc[10];
double   ddd[10];
```

Назначение переменным таких имен, как в листинге 6.2, независимо от вашего опыта программирования, приведет к большим проблемам при написании алгоритма. Именование переменных всегда было важным аспектом, поскольку имена напоминают нам о концепциях и их отношениях с данными. Эта информация теряется, если мы используем в своем коде произвольные имена. Это может не быть проблемой для компьютера, но затруднит анализ и диагностику со стороны программистов, а также повысит вероятность допущения ошибок.

Объясню, что имеется в виду под концепцией в этом контексте. *Концепция* — мысленный или абстрактный образ, который существует в нашем воображении в виде мысли или идеи. Он может быть сформирован нашим восприятием материального объекта, но может быть и полностью воображаемым. Когда вы смотрите на дерево или думаете об автомобиле, вам на ум приходят соответствующие образы, которые представляют собой две разные концепции.

Обратите внимание: иногда мы используем термин «концепция» в другом смысле — например, очевидно, что понятие «объектно-ориентированные концепции» не соответствует тому определению, которое я привел только что. Слово «концепция», будучи примененным в техническом контексте, означает принципы, которые следует понимать при изучении той или иной темы. Пока мы будем придерживаться именно такого определения.

Концепции играют важную роль в объектно-ориентированном мышлении, ведь если вы не можете держать в уме понимание объектов, то не сможете извлечь информацию о том, что они представляют и к чему относятся, как и понять их взаимоотношения.

Таким образом, объектно-ориентированное мышление основано на концепциях и связях между ними. Из этого следует, что для написания настоящей объектно-ориентированной программы необходимо иметь хорошее понимание всех соответствующих объектов, их концепций и того, как они связаны друг с другом.

Объектно-ориентированную картину, состоящую из множества концепций и их взаимоотношений, сложно передать другим людям — например, во время обсуждения задачи с коллегами. Более того, такие воображаемые концепции изменчивы и неуловимы и о них можно легко забыть. Это подчеркивает тот факт, что превращение ваших мысленных образов в идеи, которые можно объяснить, потребует их представления с помощью моделей и других инструментов.

Диаграммы связей и объектные модели

Чтобы лучше понять все то, о чем мы говорили выше, рассмотрим практический пример. Допустим, вы создали описание сцены. Цель подобного рода описания — донести до аудитории соответствующие концепции. Попробуйте взглянуть на это с такой стороны: тот, кто создает описание, имеет в своем воображении диаграмму

связей, в которой разложены по полочкам различные концепции и то, как они соединены вместе; цель данного человека — передать эту диаграмму аудитории. Вы можете сказать, что примерно так происходит любое творческое выражение мыслей; действительно, описанный процесс повторяется каждый раз, когда вы смотрите на картину, слушаете музыку или читаете роман.

Теперь взглянем на письменное описание классной комнаты. Расслабьтесь и попытайтесь представить себе то, о чем читаете. Все, что вы увидите в своем воображении, — концепция, переданная следующим описанием.

> *Наша классная комната старая, с двумя большими окнами. При входе в нее на противоположной стене можно видеть окно. Посреди нее находится несколько коричневых деревянных стульев. На этих стульях сидят пять учеников, и два из них — мальчики. На стене справа от нас висит зеленая деревянная доска, и учитель что-то говорит ученикам. Он пожилой человек в синей рубашке.*

Посмотрим, какие концепции сформировались в нашем воображении. Но имейте в виду: вы можете слишком глубоко погрузиться в свои мысли и даже не заметить этого. Поэтому постараемся придерживаться описания. Например, я могу представить, что у девочек в классе светлые волосы. Но в описании об этом не упоминается, и потому данную концепцию следует отбросить. В следующем абзаце я объясню, какая картина возникла в моем воображении, и, прежде чем продолжать, вам стоит попытаться сделать то же самое.

Я представил пять концепций (или мысленных образов, или объектов), по одной для каждого ученика в классе. Еще пять концепций — для стульев. По одной — для дерева и стекла. Еще я знаю, что стулья сделаны из дерева. Это отношение между концепциями «дерево» и «стулья». Я также знаю, что каждый ученик сидит на стуле. Из этого следует еще пять отношений — между стульями и учениками. Данное упражнение можно было бы продолжать и в итоге получить огромную и сложную диаграмму, описывающую отношения между сотнями концепций.

Теперь остановитесь на минуту и подумайте о том, чем отличается ваш подход к извлечению концепций и их отношений. Каждый делает это по-своему. То же самое происходит при решении конкретной проблемы: вначале вам необходимо создать в уме диаграмму связей. Мы будем называть данный этап *пониманием*.

Решение проблемы основано на ее концепциях и отношениях между ними, которые вам удается обнаружить. Вы объясняете его с помощью терминов этих концепций, и если кто-то захочет в нем разобраться, то это потребует понимания выведенных вами концепций и того, как они связаны между собой.

Вы, наверное, удивитесь, если я скажу, что точно так же решаются проблемы с помощью компьютера, но это именно тот случай. Проблема разбивается на объекты

(то же самое, что концепции в контексте мышления) и связи между ними, и затем делается попытка написать на основе этих объектов программу, которая в итоге решает данную проблему.

Написанная вами программа будет имитировать концепции и их отношения так, как вы это себе представляете. Компьютер выполнит ваше решение, и вы сможете проверить, работает оно или нет. Проблему решаете вы, а компьютер становится вашим коллегой, поскольку может выполнить то, что вы придумали, в виде последовательности машинных инструкций, сгенерированных из вашей диаграммы связей. К тому же он может сделать это намного быстрее и точнее.

В объектно-ориентированной программе концепции представлены объектами, а вместо диаграммы связей, которую мы держим в уме, она использует объектную модель, хранящуюся в памяти. Иными словами, если сравнивать человека с объектно-ориентированной программой, то термины *«концепция»*, *«ум»* и *«диаграмма связей»* эквивалентны таким понятиям, как *«объект»*, *«память»* и *«объектная модель»* соответственно. Это самое важное соответствие, которое вы найдете в данной главе; оно позволяет перевести то, как мы мыслим, в объектно-ориентированный код.

Но зачем нам компьютеры для имитации наших диаграмм связей? Все просто: они незаменимы с точки зрения скорости и точности. Это классический ответ на подобные вопросы, и в нашем случае он подходит. Создание и хранение большой диаграммы связей и соответствующей объектной модели — сложная задача, с которой компьютеры справляются очень хорошо. К тому же объектную модель, созданную программой, можно сохранить на диск и использовать в будущем.

Диаграмму связей, которую мы держим в уме, можно забыть, и на нее могут повлиять эмоции. Но компьютеры бесстрастны, а объектные модели куда более надежны, чем человеческие мысли. Вот почему мы должны писать объектно-ориентированный код — чтобы переводить концепции, которые представляем, в эффективные программы.

Мы пока не умеем загружать и сохранять схемы связей из человеческого ума — но все еще впереди!

В коде нет никаких объектов

Взглянув на память запущенной объектно-ориентированной программы, можно увидеть множество объектов, связанных между собой. То же самое с человеческим разумом. Если допустить, что люди — компьютеры, то можно сказать, что они всегда включены и работают, пока не умирают. Это важная аналогия. Объекты могут существовать только в запущенной программе — точно так же, как концепции могут существовать лишь в разуме живого человека. Это значит, что за пределами активной программы никаких объектов нет.

Это может прозвучать парадоксально, поскольку во время написания программы (в объектно-ориентированном стиле) она еще не существует и, следовательно, не может работать! Как же мы можем писать объектно-ориентированный код, если у нас нет ни запущенной программы, ни объектов?

В вашем объектно-ориентированном коде нет никаких объектов. Они создаются, когда вы собираете и запускаете свою программу.

На самом деле ООП не сводится к созданию объектов. Данная парадигма заключается в генерации набора инструкций, которые при запуске программы превращаются в полностью динамическую объектную модель. Поэтому объектно-ориентированный код, скомпилированный и запущенный, должен уметь создавать, изменять, связывать и даже удалять разные объекты.

Таким образом, написание объектно-ориентированного кода — непростая задача. Вам необходимо представлять себе объекты и отношения между ними в то время, когда они еще не существуют. Именно поэтому ООП может вызывать трудности и вот почему нам нужен язык программирования, который поддерживает данную парадигму. Умение вообразить то, чего еще нет, и описать различные его аспекты обычно называют *проектированием* (или *объектно-ориентированным проектированием* в контексте ООП).

В самом коде мы лишь планируем создание объектов. Где и как они будут создаваться — определяют инструкции, которые мы генерируем. Конечно, создание — еще не все. Всевозможные операции с объектами можно описывать с помощью языка программирования. Объектно-ориентированный язык предоставляет набор инструкций (и грамматических правил), позволяющих писать и планировать различные операции, связанные с объектами.

Мы уже видели, что между концепциями в нашем воображении и объектами в памяти программы существует четкая связь. Точно так же должны быть связаны между собой операции, которые можно проводить с концепциями и объектами.

Каждый объект имеет отдельный жизненный цикл. То же самое относится и к нашим мысленным концепциям. В какой-то момент нам на ум приходит идея, которая порождает мысленный образ (концепцию), но рано или поздно эта идея улетучивается. Точно так же в один момент времени объекты создаются, а в какой-то другой — уничтожаются.

В завершение следует сказать, что одни мысленные концепции могут быть прочными и долговечными, а другие — изменчивыми и мимолетными. Похоже, первые существуют вне зависимости от человеческого разума и не требуют того, чтобы их кто-то осознал. В основном это относится к области математики. Возьмем, к примеру, число 2. Оно уникально, и другое такое не сыскать во всей вселенной!

Поразительно. Это значит, мы имеем в наших мыслях одну и ту же концепцию числа 2; если попытаться ее изменить, то это будет уже нечто другое. Вот где заканчивается мир объектной ориентированности и начинается другой мир, полный неизменяемых объектов и известный под названием *«парадигма функционального программирования»*.

Атрибуты объектов

Любая концепция, которую мы себе представляем, имеет некие атрибуты. Если вернуться к нашему описанию классной комнаты, то у нас был коричневый стул с именем *chair1*. То есть каждый стул имел такой атрибут, как цвет, и в случае с *chair1* этот цвет был коричневым. Мы знаем, что в классе было еще четыре стула, и каждый из них тоже имел некий цвет (возможно, уникальный). В нашем описании все они были коричневыми, но теоретически один или два из них могли быть желтыми.

Объект может иметь целый набор атрибутов. Совокупность значений, которые присвоены этим атрибутам, называется *состоянием* объекта. Состояние можно представить в виде списка значений, каждое из которых принадлежит определенному атрибуту, относящемуся к объекту. Объект можно изменять на протяжении его времени жизни. В таком случае его называют *изменяемым* (mutable). Это просто означает, что его состояние может меняться со временем. Объекты также могут *не иметь состояния* (или каких-либо атрибутов).

Объект может быть и *неизменяемым* — точно так же, как концепция числа 2, которую нельзя изменить. Неизменяемость означает, что состояние определяется в момент создания и после этого его нельзя модифицировать.

Объект без состояния можно считать неизменяемым объектом, поскольку его состояние не меняется на протяжении его существования. Состояние нельзя изменить, если его нет.

В заключение следует отметить: неизменяемые объекты имеют особое значение. Тот факт, что их состояние нельзя изменить, является преимуществом, особенно если они разделяются в многопоточной среде.

Предметная область

У любой программы, написанной для решения конкретной задачи, даже самой незначительной, есть четко определенная предметная область (или домен). Это еще один важный термин, который широко используется в литературе, посвященной разработке ПО. Предметная область определяет рамки, в которых проявляются

возможности программы, а также требования, которым эта программа должна удовлетворять.

Для достижения этой цели предметная область использует определенную и заранее подготовленную терминологию (словарь терминов), что помогает разработчикам оставаться в ее рамках. Все участники проекта должны иметь представление о предметной области, в которой он находится.

Например, банковское программное обеспечение обычно создается для строго определенной предметной области. Оно содержит набор общеизвестных терминов, таких как «счет», «кредитные средства», «баланс», «перевод», «заем», «процентная ставка» и т. д.

Определение предметной области можно прояснить с помощью терминов, находящихся в ее словаре; например, в банковской предметной области вы не найдете упоминания таких понятий, как «пациент», «лекарства» и «дозировка».

Если язык программирования не предоставляет средств для работы с концепциями предметной области (такими как *«пациент»* или *«лекарства»* в случае со здравоохранением), написать для нее ПО будет хоть и выполнимой, но, несомненно, сложной задачей. Более того, по мере увеличения количества кода его будет все сложнее развивать и сопровождать.

Отношения между объектами

Объекты могут быть взаимосвязанными; для обозначения этих связей они могут ссылаться друг на друга. Например, если вернуться к нашему примеру с классной комнатой, то объект *student4* (четвертый студент) может иметь отношение к объекту *chair3* (третий стул) в том смысле, что *сидит на нем*. Иными словами, *student4* сидит на *chair3*. Таким образом, все объекты в системе ссылаются друг на друга и формируют сеть, известную как объектная модель. Ранее уже говорилось, что она соответствует диаграмме связей, которую мы держим в уме.

Если между двумя объектами есть связь, то изменение состояния одного из них может сказаться на состоянии другого. Объясню это на примере. Представьте, что у нас есть два объекта, `p1` и `p2`, которые описывают пикселы и не имеют никакого отношения друг к другу.

Объект `p1` содержит такие атрибуты: `{x: 53, y: 345, red: 120, green: 45, blue: 178}`. Атрибуты объекта `p2` следующие: `{x: 53, y: 346, red: 79, green: 162, blue: 23}`.

Синтаксис, который мы здесь использовали, очень похож (хоть и не до конца) на формат JSON. Атрибуты каждого объекта заключены в фигурные скобки и разделены запятыми. Каждый атрибут имеет соответствующее значение, записанное через двоеточие.

Чтобы их связать, нужно использовать дополнительный атрибут, который будет обозначать отношение между ними. Таким образом, состояние объекта `p1` поменяется на `{x: 53, y: 345, red: 120, green: 45, blue: 178, adjacent_down_ pixel: p2}`, а объекта `p2` — на `{x: 53, y: 346, red: 79, green: 162, blue: 23, adjacent_up_pixel: p1}`.

Атрибуты `adjacent_down_pixel` и `adjacent_up_pixel` говорят о том, что эти пикселы смежные; их координаты `y` отличаются всего на единицу. Благодаря этим дополнительным атрибутам объекты могут понять, что связаны между собой. Например, `p1` знает, что его атрибут `adjacent_down_pixel` равен `p2`, а `p2` знает, что его атрибут `adjacent_up_pixel` равен `p1`.

Итак, мы видим, что для установления отношений между объектами их состояние (списки значений, соответствующие их атрибутам) нужно поменять. Для этого объекты расширяются за счет новых атрибутов, и их отношения становятся частью их состояния. Конечно, это имеет последствия для такой характеристики объектов, как изменяемость и неизменяемость.

Обратите внимание: подмножество атрибутов, которые определяют состояние и неизменяемость объекта, может зависеть от предметной области и не обязательно включает в себя все атрибуты. В одной предметной области в качестве состояния могут использоваться только нессылочные атрибуты (в предыдущем примере это `x`, `y`, `red`, `green` и `blue`), а в другой они могут быть объединены вместе со ссылочными (в предыдущем примере это `adjacent_up_pixel` и `adjacent_down_pixel`).

Объектно-ориентированные операции

Объектно-ориентированный язык программирования позволяет планировать создание и уничтожение объекта, а также изменение его состояния еще до запуска программы. Для начала посмотрим, как создается объект.

Более точный термин — «построение», но в технической литературе эту операцию принято называть созданием.

Запланировать создание объекта можно двумя способами.

- Первый заключается в создании либо полностью пустого объекта (без каких-либо атрибутов в его состоянии), либо объекта с минимальным набором атрибутов.

  Остальные атрибуты будут определены и добавлены во время выполнения кода. Таким образом, у одного и того же объекта могут быть разные атрибуты в двух разных сценариях использования программы, в зависимости от изменений, обнаруженных в окружающей среде.

  Мы обращаемся с каждым объектом как с отдельной сущностью, поэтому, даже если два объекта принадлежат к одной и той же группе (или классу) и имеют

общий список атрибутов, по мере выполнения программы атрибуты в их состоянии могут отличаться.

Например, уже упомянутые объекты `p1` и `p2` являются пикселами (то есть принадлежат к одному классу под названием `pixel`), поскольку содержат одинаковые атрибуты (`x`, `y`, `red`, `green` и `blue`). Но после установления отношения в их состоянии появятся разные атрибуты: `adjacent_down_pixel` у `p1` и `adjacent_up_pixel` у `p2`.

Данный подход применяется в таких языках программирования, как JavaScript, Ruby, Python, Perl и PHP. Это в основном *интерпретируемые языки*, и атрибуты в них хранятся в виде *ассоциативного массива* (или *хеш-таблицы*) во внутренних структурах данных, которые можно легко изменять во время выполнения. Эта методика называется *прототипным ООП*.

- Второй подход состоит в создании объектов, атрибуты которых определены заранее и не меняются по ходу выполнения. Новые атрибуты добавлять нельзя, и объект сохранит свою изначальную структуру. Меняться могут только значения атрибутов и лишь в том случае, если объект изменяемый.

  Чтобы использовать эту методику, программист должен заранее подготовить *шаблон* или *класс объекта* с описанием всех атрибутов, которые данный объект должен иметь на этапе выполнения. Затем этот шаблон нужно скомпилировать и передать объектно-ориентированному языку во время работы программы.

  Во многих языках программирования, таких как Java, C++ и Python, этот шаблон называется классом, а сам подход известен как *классовое ООП*. Стоит отметить, что, помимо классового ООП, Python поддерживает и прототипное.

Класс определяет лишь список атрибутов, присутствующих в объекте, но не сами значения, которые присваиваются этим атрибутам во время выполнения.

Обратите внимание: *«объект»* и *«экземпляр»* — взаимозаменяемые понятия. Но иногда в технической литературе они могут иметь незначительные отличия. Есть еще один термин, который заслуживает отдельного упоминания и объяснения, — *«ссылка»*. Объект или экземпляр ссылается на определенное место, выделенное в памяти для его значений, в то время как ссылка похожа на указатель с адресом объекта. Поэтому у нас может быть много ссылок на один и тот же объект. У объекта обычно нет имени, а у ссылки есть.

В языке C в качестве синтаксиса для ссылок используются указатели. У нас также есть стековые объекты и объекты кучи. У стекового объекта нет имени, и обращаться к нему можно с помощью указателей. Для сравнения, объект кучи представляет собой настоящую переменную и, следовательно, имеет имя.

Оба подхода допустимы, но в C и особенно в C++ предусмотрена официальная поддержка классового ООП. Поэтому, когда программист хочет создать объект в C или C++, ему сначала нужно подготовить класс. О классах и их роли в ООП мы поговорим несколько позже.

То, о чем мы поговорим дальше, на первый взгляд имеет мало общего с данной темой, однако на самом деле это не так. Есть две точки зрения на то, как люди взрослеют, и обе они довольно точно совпадают с методами создания объектов, которые обсуждались выше. Согласно одной из них, человек появляется на свет пустым, без сущности (или состояния).

По мере того как он сталкивается с хорошими и плохими событиями в жизни, его сущность начинает расти, превращаясь в нечто имеющее независимый и зрелый характер. *Экзистенциализм* — то философское течение, которое исповедует эту идею.

Одна из его знаменитых догм звучит так: «Существование предшествует сущности». Смысл его в том, что человек сначала появляется на свет, а затем накапливает сущность по мере приобретения жизненного опыта. Эта идея крайне похожа на прототипный подход к созданию объектов, согласно которому объект изначально является пустым и эволюционирует во время выполнения программы.

Другая, более старая философия характерна в основном религиям. Согласно ей, человек создан «по образу и подобию» (или на основе некой сущности), и данный образ был определен еще до появления человека. Это больше всего напоминает то, как мы планируем создание объекта на основе шаблона или класса. Вы как создатель подготавливаете класс, и затем программа начинает создавать из него объекты.

Существует тесная корреляция между подходами, которые персонажи романов и повестей, как художественных, так и исторических, используют для преодоления определенных трудностей, и алгоритмами, с помощью которых в информатике решаются аналогичные задачи. Я глубоко убежден: то, как мы живем и познаем реальность, гармонирует с нашим пониманием алгоритмов и структур данных. То, о чем мы говорили выше, — отличный пример такой гармонии между ООП и философией.

Уничтожение объектов, как и их создание, происходит на этапе выполнения; в нашем коде мы можем его только запланировать. Все ресурсы, выделенные объектом на протяжении его существования, должны быть освобождены в момент его уничтожения. При этом необходимо изменить все связанные с ним объекты, чтобы они больше не ссылались на него. Ни один из атрибутов не должен указывать на уничтоженный объект, иначе наша объектная модель утратит *ссылочную целостность*. Это может привести к ошибкам времени выполнения, таким как повреждение памяти, ошибка сегментации, а также к логическим проблемам наподобие неправильных вычислений.

Объект (или его состояние) может меняться двумя разными путями. Это может быть просто изменение значения уже существующего атрибута или изменение самого списка атрибутов. Второй вариант возможен только при выборе прототипного подхода к созданию объектов. Помните, что модификация состояния неизменяемого объекта запрещена и обычно не допускается в объектно-ориентированных языках.

Объекты имеют поведение

Любой объект вместе со всеми своими атрибутами имеет определенный список операций, которые может выполнять. Например, автомобиль может разгоняться, замедляться, поворачивать и т. д. В ООП эти операции всегда соответствуют требованиям предметной области. Например, в банковской объектной модели клиент может заказать открытие нового счета, но не может пообедать. Конечно, клиент, как и любой другой живой человек, может есть, но если потребление еды не относится к банковской сфере, то для соответствующего объекта данная возможность считается необязательной.

Любая операция может изменить состояние объекта, модифицируя значения его атрибутов. Приведу простой пример. Автомобиль может разгоняться. Разгон — это его возможность. В момент разгона меняется скорость автомобиля (один из его атрибутов).

Таким образом, объект — просто набор атрибутов и операций. В следующих разделах мы более подробно поговорим о том, как объединить все это в один объект.

Итак, мы представили основополагающую терминологию, необходимую для изучения и понимания ООП. Далее рассмотрим такую фундаментальную концепцию, как инкапсуляция. Но прежде прервемся на минуту и порассуждаем о том, почему С не может быть объектно-ориентированным языком.

Почему язык С не является объектно-ориентированным

Язык С не объектно-ориентированный, но это не связано с его возрастом, иначе к текущему моменту мы бы уже нашли способ добавить в него поддержку ООП. Но, как вы сами увидите в главе 12, новейший стандарт данного языка программирования, С18, не пытается сделать его объектно-ориентированным.

С другой стороны, у нас есть С++ — попытка создать на основе С язык с поддержкой ООП. Если бы С было суждено уступить место объектно-ориентированному языку, то на него не было бы спроса в наши дни, в основном из-за

существования C++. Но текущая востребованность программистов на C показывает, что это не так.

Человек мыслит объектно-ориентированным образом, но машинные инструкции, которые выполняет центральный процессор, являются процедурными. Они выполняются одна за другой, хотя иногда процессору приходится переходить по другому адресу и выполнять инструкции, которые там находятся. Это очень напоминает вызовы функций в программе, написанной на процедурном языке, таком как C.

Язык C не может быть объектно-ориентированным, поскольку находится на границе между ООП и процедурным программированием. Объектно-ориентированность — человеческое понимание задачи, но процессор решает ее в процедурном стиле. Поэтому у нас должно быть нечто на стыке этих двух миров. В противном случае высокоуровневые программы, которые обычно написаны с применением ООП, нельзя было бы напрямую транслировать в процедурные инструкции, потребляемые процессором.

Если взять языки программирования высокого уровня, такие как Java, JavaScript, Python, Ruby и пр., то в их архитектуре есть компонент или слой, который связывает их среду выполнения с системной библиотекой C (стандартной библиотекой C в системах семейства Unix и Win32 API в системах Windows). Например, на платформе Java для этого предусмотрена *виртуальная машина Java* (Java Virtual Machine, JVM). И хотя не все эти среды сугубо объектно-ориентированные (например, JavaScript и Python поддерживают как ООП, так и процедурный стиль), им нужен данный слой для перевода их высокоуровневой логики в низкоуровневые процедурные инструкции.

Инкапсуляция

В предыдущих разделах вы увидели, что каждый объект имеет набор атрибутов и операций. Здесь же мы поговорим о том, как объединить эти атрибуты и операции в сущность под названием *«объект»*. Для этого воспользуемся процессом, известным под названием *«инкапсуляция»*.

Инкапсуляция означает объединение отдельных вещей в некую *капсулу*, которая представляет объект. Вначале мы это делаем в нашем воображении и затем воплощаем в коде. Каждый раз, когда вам кажется, что у объекта должны быть какие-то атрибуты и операции, вы производите мысленную инкапсуляцию; данный процесс необходимо реализовать на уровне программы.

Инкапсуляция — неотъемлемая часть языка программирования, без которой объединение связанных между собой переменных превратилось бы в непосильную задачу (я уже упоминал о том, как для этого можно использовать соглашения об именовании).

Объект состоит из набора атрибутов и операций. Все они должны быть инкапсулированы. Сначала поговорим об *инкапсуляции атрибутов*.

Инкапсуляция атрибутов

Как мы уже видели ранее, для объединения и группирования разных переменных в рамках единого объекта можно использовать их имена. Пример этого показан в листинге 6.3.

Листинг 6.3. Переменные, сгруппированные по именам и представляющие два пиксела

```
int pixel_p1_x     = 56;
int pixel_p1_y     = 34;
int pixel_p1_red   = 123;
int pixel_p1_green = 37;
int pixel_p1_blue  = 127;

int pixel_p2_x     = 212;
int pixel_p2_y     = 994;
int pixel_p2_red   = 127;
int pixel_p2_green = 127;
int pixel_p2_blue  = 0;
```

Здесь наглядно показано, как переменные можно сгруппировать в *неявные* объекты `p1` и `p2`, используя их имена. Неявными эти объекты делает тот факт, что об их существовании знает только сам программист; языку программирования о них ничего не известно.

С точки зрения языка программирования это просто десять переменных, не имеющих ничего общего. Это очень низкоуровневая инкапсуляция — вплоть до того, что к ней формально нельзя применить данный термин. Инкапсуляция по именам переменных существует во всех языках программирования (поскольку у переменных всегда есть имена), даже в ассемблере.

Что нам нужно, так это методики, предлагающие *явную* инкапсуляцию атрибутов. То есть о ней и о капсулах (объектах) должен знать как программист, так и язык программирования. Языки, не имеющие подобной возможности, использовать очень сложно.

К счастью, C поддерживает явную инкапсуляцию, и это одна из причин, почему на нем можно более или менее легко писать всевозможные объектно-ориентированные по своей сути программы. С другой стороны, как мы увидим в следующем подразделе, явной инкапсуляции поведения в C нет и ее реализация требует выработки определенных правил и условностей.

Обратите внимание: встроенная поддержка таких возможностей, как инкапсуляция, всегда приветствуется. Это касается многих других аспектов ООП, включая

наследование и полиморфизм. Их реализация на уровне языка позволяет отлавливать соответствующие ошибки на этапе компиляции, еще до выполнения программы.

Решение проблем во время выполнения — настоящий кошмар, поэтому всегда следует стремиться к тому, чтобы выявление ошибок происходило в ходе компиляции. Это главное преимущество объектно-ориентированных языков, которые прекрасно «понимают», как мы мыслим. Язык с поддержкой ООП может сообщать об ошибках и некорректных аспектах нашей архитектуры во время компиляции, избавляя нас от необходимости решать многочисленные серьезные проблемы на этапе выполнения. Именно поэтому мы наблюдаем появление все новых и более сложных языков программирования, которые пытаются сделать все явным.

К сожалению, в C нет возможности явно реализовать все объектно-ориентированные концепции. Вот почему на C так сложно писать код в стиле ООП. В языке C++ с этим дела обстоят лучше, именно поэтому его называют объектно-ориентированным.

В C инкапсуляция происходит за счет структур. Отредактируем код из листинга 6.3 (листинг 6.4).

Листинг 6.4. Структура pixel_t и объявление на ее основе двух переменных

```
typedef struct {
  int x, y;
  int red, green, blue;
} pixel_t;

pixel_t p1, p2;

p1.x = 56;
p1.y = 34;
p1.red = 123;
p1.green = 37;
p1.blue = 127;

p2.x = 212;
p2.y = 994;
p2.red = 127;
p2.green = 127;
p2.blue = 0;
```

В этом коде есть несколько важных моментов.

- Инкапсуляция атрибутов происходит, когда мы помещаем переменные `x`, `y`, `red`, `green` и `blue` в новый тип, `pixel_t`.
- Инкапсуляция всегда заключается в создании нового типа; в частности, в C это относится к атрибутам. Данный факт очень важен. Это то, как мы делаем

инкапсуляцию явной. Обратите внимание на суффикс `_t` в конце `pixel_t`. В C это общепринятый (но необязательный) способ обозначения новых типов. Мы будем использовать его на страницах этой книги.

- Во время выполнения этого кода переменные `p1` и `p2` будут явными объектами. Обе они имеют тип `pixel_t`, и все их атрибуты перечислены в соответствующей структуре. В C и особенно в C++ типы определяют атрибуты объектов.
- Новый тип `pixel_t` описывает лишь атрибуты класса (или шаблона объекта). Но, как вы помните, слово «класс», помимо атрибутов, охватывает и операции. Как следствие, структуры языка C не могут быть аналогами классов. К сожалению, с этим ничего нельзя сделать; атрибуты и операции существуют отдельно и в коде связываются косвенно. В C любой класс является неявным и представляет собой совокупность структуры и списка функций. Мы подробно поговорим об этом в будущих примерах в данной главе и в последующих.
- Как видите, объекты создаются на основе шаблона (то есть структуры `pixel_t`) с набором заранее определенных атрибутов, с которыми объект появляется на свет. И, как уже говорилось выше, структура хранит только атрибуты, но не операции.
- Создание объекта очень похоже на объявление новой переменной. Сначала идет тип, затем имя (в данном случае имя объекта). При объявлении объекта почти одновременно происходит две вещи: сначала для него выделяется память, после чего его атрибуты инициализируются с помощью значений по умолчанию. В предыдущем примере все атрибуты целочисленные, поэтому используется стандартное для языка C значение по умолчанию типа `int`: 0.
- В C, как и во многих других языках программирования, для доступа к атрибутам внутри объекта используется точка (`.`); если доступ к атрибуту структуры осуществляется не напрямую, а через адрес внутри указателя, то применяется стрелка (`->`). Выражение `p1.x` (или `p1->x`, если `p1` является указателем) следует читать как *«атрибут x в объекте p1»*.

Вы уже знаете, что в объектах можно инкапсулировать не только атрибуты. Посмотрим, как выглядит инкапсуляция поведения.

Инкапсуляция поведения

Объект — некая капсула с атрибутами и методами. Метод — еще один стандартный термин, который обычно используется для обозначения части логики или функциональности, хранящейся в объекте. Его можно считать обычной функцией языка C, у которой есть имя, список аргументов и возвращаемый тип. Атрибуты представляют собой *значения*, а методы — *поведение*. Таким образом, объект имеет список значений и может демонстрировать определенное поведение в системе.

В таких классовых объектно-ориентированных языках, как C++, атрибуты и методы легко объединить в класс. В прототипных языках наподобие JavaScript объект, как правило, создается пустым (лат. *ex nihilo* — «из ничего») или клонируется из другого объекта. Чтобы объект имел поведение, в него нужно добавить методы. Следующий пример, написанный на JavaScript, поможет лучше понять, как работают прототипные языки программирования (листинг 6.5).

Листинг 6.5. Создание объекта clientObj в JavaScript

```
// создаем пустой объект
var clientObj = {};

// устанавливаем атрибуты
clientObj.name = "John";
clientObj.surname = "Doe";

// добавляем метод для открытия банковского счета
clientObj.orderBankAccount = function () {
  ...
}

  ...

// вызываем метод
clientObj.orderBankAccount();
```

Во второй строчке данного примера мы создаем пустой объект. В следующих двух строчках добавляем в него два новых атрибута, `name` и `surname`. Дальше добавляем новый метод, `orderBankAccount`, который указывает на определение функции. На самом деле в этой строчке происходит присваивание. Справа находится *анонимная функция*, которая не имеет имени и присваивается атрибуту `orderBankAccount`, указанному слева. Иными словами, мы сохраняем функцию в атрибуте `orderBankAccount`. Это отличная демонстрация прототипных программных языков, в которых все объекты изначально пустые.

В классовом языке предыдущий пример выглядел бы иначе. Вначале мы бы написали класс, поскольку без него нельзя получить объект. В листинге 6.6 показано, как это делается в C++.

Листинг 6.6. Создание объекта clientObj в C++

```
class Client {
public:
  void orderBankAccount() {
    ...
  }
  std::string name;
  std::string surname;
};
```

```
...
Client clientObj;
clientObj.name = "John";
clientObj.surname = "Doe";
...
clientObj.orderBankAccount ();
```

Как видите, все началось с объявления нового класса, `Client`, который сразу же стал новым типом в C++. Он напоминает капсулу и находится внутри фигурных скобок. Дальше мы создали из типа `Client` объект `clientObj`.

В оставшихся строчках мы установили атрибуты и наконец вызвали из `clientObj` метод `orderBankAccount`.

В C++ методы обычно называют функциями-членами, а атрибуты — данными-членами (или просто данными класса).

Если взглянуть на способы инкапсуляции разных элементов, которые применяются в открытых и широко известных проектах на языке C, то можно заметить нечто общее. В оставшейся части этого раздела я предложу вам вариант инкапсуляции поведения, основанный на методиках, встречающихся в таких проектах.

Поскольку мы еще не раз будем возвращаться к данной методике, ей следует дать некое имя. Назовем ее *«неявная инкапсуляция»*. Неявной ее делает то, что поведение в ней инкапсулируется так, что язык C не знает об этом. Исходя из того, что нам доступно в стандарте ANSI C, нет никакой возможности сделать так, чтобы языку C было известно о существовании классов. Поэтому любые попытки реализации объектной ориентированности в C неизбежно оказываются неявными.

Неявная инкапсуляция предполагает следующее.

- Использование структур языка C для атрибутов объекта (явная инкапсуляция атрибутов). Мы будем называть их *структурами атрибутов*.
- Для инкапсуляции поведения в C применяются функции. Это так называемые *поведенческие функции*. Вероятно, вы знаете, что язык C не позволяет размещать функции в структурах. Поэтому подобные функции должны существовать за пределами структуры атрибутов (неявная инкапсуляция поведения).
- В качестве одного из аргументов (обычно первого или последнего) поведенческие функции должны принимать указатель на структуру. Данный указатель ссылается на структуру атрибутов объекта. Дело в том, что поведенческой функции может понадобиться доступ (на чтение или запись) к атрибутам — это довольно частое явление.
- Поведенческие функции должны иметь подходящие имена, которые указывают на их принадлежность к одному и тому же классу объектов. Вот почему при

использовании данного подхода крайне важно придерживаться одного соглашения об именовании. Это одно из двух соглашений, которые мы постараемся соблюдать в данных главах, чтобы сделать инкапсуляцию максимально ясной. Второе — использование суффикса `_t` в именах структур атрибутов. Но, конечно, вас никто не заставляет их задействовать; вы можете выработать собственные соглашения об именовании.

- Объявления поведенческих функций обычно размещаются в одном заголовочном файле с объявлением структуры атрибутов. Этот файл называется *заголовком объявления*.
- Определения поведенческих функций обычно находятся в одном или нескольких отдельных исходных файлах, которые подключают заголовок объявления.

Обратите внимание: при использовании неявной инкапсуляции у нас нет никаких классов, но их существование подразумевается самим программистом. В примере 6.1 (листинг 6.7) показано, как эта методика применяется в настоящей программе на C. Данная программа демонстрирует поведение автомобиля, который разгоняется до тех пор, пока не закончится бензин, а затем останавливается.

Заголовочный файл, приведенный ниже, содержит объявление нового типа, `car_t`, который представляет собой структуру атрибутов класса `Car`. Там же находятся объявления поведенческих функций этого класса. Под классом `Car` мы понимаем неявный класс, который на самом деле в коде отсутствует, но объединяет в себе структуру атрибутов и поведенческие функции.

Листинг 6.7. Объявления структуры атрибутов и поведенческих функций класса Car (ExtremeC_examples_chapter6_1.h)

```
#ifndef EXTREME_C_EXAMPLES_CHAPTER_6_1_H
#define EXTREME_C_EXAMPLES_CHAPTER_6_1_H

// эта структура содержит все атрибуты, относящиеся к объекту car
typedef struct {
  char name[32];
  double speed;
  double fuel;
} car_t;

// это объявления функций, которые составляют поведение объекта car
void car_construct(car_t*, const char*);
void car_destruct(car_t*);
void car_accelerate(car_t*);
void car_brake(car_t*);
void car_refuel(car_t*, double);

#endif
```

Как видите, структура атрибутов `car_t` имеет три поля: `name`, `speed` и `fuel`, которые являются атрибутами автомобиля. Обратите внимание: `car_t` — новый тип

в языке C, и на его основе теперь можно объявлять переменные. Поведенческие функции обычно объявляются в том же заголовочном файле, и это показано в листинге выше. Их имена начинаются с префикса `car_`, сигнализируя о том, что все они принадлежат к одному классу.

У неявной инкапсуляции есть очень важная особенность: каждая структура атрибутов относится к отдельному объекту, но все объекты разделяют одни и те же поведенческие функции. Иными словами, для каждого объекта создается своя переменная на основе структуры атрибутов, но поведенческие функции существуют в единственном экземпляре и вызываются из разных объектов.

Обратите внимание: сама по себе структура атрибутов `car_t` не является классом `Car`. Она просто содержит его атрибуты. Класс `Car` — совокупность всех объявлений. Больше примеров этого будет показано ниже.

Многие знаменитые проекты с открытым исходным кодом используют этот подход для написания частично объектно-ориентированного кода. Один из них — библиотека `libcurl`. Если взглянуть на ее исходный код, то можно увидеть много структур и функций, имена которых начинаются с `curl_`. Список таких функций можно найти по адресу https://curl.haxx.se/libcurl/c/allfuncs.html.

Исходный файл, показанный в листинге 6.8, содержит определение поведенческих функций, объявленных в примере 6.1.

Листинг 6.8. Определения поведенческих функций, входящих в класс Car (ExtremeC_examples_chapter6_1.c)

```
#include <string.h>

#include "ExtremeC_examples_chapter6_1.h"

// определения подключенных выше функций
void car_construct(car_t* car, const char* name) {
  strcpy(car->name, name);
  car->speed = 0.0;
  car->fuel = 0.0;
}

void car_destruct(car_t* car) {
  // Здесь ничего не происходит!
}

void car_accelerate(car_t* car) {
  car->speed += 0.05;
  car->fuel -= 1.0;
  if (car->fuel < 0.0) {
    car->fuel = 0.0;
  }
}
```

```
void car_brake(car_t* car) {
  car->speed -= 0.07;
  if (car->speed < 0.0) {
    car->speed = 0.0;
  }
  car->fuel -= 2.0;
  if (car->fuel < 0.0) {
    car->fuel = 0.0;
  }
}

void car_refuel(car_t* car, double amount) {
  car->fuel = amount;
}
```

В данном листинге определены поведенческие функции класса `Car`. И вы можете видеть, что все они принимают в качестве первого аргумента указатель `car_t`. Это позволяет им считывать и модифицировать атрибуты объекта. Если функция не принимает указатель на структуру атрибутов, то ее можно считать обычной функцией языка C, которая не представляет поведение объекта.

Обратите внимание: объявления поведенческих функций и соответствующей структуры атрибутов обычно находятся рядом. Причина в том, что за соответствие этих двух сущностей полностью отвечает программист, и поддержка данного кода должна быть достаточно легкой. Вот почему совместное хранение этих объявлений, обычно в одном заголовочном файле, помогает поддерживать в порядке общую структуру класса и облегчает его обслуживание в будущем.

В листинге 6.9 показан исходный файл с функцией `main`, в которой содержится основная логика. Все поведенческие функции используются здесь.

Листинг 6.9. Главная функция в примере 6.1 (ExtremeC_examples_chapter6_1_main.c)

```
#include <stdio.h>

#include "ExtremeC_examples_chapter6_1.h"

// главная функция
int main(int argc, char** argv) {

  // создаем переменную объекта
  car_t car;

  // создаем объект
  car_construct(&car, "Renault");

  // основной алгоритм
  car_refuel(&car, 100.0);
  printf("Car is refueled, the correct fuel level is %f\n", car.fuel);
  while (car.fuel > 0) {
```

```
    printf("Car fuel level: %f\n", car.fuel);
    if (car.speed < 80) {
      car_accelerate(&car);
      printf("Car has been accelerated to the speed: %f\n", car.speed);
    } else {
      car_brake(&car);
      printf("Car has been slowed down to the speed: %f\n", car.speed);
    }
  }

  printf("Car ran out of the fuel! Slowing down ...\n");
  while (car.speed > 0) {
    car_brake(&car);
    printf("Car has been slowed down to the speed: %f\n", car.speed);
  }

  // уничтожаем объект
  car_destruct(&car);

  return 0;
}
```

В качестве первой инструкции в функции `main` мы объявили переменную `car` типа `car_t`. Это наш первый объект `car`. В той же строке мы выделили память для атрибутов данного объекта. Сам он создается в следующей строчке; здесь же происходит инициализация его атрибутов. Объект можно инициализировать только после выделения памяти для его атрибутов. Конструктор принимает название автомобиля в качестве второго аргумента. Вы, вероятно, заметили, что адрес объекта `car` передается всем поведенческим функциям вида `car_*`.

Вслед за этим начинается цикл `while`, в котором функция `main` читает атрибут `fuel` и проверяет, больше ли нуля его значение. Тот факт, что функция `main`, не относящаяся к поведенческим, может обращаться к атрибутам `car` (для чтения и записи), играет важную роль. Атрибут `fuel` и `speed` — пример *публичных* атрибутов, доступ к которым имеют не только поведенческие функции, но и внешний код. Мы еще вернемся к этому замечанию в следующем подразделе.

Прежде чем покинуть функцию `main` и завершить программу, мы уничтожили объект `car`. Это просто означает, что на данном этапе были освобождены ресурсы, выделенные объектом. У нас не было нужды вмешиваться в процесс, но в ходе уничтожения объекта необходимо выполнить определенные шаги. Более подробно об этом мы поговорим в примерах ниже. Этап уничтожения обязателен и предотвращает утечки памяти в случае использования кучи.

Было бы неплохо посмотреть на то, как данный пример можно переписать на языке C++. Это может помочь вам разобраться в том, как языки с поддержкой ООП понимают классы и объекты и как это помогает сделать объектно-ориентированный код более компактным.

В листинге 6.10, который является частью примера 6.2, показан заголовочный файл с классом `Car`, написанным на C++.

Листинг 6.10. Объявление класса Car в C++ (ExtremeC_examples_chapter6_2.h)

```
#ifndef EXTREME_C_EXAMPLES_CHAPTER_6_2_H
#define EXTREME_C_EXAMPLES_CHAPTER_6_2_H

class Car {
public:
  // конструктор
  Car(const char*);
  // деструктор
  ~Car();

  void Accelerate();
  void Brake();
  void Refuel(double);

  // данные класса (то, что в C зовется атрибутами)
  char name[32];
  double speed;
  double fuel;
};

#endif
```

Главная особенность данного кода — тот факт, что C++ знает о существовании классов. Поэтому здесь мы видим явную инкапсуляцию как атрибутов, так и поведения. Более того, C++ поддерживает и другие объектно-ориентированные концепции, включая конструкторы и деструкторы.

В коде на C++ объявление атрибутов и поведения инкапсулировано в определении класса. Это явная инкапсуляция. Взгляните на первые две функции, объявленные в качестве конструктора и деструктора. Язык C не поддерживает такие абстракции, но в C++ для них предусмотрены специальные обозначения. Например, имя деструктора начинается с ~ и совпадает с именем класса.

Кроме того, поведенческие функции не принимают в качестве первого аргумента указатель на объект, поскольку все они имеют доступ к атрибутам внутри класса. В листинге 6.11 показано содержимое исходного файла с определением объявленных поведенческих функций.

Листинг 6.11. Определение класса Car в C++ (ExtremeC_examples_chapter6_2.cpp)

```
#include <string.h>

#include "ExtremeC_examples_chapter6_2.h"
```

```
Car::Car(const char* name) {
  strcpy(this->name, name);
  this->speed = 0.0;
  this->fuel = 0.0;
}

Car::~Car() {
  // здесь ничего не происходит
}

void Car::Accelerate() {
  this->speed += 0.05;
  this->fuel -= 1.0;
  if (this->fuel < 0.0) {
    this->fuel = 0.0;
  }
}

void Car::Brake() {
  this->speed -= 0.07;
  if (this->speed < 0.0) {
    this->speed = 0.0;
  }
  this->fuel -= 2.0;
  if (this->fuel < 0.0) {
    this->fuel = 0.0;
  }
}

void Car::Refuel(double amount) {
  this->fuel = amount;
}
```

Если присмотреться, то можно заметить, что указатель `car`, который мы использовали в коде на C, был заменен указателем `this`. Это ключевое слово языка C++, которое просто указывает на текущий объект. Я не стану углубляться в детали, но это элегантное решение, которое позволяет избавиться от аргумента-указателя и упрощает поведенческие функции.

И наконец, в листинге 6.12 показана функция `main`, которая использует описанный выше класс.

Листинг 6.12. Главная функция в примере 6.2 (ExtremeC_examples_chapter6_2_main.cpp)

```
// имя файла: ExtremeC_examples_chapter6_2_main.cpp
// описание: главная функция

#include <iostream>

#include "ExtremeC_examples_chapter6_2.h"
```

```
// главная функция
int main(int argc, char** argv) {

  // создаем переменную объекта и вызываем конструктор
  Car car("Renault");

  // основной алгоритм
  car.Refuel(100.0);
  std::cout << "Car is refueled, the correct fuel level is "
    << car.fuel << std::endl;
  while (car.fuel > 0) {
    std::cout << "Car fuel level: " << car.fuel << std::endl;
    if (car.speed < 80) {
      car.Accelerate();
      std::cout << "Car has been accelerated to the speed: "
        << car.speed << std::endl;
    } else {
      car.Brake();
      std::cout << "Car has been slowed down to the speed: "
        << car.speed << std::endl;
    }
  }

  std::cout << "Car ran out of the fuel! Slowing down ..."
    << std::endl;
  while (car.speed > 0) {
    car.Brake();
    std::cout << "Car has been slowed down to the speed: "
      << car.speed << std::endl;
  }
  std::cout << "Car is stopped!" << std::endl;

  // при выходе из функции объект 'car' автоматически уничтожается
  return 0;
}
```

Эта функция `main` мало чем отличается от той, которую мы написали на C, если не считать того, что она выделяет память для переменной класса, а не переменной структуры.

В C атрибуты и поведенческие функции нельзя объединить в абстракцию на уровне языка. Вместо этого их приходится группировать с помощью файлов. Но в C++ для этого есть специальный синтаксис, *определение класса*, который позволяет объединять данные-члены (атрибуты) и функции-члены (поведение).

Поскольку C++ знает об инкапсуляции, нам не нужно передавать поведенческим функциям аргумент-указатель. Действительно, в объявлениях наших функций-членов нет обязательного первого аргумента, как в нашей предыдущей версии класса `Car`, написанной на C.

Подытожим. Мы написали объектно-ориентированную программу на двух языках программирования: процедурном C и объектно-ориентированном C++. Основное отличие заключается в использовании `car.Accelerate()` вместо `car_accelerate(&car)` и `car.Refuel(1000.0)` вместо `car_refuel(&car, 1000.0)`.

Иными словами, если в процедурном языке программирования мы делаем вызов вида `func(obj, a, b, c, ...)`, то в объектно-ориентированном это можно записать так: `obj.func(a, b, c, ...)`. Выражения эквивалентны, но принадлежат к разным парадигмам. Как уже говорилось ранее, существует множество проектов на языке C, которые используют данный подход.

В главе 9 вы увидите, что точно такой же подход применяется для перевода высокоуровневых вызовов функций языка C++ в низкоуровневые вызовы на языке C.

В заключение следует отметить: C и C++ существенно различаются тем, как происходит уничтожение объектов. В C++ функция-деструктор вызывается автоматически, когда объект, выделенный на вершине стека, выходит из области видимости, как любая другая стековая переменная. Это большое достижение в управлении памятью, поскольку в C вы можете легко забыть вызвать функцию-деструктор и в конечном счете получить утечку памяти.

Теперь пришло время поговорить о других аспектах инкапсуляции. В следующем подразделе речь пойдет о ее побочном эффекте: принципе сокрытия.

Принцип сокрытия информации

Вы уже знаете, как инкапсуляция объединяет атрибуты (которые представляют значения) и операции (которые представляют поведение) в одном объекте. Но это еще не все.

У инкапсуляции есть еще одно важное назначение или следствие — *сокрытие информации*. Это принцип, защищающий (или скрывающий) те или иные атрибуты и аспекты поведения, которые не должны быть видны извне. Под «извне» имеется в виду любой код, не относящийся к поведению объекта. Согласно данному определению, к приватному атрибуту или поведению объекта, которые не являются частью публичного интерфейса класса, не могут обращаться никакие другие функции.

Обратите внимание: поведенческие функции двух объектов одного типа, таких как `car1` и `car2`, полученных из класса `Car`, имеют доступ к атрибутам любого объекта того же типа. Это обусловлено тем фактом, что поведенческие функции создаются в одном экземпляре сразу для всех объектов класса.

В примере 6.1 мы видели, что функция `main` легко обращалась к полям `speed` и `fuel` структуры атрибутов `car_t`. Это значит, все атрибуты в типе `car_t` были публичными. Наличие публичных атрибутов или поведенческих функций может иметь отрицательные долгосрочные последствия, такие как, например, утечка деталей реализации.

Допустим, вы собираетесь использовать объект `car`. Вам важно только то, что у него есть поведение, которое позволяет разгонять автомобиль, и неинтересно, как это реализовано. В процессе разгона могут участвовать и другие внутренние атрибуты объекта, но делать их доступными для клиентской логики нет никакого смысла.

Например, роль атрибута может играть сила электрического тока, поданного на стартер, но она должна быть изолирована внутри объекта. Это относится и к определенным аспектам поведения объекта. Скажем, впрыск топлива в камеру сгорания — внутреннее поведение, которое не должно быть доступно пользователям, иначе они могут повлиять на него и нарушить работу двигателя.

С другой стороны, детали реализации (как работает автомобиль) могут варьироваться от одного автопроизводителя к другому, но возможность разгона присутствует в любом автомобиле. Принято говорить, что разгон — часть *публичного API*, или *публичного интерфейса*, класса `Car`.

Обычно код, который использует объект, становится зависимым от публичных атрибутов и поведения данного объекта. Это серьезная проблема. Если атрибуты, являющиеся частью публичного интерфейса, сделать приватными, то это может фактически нарушить сборку зависимого кода. После такого изменения все остальные участки кода, которые используют данные атрибуты публично, должны перестать компилироваться. Это означает нарушение обратной совместимости. Вот почему мы предпочитаем осторожный подход и изначально оставляем каждый атрибут приватным, пока не найдем разумной причины сделать его публичным.

Проще говоря, открытие доступа к внутреннему коду класса фактически означает, что мы теперь зависим не от легковесного публичного интерфейса, а от «толстой» реализации. Последствия этого могут быть серьезными и потребовать переписывания существенной части проекта. Поэтому атрибуты и поведение необходимо делать настолько приватными, насколько возможно.

Листинг 6.13 является частью примера 6.3 и демонстрирует реализацию приватных атрибутов и поведенческих функций на языке C. Рассматривается класс `List`, предназначенный для хранения целочисленных значений.

Листинг 6.13. Публичный интерфейс класса List (ExtremeC_examples_chapter6_3.h)

```
#ifndef EXTREME_C_EXAMPLES_CHAPTER_6_3_H
#define EXTREME_C_EXAMPLES_CHAPTER_6_3_H
```

```
#include <unistd.h>

// структура атрибутов без публично доступных полей
struct list_t;

// функция выделения памяти
struct list_t* list_malloc();

// конструктор и деструктор
void list_init(struct list_t*);
void list_destroy(struct list_t*);

// публичные поведенческие функции
int list_add(struct list_t*, int);
int list_get(struct list_t*, int, int*);
void list_clear(struct list_t*);
size_t list_size(struct list_t*);
void list_print(struct list_t*);

#endif
```

Здесь вы можете видеть, как мы делаем атрибуты приватными. Если другой исходный файл (например, содержащий функцию `main`) подключит данный заголовок, то не будет иметь доступ к атрибутам внутри типа `list_t`. Это легко объяснить: `list_t` — просто объявление без определения, которое не позволяет обращаться к полям структуры. Вы даже не сможете объявить переменную на основе данного типа. Таким образом, мы соблюдаем принцип сокрытия. Это большое достижение.

Напомним: перед созданием и публикацией заголовочного файла обязательно следует перепроверить, что в нем должно быть публичным, а что — нет. Открывая доступ к публичному поведению или атрибуту, вы создаете зависимости, нарушение которых будет стоить вам дополнительных усилий, времени и в конечном счете денег.

В коде листинга 6.14 показано определение структуры атрибутов `list_t`. Заметьте, что оно находится в исходном файле, а не в заголовке.

Листинг 6.14. Определение класса List (ExtremeC_examples_chapter6_3.c)

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

#define MAX_SIZE 10

// создаем псевдоним bool_t
typedef int bool_t;

// определяем тип list_t
typedef struct {
  size_t size;
```

```
  int* items;
} list_t;

// приватное поведение, которое проверяет, заполнен ли список
bool_t __list_is_full(list_t* list) {
  return (list->size == MAX_SIZE);
}

// еще одно приватное поведение для проверки индекса
bool_t __check_index(list_t* list, const int index) {
  return (index >= 0 && index <= list->size);
}

// выделяет память для объекта list
list_t* list_malloc() {
  return (list_t*)malloc(sizeof(list_t));
}

// конструктор объекта list
void list_init(list_t* list) {
  list->size = 0;
  // выделяет из кучи
  list->items = (int*)malloc(MAX_SIZE * sizeof(int));
}

// деструктор объекта list
  void list_destroy(list_t* list) {
  // освобождает выделенную память
  free(list->items);
}

int list_add(list_t* list, const int item) {
  // использование приватного поведения
  if (__list_is_full(list)) {
    return -1;
  }
  list->items[list->size++] = item;
  return 0;
}

int list_get(list_t* list, const int index, int* result) {
  if (__check_index(list, index)) {
    *result = list->items[index];
    return 0;
  }
  return -1;
}

void list_clear(list_t* list) {
  list->size = 0;
}
```

```
size_t list_size(list_t* list) {
  return list->size;
}

void list_print(list_t* list) {
  printf("[");
  for (size_t i = 0; i < list->size; i++) {
    printf("%d ", list->items[i]);
  }
  printf("]\n");
}
```

Все определения, приведенные в этом листинге, являются приватными. Внешней логике, которая будет использовать объект `list_t`, ничего о них неизвестно, и полагаться она может только на код в заголовочном файле.

Заметьте, что в данном листинге не было подключено никаких заголовков! Совпадений сигнатур функций с объявлениями в заголовочном файле достаточно. Но все же заголовки рекомендуется подключать, поскольку это гарантирует совместимость между объявлениями и соответствующими определениями. Как вы уже видели в главе 2, исходные файлы компилируются отдельно и только потом объединяются.

На самом деле компоновщик формирует программу из сочетания приватных определений и публичных объявлений.

Приватные поведенческие функции можно обозначать и по-другому. Мы можем использовать в их именах префикс __. Например, функция __check_index — приватная. Обратите внимание: у приватных функций нет соответствующих объявлений в заголовочном файле.

Листинг 6.15 содержит функцию `main` из примера 6.3, которая создает два списка, заполняет первый из них и затем использует второй список для хранения тех же элементов, но в обратном порядке. В конце выводится их содержимое.

Листинг 6.15. Главная функция примера 6.3 (ExtremeC_examples_chapter6_3_main.c)

```
#include <stdlib.h>

#include "ExtremeC_examples_chapter6_3.h"

int reverse(struct list_t* source, struct list_t* dest) {
  list_clear(dest);
  for (size_t i = list_size(source) - 1; i >= 0; i--) {
    int item;
    if(list_get(source, i, &item)) {
      return -1;
```

```
    }
    list_add(dest, item);
  }
  return 0;
}

int main(int argc, char** argv) {
  struct list_t* list1 = list_malloc();
  struct list_t* list2 = list_malloc();

  // создание объектов
  list_init(list1);
  list_init(list2);

  list_add(list1, 4);
  list_add(list1, 6);
  list_add(list1, 1);
  list_add(list1, 5);

  list_add(list2, 9);

  reverse(list1, list2);

  list_print(list1);
  list_print(list2);

  // уничтожение объектов
  list_destroy(list1);
  list_destroy(list2);

  free(list1);
  free(list2);
  return 0;
}
```

Здесь вы можете видеть функции `main` и `reverse`, написанные исходя лишь из того, что было объявлено в заголовочном файле. Иными словами, эти функции используют только публичный API класса `List`: объявления структуры атрибутов `list_t` и его поведенческих функций. Это хорошая демонстрация того, как избавиться от зависимостей и скрыть детали реализации от других участков кода.

Используя публичный API, можно написать программу, которая компилируется, но не работает, если не предоставить и не скомпоновать соответствующие объектные файлы с приватным кодом.

Более подробно обсудим некоторые аспекты приведенного выше листинга. Чтобы выделить память для объекта `list_t`, нам нужна функция `list_malloc`. Если мы

закончим работать с объектом, то выделенную для него память можно освободить с помощью функции `free`.

В этом примере нельзя напрямую использовать функцию `malloc`. Если вызывать ее прямо в функции `main`, то необходимо указать количество выделяемых байтов, используя выражение `sizeof(list_t)`. Но `sizeof` умеет работать с неполными типами.

Тип `list_t`, подключенный из заголовочного файла, является *неполным*, поскольку это просто объявление, которое не несет в себе никакой информации о внутренних полях, и в момент компиляции мы не знаем его размер. Настоящий размер определяется только на этапе компоновки, когда становятся известными детали реализации. Чтобы решить эту проблему, мы определили функцию `list_malloc` и использовали `malloc` лишь там, где можно было вычислить `sizeof(list_t)`.

Чтобы собрать пример 6.3, нам нужно сначала скомпилировать его исходные файлы. Представленная в терминале 6.1 команда генерирует необходимые объектные файлы, которые затем следует скомпоновать.

Терминал 6.1. Компиляция примера 6.3

```
$ gcc -c ExtremeC_examples_chapter6_3.c -o private.o
$ gcc -c ExtremeC_examples_chapter6_3_main.c -o main.o
```

Как видите, мы скомпилировали приватную часть кода в `private.o`, а главную — в `main.o`. Напомню, что заголовочные файлы не компилируют. Публичные объявления, которые в них содержатся, становятся частью объектного файла `main.o`.

Теперь нам нужно скомпоновать эти объектные файлы, поскольку исполняемую программу нельзя получить только из `main.o`. Если все же попытаться это сделать, то возникнут ошибки, показанные в терминале 6.2.

Терминал 6.2. Попытка скомпоновать пример 6.3 на основе одного файла main.o

```
$ gcc main.o -o ex6_3.out
main.o: In function 'reverse':
ExtremeC_examples_chapter6_3_main.c:(.text+0x27): undefined
reference to 'list_clear'
...
main.o: In function 'main':
ExtremeC_examples_chapter6_3_main.c:(.text+0xa5): undefined
reference to 'list_malloc'
...
collect2: error: ld returned 1 exit status
$
```

Здесь видно, что компоновщику не удается найти определения функций, объявленных в заголовочном файле. Скомпоновать этот пример следует так (терминал 6.3).

Терминал 6.3. Компоновка и запуск примера 6.3

```
$ gcc main.o private.o -o ex6_3.out
$ ./ex6_3.out
[4 6 1 5 ]
[5 1 6 4 ]
$
```

Но что произойдет, если поменять реализацию класса `List`?

Представьте: вместо массива мы задействовали связный список. Похоже, нам не пришлось бы заново генерировать `main.o`, поскольку этот файл не зависит от деталей реализации списка, который используется в нем. Таким образом, нам нужно было бы скомпилировать новый объектный файл только для обновленной реализации; например, `private2.o`. Дальше следовало бы еще раз скомпоновать объектные файлы и получить новую исполняемую программу (терминал 6.4).

Терминал 6.4. Компоновка и запуск примера 6.3 с другой реализацией класса List

```
$ gcc main.o private2.o -o ex6_3.out
$ ./ex6_3.out
[4 6 1 5 ]
[5 1 6 4 ]
$
```

С точки зрения пользователя ничего не изменилось, но внутренняя реализация была заменена. Это важное достижение, и данный подход активно применяется в проектах на языке C.

Но что, если нам не хочется повторять этап компоновки при каждом изменении реализации списка? В таком случае для хранения приватного объектного файла можно использовать разделяемую библиотеку (файл `.so`). Ее можно загружать динамически во время выполнения, что избавляет нас от необходимости заново компоновать исполняемый файл. Разделяемые библиотеки обсуждались в главе 3.

На этом данная глава подходит к концу, но мы продолжим рассматривать начатую в ней тему. Следующие две главы посвящены отношениям, которые могут существовать между двумя классами.

Резюме

В этой главе мы:

- подробно рассмотрели объектно-ориентированную философию и то, как получить объектную модель из диаграммы связей;

- познакомились с понятием предметной области и увидели, как с ее помощью оставить в диаграмме связей только нужные нам концепции и идеи;
- поговорили об атрибутах и поведении отдельного объекта, а также о том, как их можно сформировать на основе диаграммы связей или требований, перечисленных в описании предметной области;
- выяснили, почему язык C не может быть объектно-ориентированным, и исследовали его роль в переводе ООП-программ в низкоуровневые ассемблерные инструкции, которые на самом деле выполняет центральный процессор;
- обсудили одну из концепцией ООП — инкапсуляцию. Мы используем ее для создания капсул (объектов), содержащих набор атрибутов (вместо которых подставляются значения) и поведенческих функций (вместо которых подставляется логика);
- поговорили о принципе сокрытия, включая то, как на его основе создаются интерфейсы (или API), которые можно использовать, не задействуя зависимость от внутренней реализации;
- увидели в ходе обсуждения принципа сокрытия, как атрибуты и методы в коде на C можно делать приватными.

Следующая глава положит начало теме возможных отношений между классами. В ней мы поговорим о композиции, а в главе 8 перейдем к наследованию и полиморфизму.

7 Композиция и агрегация

В предыдущей главе мы обсуждали инкапсуляцию и принцип сокрытия. Здесь же продолжим тему объектной ориентированности в языке C и уделим внимание различным отношениям, которые могут существовать между двумя классами. В конечном счете это поможет расширять наши объектные модели и выражать связи между объектами в рамках следующих глав.

Мы обсудим следующие темы.

- Типы отношений, которые могут существовать между двумя объектами и их соответствующими классами. Мы поговорим об отношениях типа *«иметь»* и *«быть»*, но основное внимание будет уделено первому типу.
- Первой разновидностью отношений типа *«иметь»* будет *композиция*. Мы покажем пример настоящей композиции двух классов. С его помощью мы исследуем структуру памяти, характерную для композиции.
- Второй разновидностью отношений типа *«иметь»* будет *агрегация*. Она похожа на композицию, поскольку принадлежит к тому же типу отношений. Но имеет и свои особенности. Мы рассмотрим отдельный пример использования агрегации. Разница между агрегацией и композицией отчетливо проявляется в том, как они структурируют память.

Это вторая из четырех глав, посвященных ООП в C. Отношение типа *«быть»*, известное как *наследование*, будет рассмотрено в главе 8.

Отношения между классами

Объектная модель — набор взаимосвязанных объектов. Количество связей в модели может быть огромным, но отношения между двумя объектами (или их соответствующими классами) могут быть организованы лишь несколькими способами. Существует две основные категории отношений: *«иметь»* и *«быть»*. Первую мы подробно обсудим в этой главе, а вторую — в следующей. Мы также поговорим о том, как взаимоотношения объектов могут влиять на отношения между их соответствующими классами. Но сначала следует провести четкую границу между классом и объектом.

Объекты и классы

Как вы помните, существует два подхода к созданию объектов: *прототипный* и *классовый*.

В рамках прототипного подхода объект изначально создается пустым (без каких-либо атрибутов и без поведения) или клонируется из существующего объекта. В этом контексте *экземпляр* и *объект* означают одно и то же. Таким образом, можно сказать, что прототипный подход основан на объектах; то есть все начинается с пустых объектов, а не с классов.

В случае с классовым подходом мы не можем создать объект без «схемы», известной как *класс*. Поэтому сначала пишем класс, а затем создаем из него объект. В предыдущей главе мы рассмотрели методику неявной инкапсуляции, с точки зрения которой класс представляет собой набор объявлений, находящихся в заголовочном файле. Мы также показали несколько примеров того, как это работает в языке C.

В текущем разделе мы поговорим о различиях между классом и объектом. Они могут показаться очевидными, но мы постараемся изучить их на более глубоком уровне. Начнем с практического примера.

Представьте, что мы определили класс `Person` с атрибутами `name`, `surname` и `age`. Мы не станем останавливаться на его поведении, так как основные отличия связаны именно с атрибутами.

В языке C мы можем написать этот класс с помощью публичных атрибутов (листинг 7.1).

Листинг 7.1. Структура атрибутов Person на языке C

```
typedef struct {
  char name[32];
  char surname[32];
  unsigned int age;
} person_t;
```

В C++ это будет выглядеть следующим образом (листинг 7.2).

Листинг 7.2. Класс Person на языке C++

```
class Person {
public:
  std::string name;
  std::string family;
  uint32_t age;
};
```

Оба листинга почти идентичны. На самом деле все, о чем мы здесь говорим, применимо и к C, и к C++, и даже к другим объектно-ориентированным языкам, таким

как Java. Класс (или шаблон объекта) — это схема, которая определяет лишь то, какие атрибуты должны присутствовать в каждом объекте, но *не* их значения. В действительности у каждого объекта есть отдельный набор значений для атрибутов, которые имеются в других объектах, полученных из того же класса.

Когда объект создается из класса, для него сначала выделяется память. Это место, в котором будут размещаться значения атрибутов. Затем атрибуты нужно инициализировать. Это важный этап, без которого свежесозданный объект может оказаться в некорректном состоянии.

За создание объекта обычно отвечает отдельная функция, которую называют *конструктором*. В предыдущей главе примерами конструкторов были функции `list_init` и `car_construct`. Вполне возможно, что во время создания объекта необходимо выделить дополнительную память для таких ресурсов, как другие объекты, буферы, массивы, потоки и т. д. Это ресурсы, от которых зависит (или которыми владеет) объект, и их следует освобождать перед его уничтожением.

У нас есть еще одна функция, похожая на конструктор, которая отвечает за освобождение выделенных ресурсов. Она называется *деструктором*. Точно так же, если вернуться к предыдущей главе, примерами деструкторов были функции `list_destroy` и `car_destruct`. После уничтожения объекта освобождается его память, но перед этим необходимо позаботиться об освобождении памяти, которую занимают все его ресурсы.

Прежде чем двигаться дальше, подытожим все вышесказанное:

- класс — это схема, которая используется в качестве «чертежа» для создания объектов;
- из одного класса можно создать много объектов;
- класс определяет, какие атрибуты должны присутствовать в каждом объекте, создаваемом на его основе. Но он не имеет никакого отношения к значениям, которые могут быть присвоены этим атрибутам;
- сам класс не потребляет никакой памяти (по крайней мере не в C или C++) и существует лишь в виде исходного кода и только на этапе компиляции. Но объекты существуют во время выполнения и потребляют память;
- создание объекта начинается с выделения памяти. Освобождение памяти — последняя операция, которую выполняет объект;
- объект должен создаваться сразу после выделения памяти и уничтожаться непосредственно перед ее освобождением;
- объект может владеть частью ресурсов наподобие потоков, буферов, массивов и пр., которые должны быть освобождены перед его уничтожением.

Теперь, разобравшись с различиями между классами и объектами, мы можем продолжить наше обсуждение. Рассмотрим разные виды отношений, которые могут

существовать между двумя объектами и их соответствующими классами. Начнем с композиции.

Композиция

Отношение «композиция» между двумя объектами означает, что один из них обладает другим. Иными словами, один объект состоит из другого.

Например, у автомобиля есть двигатель; то есть объект «автомобиль» содержит объект «двигатель». Таким образом, эти два объекта имеют отношение типа «композиция». При его использовании должно выполняться одно важное условие: *время жизни внутреннего объекта привязано к времени жизни внешнего объекта (контейнера)*.

Если существует контейнер, то должен существовать и внутренний объект. Вместе с тем последний необходимо уничтожить непосредственно перед уничтожением контейнера. Это условие часто подразумевает, что внутренний объект является приватным для контейнера.

Некоторые элементы контейнера могут быть доступны через публичный интерфейс (или поведенческие функции) его класса, но время жизни внутренних объектов должно определяться самим контейнером. Если какой-то участок кода может уничтожить внутренний объект, не уничтожая внешний, то отношение между объектами больше нельзя считать композицией.

В примере 7.1 показана композиция объектов «автомобиль» и «двигатель». Он состоит из пяти файлов: двух заголовочных, которые объявляют публичные интерфейсы классов `Car` и `Engine`, двух исходных, содержащих реализацию этих классов, и еще одного исходного файла с функцией `main`, выполняющей простой сценарий с объектами `car` и `engine`.

Следует отметить, что в некоторых областях двигатель может находиться за пределами автомобиля; например, в системе автоматизированного проектирования для машиностроения. Поэтому типы отношений между различными объектами определяются проблемной областью. Чтобы не усложнять наш пример, представим такую область, в которой двигатель не может существовать вне автомобиля.

В листинге 7.3 показан заголовочный файл для класса `Car`.

Листинг 7.3. Публичный интерфейс класса Car (ExtremeC_examples_chapter7_1_car.h)

```
#ifndef EXTREME_C_EXAMPLES_CHAPTER_7_1_CAR_H
#define EXTREME_C_EXAMPLES_CHAPTER_7_1_CAR_H

struct car_t;
```

```
// аллокатор памяти
struct car_t* car_new();

// конструктор
void car_ctor(struct car_t*);

// деструктор
void car_dtor(struct car_t*);

// поведенческие функции
void car_start(struct car_t*);
void car_stop(struct car_t*);
double car_get_engine_temperature(struct car_t*);

#endif
```

Как видите, объявления здесь делаются по тому же принципу, что и в примере 6.3 с классом `List`, приведенном в главе 6. Разница в том, что мы выбрали другой суффикс для конструктора: `car_new` вместо `car_construct`. Еще одно отличие состоит в объявлении структуры атрибутов `car_t`. Мы не определили ее поля — это называется *предварительным объявлением*. Определение структуры `car_t` будет находиться в исходном файле, представленном в листинге 7.5. Обратите внимание: в этом заголовке `car_t` считается неполным типом, который еще не определен.

В листинге 7.4 показан заголовочный файл для класса `Engine`.

Листинг 7.4. Публичный интерфейс класса Engine (ExtremeC_examples_chapter7_1_engine.h)

```
#ifndef EXTREME_C_EXAMPLES_CHAPTER_7_1_ENGINE_H
#define EXTREME_C_EXAMPLES_CHAPTER_7_1_ENGINE_H

struct engine_t;

// аллокатор памяти
struct engine_t* engine_new();

// конструктор
void engine_ctor(struct engine_t*);

// деструктор
void engine_dtor(struct engine_t*);

// поведенческие функции
void engine_turn_on(struct engine_t*);
void engine_turn_off(struct engine_t*);
double engine_get_temperature(struct engine_t*);

#endif
```

В следующих листингах содержатся реализации классов `Car` (листинг 7.5) и `Engine` (листинг 7.6). Сначала рассмотрим определение класса `Car`.

Листинг 7.5. Определение класса Car (ExtremeC_examples_chapter7_1_car.c)

```
#include <stdlib.h>

// Car может работать только с публичным интерфейсом Engine
#include "ExtremeC_examples_chapter7_1_engine.h"

typedef struct {
  // благодаря этому атрибуту устанавливается отношение композиции
  struct engine_t* engine;
} car_t;

car_t* car_new() {
  return (car_t*)malloc(sizeof(car_t));
}

void car_ctor(car_t* car) {
  // выделяем память для объекта engine
  car->engine = engine_new();

  // создаем объект engine
  engine_ctor(car->engine);
}

void car_dtor(car_t* car) {
  // уничтожаем объект engine
  engine_dtor(car->engine);

  // освобождаем память, выделенную для объекта engine
  free(car->engine);
}

void car_start(car_t* car) {
  engine_turn_on(car->engine);
}

void car_stop(car_t* car) {
  engine_turn_off(car->engine);
}

double car_get_engine_temperature(car_t* car) {
  return engine_get_temperature(car->engine);
}
```

В данном листинге видно, как объект `car` содержит `engine`. В структуре атрибутов `car_t` есть новое поле типа `struct engine_t*`. Благодаря ему устанавливается отношение композиции.

Внутри этого исходного файла тип `struct engine_t*` по-прежнему остается неполным, но во время выполнения сможет сослаться на объект полного типа `engine_t`. Данный атрибут будет указывать на объект, который станет создаваться в конструкторе класса `Car`, а освобождаться — внутри деструктора. В обоих случаях

объект `car` все еще существует; это значит, его время жизни включает в себя время жизни объекта `engine`.

Указатель `engine` является приватным, доступным только внутри реализации. Это важный факт. Когда вы реализуете композицию, ни один указатель не должен быть виден снаружи, иначе внешний код сможет изменять состояние внутреннего объекта. Как и в случае с инкапсуляцией, это касается любых указателей, которые дают прямой доступ к приватным частям объекта. Обращение к этим приватным частям всегда должно происходить косвенно, через поведенческие функции.

Функция `car_get_engine_temperature` в приведенном выше листинге дает доступ к атрибуту `temperature` объекта `engine`. Но она имеет одну важную особенность: использует публичный интерфейс `engine`. Если присмотреться, то можно заметить, что *приватная реализация* `car` потребляет *публичный интерфейс* `engine`.

Это значит, самому объекту `car` ничего не известно о деталях реализации `engine`. Так и должно быть.

В большинстве случаев два объекта разных типов не должны знать о деталях реализации друг друга. Это продиктовано принципом сокрытия. Напомню, что поведение `car` считается внешним относительно `engine`.

Благодаря этому мы можем использовать альтернативную реализацию `engine`, и она будет работать, если имеет определения для всех публичных функций, объявленных в заголовочном файле `engine`.

Теперь взглянем на реализацию класса `Engine` (см. листинг 7.6).

Листинг 7.6. Определение класса Engine (ExtremeC_examples_chapter7_1_engine.c)

```
#include <stdlib.h>

typedef enum {
  ON,
  OFF
} state_t;

typedef struct {
  state_t state;
  double temperature;
} engine_t;

// аллокатор памяти
engine_t* engine_new() {
  return (engine_t*)malloc(sizeof(engine_t));
}

// конструктор
void engine_ctor(engine_t* engine) {
  engine->state = OFF;
```

```
  engine->temperature = 15;
}

// деструктор
void engine_dtor(engine_t* engine) {
  // Здесь ничего не происходит
}

// поведенческие функции
void engine_turn_on(engine_t* engine) {
  if (engine->state == ON) {
    return;
  }
  engine->state = ON;
  engine->temperature = 75;
}

void engine_turn_off(engine_t* engine) {
  if (engine->state == OFF) {
    return;
  }
  engine->state = OFF;
  engine->temperature = 15;
}

double engine_get_temperature(engine_t* engine) {
  return engine->temperature;
}
```

Для приватной реализации в этом коде применяется подход с неявной инкапсуляцией, очень похожий на тот, который использовался в предыдущих примерах. Но здесь следует отметить один важный момент. Как видите, объект `engine` не знает, что будет содержаться во внешнем объекте в рамках композиции. Все как в жизни: когда компания производит двигатели, она не знает, в какие именно автомобили те будут устанавливаться. Конечно, мы могли бы оставить указатель на контейнер `car`, но в данном примере этого делать не пришлось.

В листинге 7.7 показан сценарий, в котором мы создаем объект `car` и обращаемся к его публичному API, чтобы получить информацию о двигателе автомобиля.

Листинг 7.7. Главная функция в примере 7.1 (ExtremeC_examples_chapter7_1_main.c)

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

#include "ExtremeC_examples_chapter7_1_car.h"

int main(int argc, char** argv) {

  // выделяем память для объекта car
  struct car_t *car = car_new();
```

```
  // создаем объект car
  car_ctor(car);

  printf("Engine temperature before starting the car: %f\n",
         car_get_engine_temperature(car));
  car_start(car);
  printf("Engine temperature after starting the car: %f\n",
         car_get_engine_temperature(car));
  car_stop(car);
  printf("Engine temperature after stopping the car: %f\n",
         car_get_engine_temperature(car));

  // уничтожаем объект car
  car_dtor(car);

  // освобождаем память, выделенную для объекта car
  free(car);

  return 0;
}
```

Чтобы собрать этот пример, нужно сначала скомпилировать три предыдущих исходных файла. Затем их следует скомпоновать, чтобы сгенерировать итоговый исполняемый объектный файл. Обратите внимание: главный исходный файл (тот, который содержит функцию `main`) зависит только от публичного интерфейса `car`. Поэтому при компоновке ему нужна лишь его приватная реализация. Однако приватная реализация объекта `car` зависит от публичного интерфейса `engine`, поэтому приватную реализацию последнего нужно предоставить во время компоновки. Таким образом, чтобы получить итоговую исполняемую программу, нам нужно скомпоновать все три объектных файла.

В терминале 7.1 показаны команды, которые собирают этот пример и запускают итоговый исполняемый файл.

Терминал 7.1. Компиляция, компоновка и запуск примера 7.1

```
$ gcc -c ExtremeC_examples_chapter7_1_engine.c -o engine.o
$ gcc -c ExtremeC_examples_chapter7_1_car.c -o car.o
$ gcc -c ExtremeC_examples_chapter7_1_main.c -o main.o
$ gcc engine.o car.o main.o -o ex7_1.out
$ ./ex7_1.out
Engine temperature before starting the car: 15.000000
Engine temperature after starting the car: 75.000000
Engine temperature after stopping the car: 15.000000
$
```

В этом разделе мы рассмотрели один из типов отношений, которые могут существовать между двумя объектами. Далее поговорим еще об одном отношении. По своему принципу оно похоже на композицию, но при этом обладает некоторыми значительными особенностями.

Агрегация

В агрегации тоже применяется контейнер, который содержит другой объект. Основное отличие в том, что время жизни контейнера никак не связано с временем жизни содержащегося в нем объекта.

При использовании агрегации внутренний объект можно создать даже раньше контейнера. Для сравнения, композиция подразумевает, что время жизни контейнера должно быть не короче времени жизни внутреннего объекта.

Отношение типа «агрегация» продемонстрировано в примере 7.2. Здесь рассматривается очень простой сценарий, в котором игрок подбирает ружье, делает из него несколько выстрелов и роняет его на землю.

Объект `player` будет какое-то время играть роль контейнера, а `gun` будет выступать внутренним объектом, пока игрок держит его в руках. Эти два объекта имеют независимое время жизни.

В листинге 7.8 показан заголовочный файл класса `Gun`.

Листинг 7.8. Публичный интерфейс класса Gun (ExtremeC_examples_chapter7_2_gun.h)

```
#ifndef EXTREME_C_EXAMPLES_CHAPTER_7_2_GUN_H
#define EXTREME_C_EXAMPLES_CHAPTER_7_2_GUN_H

typedef int bool_t;

// предварительное объявление типа
struct gun_t;

// аллокатор памяти
struct gun_t* gun_new();

// конструктор
void gun_ctor(struct gun_t*, int);

// деструктор
void gun_dtor(struct gun_t*);

// поведенческие функции
bool_t gun_has_bullets(struct gun_t*);
void gun_trigger(struct gun_t*);
void gun_refill(struct gun_t*);

#endif
```

Здесь находится только объявление структуры атрибутов `gun_t`, поскольку мы еще не определили ее поля. Как уже объяснялось ранее, это называется предварительным объявлением, и в итоге получается неполный тип, из которого нельзя создать объект.

В листинге 7.9 показан заголовочный файл класса `Player`.

Листинг 7.9. Публичный интерфейс класса Player (ExtremeC_examples_chapter7_2_player.h)

```
#ifndef EXTREME_C_EXAMPLES_CHAPTER_7_2_PLAYER_H
#define EXTREME_C_EXAMPLES_CHAPTER_7_2_PLAYER_H

// предварительное объявление типа
struct player_t;
struct gun_t;

// аллокатор памяти
struct player_t* player_new();

// конструктор
void player_ctor(struct player_t*, const char*);

// деструктор
void player_dtor(struct player_t*);

// поведенческие функции
void player_pickup_gun(struct player_t*, struct gun_t*);
void player_shoot(struct player_t*);
void player_drop_gun(struct player_t*);

#endif
```

Опять же мы видим предварительное объявление структур `gun_t` и `player_t`. Тип `gun_t` нужно объявить в связи с тем, что некоторые поведенческие функции класса `Player` принимают аргументы этого типа.

Реализация класса `Player` выглядит следующим образом (листинг 7.10).

Листинг 7.10. Определение класса Player (ExtremeC_examples_chapter7_2_player.c)

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

#include "ExtremeC_example s_chapter7_2_gun.h"

// структура атрибутов
typedef struct {
  char* name;
  struct gun_t* gun;
} player_t;

// аллокатор памяти
player_t* player_new() {
  return (player_t*)malloc(sizeof(player_t));
}
```

```
// конструктор
void player_ctor(player_t* player, const char* name) {
  player->name =
      (char*)malloc((strlen(name) + 1) * sizeof(char));
  strcpy(player->name, name);
  // Это важно. Указатели агрегации, которые не должны быть заданы
  // в конструкторе, необходимо обнулить
  player->gun = NULL;
}

// деструктор
void player_dtor(player_t* player) {
  free(player->name);
}

// поведенческие функции
void player_pickup_gun(player_t* player, struct gun_t* gun) {
  // после следующей строчки начинается отношение типа «агрегация»
  player->gun = gun;
}

void player_shoot(player_t* player) {
  // нам нужно проверить, подобрал ли игрок ружье,
  // иначе стрельба не имеет смысла
  if (player->gun) {
    gun_trigger(player->gun);
  } else {
    printf("Player wants to shoot but he doesn't have a gun!");
    exit(1);
  }
}

void player_drop_gun(player_t* player) {
  // После следующей строчки завершается агрегация двух объектов.
  // Заметьте, что объект gun не нужно освобождать, поскольку
  // данный объект не является его владельцем (как в композиции).
  player->gun = NULL;
}
```

Внутри структуры `player_t` объявлен атрибут-указатель `gun`, который скоро будет ссылаться на одноименный объект. В конструкторе его нужно обнулить, а не присваивать ему значение, как в случае с композицией.

Если указатель агрегации необходимо установить в процессе создания, то адрес соответствующего объекта следует передать конструктору в качестве аргумента. Такую агрегацию называют *принудительной*.

Если во время создания объекта указатель можно оставить обнуленным, как в приведенном выше листинге, то это называется *необязательной агрегацией*. В таких случаях обнуление необходимо выполнять в конструкторе.

Агрегация начинается в функции `player_pickup_gun`, а заканчивается в `player_drop_gun`, когда игрок роняет ружье.

Следует отметить, что после разрыва отношения типа «агрегация» указатель `gun` следует обнулить. В отличие от композиции, контейнер здесь не является *владельцем* внутреннего объекта, поэтому не может контролировать его жизненный цикл. Таким образом, нам не обязательно освобождать объект `gun` на каком-либо участке внутренней реализации `player`.

При использовании необязательной агрегации контейнерный объект может оставаться без определенного значения. Поэтому с указателем агрегации нужно обращаться осторожно, поскольку любое обращение к неустановленному или нулевому указателю приводит к ошибке сегментации. Вот почему мы проверяем указатель `gun` в функции `player_shoot`. Если он обнулен, то код, который работает с объектом `player`, не должен к нему обращаться. Если же обращение все же происходит, то мы прерываем выполнение программы и возвращаем `1` в качестве кода *выхода* из процесса.

В листинге 7.11 показана реализация класса `Gun`.

Листинг 7.11. Определение класса Gun (ExtremeC_examples_chapter7_2_gun.c)

```
#include <stdlib.h>

typedef int bool_t;

// структура атрибутов
typedef struct {
  int bullets;
} gun_t;

// аллокатор памяти
gun_t* gun_new() {
  return (gun_t*)malloc(sizeof(gun_t));
}

// конструктор
void gun_ctor(gun_t* gun, int initial_bullets) {
  gun->bullets = 0;
  if (initial_bullets > 0) {
    gun->bullets = initial_bullets;
  }
}

// деструктор
void gun_dtor(gun_t* gun) {
  // здесь ничего не происходит
}
```

```
// поведенческие функции
bool_t gun_has_bullets(gun_t* gun) {
  return (gun->bullets > 0);
}

void gun_trigger(gun_t* gun) {
  gun->bullets--;
}

void gun_refill(gun_t* gun) {
  gun->bullets = 7;
}
```

Здесь все должно быть понятно. Этот код написан таким образом, что объекту gun неизвестно о том, что он содержится внутри другого объекта.

Наконец, в листинге 7.12 показан короткий сценарий, в котором создаются объекты player и gun. Игрок подбирает ружье и начинает стрелять, пока не закончатся патроны. После перезарядки все повторяется. В конце он роняет ружье.

Листинг 7.12. Главная функция в примере 7.2 (ExtremeC_examples_chapter7_2_main.c)

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

#include "ExtremeC_examples_chapter7_2_player.h"
#include "ExtremeC_examples_chapter7_2_gun.h"

int main(int argc, char** argv) {

  // создаем и инициализируем объект gun
  struct gun_t* gun = gun_new();
  gun_ctor(gun, 3);

  // создаем и инициализируем объект player
  struct player_t* player = player_new();
  player_ctor(player, "Billy");

  // начинаем агрегацию
  player_pickup_gun(player, gun);

  // стреляем, пока не закончатся патроны
  while (gun_has_bullets(gun)) {
    player_shoot(player);
  }

  // перезаряжаем ружье
  gun_refill(gun);

  // стреляем, пока не закончатся патроны
  while (gun_has_bullets(gun)) {
```

```
    player_shoot(player);
  }

  // завершаем агрегацию
  player_drop_gun(player);

  // уничтожаем и освобождаем объект player
  player_dtor(player);
  free(player);

  // уничтожаем и освобождаем объект gun
  gun_dtor(gun);
  free(gun);

  return 0;
}
```

Как видите, объекты `gun` и `player` не зависят друг от друга. Логика, которая отвечает за их создание и уничтожение, находится в функции `main`. На каком-то этапе выполнения они формируют отношение типа «агрегация», выполняют то, что им предписано, и снова разъединяются. Важной особенностью агрегации является то, что контейнер не должен влиять на время жизни внутреннего объекта; если следовать данному правилу, то никаких проблем с памятью возникнуть не должно.

В терминале 7.2 показано, как собрать этот пример и запустить полученный исполняемый файл. Как видите, функция `main` ничего не выводит.

Терминал 7.2. Компиляция, компоновка и запуск примера 7.2

```
$ gcc -c ExtremeC_examples_chapter7_2_gun.c -o gun.o
$ gcc -c ExtremeC_examples_chapter7_2_player.c -o player.o
$ gcc -c ExtremeC_examples_chapter7_2_main.c -o main.o
$ gcc gun.o player.o main.o -o ex7_2.out
$ ./ex7_2.out
$
```

В объектной модели реального проекта агрегация обычно применяется чаще, чем композиция. Кроме того, агрегация лучше видна снаружи, поскольку ее работа требует наличия отдельных поведенческих функций (по крайней мере в публичном интерфейсе контейнера) для установки и сброса внутреннего объекта.

Как можно видеть в приведенном выше примере, объекты `gun` и `player` изначально разделены. Связь между ними устанавливается лишь на короткое время, после чего они опять становятся независимыми. Это значит, что агрегация, в отличие от композиции, является временным отношением между двумя объектами. Таким образом, композицию можно считать усиленной разновидностью *владения* (отношения типа *«иметь»*), а агрегацию — ослабленной.

Теперь встает вопрос: если агрегация носит временный характер для двух объектов, то является ли она временной для их соответствующих классов? Ответ отрицательный. Отношение агрегации между двумя типами остается навсегда. Если существует малейшая вероятность того, что в будущем два объекта сформируют связь на основе агрегации, то их типы должны поддерживать это отношение на постоянной основе. То же самое относится и к композиции.

Даже небольшая вероятность возникновения отношения типа «агрегация» — повод для объявления некоторых указателей в структуре атрибутов контейнера, что означает внесение постоянного изменения. Конечно, это относится только к классовым языкам программирования.

Композиция и агрегация описывают владение некими объектами. Иными словами, они представляют отношение типа *«иметь»*; игрок *имеет* пистолет, автомобиль *имеет* двигатель. Если вам кажется, что один объект владеет другим, то между ними (и их соответствующими классами) следует установить отношение композиции или агрегации.

В следующей главе мы продолжим обсуждение разных типов отношений и уделим внимание *наследованию* и *расширению*.

Резюме

В этой главе были рассмотрены следующие темы:

- возможные типы отношений между классами и объектами;
- различия и сходства между классом, объектом, экземпляром и ссылкой;
- композиция подразумевает, что внутренний объект полностью зависит от своего контейнера;
- агрегация — внутренний объект может свободно существовать без зависимости от своего контейнера;
- агрегация может быть временной между объектами, но между их типами (или классами) должна быть постоянной.

В следующей главе мы продолжим исследовать ООП и сосредоточимся еще на двух концепциях, на которых основана эта парадигма: на наследовании и полиморфизме.

8 Наследование и полиморфизм

Эта глава продолжает две предыдущие, в которых вы увидели, как реализовать ООП в языке C, и познакомились с такими концепциями, как композиция и агрегация. Здесь же мы в основном продолжим рассматривать отношения между объектами и их соответствующими классами. На сей раз речь пойдет о наследовании и полиморфизме. Это заключительная часть данной темы, а в следующей главе мы поговорим об *абстракции данных*.

То, что мы будем здесь обсуждать, основано на теоретическом материале, пройденном в предыдущих двух главах, который касался возможных отношений между классами. Мы уже познакомились с *композицией* и *агрегацией*, и теперь пришло время поговорить о *расширении (наследовании)* наряду с несколькими другими темами.

Ниже перечислены аспекты ООП, на которых мы сконцентрируемся.

- Как уже упоминалось выше, в первую очередь мы рассмотрим наследование. Будут представлены методы реализации наследования в языке C и их сравнение.
- Следующей большой темой станет *полиморфизм*. Он позволяет наделять классы разными версиями одних и тех же операций на случай, если один из них наследует другой. Мы обсудим, как реализовать полиморфные функции в C; это будет первым шагом на пути к пониманию того, как полиморфизм устроен в C++.

Начнем с наследования.

Наследование

В предыдущей главе мы обсудили отношения типа *«иметь»*, что в итоге привело нас к композиции и агрегации. В этом разделе речь пойдет об отношениях типа *«быть»*, примером которых является наследование.

Наследование также называют *расширением*, поскольку оно только добавляет новые атрибуты и операции в существующий объект или класс. Далее мы объясним принцип работы этого отношения и то, как его можно реализовать в C.

Бывают случаи, когда один объект должен иметь те же атрибуты, которые есть в другом. Иными словами, новый объект — расширение существующего.

Например, студент обладает всеми атрибутами человека, однако не ограничен ими (листинг 8.1).

Листинг 8.1. Структуры атрибутов для классов Person и Student

```
typedef struct {
  char first_name[32];
  char last_name[32];
  unsigned int birth_year;
} person_t;

typedef struct {
  char first_name[32];
  char last_name[32];
  unsigned int birth_year;
  char student_number[16]; // дополнительный атрибут
  unsigned int passed_credits; // дополнительный атрибут
} student_t;
```

Этот пример наглядно показывает, как `student_t` расширяет `person_t` за счет новых атрибутов `student_number` и `passed_credits`, которые характерны студентам.

Как уже отмечалось ранее, наследование (или расширение) — это отношение типа *«быть»*. Напомню, что композиция и агрегация относятся к типу *«иметь»*. Поэтому мы можем сказать: «Студент является человеком», что звучит логично в контексте образовательного программного обеспечения. Если в предметной области существует отношение типа *«быть»*, то это, скорее всего, наследование. В показанном выше примере `person_t` обычно называется *супертипом* или просто *базовым/родительским* типом, а `student_t` — *дочерним* типом или *унаследованным подтипом*.

Природа наследования

Если копнуть глубже и попытаться понять, что же на самом деле представляет собой наследование, то можно прийти к выводу: по своей природе это композиция. Например, мы можем сказать, что студент имеет человеческую сущность. Иными словами, мы можем предположить, что внутри структуры атрибутов класса `Student` есть приватный объект `person`. Таким образом, наследование может быть эквивалентом композиции вида «один к одному».

Поэтому структуру из листинга 8.1 можно записать следующим образом (листинг 8.2).

Листинг 8.2. Структуры атрибутов для классов Person и Student, но на этот раз вложенные

```
typedef struct {
  char first_name[32];
  char last_name[32];
  unsigned int birth_year;
} person_t;

typedef struct {
  person_t person;
  char student_number[16]; // Дополнительный атрибут
  unsigned int passed_credits; // Дополнительный атрибут
} student_t;
```

Данный синтаксис полностью корректен с точки зрения C. На самом деле вложение структур с помощью структурных переменных (а не указателей) имеет большой потенциал. Это позволяет поместить в новую структуру переменную сложного типа так, чтобы данная структура являлась его расширением.

В показанном выше листинге наличие первого поля типа `person_t` позволяет привести указатель `student_t` к этому типу и сделать так, чтобы они указывали на один и тот же адрес в памяти.

Это так называемое *восходящее приведение* (или преобразование, upcasting) — то есть приведение структуры атрибутов потомка к типу структуры атрибутов родителя. Имейте в виду, что со структурными переменными этого делать нельзя.

Данный подход показан в примере 8.1 (листинг 8.3).

Листинг 8.3. Пример 8.1, демонстрирующий восходящее преобразование указателей на объекты Student и Person (ExtremeC_examples_chapter8_1.c)

```
#include <stdio.h>

typedef struct {
  char first_name[32];
  char last_name[32];
  unsigned int birth_year;
} person_t;

typedef struct {
  person_t person;
  char student_number[16]; // дополнительный атрибут
  unsigned int passed_credits; // дополнительный атрибут
} student_t;
```

```
int main(int argc, char** argv) {
  student_t s;
  student_t* s_ptr = &s;
  person_t* p_ptr = (person_t*)&s;
  printf("Student pointer points to %p\n", (void*)s_ptr);
  printf("Person pointer points to %p\n", (void*)p_ptr);
  return 0;
}
```

Мы ожидаем, что `s_ptr` и `p_ptr` указывают на один и тот же адрес в памяти. В терминале 8.1 показан вывод примера 8.1 после его сборки и запуска.

Терминал 8.1. Вывод примера 8.1

```
$ gcc ExtremeC_examples_chapter8_1.c -o ex8_1.out
$ ./ex8_1.out
Student pointer points to 0x7ffeecd41810
Person pointer points to 0x7ffeecd41810
$
```

Действительно, они указывают на один и тот же адрес. Обратите внимание: данный вывод может меняться с каждым запуском, но суть в том, что указатели всегда ссылаются на один участок памяти. Это значит, что структурная переменная типа `student_t` в действительности наследует структуру `person_t` в своей схеме размещения. Отсюда следует, что поведенческие функции класса `Person` можно вызывать с помощью указателя на объект `student`. Иными словами, поведенческие функции класса `Person` могут использоваться объектами `student`. Это важное достижение.

Обратите внимание на то, что код, приведенный в листинге 8.4, некорректен и его нельзя скомпилировать.

Листинг 8.4. Отношение наследования, которое не компилируется!

```
struct person_t;

typedef struct {
  struct person_t person; // генерирует ошибку!
  char student_number[16]; // дополнительный атрибут
  unsigned int passed_credits; // дополнительный атрибут
} student_t;
```

Строчка, в которой объявляется поле `person`, генерирует ошибку, поскольку мы не можем создать переменную из *неполного типа*. Не забывайте, что неполный тип — результат предварительного объявления структуры (подобного первой строчке в листинге 8.4). Его могут иметь только указатели, но *не* переменные. Как вы уже сами видели, память для неполного типа нельзя выделить даже в куче.

Так что же это означает? Получается, в случае реализации наследования с помощью вложенных структурных переменных структура `student_t` должна иметь доступ к определению `person_t`, которое согласно тому, что мы знаем об инкапсуляции, должно быть приватным и недоступным для любых других классов.

В связи с этим наследование можно реализовать двумя путями:

- сделать так, чтобы дочерний класс имел доступ к приватной реализации (определению) базового класса;
- сделать так, чтобы дочерний класс мог обращаться только к публичному интерфейсу базового класса.

Первый подход к наследованию в C

Мы продемонстрируем первый подход в примере 8.2, а второй — в примере 8.3, который находится в пункте «Второй подход к наследованию в C» далее, на с. 270. В обоих случаях представлены одни и те же классы, `Student` и `Person`, содержащие ряд поведенческих функций. Объекты на их основе будут взаимодействовать в некоем простом сценарии в функции `main`.

Начнем с примера 8.2, в котором классу `Student` нужен доступ к приватному определению структуры атрибутов класса `Person`. В листингах ниже показаны заголовки и исходники классов `Student` и `Person`, а также функция `main`. В листинге 8.5 можно видеть заголовочный файл, в котором объявляется класс `Person`.

Листинг 8.5. Публичный интерфейс класса Person в примере 8.2 (ExtremeC_examples_chapter8_2_person.h)

```
#ifndef EXTREME_C_EXAMPLES_CHAPTER_8_2_PERSON_H
#define EXTREME_C_EXAMPLES_CHAPTER_8_2_PERSON_H

// предварительное объявление
struct person_t;

// аллокатор памяти
struct person_t* person_new();

// конструктор
void person_ctor(struct person_t*,
                 const char*  /* имя */,
                 const char*  /* фамилия */,
                 unsigned int /* год рождения */);

// деструктор
void person_dtor(struct person_t*);

// поведенческие функции
void person_get_first_name(struct person_t*, char*);
```

```
void person_get_last_name(struct person_t*, char*);
unsigned int person_get_birth_year(struct person_t*);

#endif
```

Взгляните на конструктор. Он принимает все значения, необходимые для создания объекта `person`: `first_name`, `second_name` и `birth_year`. Структура атрибутов `person_t` является неполной, поэтому класс `Student` не может использовать этот заголовочный файл для наследования (см. предыдущий раздел).

С другой стороны, этот заголовок не должен содержать определение структуры атрибутов `person_t`, поскольку будет использоваться на других участках кода, которые не должны ничего знать о внутренностях класса `Person`. Что же делать? Мы хотим, чтобы одна часть кода знала об определении структуры, а другая — нет. Здесь нам пригодятся *приватные заголовочные файлы*.

Приватным называют заголовочный файл, который должен подключаться и использоваться только на определенных участках кода или в классах, которым он действительно нужен. Если вернуться к примеру 8.2, то в приватном заголовке должно находиться определение `person_t`. Образец этого показан в листинге 8.6.

Листинг 8.6. Приватный заголовочный файл с определением person_t (ExtremeC_examples_chapter8_2_person_p.h)

```
#ifndef EXTREME_C_EXAMPLES_CHAPTER_8_2_PERSON_P_H
#define EXTREME_C_EXAMPLES_CHAPTER_8_2_PERSON_P_H

// приватное определение
typedef struct {
  char first_name[32];
  char last_name[32];
  unsigned int birth_year;
} person_t;

#endif
```

Как видите, здесь содержится только определение структуры `person_t` и ничего больше. Это часть класса `Person`, которая должна оставаться приватной, но притом быть доступной классу `Student`. Данный заголовок нам понадобится при определении структуры атрибутов `student_t`. В листинге 8.7 показана приватная реализация класса `Person`.

Листинг 8.7. Определение класса Person (ExtremeC_examples_chapter8_2_person.c)

```
#include <stdlib.h>
#include <string.h>

// определение person_t находится в следующем заголовке
#include "ExtremeC_examples_chapter8_2_person_p.h"
```

```
// аллокатор памяти
person_t* person_new() {
  return (person_t*)malloc(sizeof(person_t));
}

// конструктор
void person_ctor(person_t* person,
                 const char* first_name,
                 const char* last_name,
                 unsigned int birth_year) {
  strcpy(person->first_name, first_name);
  strcpy(person->last_name, last_name);
  person->birth_year = birth_year;
}

// деструктор
void person_dtor(person_t* person) {
  // здесь ничего не происходит
}

// поведенческие функции
void person_get_first_name(person_t* person, char* buffer) {
  strcpy(buffer, person->first_name);
}

void person_get_last_name(person_t* person, char* buffer) {
  strcpy(buffer, person->last_name);
}

unsigned int person_get_birth_year(person_t* person) {
  return person->birth_year;
}
```

В определении класса `Person` нет ничего особенного; все это мы уже видели в предыдущих примерах. В листинге 8.8 представлен публичный интерфейс класса `Student`.

Листинг 8.8. Публичный интерфейс класса Student (ExtremeC_examples_chapter8_2_student.h)

```
#ifndef EXTREME_C_EXAMPLES_CHAPTER_8_2_STUDENT_H
#define EXTREME_C_EXAMPLES_CHAPTER_8_2_STUDENT_H

// предварительное объявление
struct student_t;

// аллокатор памяти
struct student_t* student_new();

// конструктор
void student_ctor(struct student_t*,
                  const char*  /* имя */,
                  const char*  /* фамилия */,
```

```
                  unsigned int /* год рождения */,
                  const char*  /* номер студенческого билета */,
                  unsigned int /* засчитанные кредиты */);

// деструктор
void student_dtor(struct student_t*);

// поведенческие функции
void student_get_student_number(struct student_t*, char*);
unsigned int student_get_passed_credits(struct student_t*);

#endif
```

Аргументы, которые принимает конструктор, похожи на аргументы конструктора из класса `Person`. Это вызвано тем, что объект `student` фактически содержит объект `person` и должен передать ему эти значения.

Отсюда следует, что конструктор `student` должен установить атрибуты `person`, которые являются его частью.

Обратите внимание: класс `Student` содержит всего две дополнительные поведенческие функции, так как, помимо них, объекты `student` доступны функции класса `Person`.

В листинге 8.9 показана приватная реализация класса `Student`.

Листинг 8.9. Приватное определение класса Student (ExtremeC_examples_chapter8_2_student.c)

```
#include <stdlib.h>
#include <string.h>

#include "ExtremeC_examples_chapter8_2_person.h"

// определение person_t находится в следующем заголовке, и оно нам здесь нужно
#include "ExtremeC_examples_chapter8_2_person_p.h"

// предварительное объявление
typedef struct {
  // благодаря этой вложенности мы наследуем все атрибуты класса Person
  // и получаем доступ ко всем его поведенческим функциям
  person_t person;
  char* student_number;
  unsigned int passed_credits;
} student_t;

// аллокатор памяти
student_t* student_new() {
  return (student_t*)malloc(sizeof(student_t));
}
```

```
// конструктор
void student_ctor(student_t* student,
                  const char* first_name,
                  const char* last_name,
                  unsigned int birth_year,
                  const char* student_number,
                  unsigned int passed_credits) {
  // вызываем конструктор родительского класса
  person_ctor((struct person_t*)student,
          first_name, last_name, birth_year);
  student->student_number = (char*)malloc(16 * sizeof(char));
  strcpy(student->student_number, student_number);
  student->passed_credits = passed_credits;
}

// деструктор
void student_dtor(student_t* student) {
  // сначала нужно уничтожить дочерний объект
  free(student->student_number);
  // затем следует вызвать деструктор родительского класса
  person_dtor((struct person_t*)student);
}

// поведенческие функции
void student_get_student_number(student_t* student,
                                char* buffer) {
  strcpy(buffer, student->student_number);
}

unsigned int student_get_passed_credits(student_t* student) {
  return student->passed_credits;
}
```

Этот листинг содержит самый важный код с точки зрения наследования. Прежде всего, нам нужно было подключить приватный заголовок класса `Person`, так как мы хотели, чтобы первое поле в определении `student_t` имело тип `person_t`. И, поскольку это поле является переменной, а не указателем, структура `person_t` уже должна быть определена. Имейте в виду: данная переменная должна быть *первым полем* структуры, иначе мы теряем возможность использования поведенческих функций класса `Person`.

В данном листинге мы снова вызываем конструктор родителя в конструкторе класса `Student`, чтобы инициализировать родительский объект. Обратите внимание: мы привели указатель `student_t` к типу `person_t` при передаче его в функцию `person_ctor`. Это возможно лишь благодаря тому, что поле `person` — первый атрибут структуры `student_t`.

Аналогичным образом мы вызвали родительский деструктор в деструкторе класса `Student`. Уничтожить нужно сначала дочерний объект, а затем родительский —

в обратном порядке по сравнению с созданием. В листинге 8.10 находится главная функция примера 8.2, в которой на основе класса `Student` создается объект соответствующего типа.

Листинг 8.10. Главная функция примера 8.2 (ExtremeC_examples_chapter8_2_main.c)

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

#include "ExtremeC_examples_chapter8_2_person.h"
#include "ExtremeC_examples_chapter8_2_student.h"

int main(int argc, char** argv) {
  // создаем объект student
  struct student_t* student = student_new();
  student_ctor(student, "John", "Doe",
          1987, "TA5667", 134);

  // теперь мы используем поведенческие функции объекта person
  // для чтения атрибутов объекта student
  char buffer[32];

  // восходящее приведение указателя к родительскому типу
  struct person_t* person_ptr = (struct person_t*)student;

  person_get_first_name(person_ptr, buffer);
  printf("First name: %s\n", buffer);

  person_get_last_name(person_ptr, buffer);
  printf("Last name: %s\n", buffer);

  printf("Birth year: %d\n", person_get_birth_year(person_ptr));

  // теперь мы читаем атрибуты, принадлежащие объекту student
  student_get_student_number(student, buffer);
  printf("Student number: %s\n", buffer);

  printf("Passed credits: %d\n",
          student_get_passed_credits(student));

  // уничтожаем и освобождаем объект student
  student_dtor(student);
  free(student);

  return 0;
}
```

В главной функции подключаются публичные интерфейсы обоих классов, `Person` и `Student` (но не приватный заголовочный файл), но создается только объект `student`. Как видите, этот объект унаследовал все атрибуты своего внутреннего

объекта, `person`, и для их чтения может использовать поведенческие функции класса `Person`.

В терминале 8.2 показано, как скомпилировать пример 8.2.

Терминал 8.2. Сборка и запуск примера 8.2

```
$ gcc -c ExtremeC_examples_chapter8_2_person.c -o person.o
$ gcc -c ExtremeC_examples_chapter8_2_student.c -o student.o
$ gcc -c ExtremeC_examples_chapter8_2_main.c -o main.o
$ gcc person.o student.o main.o -o ex8_2.out
$ ./ex8_2.out
First name: John
Last name: Doe
Birth year: 1987
Student number: TA5667
Passed credits: 134
$
```

В следующем примере, под номером 8.3, мы обсудим второй подход к реализации наследования в языке C. Его вывод должен быть очень похож на то, что мы только видели.

Второй подход к наследованию в C

При использовании первого подхода мы хранили структурную переменную в качестве первого поля структуры атрибутов потомка. Здесь же будем использовать указатель на структурную переменную родителя. Это позволит сделать дочерний класс независимым от реализации родительского, что является положительным фактором с точки зрения принципа сокрытия информации.

Второй подход имеет свои преимущества и недостатки. После демонстрации примера 8.3 мы сравним эти два метода, и вы сможете увидеть их сильные и слабые стороны.

Пример 8.3, представленный ниже, удивительно похож на пример 8.2, особенно с точки зрения вывода и итоговых результатов. Его основное отличие в том, что класс `Student` зависит только от публичного интерфейса класса `Person` и не требует доступа к его приватному определению. Это важно, поскольку так мы можем изолировать классы друг от друга и легко изменять реализацию родителя, не меняя реализацию потомка.

В предыдущем примере класс `Student`, строго говоря, не нарушал принцип сокрытия, но имел такую возможность, поскольку имел доступ к определению структуры `person_t` и к ее полям. Как результат, он мог читать или изменять ее поля в обход поведенческих функций класса `Person`.

Как уже отмечалось, примеры 8.3 и 8.2 мало чем отличаются, но у 8.3 есть некоторые фундаментальные особенности. Класс `Person` имеет тот же публичный интерфейс, который мы видели ранее, чего нельзя сказать о `Student`. Новый публичный интерфейс класса `Student` показан в листинге 8.11.

Листинг 8.11. Новый публичный интерфейс класса Student (ExtremeC_examples_chapter8_3_student.h)

```
#ifndef EXTREME_C_EXAMPLES_CHAPTER_8_3_STUDENT_H
#define EXTREME_C_EXAMPLES_CHAPTER_8_3_STUDENT_H

// предварительное объявление
struct student_t;

// аллокатор памяти
struct student_t* student_new();

// конструктор
void student_ctor(struct student_t*,
                  const char*  /* имя */,
                  const char*  /* фамилия */,
                  unsigned int /* год рождения */,
                  const char*  /* номер студенческого билета */,
                  unsigned int /* засчитанные кредиты */);

// деструктор
void student_dtor(struct student_t*);

// поведенческие функции
void student_get_first_name(struct student_t*, char*);
void student_get_last_name(struct student_t*, char*);
unsigned int student_get_birth_year(struct student_t*);
void student_get_student_number(struct student_t*, char*);
unsigned int student_get_passed_credits(struct student_t*);

#endif
```

По причинам, которые станут очевидными чуть позже, класс `Student` должен повторять все поведенческие функции, объявленные внутри `Person`. Дело в том, что мы больше не можем привести указатель `student_t` к типу `person_t`. Иными словами, для указателей `Student` и `Person` больше не работает восходящее преобразование.

Если публичный интерфейс класса `Person` остался прежним по сравнению с примером 8.2, то его реализация претерпела изменения. Вы можете видеть ее в листинге 8.12.

Листинг 8.12. Новая реализация класса Person (ExtremeC_examples_chapter8_3_person.c)

```
#include <stdlib.h>
#include <string.h>

// приватное определение
typedef struct {
  char first_name[32];
  char last_name[32];
  unsigned int birth_year;
} person_t;

// аллокатор памяти
person_t* person_new() {
  return (person_t*)malloc(sizeof(person_t));
}

// конструктор
void person_ctor(person_t* person,
                 const char* first_name,
                 const char* last_name,
                 unsigned int birth_year) {
  strcpy(person->first_name, first_name);
  strcpy(person->last_name, last_name);
  person->birth_year = birth_year;
}

// деструктор
void person_dtor(person_t* person) {
  // Здесь ничего не происходит
}

// поведенческие функции
void person_get_first_name(person_t* person, char* buffer) {
  strcpy(buffer, person->first_name);
}

void person_get_last_name(person_t* person, char* buffer) {
  strcpy(buffer, person->last_name);
}

unsigned int person_get_birth_year(person_t* person) {
  return person->birth_year;
}
```

Приватное определение `person_t` теперь находится в исходном файле, и мы больше не используем приватный заголовок. Это значит, мы не станем делать его доступным ни для каких других классов, включая `Student`. Мы хотим добиться полноценной инкапсуляции класса `Person` и скрыть все детали его реализации.

В листинге 8.13 показана приватная реализация класса `Student`.

Листинг 8.13. Новая реализация класса Student (ExtremeC_examples_chapter8_3_student.c)

```
#include <stdlib.h>
#include <string.h>

// публичный интерфейс класса Person
#include "ExtremeC_examples_chapter8_3_person.h"

// предварительное объявление
typedef struct {
  char* student_number;
  unsigned int passed_credits;
  // здесь нам нужен указатель, так как тип person_t является неполным
  struct person_t* person;
} student_t;

// аллокатор памяти
student_t* student_new() {
  return (student_t*)malloc(sizeof(student_t));
}

// конструктор
void student_ctor(student_t* student,
                  const char* first_name,
                  const char* last_name,
                  unsigned int birth_year,
                  const char* student_number,
                  unsigned int passed_credits) {
  // выделяем память для родительского объекта
  student->person = person_new();
  person_ctor(student->person, first_name,
          last_name, birth_year);
  student->student_number = (char*)malloc(16 * sizeof(char));
  strcpy(student->student_number, student_number);
  student->passed_credits = passed_credits;
}

// деструктор
void student_dtor(student_t* student) {
  // сначала нужно уничтожить дочерний объект
  free(student->student_number);
  // затем следует вызвать деструктор родительского класса
  person_dtor(student->person);
  // необходимо также освободить память, выделенную для родительского объекта
  free(student->person);
}

// поведенческие функции
void student_get_first_name(student_t* student, char* buffer) {
```

```
  // мы должны использовать поведенческие функции класса Person
  person_get_first_name(student->person, buffer);
}

void student_get_last_name(student_t* student, char* buffer) {
  // мы должны использовать поведенческие функции класса Person
  person_get_last_name(student->person, buffer);
}

unsigned int student_get_birth_year(student_t* student) {
  // мы должны использовать поведенческие функции класса Person
  return person_get_birth_year(student->person);
}

void student_get_student_number(student_t* student, char* buffer) {
  strcpy(buffer, student->student_number);
}

unsigned int student_get_passed_credits(student_t* student) {
  return student->passed_credits;
}
```

В этом листинге мы использовали публичный интерфейс класса `Person`, подключив его заголовочный файл. Кроме того, добавили в определение `student_t` поле-указатель, ссылающийся на родительский объект `Person`. Это должно напомнить вам об отношении «композиция», которое мы реализовали в предыдущей главе.

Следует отметить, что данное поле не обязательно делать первым в структуре атрибутов. Это отличается от того, что мы видели при реализации первого подхода. Указатели типов `student_t` и `person_t` больше не являются взаимозаменяемыми, так как ссылаются на разные адреса в памяти, которые могут не быть смежными. Это еще одно отличие от примера 8.2.

Обратите внимание: в конструкторе класса `Student` создается родительский объект. Затем он инициализируется путем вызова конструктора `Person` и передачи ему необходимых параметров. То же самое касается деструкторов: родительский объект в деструкторе класса `Student` уничтожается в последнюю очередь.

Мы не можем использовать поведение класса `Person` для чтения унаследованных приватных атрибутов, класс `Student` должен предоставлять свой набор поведенческих функций для доступа к этим атрибутам.

Иными словами, класс `Student` должен содержать функции-обертки для работы с приватными атрибутами родительского объекта `person`. Обратите внимание: самому объекту `student` ничего не известно о приватных атрибутах объекта `person`. Это отличается от того, что мы видели в предыдущем подходе.

Главная функция тоже очень похожа на ту, которая использовалась в примере 8.2. Это показано в листинге 8.14.

Листинг 8.14. Главная функция примера 8.3 (ExtremeC_examples_chapter8_3_main.c)

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

#include "ExtremeC_examples_chapter8_3_student.h"

int main(int argc, char** argv) {
  // создаем объект student
  struct student_t* student = student_new();
  student_ctor(student, "John", "Doe", 1987, "TA5667", 134);

  // Мы должны использовать поведенческие функции объекта student,
  // так как его указатель нельзя привести к типу person и мы
  // не имеем доступа к приватному родительскому указателю
  // в объекте student.
  char buffer[32];
  student_get_first_name(student, buffer);
  printf("First name: %s\n", buffer);

  student_get_last_name(student, buffer);
  printf("Last name: %s\n", buffer);

  printf("Birth year: %d\n", student_get_birth_year(student));

  student_get_student_number(student, buffer);
  printf("Student number: %s\n", buffer);

  printf("Passed credits: %d\n", student_get_passed_credits(student));

  // уничтожаем и освобождаем объект student
  student_dtor(student);
  free(student);

  return 0;
}
```

Если сравнивать с главной функцией в примере 8.2, то мы не подключили публичный интерфейс класса `Person`. Нам также пришлось использовать поведенческие функции класса `Student`, поскольку указатели `student_t` и `person_t` больше не взаимозаменяемы.

В терминале 8.3 показано, как скомпилировать и запустить пример 8.3. Как вы уже, наверное, догадались, вывод ничем не отличается от предыдущего примера.

Терминал 8.3. Сборка и запуск примера 8.3

```
$ gcc -c ExtremeC_examples_chapter8_3_person.c -o person.o
$ gcc -c ExtremeC_examples_chapter8_3_student.c -o student.o
$ gcc -c ExtremeC_examples_chapter8_3_main.c -o main.o
$ gcc person.o student.o main.o -o ex8_3.out
$ ./ex8_3.out
First name: John
Last name: Doe
Birth year: 1987
Student number: TA5667
Passed credits: 134
$
```

Далее мы сравним рассмотренные выше подходы к реализации наследования в C.

Сравнение двух подходов

Итак, вы познакомились с двумя подходами к реализации наследования в языке C. Теперь сравним их. Ниже перечислены их основные сходства и различия.

- Оба подхода, по сути, демонстрируют отношение типа «композиция».
- В первом подходе структурная переменная находится в структуре атрибутов потомка. Здесь требуется доступ к приватной реализации родительского класса. Но во втором подходе используется структурный указатель неполного типа, принадлежащего структуре атрибутов родителя, поэтому теряется зависимость от приватной реализации родительского класса.
- В первом подходе родительский и дочерний типы тесно связаны. Во втором классы независимы друг от друга и все, что находится в родительской реализации, скрыто от потомка.
- В первом подходе может быть только один родитель. То есть это реализация *одиночного наследования* в C. Во втором же подходе родителей может быть сколько угодно; так выглядит концепция *множественного наследования*.
- В первом подходе структурная переменная родителя должна быть первым полем структуры атрибутов дочернего класса. Во втором указатели на родительские объекты могут находиться в любом месте структуры.
- В первом подходе у нас не было двух отдельных объектов. Родительский объект был частью дочернего, и указатель на дочерний объект являлся также указателем на родительский.
- В первом подходе мы могли использовать поведенческие функции родительского класса. Во втором нам пришлось делать для них обертки в дочернем классе.

До сих пор мы обсуждали только саму концепцию наследования и не видели, как она используется. Один из важнейших способов применения наследования — реализация *полиморфизма* в объектной модели. В следующем разделе мы поговорим о полиморфизме и о том, как он реализован в языке C.

Полиморфизм

На самом деле полиморфизм — не отношение между двумя классами. Прежде всего это метод использования одного и того же кода для реализации разного поведения. С его помощью можно расширять код и добавлять новые возможности, не прибегая к перекомпиляции всей кодовой базы.

В данном разделе мы поговорим о том, что такое полиморфизм и как его можно добавить в язык C. Это поможет нам лучше понять, как эта концепция реализована в современных языках программирования, таких как C++. Начнем с определения.

Что такое полиморфизм

Полиморфизм — просто предоставление разного поведения с помощью одного и того же публичного интерфейса (или набора поведенческих функций).

Представьте, что у вас есть два класса, `Cat` и `Duck`, каждый из которых имеет поведенческую функцию `sound` для вывода издаваемых ими звуков. Объяснение концепции полиморфизма — непростая задача; я начну с верхнего уровня и буду двигаться по нисходящей. Для начала попробую показать, как выглядит и ведет себя полиморфный код, а затем мы перейдем к его реализации в C. Когда вы разберетесь с общей идеей, вам будет легче понять ее практические аспекты. В следующих листингах мы сначала создадим некоторые объекты, а затем посмотрим, какого поведения следовало бы ожидать от их функций, если бы они были полиморфными. Итак, нам понадобится три объекта. Мы изначально исходим из того, что `Cat` и `Duck` — потомки класса `Animal` (листинг 8.15).

Листинг 8.15. Создание трех объектов с типами Animal, Cat и Duck

```
struct animal_t* animal = animal_malloc();
animal_ctor(animal);

struct cat_t* cat = cat_malloc();
cat_ctor(cat);

struct duck_t* duck = duck_malloc();
duck_ctor(duck);
```

Без полиморфизма операцию `sound` пришлось бы вызывать из каждого объекта следующим образом (листинг 8.16).

Листинг 8.16. Вызов операции sound из созданных ранее объектов

```
// в отсутствие полиморфизма
animal_sound(animal);
cat_sound(cat);
duck_sound(duck);
```

Мы бы получили такой вывод (терминал 8.4).

Терминал 8.4. Вывод в результате вызова функций

```
Animal: Beeeep
Cat: Meow
Duck: Quack
```

На отсутствие полиморфизма в приведенных выше листингах указывает тот факт, что для вызова определенных операций из объектов `Cat` и `Duck` использовались разные функции: `cat_sound` и `duck_sound`. В листинге 8.17 показано, как должны выглядеть вызовы полиморфных функций. Это безупречный пример полиморфизма.

Листинг 8.17. Вызов одной и той же операции sound из всех трех объектов

```
// это полиморфизм
animal_sound(animal);
animal_sound((struct animal_t*)cat);
animal_sound((struct animal_t*)duck);
```

Несмотря на то что во всех трех случаях вызывается одна и та же функция `animal_sound`, ее поведение должно меняться. Похоже, это обусловлено передачей указателей на разные объекты. В терминале 8.5 показан вывод кода из листинга 8.17 в случае, если функция `animal_sound` полиморфна.

Терминал 8.5. Вывод в результате вызова функции

```
Animal: Beeeep
Cat: Meow
Duck: Quake
```

Как видите, мы использовали одну и ту же функцию, `animal_sound`, но с разными указателями, поэтому внутри выполнялись разные операции.

Если вам не удается разобраться в приведенном выше коде, то, пожалуйста, не листайте дальше. Попробуйте перечитать предыдущий раздел.

В этом полиморфном коде подразумевается наличие отношения наследования между классом `Animal` и двумя другими классами, `Cat` и `Duck`, поскольку мы должны иметь возможность приводить указатели `duck_t` и `cat_t` к типу `animal_t`. Еще одна особенность данного кода состоит в том, что для использования преимуществ этой разновидности полиморфизма нам следует применять первый подход к наследованию в C.

Вы, наверное, помните, что в первом подходе у дочернего класса был доступ к приватной реализации родительского класса, поэтому здесь структурная переменная типа `animal_t` должна быть первым полем в структурах атрибутов `duck_t` и `cat_t`. В листинге 8.18 показаны отношения между этими тремя классами.

Листинг 8.18. Определение структур атрибутов для классов Animal, Cat и Duck

```
typedef struct {
  ...
} animal_t;

typedef struct {
  animal_t animal;
  ...
} cat_t;

typedef struct {
  animal_t animal;
  ...
} duck_t;
```

Благодаря такой конфигурации мы можем приводить указатели `duck_t` и `cat_t` к типу `animal_t` и затем использовать одни и те же поведенческие функции для обоих дочерних классов.

Итак, было показано, как должны себя вести полиморфные функции и как следует определить отношение наследования между классами. Теперь осталось выяснить, как реализовать это полиморфное поведение. Иными словами, мы должны обсудить сам механизм, который стоит за полиморфизмом.

Представим, что функция `animal_sound` определена так, как показано в листинге 8.19. Независимо от того, какой указатель она получает в качестве аргумента, ее поведение остается неизменным, поэтому без отдельного внутреннего механизма ее вызовы не будут полиморфными. Данный механизм будет показан в примере 8.4, который мы рассмотрим чуть позже.

Листинг 8.19. Функция animal_sound еще не полиморфная!

```
void animal_sound(animal_t* ptr) {
  printf("Animal: Beeeep");
}

// Эти вызовы могли бы быть полиморфными, но таковыми НЕ являются!
animal_sound(animal);
animal_sound((struct animal_t*)cat);
animal_sound((struct animal_t*)duck);
```

Как вы вскоре увидите, вызов поведенческой функции `animal_sound` с разными параметрами не влияет на ее логику; то есть она не полиморфная (терминал 8.6). В примере 8.4 мы это исправим.

Терминал 8.6. Результат вызовов функции из листинга 8.19

```
Animal: Beeeep
Animal: Beeeep
Animal: Beeeep
```

Какой же внутренний механизм позволит нам сделать поведение функций полиморфным? На этот вопрос я отвечу ниже, но прежде объясню, зачем нам вообще нужен полиморфизм.

Зачем нужен полиморфизм

Прежде чем продолжать разговор о том, как мы будем реализовывать полиморфизм в языке C, следует уделить внимание причинам, по которым нам нужна эта концепция. Полиморфизм прежде всего нужен потому, что мы хотим использовать один и тот же код при работе с разными подтипами базового типа. Соответствующие примеры будут показаны чуть позже.

Нам не хочется модифицировать нашу логику при добавлении в систему новых подтипов или когда один из подтипов меняет свое поведение. Конечно, при появлении новой возможности нельзя совсем обойтись без обновления кода — какие-то изменения обязательно потребуются. Но благодаря полиморфизму мы можем существенно уменьшить количество необходимых изменений.

Еще один повод для использования полиморфизма связан с понятием *абстракции*. Абстрактные типы (или классы) обычно содержат какие-то неопределенные или нереализованные поведенческие функции, которые необходимо переопределять в дочерних классах, и полиморфизм играет в этом ключевую роль.

Поскольку мы хотим писать свою логику с помощью абстрактных типов, нам нужно как-то вызывать подходящую реализацию при работе с ними. Здесь тоже помогает полиморфизм. Полиморфное поведение необходимо в любом языке, иначе стоимость сопровождения крупных проектов будет быстро расти — например, при добавлении в код нового подтипа.

Итак, мы убедились, насколько важен полиморфизм. Теперь пришло время узнать, как реализовать его в языке C.

Полиморфное поведение в языке C

Чтобы получить полиморфизм в C, необходимо использовать первый подход к реализации наследования, который мы исследовали. Вдобавок можно применять *указатели на функции*. Однако на сей раз их следует оформить в виде полей структуры атрибутов. Рассмотрим это на нашем примере со звуками, которые издают животные.

У нас есть три класса: `Animal`, `Cat` и `Duck`. Последние два — подклассы первого. У каждого из них есть по одному заголовку и исходнику. У класса `Animal` есть также дополнительный приватный заголовочный файл с определением его структуры атрибутов; это нужно в связи с тем, что мы выбрали первый подход к реализации наследования. Приватный заголовок будет использоваться классами `Cat` и `Duck`.

В листинге 8.20 показан публичный интерфейс класса `Animal`.

Листинг 8.20. Публичный интерфейс класса Animal (ExtremeC_examples_chapter8_4_animal.h)

```
#ifndef EXTREME_C_EXAMPLES_CHAPTER_8_4_ANIMAL_H
#define EXTREME_C_EXAMPLES_CHAPTER_8_4_ANIMAL_H

// предварительное объявление
struct animal_t;

// аллокатор памяти
struct animal_t* animal_new();

// конструктор
void animal_ctor(struct animal_t*);

// деструктор
void animal_dtor(struct animal_t*);

// поведенческие функции
void animal_get_name(struct animal_t*, char*);
void animal_sound(struct animal_t*);

#endif
```

У класса `Animal` есть две поведенческие функции. Первая, `animal_sound`, должна быть полиморфной и доступной для переопределения со стороны дочерних классов; вторая, `animal_get_name`, будет обычной функцией, которую дочерние классы не смогут переопределять.

В листинге 8.21 представлено приватное определение структуры атрибутов `animal_t`.

Листинг 8.21. Приватный заголовок класса Animal (ExtremeC_examples_chapter8_4_animal_p.h)

```
#ifndef EXTREME_C_EXAMPLES_CHAPTER_8_4_ANIMAL_P_H
#define EXTREME_C_EXAMPLES_CHAPTER_8_4_ANIMAL_P_H

// тип указателя, необходимого для обращения к разным версиям animal_sound
typedef void (*sound_func_t)(void*);

// предварительное объявление
typedef struct {
  char* name;
  // этот член класса является указателем на функцию,
  // которая отвечает за вывод звуков
  sound_func_t sound_func;
} animal_t;

#endif
```

При использовании полиморфизма каждый дочерний класс может предоставить собственную версию функции `animal_sound`. Иными словами, каждый подкласс может переопределить функцию, унаследованную от родительского. Поэтому любой потомок, который хочет выполнить переопределение, должен иметь свою разновидность функции. В случае с функцией `animal_sound` это означает, что мы будем вызывать ту ее версию, которую переопределил потомок.

Вот почему мы используем указатели на функции. В каждом экземпляре `animal_t` будет указатель, предназначенный специально для операции `animal_sound`; он станет ссылаться на определение соответствующей полиморфной функции внутри класса.

Для каждой полиморфной поведенческой функции следует предусмотреть отдельный указатель. Здесь я покажу, как с помощью такого указателя вызывать подходящую функцию в каждом подклассе. Иными словами, вы увидите то, как на самом деле работает полиморфизм.

В листинге 8.22 показано определение класса `Animal`.

Листинг 8.22. Определение класса Animal (ExtremeC_examples_chapter8_4_animal.c)

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

#include "ExtremeC_examples_chapter8_4_animal_p.h"

// родительское определение animal_sound, которое используется по умолчанию
void __animal_sound(void* this_ptr) {
  animal_t* animal = (animal_t*)this_ptr;
  printf("%s: Beeeep\n", animal->name);
}

// аллокатор памяти
animal_t* animal_new() {
  return (animal_t*)malloc(sizeof(animal_t));
}

// конструктор
void animal_ctor(animal_t* animal) {
  animal->name = (char*)malloc(10 * sizeof(char));
  strcpy(animal->name, "Animal");
  // присваиваем указателю на функцию адрес определения по умолчанию
  animal->sound_func = __animal_sound;
}

// деструктор
void animal_dtor(animal_t* animal) {
  free(animal->name);
}
```

```
// поведенческие функции
void animal_get_name(animal_t* animal, char* buffer) {
  strcpy(buffer, animal->name);
}

void animal_sound(animal_t* animal) {
  // вызываем функцию, на которую ссылается указатель
  animal->sound_func(animal);
}
```

Само полиморфное поведение находится в функции `animal_sound`. На случай, если дочерний класс решит ее не переопределять, предусмотрено поведение по умолчанию — приватная функция `__animal_sound`. В следующей главе вы увидите, что у полиморфных поведенческих функций есть определение по умолчанию, которое наследуется и используется, если подкласс не предоставляет переопределенную версию.

Идем дальше. Внутри конструктора `animal_ctor` мы сохраняем в поле `sound_func` объекта `animal` адрес `__animal_sound`. В такой конфигурации этот указатель на определение по умолчанию, `__animal_sound`, наследуется каждым потомком.

И в завершение внутри поведенческой функции `animal_sound` вызывается операция, на которую указывает поле `sound_func`. Это поле — указатель на фактическое определение операции, роль которого в данном случае играет `__animal_sound`. Обратите внимание: `animal_sound`, в сущности, перенаправляет вызов к настоящей поведенческой функции.

Таким образом, если поле `sound_func` указывает на другую функцию, именно она будет выполняться при вызове `animal_sound`. Мы будем использовать этот прием, чтобы переопределить стандартную операцию `sound` в классах `Cat` и `Duck`.

Итак, посмотрим, как выглядят эти классы. В следующих листингах показан публичный интерфейс класса `Cat` и его приватное определение. Начнем с заголовочного файла (листинг 8.23).

Листинг 8.23. Публичный интерфейс класса Cat (ExtremeC_examples_chapter8_4_cat.h)

```
#ifndef EXTREME_C_EXAMPLES_CHAPTER_8_4_CAT_H
#define EXTREME_C_EXAMPLES_CHAPTER_8_4_CAT_H

// предварительное объявление
struct cat_t;

// аллокатор памяти
struct cat_t* cat_new();

// конструктор
void cat_ctor(struct cat_t*);
```

```
// деструктор
void cat_dtor(struct cat_t*);

// все поведенческие функции наследуются от класса Animal

#endif
```

Как вы вскоре увидите, этот код унаследует операцию `sound` от своего родительского класса, `Animal`.

В листинге 8.24 показано определение класса `Cat`.

Листинг 8.24. Приватная реализация класса Cat (ExtremeC_examples_chapter8_4_cat.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "ExtremeC_examples_chapter8_4_animal.h"
#include "ExtremeC_examples_chapter8_4_animal_p.h"

typedef struct {
  animal_t animal;
} cat_t;

// определяем новое поведение для операции sound
void __cat_sound(void* ptr) {
  animal_t* animal = (animal_t*)ptr;
  printf("%s: Meow\n", animal->name);
}

// аллокатор памяти
cat_t* cat_new() {
  return (cat_t*)malloc(sizeof(cat_t));
}

// конструктор
void cat_ctor(cat_t* cat) {
  animal_ctor((struct animal_t*)cat);
  strcpy(cat->animal.name, "Cat");
  // указываем на новую поведенческую функцию
  // переопределение происходит именно здесь
  cat->animal.sound_func = __cat_sound;
}

// деструктор
void cat_dtor(cat_t* cat) {
  animal_dtor((struct animal_t*)cat);
}
```

Как видите, мы определили новую функцию, `__cat_sound`, которая будет издавать кошачьи звуки. Затем внутри конструктора сделали так, чтобы указатель `sound_func` ссылался на эту функцию.

Теперь происходит переопределение, и с этого момента все объекты `cat` будут вызывать `__cat_sound` вместо `__animal_sound`. Тот же прием используется и для класса `Duck`.

Публичный интерфейс класса `Duck` показан в листинге 8.25.

Листинг 8.25. Публичный интерфейс класса Duck (ExtremeC_examples_chapter8_4_duck.h)

```
#ifndef EXTREME_C_EXAMPLES_CHAPTER_8_4_DUCK_H
#define EXTREME_C_EXAMPLES_CHAPTER_8_4_DUCK_H

// предварительное объявление
struct duck_t;

// аллокатор памяти
struct duck_t* duck_new();

// конструктор
void duck_ctor(struct duck_t*);

// деструктор
void duck_dtor(struct duck_t*);

// все поведенческие функции наследуются от класса Animal

#endif
```

Как видите, это очень похоже на класс `Cat`. Посмотрим на приватное определение класса `Duck` (листинг 8.26).

Листинг 8.26. Приватная реализация класса Duck (ExtremeC_examples_chapter8_4_duck.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "ExtremeC_examples_chapter8_4_animal.h"
#include "ExtremeC_examples_chapter8_4_animal_p.h"

typedef struct {
  animal_t animal;
} duck_t;

// определяем новое поведение для операции sound
void __duck_sound(void* ptr) {
```

```
  animal_t* animal = (animal_t*)ptr;
  printf("%s: Quacks\n", animal->name);
}

// аллокатор памяти
duck_t* duck_new() {
  return (duck_t*)malloc(sizeof(duck_t));
}

// конструктор
void duck_ctor(duck_t* duck) {
  animal_ctor((struct animal_t*)duck);
  strcpy(duck->animal.name, "Duck");
  // указываем на новую поведенческую функцию
  // переопределение происходит именно здесь
  duck->animal.sound_func = __duck_sound;
}

// деструктор
void duck_dtor(duck_t* duck) {
  animal_dtor((struct animal_t*)duck);
}
```

Данная методика используется для переопределения стандартной операции `sound`. Мы определили новую приватную поведенческую функцию, `duck_sound`, которая издает утиные звуки, и на нее теперь ссылается указатель `sound_func`. В целом это то, как полиморфизм реализован в C++. Мы вернемся к этому в следующей главе.

Наконец, в листинге 8.27 представлена главная функция примера 8.4.

Листинг 8.27. Главная функция примера 8.4 (ExtremeC_examples_chapter8_4_main.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// только публичные интерфейсы
#include "ExtremeC_examples_chapter8_4_animal.h"
#include "ExtremeC_examples_chapter8_4_cat.h"
#include "ExtremeC_examples_chapter8_4_duck.h"

int main(int argc, char** argv) {
  struct animal_t* animal = animal_new();
  struct cat_t* cat = cat_new();
  struct duck_t* duck = duck_new();

  animal_ctor(animal);
  cat_ctor(cat);
  duck_ctor(duck);
```

```
  animal_sound(animal);
  animal_sound((struct animal_t*)cat);
  animal_sound((struct animal_t*)duck);

  animal_dtor(animal);
  cat_dtor(cat);
  duck_dtor(duck);

  free(duck);
  free(cat);
  free(animal);
  return 0;
}
```

Здесь используются только публичные интерфейсы классов `Animal`, `Cat` и `Duck`. Поэтому функция `main` ничего не знает об их внутренней реализации. Чтобы продемонстрировать полиморфизм в действии, мы вызываем функцию `animal_sound`, передавая ей разные указатели. Посмотрим на вывод, который генерирует этот пример.

В терминале 8.7 показано, как скомпилировать и запустить пример 8.4.

Терминал 8.7. Компиляция, запуск и вывод примера 8.4

```
$ gcc -c ExtremeC_examples_chapter8_4_animal.c -o animal.o
$ gcc -c ExtremeC_examples_chapter8_4_cat.c -o cat.o
$ gcc -c ExtremeC_examples_chapter8_4_duck.c -o duck.o
$ gcc -c ExtremeC_examples_chapter8_4_main.c -o main.o
$ gcc animal.o cat.o duck.o main.o -o ex8_4.out
$ ./ex8_4.out
Animal: Beeeep
Cat: Meow
Duck: Quake
$
```

Как видите, в классовом языке программирования поведенческие функции, которые должны быть полиморфными, требуют особого внимания и обращения. В противном случае поведенческая функция, не имеющая представленного выше внутреннего механизма, не может быть полиморфной. Вот почему эти функции называются особым образом и в языках вроде C++ обозначаются специальными ключевыми словами. Это *виртуальные функции* — операции, которые могут быть переопределены дочерними классами. Они должны отслеживаться компилятором, и при их переопределении в соответствующих объектах должны использоваться указатели на сами операции. Во время выполнения с помощью этих указателей вызывается подходящая версия функции.

В следующей главе мы покажем, как объектно-ориентированные отношения между классами организованы в C++. Кроме того, вы узнаете, как в этом языке реализован полиморфизм. Еще мы обсудим *абстракцию данных* — прямое следствие полиморфизма.

Резюме

Здесь мы продолжили наше исследование концепций ООП, начатое в предыдущей главе. Были рассмотрены следующие темы:

- мы выяснили, как работает наследование, и обсудили два подхода к его реализации в языке C;
- первый подход дает прямой доступ ко всем приватным атрибутам родительского класса; второй более консервативен и прячет эти атрибуты;
- мы сравнили оба подхода и увидели, что каждый из них подходит для определенных сценариев использования;
- мы исследовали полиморфизм. Говоря простым языком, эта концепция позволяет создавать разные версии одного и того же поведения и затем вызывать те из них, которые нам нужны, с помощью публичного API абстрактного супертипа;
- мы обсудили, как писать полиморфный код на C, и увидели, какую роль играют функции на указатели при выборе подходящей версии конкретной операции на этапе выполнения.

Мы подходим к последней главе, посвященной объектной ориентированности. В ней мы исследуем то, как в C++ работают инкапсуляция, наследование и полиморфизм. Кроме того, обсудим принцип абстракции данных и покажем, как он привел к появлению причудливых *абстрактных классов*, из которых нельзя создавать объекты.

9 Абстракция данных и ООП в C++

Это последняя глава, посвященная ООП в C. В ней мы рассмотрим оставшиеся темы и познакомимся с новой парадигмой программирования. Кроме того, исследуем язык C++ и посмотрим, как у него внутри реализованы объектно-ориентированные концепции.

Данная глава охватывает следующие темы.

- Вначале мы обсудим *абстракцию данных*. Это будет продолжение нашего разговора о наследовании и полиморфизме, и так мы завершим обсуждение ООП в языке C. Вы увидите, как абстракция данных помогает проектировать максимально расширяемые объектные модели с минимальным количеством зависимостей между их разными компонентами.
- Мы поговорим о том, как объектно-ориентированные концепции были реализованы в популярном компиляторе для C++, `g++`. В ходе разговора вы узнаете, насколько близка эта реализация к тем подходам, которые мы обсуждали ранее.

Начнем с абстракции данных.

Абстракция данных

Понятие абстракции данных может иметь разные значения в разных сферах науки и техники. Но в программировании, и особенно в ООП, оно, в сущности, относится к *абстрактным типам данных*. В классовой объектной ориентированности это то же самое, что и *абстрактные классы*. Абстрактными называют специальные классы, из которых нельзя создавать объекты; они не завершены и не готовы для использования в создании объектов. Так зачем же они нужны? Дело в том, что, работая с абстрактными и обычными типами данных, мы можем избежать возникновения жестких зависимостей между различными частями кода.

Например, между классами *Человек* и *Яблоко* могут быть следующие отношения:

Объект класса Человек ест объект класса Яблоко.

Объект класса Человек ест объект класса Апельсин.

Если к *Яблоку* и *Апельсину* нужно добавить другие классы, которые может есть объект класса *Человек*, то у последнего появятся дополнительные связи. Вместо этого мы можем создать абстрактный класс *Фрукт*, который будет родителем *Яблока* и *Апельсина*. Это позволит ограничиться лишь одним отношением — между *Человеком* и *Фруктом*. Таким образом, мы можем свести два предыдущих утверждения к одному:

Объект класса Человек ест объект одного из подтипов класса Фрукт.

Класс *Фрукт* — абстрактный, поскольку не несет в себе информацию о форме, вкусе, запахе, цвете и многих других атрибутах, свойственных фруктам. Значения этих атрибутов становятся известными, только когда мы получаем яблоко или апельсин. Классы *Яблоко* и *Апельсин* называют *конкретными типами*.

Мы можем повысить уровень абстракции. Класс *Человек* может также есть *Салат* и *Шоколад*. Поэтому допустимо сказать следующее:

Объект типа Человек может есть объект одного из подтипов класса Пища.

Как видите, *Пища* находится на еще более высоком уровне абстракции, чем *Фрукт*. Это отличный подход к проектированию объектных моделей с минимальной зависимостью от конкретных типов и возможностью легкого расширения, если в будущем в систему понадобится добавить другие конкретные типы.

Возвращаясь к предыдущему примеру, мы могли бы абстрагироваться еще сильнее, используя тот факт, что *Человеку* свойственно *потреблять пищу*. Таким образом, можно получить еще более абстрактное утверждение:

Объект одного из подтипов класса Едок ест объект одного из подтипов класса Пища.

Мы можем продолжить абстрагирование объектной модели и подобрать типы данных, уровень абстракции которых выше того, который необходим для решения нашей задачи. Это так называемое *чрезмерное абстрагирование*. Оно случается, когда вы пытаетесь создать абстрактные типы, не имеющие реального применения в контексте ваших текущих или будущих потребностей. Этого следует избегать любой ценой, поскольку абстракция, несмотря на все свои преимущества, может создать проблемы.

При определении того, насколько сильно нужно абстрагировать объектную модель, можно руководствоваться *принципом абстракции*. На посвященной ему странице в «Википедии», https://en.wikipedia.org/wiki/Abstraction_principle_(computer_programming), утверждается следующее.

> *Каждая существенная часть функциональности программы должна быть реализована только на одном участке кода. В целом, если разные участки кода имеют похожие функции, то их имеет смысл объединить путем абстрагирования того, чем они отличаются.*

На первый взгляд в этом утверждении нет никаких признаков объектной ориентированности или наследования. Но, немного подумав, можно заметить, что данный принцип лежит в основе рассмотренных нами отношений между объектами. Поэтому на практике, если вы не ожидаете, что конкретная логика будет варьироваться в будущем, на данном этапе не нужно вводить абстракцию.

Для создания абстракций в языках программирования используются две возможности: наследование и полиморфизм. Абстрактный класс, такой как *Пища*, — супертип по отношению к своим конкретным классам, таким как *Яблоко*. И это достигается за счет наследования.

Полиморфизм тоже играет важную роль. Некое поведение абстрактного типа *не может* иметь реализации по умолчанию на этом абстрактном уровне. Например, у атрибута *вкус*, реализованного в классе *Пища* в виде поведенческой функции `eatable_get_taste`, не может быть определенного значения. Иными словами, мы не можем создать объект напрямую из класса *Пища*, не зная, как определить поведенческую функцию `eatable_get_taste`.

Эта функция может иметь определение, только когда дочерний класс является достаточно конкретным. Например, мы знаем, что у *Яблок* атрибут *вкус* должен быть *сладким* (предположим, что кислых яблок не существует). Вот где на помощь приходит полиморфизм. Он позволяет дочернему классу переопределить поведение родителя и, к примеру, вернуть подходящий *вкус*.

Как вы помните по предыдущей главе, поведенческие функции, которые могут переопределяться дочерними классами, называются *виртуальными*. Имейте в виду, что у виртуальной функции может вообще не быть никакого определения. И это, конечно, делает абстрактным класс, к которому она принадлежит.

При достижении определенного уровня абстракции у нас получаются классы без каких-либо атрибутов или определений по умолчанию, а только с виртуальными функциями. Такие классы называются *интерфейсами*. Иными словами, они предоставляют возможности, но не предлагают никакой реализации; обычно с их помощью в программных проектах создаются зависимости между различными компонентами. В наших предыдущих примерах роль интерфейсов играли классы *Едок* и *Пища*. Обратите внимание: как и в случае с абстрактными классами, из интерфейсов нельзя создавать объекты. В представленном ниже коде показано, почему эту концепцию нельзя воплотить в языке C.

Листинг 9.1 реализует упомянутый выше интерфейс *Пища* (`Eatable`) на C, используя приемы для организации наследования и полиморфизма, с которыми вы познакомились в предыдущей главе.

Листинг 9.1. Интерфейс Eatable в C

```
typedef enum {SWEET, SOUR} taste_t;

// тип указателя на функцию
typedef taste_t (*get_taste_func_t)(void*);
```

```
typedef struct {
  // указатель на определение виртуальной функции
  get_taste_func_t get_taste_func;
} eatable_t;

eatable_t* eatable_new() { ... }

void eatable_ctor(eatable_t* eatable) {
  // у этой виртуальной функции нет определения по умолчанию
  eatable->get_taste_func = NULL;
}

// виртуальная поведенческая функция
taste_t eatable_get_taste(eatable_t* eatable) {
  return eatable->get_taste_func(eatable);
}
```

Внутри конструктора мы присвоили функции `get_taste_func` значение `NULL`. Поэтому вызов виртуальной функции `eatable_get_taste` повлечет ошибку сегментации. Это практический аспект того, почему из интерфейса `Eatable` нельзя создавать объекты; другие причины обусловлены самим понятием «интерфейс» и правилами проектирования.

В листинге 9.2 показано, как создание объекта из интерфейса `Eatable` может привести к сбою и почему так не следует делать, хотя это никоим образом не противоречит синтаксису языка C как таковому.

Листинг 9.2. Ошибка сегментации при создании объекта из интерфейса Eatable и вызове из него чисто сугубо виртуальной функции

```
eatable_t *eatable = eatable_new();
eatable_ctor(eatable);
taste_t taste = eatable_get_taste(eatable); // Ошибка сегментации!
free(eatable);
```

Чтобы избежать создания объекта абстрактного типа, из публичного интерфейса класса можно убрать *функцию-аллокатор*. Если вы помните подходы к реализации наследования в языке C, которые мы рассматривали в предыдущей главе, то в результате удаления аллокатора создание объектов из структуры атрибутов родителя становится доступным только дочерним классам.

Внешний код больше не может этого делать. Представьте, что в предыдущем примере мы не хотели, чтобы внешний код мог создавать какие-либо объекты из структуры `eatable_t`. Для достижения этого структуру атрибутов нужно предварительно объявить и тем самым сделать ее неполным типом. Затем из класса следует удалить публичный аллокатор `eatable_new`.

Если подытожить, то для получения абстрактного класса в C необходимо обнулить указатели на виртуальные функции, у которых не должно быть определения

по умолчанию на абстрактном уровне. На высочайшем уровне абстракции у нас получится интерфейс, у которого обнулены все указатели на функции. Чтобы внешний код не мог создавать объекты абстрактных типов, следует удалить функцию-аллокатор из публичного интерфейса.

В следующем разделе мы сравним аналогичные объектно-ориентированные возможности в C и C++. Это поможет вам понять, как C++ получился из чистого языка C.

Объектно-ориентированные концепции в C++

В этом разделе мы сравним то, что нам удалось реализовать на языке C, с помощью внутренних механизмов известного компилятора `g++`, предназначенных для поддержки инкапсуляции, наследования, полиморфизма и абстракции данных в C++.

Я хочу показать, что для реализации объектно-ориентированных концепций в C и C++ используются похожие методы. Обратите внимание: с этого момента под C++ мы будем понимать реализацию данного языка в компиляторе `g++`, а не его стандарт. У нас нет оснований полагать, что в других компиляторах все реализовано совершенно иначе, но некоторые различия точно присутствуют. Вдобавок отмечу, что `g++` будет использоваться в 64-битной системе под управлением Linux.

Сначала мы применим методики для написания объектно-ориентированного кода на C, рассмотренные ранее, а затем напишем ту же программу на C++. В конце подведем итоги.

Инкапсуляция

Углубиться во внутренности компилятора C++ и проанализировать, как он использует рассмотренные нами методики для создания итогового исполняемого файла, не так-то просто, но существует один элегантный прием, который позволит нам это сделать. Мы сравним инструкции ассемблера, сгенерированные для двух похожих программ на C и C++.

Так мы сможем увидеть: компилятор C++ в конечном счете генерирует тот же ассемблерный код, что и программа на языке C, которая использует уже знакомые нам подходы к реализации ООП.

В примере 9.1 рассматриваются два простых проекта на C и C++, имеющих похожую объектно-ориентированную логику. У нас есть класс `Rectangle` с поведенческой функцией для вычисления площади прямоугольника. Сравним ассемблерный код, который генерируется для этой функции в обеих программах. Версия на языке C показана в листинге 9.3.

Листинг 9.3. Пример инкапсуляции в C (ExtremeC_examples_chapter9_1.c)

```
#include <stdio.h>

typedef struct {
  int width;
  int length;
} rect_t;

int rect_area(rect_t* rect) {
  return rect->width * rect->length;
}

int main(int argc, char** argv) {
  rect_t r;
  r.width = 10;
  r.length = 25;
  int area = rect_area(&r);
  printf("Area: %d\n", area);
  return 0;
}
```

А в листинге 9.4 показана та же программа, только на C++.

Листинг 9.4. Пример инкапсуляции в C++ (ExtremeC_examples_chapter9_1.cpp)

```
#include <iostream>

class Rect {
public:
  int Area() {
    return width * length;
  }
  int width;
  int length;
};

int main(int argc, char** argv) {
  Rect r;
  r.width = 10;
  r.length = 25;
  int area = r.Area();
  std::cout << "Area: " << area << std::endl;
  return 0;
}
```

Сгенерируем ассемблерный код для этих программ (терминал 9.1).

Терминал 9.1. Генерация ассемблерного вывода для кода на C и C++

```
$ gcc -S ExtremeC_examples_chapter9_1.c -o ex9_1_c.s
$ g++ -S ExtremeC_examples_chapter9_1.cpp -o ex9_1_cpp.s
$
```

Теперь откроем файлы `ex9_1_c.s` и `ex9_1_cpp.s` и поищем определение поведенческих функций. В `ex9_1_c.s` следует искать символ `rect_area`, а в `ex9_1_cpp.s` — символ `_ZN4Rect4AreaEv`. Обратите внимание: C++ декорирует имена символов, вот почему вторая строка выглядит так странно. Декорирование имен в C++ обсуждалось в главе 2.

В программе на языке C ассемблерные инструкции функции `rect_area` выглядят следующим образом (терминал 9.2).

Терминал 9.2. Ассемблерный код, сгенерированный для функции rect_area

```
$ cat ex9_1_c.s
...
rect_area:
.LFB0:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    movq    %rdi, -8(%rbp)
    movq    -8(%rbp), %rax
    movl    (%rax), %edx
    movq    -8(%rbp), %rax
    movl    4(%rax), %eax
    imull   %edx, %eax
    popq    %rbp
    .cfi_def_cfa 7, 8
    Ret
    .cfi_endproc
...
$
```

Следующие ассемблерные инструкции были сгенерированы для функции `Rect::Area` (терминал 9.3).

Терминал 9.3. Ассемблерный код, сгенерированный для функции Rect::Area

```
$ cat ex9_1_cpp.s
...
_ZN4Rect4AreaEv:
.LFB1493:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
```

```
    movq    %rdi, -8(%rbp)
    movq    -8(%rbp), %rax
    movl    (%rax), %edx
    movq    -8(%rbp), %rax
    movl    4(%rax), %eax
    imull   %edx, %eax
    popq    %rbp
    .cfi_def_cfa 7, 8
    Ret
    .cfi_endproc
...
$
```

Невероятно, но они абсолютно одинаковы! Мне сложно сказать, как код на C++ превратился в представленные выше ассемблерные инструкции, но в одном я точно уверен: для функций на C и C++ был сгенерирован ассемблерный код с высокой степенью схожести.

Можно сделать вывод: компилятор C++ использует подход, аналогичный тому, с которым мы познакомились при обсуждении *неявной инкапсуляции* в главе 6. В листинге 9.3 (см. выше) можно видеть, что указатель на структуру атрибутов передается функции `rect_area` в качестве первого аргумента.

Обратите внимание на ассемблерные инструкции, выделенные в обоих терминалах жирным шрифтом. Переменные `width` и `length` считываются путем прибавления к адресу памяти, переданному в первом аргументе. Первый аргумент-указатель находится в регистре `%rdi`, что соответствует спецификации *System V ABI*. Из этого можно сделать вывод: компилятор C++ изменил функцию `Area` так, чтобы ее первым аргументом был указатель на сам объект.

Подведем итог. Мы увидели, что C и C++ очень похожи с точки зрения инкапсуляции, по крайней мере в этом простом примере. Посмотрим, можно ли то же самое сказать о наследовании.

Наследование

Наследование легче поддается анализу, чем инкапсуляция. В C++ указатели дочерних типов можно присваивать указателю родительского типа. Кроме того, у дочернего класса должен быть доступ к приватному определению родителя.

Оба факта говорят о том, что в C++ используется первый из двух подходов к наследованию, которые мы обсудили в главе 8. Можете пролистать назад и освежить память, если в этом есть необходимость.

Но не все так просто. C++ поддерживает множественное наследование, которое исключено в первом подходе. В этом подразделе мы исследуем структуру памяти двух объектов, основанных на двух похожих классах на C и C++.

В примере 9.2 содержится два простых класса, один из которых наследует другой, и ни у одного из них нет поведенческих функций. Версия на языке C выглядит так (листинг 9.5).

Листинг 9.5. Пример наследования в C (ExtremeC_examples_chapter9_2.c)

```
#include <string.h>

typedef struct {
  char c;
  char d;
} a_t;

typedef struct {
  a_t parent;
  char str[5];
} b_t;

int main(int argc, char** argv) {
  b_t b;
  b.parent.c = 'A';
  b.parent.d = 'B';
  strcpy(b.str, "1234");
  // чтобы исследовать структуру памяти, здесь нужно создать точку останова
  return 0;
}
```

Версия, написанная на C++, показана в листинге 9.6.

Листинг 9.6. Пример наследования в C++ (ExtremeC_examples_chapter9_2.cpp)

```
#include <string.h>

class A {
public:
  char c;
  char d;
};

class B : public A {
public:
  char str[5];
};

int main(int argc, char** argv) {
  B b;
  b.c = 'A';
  b.d = 'B';
  strcpy(b.str, "1234");
  // чтобы исследовать структуру памяти, здесь нужно создать точку останова
  return 0;
}
```

Для начала скомпилируем программу на языке C и используем gdb, чтобы создать точку останова в последней строчке функции main (терминал 9.4). Когда выполнение остановится, мы сможем исследовать структуру памяти и имеющиеся значения.

Терминал 9.4. Выполнение в gdb примера 9.2, написанного на C

```
$ gcc -g ExtremeC_examples_chapter9_2.c -o ex9_2_c.out
$ gdb ./ex9_2_c.out
...
(gdb) b ExtremeC_examples_chapter9_2.c:19
Breakpoint 1 at 0x69e: file ExtremeC_examples_chapter9_2.c, line 19.
(gdb) r
Starting program: .../ex9_2_c.out

Breakpoint 1, main (argc=1, argv=0x7fffffffe358) at ExtremeC_
examples_chapter9_2.c:20
20    return 0;
(gdb) x/7c &b
0x7fffffffe261: 65 'A'  66 'B'  49 '1'  50 '2'  51 '3'  52 '4'  0 '\000'
(qdb) c
[Inferior 1 (process 3759) exited normally]
(qdb) q
$
```

Итак, мы вывели семь символов, начиная с адреса объекта b: 'A', 'B', '1', '2', '3', '4', '\0'. Сделаем то же самое для кода на C++ (терминал 9.5).

Терминал 9.5. Выполнение в gdb примера 9.2, написанного на C++

```
$ g++ -g ExtremeC_examples_chapter9_2.cpp -o ex9_2_cpp.out
$ gdb ./ex9_2_cpp.out
...
(gdb) b ExtremeC_examples_chapter9_2.cpp:20
Breakpoint 1 at 0x69b: file ExtremeC_examples_chapter9_2.cpp, line 20.
(gdb) r
Starting program: .../ex9_2_cpp.out

Breakpoint 1, main (argc=1, argv=0x7fffffffe358) at ExtremeC_
examples_chapter9_2.cpp:21
21    return 0;
(gdb) x/7c &b
0x7fffffffe251: 65 'A'  66 'B'  49 '1'  50 '2'  51 '3'  52 '4'  0 '\000'
(qdb) c
[Inferior 1 (process 3804) exited normally]
(qdb) q
$
```

Структура памяти в этих двух терминалах и значения, хранящиеся в атрибутах, ничем не отличаются. Не стоит удивляться тому, что в версии на C++ поведенческие функции и атрибуты находятся внутри класса; при выполнении они интер-

претируются отдельно. В C++ атрибуты класса всегда собираются в одном блоке памяти соответствующего объекта, вне зависимости от того, где вы их укажете, а функции всегда отделены от атрибутов. Все это мы уже видели при обсуждении *неявной инкапсуляции* в главе 6.

Этот пример демонстрирует *одиночное наследование*. Но как насчет *множественного*? В предыдущей главе мы выяснили, почему первый подход к реализации наследования в C не допускает наличия нескольких родителей. Еще раз рассмотрим причину в листинге 9.7.

Листинг 9.7. Демонстрация того, почему первый подход к реализации наследования в C не поддерживает больше одного родителя

```
typedef struct { ... } a_t;
typedef struct { ... } b_t;

typedef struct {
  a_t a;
  b_t b;
  ...
} c_t;

c_t c_obj;
a_t* a_ptr = (a_ptr*)&c_obj;
b_t* b_ptr = (b_ptr*)&c_obj;
c_t* c_ptr = &c_obj;
```

Здесь видно, что класс `c_t` хочет быть наследником сразу двух классов: `a_t` и `b_t`. После объявления всех типов мы создаем объект `c_obj`. Дальше создаются разные указатели.

Необходимо отметить, что *все эти указатели должны содержать один и тот же адрес*. Указатели `a_ptr` и `c_ptr` можно безопасно использовать с любыми поведенческими функциями из классов `a_t` и `c_t`, чего нельзя сказать об указателе `b_ptr`; он ссылается на поле в классе `c_t`, которое является объектом `a_t`. Попытка обратиться к этому полю внутри `b_t` с помощью указателя `b_ptr` приведет к непредсказуемому поведению.

В листинге 9.8 показана корректная версия предыдущего кода, в которой с указателями можно работать безопасно.

Листинг 9.8. Демонстрация того, как следует приводить типы, чтобы указатели ссылались на правильные поля

```
c_t c_obj;
a_t* a_ptr = (a_ptr*)&c_obj;
b_t* b_ptr = (b_ptr*)(&c_obj + sizeof(a_t));
c_t* c_ptr = &c_obj;
```

В третьей строке можно видеть, что мы прибавили размер объекта `a_t` к адресу `c_obj`; благодаря этому указатель теперь ссылается на поле `b` внутри `c_t`. Имейте в виду: приведение типов в C не делает ничего магического; оно не влияет на передаваемые

значения (в данном случае на адрес в памяти). После присваивания адрес, указанный справа, в конечном счете будет скопирован в левую часть выражения.

Посмотрим, как тот же код будет выглядеть в C++. Представьте, что у нас есть класс D, который является наследником трех разных классов: A, B и C. В листинге 9.9 показан код примера 9.3.

Листинг 9.9. Множественное наследование в C++ (ExtremeC_examples_chapter9_3.cpp)

```
#include <string.h>

class A {
public:
  char a;
  char b[4];
};

class B {
public:
  char c;
  char d;
};

class C {
public:
  char e;
  char f;
};

class D : public A, public B, public C {
public:
  char str[5];
};

int main(int argc, char** argv) {
  D d;
  d.a = 'A';
  strcpy(d.b, "BBB");
  d.c = 'C';
  d.d = 'D';
  d.e = 'E';
  d.f = 'F';
  strcpy(d.str, "1234");
  A* ap = &d;
  B* bp = &d;
  C* cp = &d;
  D* dp = &d;
  // в этой строке нужно создать точку останова
  return 0;
}
```

Скомпилируем этот пример и запустим его с помощью отладчика gdb (терминал 9.6).

Терминал 9.6. Компиляция и выполнение примера 9.3 с помощью gdb

```
$ g++ -g ExtremeC_examples_chapter9_3.cpp -o ex9_3.out
$ gdb ./ex9_3.out
...
(gdb) b ExtremeC_examples_chapter9_3.cpp:40
Breakpoint 1 at 0x100000f78: file ExtremeC_examples_chapter9_3.
cpp, line 40.
(gdb) r
Starting program: .../ex9_3.out

Breakpoint 1, main (argc=1, argv=0x7fffffffe358) at ExtremeC_
examples_chapter9_3.cpp:41
41    return 0;
(gdb) x/14c &d
0x7fffffffe25a: 65 'A'  66 'B'  66 'B'  66 'B'  0 '\000'    67 'C' 68 'D'
69 'E' 0x7fffffffe262: 70 'F'  49 '1'  50 '2'  51 '3'  52 '4'  0 '\000'
(gdb)
$
```

Как видите, атрибуты находятся по соседству. Это говорит о том, что в памяти объекта d находится сразу несколько объектов родительских классов. Но что насчет указателей ap, bp, cp и dp? В C++ приведение типов в ходе присваивания дочернего указателя родительскому (восходящее приведение) может происходить неявно.

Теперь посмотрим на значения этих указателей во время текущего сеанса работы программы (терминал 9.7).

Терминал 9.7. Вывод адресов, хранящихся в указателях в примере 9.3

```
(gdb) print ap
$1 = (A *) 0x7fffffffe25a
(gdb) print bp
$2 = (B *) 0x7fffffffe25f
(gdb) print cp
$3 = (C *) 0x7fffffffe261
(gdb) print dp
$4 = (D *) 0x7fffffffe25a
(gdb)
```

Как видите, начальный адрес объекта d ($4) совпадает с адресом, на который указывает ap ($1). Это явное свидетельство того, что компилятор C++ размещает объект типа A в первом поле соответствующей структуры атрибутов класса D. Учитывая адреса в указателях и результат выполнения команды x, можно утверждать, что объект типа B и затем объект типа C были размещены на одном участке памяти, принадлежащем объекту d.

Кроме того, эти адреса показывают, что приведение типов в C++ не является пассивной операцией; когда один тип приводится к другому, мы можем выполнять арифметические действия с передаваемым адресом. Например, в листинге 9.9 во время присваивания указателя `bp` в функции `main` к адресу `d` было добавлено 5 байт (или `sizeof(A)`). Это было сделано для того, чтобы преодолеть проблему, с которой мы столкнулись при реализации множественного наследования в C. Теперь эти указатели можно легко использовать во всех поведенческих функциях, и арифметические операции притом не нужно выполнять вручную. Необходимо отметить, что в C и C++ приведение типов отличается, поэтому вы можете столкнуться с непредвиденным поведением, если будете предполагать, что в C++ данная операция такая же пассивная, как и в C.

Теперь пришло время рассмотреть сходства, которые C и C++ имеют в контексте полиморфизма.

Полиморфизм

Сравнение внутренних механизмов, стоящих за полиморфизмом в C и C++, — нелегкая задача. В предыдущей главе был продемонстрирован простой способ реализации полиморфизма в C, но в C++ используется куда более замысловатый подход, хотя основной принцип все тот же. Обобщить то, как мы реализовывали полиморфизм в языке C, можно на примере следующего псевдокода (листинг 9.10).

Листинг 9.10. Псевдокод, демонстрирующий, как в C можно объявлять и определять виртуальные функции

```
// определение типов для указателей на функции
typedef void* (*func_1_t)(void*, ...);
typedef void* (*func_2_t)(void*, ...);
...
typedef void* (*func_n_t)(void*, ...);

// структура атрибутов родительского класса
typedef struct {
  // атрибуты
  ...
  // указатели на функции
  func_1_t func_1;
  func_2_t func_2;
  ...
  func_n_t func_t;
} parent_t;

// приватные определения виртуальных поведенческих функций по умолчанию
void* __default_func_1(void* parent, ...) { // определение по умолчанию
}
void* __default_func_2(void* parent, ...) { // определение по умолчанию
}
```

```
...
void* __default_func_n(void* parent, ...) { // определение по умолчанию
}
// конструктор
void parent_ctor(parent_t *parent) {
  // инициализируем атрибуты
  ...
  // назначаем определения по умолчанию для поведенческих функций
  parent->func_1 = __default_func_1;
  parent->func_2 = __default_func_2;
  ...
  parent->func_n = __default_func_n;
}

// публичные и невиртуальные поведенческие функции
void* parent_non_virt_func_1(parent_t* parent, ...) { // Код }
void* parent_non_virt_func_2(parent_t* parent, ...) { // Код }
...
void* parent_non_virt_func_m(parent_t* parent, ...) { // Код }

// сами публичные виртуальные поведенческие функции
void* parent_func_1(parent_t* parent, ...) {
  return parent->func_1(parent, ...);
}
void* parent_func_2(parent_t* parent, ...) {
  return parent->func_2(parent, ...);
}
...
void* parent_func_n(parent_t* parent, ...) {
  return parent->func_n(parent, ...);
}
```

Как видите, родительский класс должен хранить в своей структуре атрибутов список указателей на функции. Эти указатели (в родительском классе) либо ссылаются на определения виртуальных функций по умолчанию, либо равны нулю. У псевдокласса, представленного в данном листинге, есть `m` невиртуальных поведенческих функций и `n` виртуальных.

Обратите внимание на то, что все поведенческие функции полиморфны. Их называют виртуальными. В некоторых языках, таких как Java, они называются виртуальными методами.

Невиртуальные функции не полиморфны, и при их вызове поведение никогда не будет меняться. Иными словами, это обычная функция, которая просто выполняет логику внутри определения и не перенаправляет вызов другой функции. Для сравнения, виртуальные функции должны перенаправлять вызовы подходящим определениям, установленным в конструкторе родителя или потомка. Если дочерний класс хочет переопределить какие-либо унаследованные виртуальные функции, то должен обновить их указатели.

Вместо типа void* исходящим переменным можно назначить любой другой тип указателя. Я использовал обобщенный указатель с целью показать: функции в псевдокоде могут возвращать что угодно.

В следующем псевдокоде показано, как дочерний класс переопределяет несколько виртуальных функций (листинг 9.11).

Листинг 9.11. Псевдокод, демонстрирующий, как дочерний класс в C может переопределить виртуальные функции, унаследованные от родительского класса

```
Include everything related to parent class ...

typedef struct {
  parent_t parent;
  // дочерние атрибуты
  ...
} child_t;

void* __child_func_4(void* parent, ...) { // переопределение функции
}
void* __child_func_7(void* parent, ...) { // переопределение функции
}

void child_ctor(child_t* child) {
  parent_ctor((parent_t*)child);
  // инициализируем дочерние атрибуты
  ...
  // обновляем указатели на функции
  child->parent.func_4 = __child_func_4;
  child->parent.func_7 = __child_func_7;
}

// поведенческие функции потомка
...
```

Как видите, дочернему классу достаточно обновить лишь несколько указателей в структуре атрибутов родителя. В C++ применяется похожий подход. Когда вы объявляете поведенческую функцию как виртуальную (с помощью ключевого слова `virtual`), C++ создает массив указателей на функции, похожий на тот, который мы только что видели.

Мы также добавили по одному атрибуту с указателем для каждой виртуальной функции, но в C++ это делается более элегантно. Для этого используется массив, который называется *виртуальной таблицей* (virtual table, *vtable*). Она создается непосредственно перед созданием самого объекта и заполняется во время вызова сначала конструктора базового класса, а затем конструктора потомка — точно так же, как было показано в листингах 9.10 и 9.11.

Поскольку виртуальная таблица заполняется только внутри конструкторов, вызова полиморфных методов из конструктора родительского или дочернего класса следует избегать, поскольку их указатели могут еще не подвергнуться обновлению и ссылаться не на то определение.

В завершение нашего разговора о внутренних механизмах для реализации различных объектно-ориентированных концепций в C и C++ рассмотрим абстракцию данных.

Абстрактные классы

Абстракция в C++ возможна благодаря *чистым виртуальным* функциям. Если сделать метод класса виртуальным и присвоить ему ноль, то получится чистая виртуальная функция. Взгляните на следующий пример (листинг 9.12).

Листинг 9.12. Интерфейс Eatable в C++

```
enum class Taste { Sweet, Sour };

// это интерфейс
class Eatable {
public:
  virtual Taste GetTaste() = 0;
};
```

Внутри класса `Eatable` находится виртуальная функция `GetTaste` с нулевым значением. Это чистая виртуальная функция, которая делает весь класс абстрактным. Вы больше не можете создавать объекты типа `Eatable` — язык C++ этого не позволяет. Кроме того, `Eatable` является интерфейсом, поскольку все его функции-члены чисто виртуальные. `GetTaste` можно переопределить в дочернем классе.

В листинге 9.13 показан класс, который переопределяет функцию `GetTaste`.

Листинг 9.13. Два дочерних класса, реализующих интерфейс Eatable

```
enum class Taste { Sweet, Sour };

// это интерфейс
class Eatable {
public:
  virtual Taste GetTaste() = 0;
};

class Apple : public Eatable {
public:
  Taste GetTaste() override {
    return Taste::Sweet;
  }
};
```

Чистые виртуальные функции очень похожи на обычные виртуальные. Адреса их определений точно так же хранятся в виртуальной таблице, но с одним отличием. Указатель на чистую виртуальную функцию изначально равен `NULL`; для сравнения, во время выполнения конструктора указатель на обычную виртуальную функцию должен ссылаться на определение по умолчанию.

В отличие от компилятора языка C, которому ничего не известно об абстрактных типах, компилятор C++ о них знает и при попытке создать из них объект генерирует ошибку компиляции.

В этом разделе мы взяли разные объектно-ориентированные концепции и сравнили то, как они реализуются в C (с помощью методик, с которыми познакомились в предыдущих трех главах) и C++ (задействуя компилятор `g++`). Вы могли убедиться в том, что в большинстве случаев применяемые нами подходы соответствовали методам, которые используются в `g++`.

Резюме

Данную главу мы начали с такого понятия, как абстракция, и затем продемонстрировали сходство между C и C++ в отношении объектно-ориентированных концепций. На этом наше обсуждение ООП завершается.

В этой главе мы:

- вначале поговорили об абстрактных классах и интерфейсах. Мы можем создать интерфейс или частично абстрактный класс, который позволяет создавать конкретные дочерние классы с полиморфным и варьирующимся поведением;
- сравнили ассемблерный код, генерирующий методики, которые мы задействовали для реализации некоторых аспектов ООП в C, и компилятор `g++`. Результаты оказались очень похожими. Отсюда мы сделали вывод о том, что использованные нами подходы могут иметь очень похожие продукты компиляции;
- подробно обсудили виртуальные таблицы;
- увидели, как с помощью чистых виртуальных функций (концепции языка C++, у которой нет аналога в C) можно объявлять виртуальное поведение, не имеющее определения по умолчанию.

Следующая глава посвящена системе Unix, истории ее развития, ее многоуровневой архитектуре и тому, как она связана с C. Мы также рассмотрим историю создания языка C.

10 История и архитектура Unix

Вы, наверное, уже задавались вопросом, что делает глава о Unix посреди книги, посвященной углубленному изучению C. Если нет, то я предлагаю вам спросить себя, каким образом могут переплетаться обе темы, C и Unix, что на это пришлось выделить целых две главы (текущую и следующую)?

Ответ прост: если вам кажется, что они никак не связаны между собой, то вы допускаете большую ошибку. Отношение между ними довольно очевидное; Unix — первая операционная система, реализованная на достаточно высокоуровневом языке программирования, C, который, в свою очередь, был разработан специально для этой цели и получил известность и влияние благодаря Unix. Хотя, конечно, стоит отметить, что C больше не считается языком программирования высокого уровня.

Если бы в 1970-е и 1980-е годы инженеры в Bell Labs решили перейти с C на другой язык для разработки новой версии Unix, то мы бы сегодня говорили именно об этом языке и наша книга называлась бы по-другому. Остановимся на минуту и прочтем высказывание Денниса М. Ритчи, одного из пионеров C, о роли Unix в успехе данного языка.

> *Успех самой системы Unix, несомненно, сыграл самую важную роль; благодаря ему этот язык стал доступен сотням тысяч людей. С другой стороны, конечно, использование C в Unix обеспечило переносимость этой системы на широкий спектр компьютеров, что было важно для ее успеха.*
>
> *Деннис М. Ритчи,*
> *The Development of the C Language*

Сам документ доступен по адресу https://www.bell-labs.com/usr/dmr/www/chist.html.

В процессе чтения этой главы мы:

- кратко пройдемся по истории Unix, включая создание языка C;
- увидим, почему разработка языка C была основана на B и BCPL;
- обсудим многослойную архитектуру Unix и то, как она была спроектирована в соответствии с философией этой системы;

- рассмотрим прикладной пользовательский уровень вместе с кольцом командной оболочки и увидим, как программы задействуют API этого кольца. В рамках данного раздела будут рассмотрены стандарты SUS и POSIX;
- обсудим слой ядра и посмотрим, какие возможности и функции должны присутствовать в ядре Unix;
- поговорим о Unix-устройствах и о том, как их можно использовать в системе Unix.

Начнем главу с обзора истории Unix.

История Unix

В этом разделе мы совершим небольшой экскурс в историю Unix. Мы постараемся сделать его коротким, не углубляясь в подробности. Наша цель — закрепить в вашем воображении неразрывную связь между Unix и C.

Multics OS и Unix

Еще до Unix существовала система Multics OS — совместный проект MIT, General Electric и Bell Labs, запущенный в 1964 году. Это был первый в мире пример рабочей и безопасной операционной системы, что обеспечило ее успех. Multics устанавливали везде, от университетов до правительственных учреждений. Если перенестись обратно в наше время, то все современные ОС заимствуют те или иные идеи из Multics (хоть и косвенно, через Unix).

В 1969 году по разным причинам, о которых мы вскоре поговорим, часть сотрудников Bell Labs, особенно такие пионеры, как Кен Томпсон и Деннис Ритчи, разочаровались в Multics, в результате чего проект был заброшен. Но на этом история не закончилась. Компания Bell Labs разработала собственную, более простую и эффективную операционную систему под названием Unix.

Более подробно о проекте Multics и его истории можно почитать на сайте https://multicians.org/history.html. По ссылке https://www.quora.com/Why-did-Unix-succeed-and-not-Multics также можно найти хорошее объяснение того, почему система Unix выжила, а Multics — нет.

Сравним эти операционные системы. Ниже перечислены их сходства и различия.

- *Внутренняя структура обеих ОС имеет многослойную архитектуру*. Это значит, их архитектура состоит примерно из одних и тех же колец. Особенно это касается ядра и командной оболочки. Следовательно, программисты могут писать свои программы поверх оболочки. Кроме того, Unix и Multics предоставляют ряд собственных утилит, таких как `ls` и `pwd`. В следующих разделах мы рассмотрим различные кольца, которые составляют архитектуру Unix.

- *Системе Multics для работы требовались дорогие ресурсы и оборудование.* Ее нельзя было установить на обычный потребительский компьютер, и это был один из основных недостатков, который открыл Unix дорогу и сделал Multics устаревшей системой.
- *Архитектура Multics была сложной.* В этом состояла основная причина разочарования работников Bell Labs, и именно она, как уже упоминалось, инициировала их уход из данного проекта. Систему Unix изначально пытались сделать простой. В ее первой версии не было даже многозадачности и многопользовательского режима!

Больше о Unix и Multics, а также о событиях, происходивших в ту эпоху, можно почитать в Интернете. Оба проекта были успешными, но Unix удалось преуспеть и дожить до наших дней.

Вдобавок стоит отметить, что компания Bell Labs работала над новой распределенной операционной системой под названием *Plan 9*, которая была основана на проекте Unix (рис. 10.1). Более подробную информацию о ней можно получить в «Википедии»: https://ru.wikipedia.org/wiki/Plan_9.

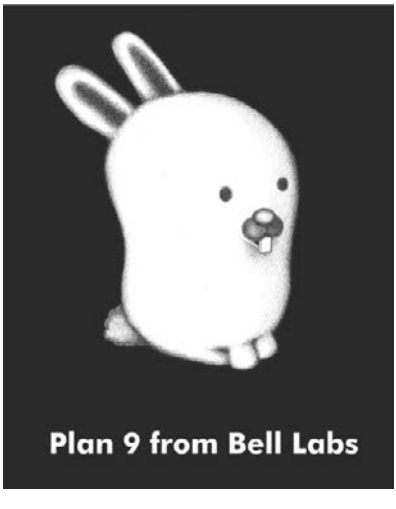

Рис. 10.1. Plan 9 от Bell Labs (из «Википедии»)

Полагаю, достаточно сказать, что система Unix являлась упрощенным выражением идей и инноваций, представленных в Multics; она не была чем-то новым. И на этом я могу завершить обзор истории обеих систем.

До сих пор я ни разу не упомянул о языке C, поскольку на данном этапе его еще не существовало. Первая версия Unix была полностью написана на ассемблере. Язык C начали применять только в версии 4, которая вышла в 1973 году.

Мы уже подходим к главной теме нашего обсуждения, но сначала затронем BCPL и B, которые дали дорогу языку C.

BCPL и B

Язык программирования BCPL был создан Мартином Ричардсом для написания компиляторов. Сотрудники Bell Labs познакомились с этим языком во время работы над Multics. После закрытия проекта компания Bell Labs начала писать Unix на ассемблере. В те времена это был единственный приемлемый вариант!

Например, тот факт, что участники проекта Multics использовали в разработке PL/1, считался чем-то необычным, но в то же время являлся доказательством того,

что операционную систему можно успешно написать на языке программирования высокого уровня. И это стало основной причиной, почему разработка Unix была переведена на другой язык.

Кен Томпсон и Деннис Ритчи продолжили попытки написания модулей операционной системы на языке, отличном от ассемблера. Они попробовали использовать BCPL, но оказалось, что для применения этого языка на компактных компьютерах наподобие DEC PDP-7 его необходимо модифицировать. Эти модификации привели к появлению языка программирования B.

Мы не станем слишком углубляться в особенности языка B. Больше о нем и о том, как он разрабатывался, можно почитать на следующих страницах:

- язык программирования B, https://ru.wikipedia.org/wiki/Би_(язык_программирования);
- *The Development of the C Language*, https://www.bell-labs.com/usr/dmr/www/chist.html.

Вторую статью написал сам Деннис Ритчи. Помимо объяснения процесса создания языка C, в ней приводится ценная информация о проекте B и его характеристиках.

Язык B, будучи системным языком программирования, имел недостатки. В нем не было системы типов, из-за чего в каждой операции он позволял работать только с *машинными словами* (а не с байтами). Это затрудняло его применение на компьютерах с разной длиной машинных слов.

Вот почему язык B со временем претерпел ряд изменений, которые привели к появлению языка *NB* (new B — «новый B»), унаследовавшего структуры из B. В языке B эти структуры не имели типа, но в C стали типизированными. Наконец, в 1973 году вышла четвертая версия Unix, написанная с использованием C, хотя в ней по-прежнему было много ассемблерного кода.

В следующем подразделе я расскажу об отличиях B и C и покажу, почему C остается первоклассным современным языком программирования для написания операционных систем.

Путь к C

Не думаю, что кто-то мог бы объяснить причины создания C после возникновения трудностей с использованием B лучше, чем сам Деннис Ритчи. В этом подразделе мы увидим, почему он, Кен Томпсон и другие инженеры решили создать новый язык программирования, вместо того чтобы писать Unix на B.

Ниже перечислены недостатки B, которые привели к появлению C.

- *Язык B позволял работать только с машинными словами в памяти.* Каждую операцию без исключения нужно было выполнять с машинными словами. В те времена о языке программирования, который позволял бы работать с байта-

ми, можно было только мечтать. Это обуславливалось доступным на тот момент обрудованием, в котором адресация памяти основывалась на словах.

- *Язык B не поддерживал типы.* Если более точно, то B был однотипным языком. Все переменные имели один и тот же тип: машинное слово. Поэтому при наличии строки из 20 символов (плюс нулевой символ в конце) нужно было разделить ее на слова и сохранить в несколько переменных. Например, если слово занимало 4 байта, то для хранения 21-символьной строки требовалось шесть переменных.
- Отсутствие типов привело к тому, что многие операции, ориентированные на байты (такие как алгоритмы для работы со строками), были неэффективно написаны на B. Причина — язык B использовал для работы с памятью слова, а не байты, что не позволяло эффективно управлять многобайтными типами данных, такими как целые числа и строки.
- *Язык B не поддерживал операции с плавающей запятой.* В то время подобные операции становились все более доступными в новом оборудовании, но их поддержка в языке B отсутствовала.
- Несмотря на существование таких компьютеров, как PDP-1, которые могли работать с памятью побайтно, язык B демонстрировал низкую эффективность при адресации байтов памяти. Особенно это касалось указателей, которые могли ссылаться только на машинные слова, но не на байты. То есть если программе нужно было обратиться к определенному байту или диапазону байтов в памяти, то ей приходилось выполнять дополнительные действия, чтобы вычислить индексы подходящих слов.

Недостатки языка B, особенно его медленное развитие и низкая производительность на современных по тем временам компьютерах, вынудили Денниса Ритчи создать новый язык, NB (new B), который в итоге превратился в C.

Этот новый язык пытался исправить изъяны и трудности, присущие B, и стал фактически стандартом в мире системного программирования, заменив ассемблер. Менее чем через десять лет новые версии Unix уже были полностью написаны на C, и с тех пор все ОС, основанные на Unix, полагаются на данный язык и его важнейшую роль в системе.

Как видите, C имеет немного необычную историю появления по сравнению с другими языками программирования; он разрабатывался с учетом полного списка требований, и на сегодня у него нет конкурентов. Такие языки, как Java, Python и Ruby, — более высокоуровневые, но их нельзя использовать для тех же задач. Например, на Java или Python невозможно написать драйвер устройства или модуль ядра; эти языки сами по себе содержат в своей основе слой, написанный на C.

В отличие от многих других языков программирования C является стандартом ISO, и при добавлении в него новых возможностей этот стандарт следует соответствующим образом отредактировать.

В следующем разделе мы обсудим архитектуру Unix. Это одна из фундаментальных концепций, необходимая для понимания жизненного цикла программы в среде Unix.

Архитектура Unix

В этом разделе мы исследуем философию, которой руководствовались создатели Unix, и то, какой результат они ожидали получить, проектируя архитектуру данной системы.

Как уже объяснялось в предыдущем разделе, сотрудники Bell Labs, имевшие отношение к Unix, изначально работали над Multics. Это был более крупный проект со сложной архитектурой, рассчитанный на дорогое оборудование. Но следует помнить, что, несмотря на все трудности, перед Multics стояли масштабные цели. Идея, стоявшая за этим проектом, полностью изменила наше представление об операционных системах.

Несмотря на проблемы и трудности, рассмотренные ранее, идеи, легшие в основу Multics, оказались удачными, поскольку этой системе удалось прожить около 40 лет, вплоть до 2000 года. Более того, проект стал огромным источником доходов для компании, которая им владела.

Такие люди, как Кен Томпсон и его коллеги, привнесли эти идеи в Unix, хотя изначально система должна была быть простой. Multics и Unix пытались реализовать похожие архитектуры, но их судьбы оказались совершенно разными. На рубеже столетий о Multics начали забывать, а вот проект Unix и семейство основанных на нем операционных систем, таких как BSD, продолжают развиваться.

Теперь поговорим о философии Unix. Это просто набор общих требований, на которых основана данная система. Затем мы обсудим ее многокольцевую, многослойную архитектуру и роль каждого кольца в ее поведении.

Философия

Создатели системы Unix уже несколько раз изложили ее философию. Поэтому ее подробное рассмотрение выходит за рамки данной книги. Я лишь проведу общий обзор всех основных аспектов.

Но сначала ознакомьтесь со списком отличных дополнительных материалов, которые могут помочь вам лучше понять эту тему:

- «Википедия», «Философия Unix»: https://ru.wikipedia.org/wiki/Философия_Unix;
- *The Unix Philosophy: A Brief Introduction*: http://www.linfo.org/unix_philosophy.html;

- Эрик Стивен Реймонд, «Искусство программирования для Unix»[1]: https://homepage.cs.uri.edu/~thenry/resources/unix_art/ch01s06.html.

А по ссылке, приведенной далее, вы найдете кардинально противоположный взгляд на философию Unix. В этом мире нет ничего идеального, и потому всегда лучше знать обе стороны: *The Collapse of UNIX Philosophy*: https://kukuruku.co/post/the-collapse-of-the-unix-philosophy/.

Чтобы подытожить перечисленные выше точки зрения, приведу ключевые аспекты философии Unix.

- *Система Unix проектировалась и разрабатывалась в основном для программистов, а не для обычных пользователей.* В связи с этим многие требования, относящиеся к пользовательскому интерфейсу и взаимодействию с пользователями, не являются частью архитектуры Unix.
- *Unix состоит из множества мелких и простых программ.* Каждая из них предназначена для выполнения небольшой и простой задачи. Существует множество примеров таких программ, включая `ls`, `mkdir`, `ifconfig`, `grep` и `sed`.
- *Сложную задачу можно решить путем последовательного выполнения таких мелких и простых программ.* Это значит, в решении крупной и сложной задачи принимает участие несколько программ, каждая из которых может выполняться многократно. Хороший пример — использование скриптов командной оболочки вместо написания программ с нуля. Обратите внимание: такие скрипты зачастую можно переносить между разными системами Unix, что поощряет программистов разбивать свои большие и сложные проекты на мелкие и простые программы.
- *Каждая мелкая и простая программа должна уметь подавать свой вывод на вход другой программе, продолжая цепочку выполнения.* Таким образом, мелкие программы можно объединять в цепочку для выполнения сложных задач. В ней каждая программа играет роль преобразователя, который получает вывод предыдущей программы, изменяет его в соответствии со своей логикой и передает его следующей программе. Отличным примером тому служит объединение команд Unix в *конвейер* с помощью вертикальной черты; например, `ls -l | grep a.out`.
- *Система Unix строго ориентирована на работу с текстом.* Вся конфигурация хранится в текстовых файлах, и командная строка тоже текстовая. Скрипты командной оболочки тоже представляют собой текстовые файлы с простым синтаксисом для написания алгоритмов, которые выполняют другие консольные команды Unix.

[1] Реймонд, Э. С. Искусство программирования для Unix. — М.: Вильямс, 2005.

- *Unix поощряет выбор простоты перед совершенством.* Например, если простое решение работает в большинстве случаев, то не стоит проектировать сложный механизм, который будет лишь немногим лучше.
- Программы, написанные для определенной операционной системы, совместимой с Unix, должны легко переноситься на другие системы данного семейства. Это в основном достигается благодаря наличию единой кодовой базы, которую можно собирать и выполнять на различных Unix-подобных ОС.

Эти требования были выработаны и интерпретированы разными людьми, но в целом их можно считать основными принципами, лежащими в основе философии Unix и, как следствие, сформировавшими архитектуру данной системы.

Если вы уже имели дело с Unix-подобной ОС, такой как Linux, то ваш опыт должен соответствовать изложенным выше принципам. Как уже объяснялось в предыдущем подразделе, посвященном истории Unix, это должна была быть упрощенная версия Multics; именно впечатления от работы с Multics позволили создателям Unix сформировать данную философию.

Но вернемся к основной теме нашей книги. Вы можете спросить, какую роль в этом играет язык C? Дело в том, что почти все ключевые элементы (то есть мелкие и простые программы, лежащие в основе Unix), о которых шла речь выше, написаны на данном языке.

Но лучше один раз показать, чем сто раз рассказать. Поэтому рассмотрим пример. Исходный код программы `ls` из NetBSD можно найти на странице http://cvsweb.netbsd.org/bsdweb.cgi/~checkout~/src/bin/ls/ls.c?rev=1.67. Как вы уже должны знать, программа `ls` занимается лишь тем, что выводит содержимое каталога; пройдя по ссылке, вы увидите: ее простая логика написана на C. Но этим роль C в Unix не ограничивается. Более подробно об этом мы поговорим ниже, где речь пойдет о стандартной библиотеке C.

Многослойная структура Unix

Пришло время исследовать архитектуру Unix. В целом, как уже упоминалось, она имеет *многослойную структуру*, похожую на луковицу. Структура состоит из *колец*, каждое из которых оборачивает внутренние кольца.

Эта знаменитая модель показана на рис. 10.2.

На первый взгляд, эта модель выглядит довольно просто. Но, чтобы как следует в ней разобраться, необходимо написать несколько программ для Unix. Только после этого вы поймете, для чего в действительности предназначено каждое кольцо. Я попробую описать данную архитектуру максимально просто и тем самым заложу фундамент для написания настоящих примеров.

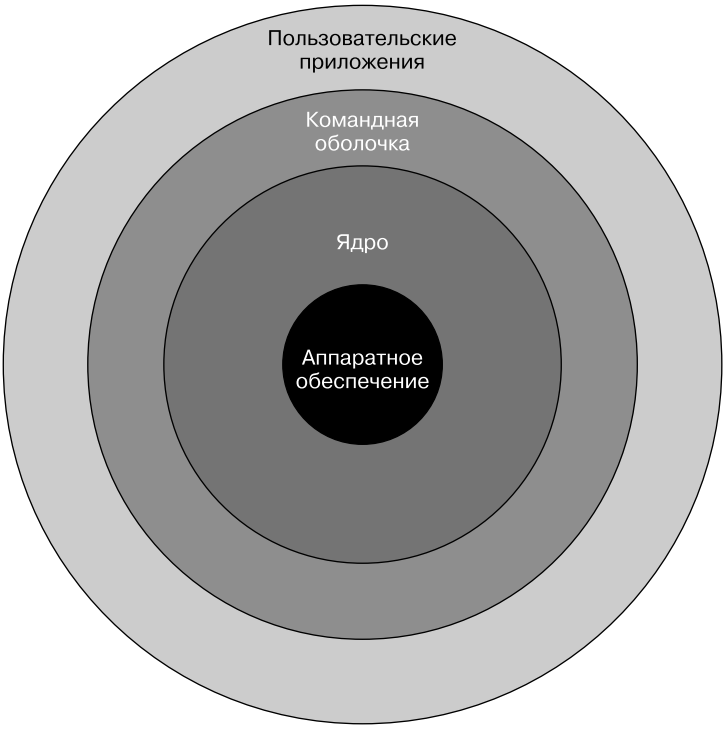

Рис. 10.2. Многослойная модель архитектуры Unix

Разберем многослойную модель, начиная с внутреннего кольца.

В самом центре модели находится *аппаратное обеспечение*. Как мы знаем, основная задача операционной системы — дать пользователю возможность взаимодействовать с оборудованием. Вот почему аппаратное обеспечение занимает центральное место на рис. 10.2. Таким образом нам показывают одну из важнейших задач, стоящих перед Unix: сделать оборудование доступным для программ, которые хотят к нему обращаться. Все, что вы прочитали выше о философии Unix, направлено на предоставление этой возможности максимально оптимальным образом.

В роли кольца, в которое заключено аппаратное обеспечение, выступает *ядро*. Это самая важная часть операционной системы. Оно ближе всего находится к оборудованию и служит оберткой для предоставления возможностей, которыми обладает данное оборудование. Благодаря прямому доступу к аппаратному уровню ядро имеет наивысшие привилегии и может использовать все ресурсы, доступные в системе. Этот неограниченный доступ ко всем компонентам — лучшее обоснование наличия в данной архитектуре остальных колец, которые имеют менее широкие привилегии. На самом деле это послужило причиной разделения между

пространством ядра и *пространством пользователя*. Данную тему мы подробно рассмотрим в текущей и следующей главах.

Обратите внимание: на написание ядра уходит львиная доля усилий, необходимых для создания новой Unix-подобной операционной системы, и, как вы можете видеть, его кольцо изображено более толстым по сравнению с другими. Ядро Unix состоит из множества модулей, каждый из которых — неотъемлемая часть экосистемы. Позже в этой главе мы поближе познакомимся с внутренней структурой ядра Unix.

Следующее кольцо называется *командной оболочкой*. Это обертка, которая позволяет пользовательским приложениям взаимодействовать с ядром и применять его многочисленные функции. Стоит отметить, что данное кольцо само по себе лежит в основе большинства требований, которые пытается удовлетворить философия Unix (см. предыдущий подраздел). Подробнее об этом — в следующих абзацах.

Кольцо командной оболочки состоит из множества мелких программ, в совокупности составляющих набор инструментов, предоставляющий приложениям и пользователям доступ к возможностям ядра. Оно также содержит набор библиотек, полностью написанных на C, с помощью которых программист может разрабатывать новые приложения для Unix.

Используя библиотеки, описанные в спецификации *SUS* (Simple Unix Specification — простая спецификация Unix), кольцо командной оболочки должно предоставлять программистам стандартный и строго определенный интерфейс. Такая стандартизация делает Unix-программы переносимыми или как минимум совместимыми с разными реализациями Unix. Чуть позже я открою вам шокирующие тайны об этом кольце!

Наконец, внешнее кольцо, *пользовательские приложения*, состоит из прикладных программ, предназначенных для выполнения в системах Unix. Речь идет о базах данных, веб-сервисах, почтовых серверах, браузерах, электронных таблицах и текстовых процессорах.

Эти приложения должны использовать API и инструменты, предоставляемые командной оболочкой, не обращаясь к ядру напрямую (через *системные вызовы*, которые мы вскоре обсудим). Этого требует принцип переносимости, являющийся частью философии Unix. Обратите внимание: в нашем текущем контексте под *пользователем* обычно имеется в виду *пользовательское приложение*, а не живой человек, который работает с приложением.

Использование исключительно кольца командной оболочки помогает сделать эти приложения совместимыми с различными Unix-подобными операционными системами, которые не до конца соответствуют SUS. Нам хочется, чтобы одна кодовая база работала как на Unix-совместимых, так и на Unix-подобных ОС.

По ходу чтения данной главы вы постепенно познакомитесь с отличиями этих систем.

Характерная черта архитектуры Unix — тот факт, что внешние кольца должны взаимодействовать с внутренними через определенный интерфейс. На самом деле эти интерфейсы даже важнее колец. Например, нас больше интересует, как использовать возможности ядра, а не его внутреннее устройство, которое зависит от конкретной реализации Unix.

То же касается кольца командной оболочки и интерфейса, которое предоставляется пользовательским приложениям. На самом деле эти интерфейсы будут основной темой обсуждения в двух главах, посвященных Unix. В следующем разделе мы рассмотрим каждое кольцо в отдельности и подробно поговорим о предоставляемых им интерфейсах.

Интерфейс командной оболочки для пользовательских приложений

Чтобы применить возможности, доступные в системе Unix, *живой пользователь* работает либо с терминалом, либо с отдельной графической программой, такой как браузер. И то и другое принадлежит к категории пользовательских приложений (или просто приложений/программ), которые позволяют обращаться к оборудованию через кольцо командной оболочки. Оперативная память, центральный процессор, сетевой адаптер и жесткие диски — все это типичные примеры аппаратного обеспечения, которое большинство Unix-программ задействуют через API кольца командной оболочки. Этот интерфейс будет одной из тем, которые мы обсудим ниже.

С точки зрения разработчика, приложение мало чем отличается от программы. Но в понимании пользователя приложение — программа, с которой можно взаимодействовать через графический или консольный интерфейс (GUI и CLI соответственно), а программа — просто программный компонент, который выполняется на компьютере без какого-либо пользовательского интерфейса (как в случае с сервисом). В книге я не провожу такое разграничение; для нас эти термины взаимозаменяемы.

Для Unix был написан широкий спектр программ на языке C. Базы данных, веб-серверы, почтовые серверы, игры, офисные приложения — это лишь несколько примеров ПО, существующего в среде Unix. Всем им присуща общая черта: их код можно переносить между разными Unix-подобными ОС с минимальными изменениями. Но как это возможно? Как написать такую программу на C, которую можно собрать в разных версиях Unix и на разных аппаратных платформах?

Ответ прост: все системы Unix предоставляют для своей командной оболочки один и тот же *интерфейс прикладного программирования* (Application Programming Interface, API). Фрагмент кода на языке C, который использует только этот стандартный интерфейс, можно собрать и запустить на любой Unix-системе.

Но что именно имеется в виду под предоставлением API? Как уже объяснялось ранее, API — куча заголовочных файлов с объявлениями. Эти заголовки и объявленные в них функции остаются неизменными для всех Unix-систем, но их реализации (то есть статические и динамические библиотеки, написанные для каждой Unix-совместимой ОС) могут отличаться.

Обратите внимание: мы рассматриваем Unix в качестве стандарта, а не операционной системы. Существуют ОС, полностью совместимые с этим стандартом, такие как BSD Unix; мы их называем *Unix-совместимыми*. Есть и частично совместимые системы наподобие Linux, которые мы называем *Unix-подобными*.

Все системы Unix предоставляют примерно одинаковый API для кольца командной оболочки. Например, согласно стандарту Unix функция `printf` всегда должна быть объявлена в заголовочном файле `stdio.h`. Если вам нужно послать что-то в стандартный вывод Unix-совместимой системы, то вы должны использовать `printf` или `fprintf` из заголовка `stdio.h`.

На самом деле заголовок `stdio.h` не является частью C, несмотря на то что он и объявленные в нем функции фигурируют во всех книгах о данном языке. Это часть стандартной библиотеки C, описанной в стандарте SUS. Программе, написанной на C для Unix, ничего не известно о реализации тех или иных функций, таких как `printf` или `fopen`. Иными словами, программы во внешнем кольце воспринимают командную оболочку как некий черный ящик.

В стандарте SUS собраны различные API, предоставляемые кольцом командной оболочки. Этот стандарт развивается консорциумом *The Open Group* и имеет несколько версий, вышедших с момента создания Unix. Самая последняя версия SUS под номером 4 была выпущена в 2008 году, хотя за это время у нее появилось несколько ревизий (в 2013, 2016 и, наконец, 2018 годах).

По адресу http://www.unix.org/version4/GS5_APIs.pdf находится документ, в котором описаны интерфейсы, доступные в SUS версии 4. Как видите, кольцо командной оболочки предоставляет доступ к разного рода API. Одни из них обязательны, а другие — нет. Все они перечислены ниже.

- *Системные интерфейсы* — сюда входят все функции, которыми могут пользоваться любые программы на языке C. В стандарте SUS v4 предусмотрена 1191 функция, которая должна быть реализована в Unix-системе. В таблице, расположенной по приведенной выше ссылке, указано, какие из этих функций являются обязательными или дополнительными в конкретной версии C. Заметьте, что нас интересует версия C99.

- *Заголовочные интерфейсы* — список заголовочных файлов, которые могут быть доступны в Unix-системе, совместимой с SUS v4. В этой версии SUS перечислены 82 заголовочных файла, к которым может обращаться любая программа на языке C. Если пройтись по этому списку, то можно найти много известных заголовков, таких как `stdio.h`, `stdlib.h`, `math.h` и `string.h`. В зависимости от версий Unix и языка C одни из них являются обязательными, а другие — нет. Последние могут отсутствовать, тогда как обязательные непременно хранятся где-то в файловой системе.
- *Вспомогательные интерфейсы* — список консольных утилит или программ командной строки, которые должны быть доступны в Unix-системе, совместимой с SUS v4. Если пройтись по таблицам, то можно встретить много знакомых нам команд, таких как `mkdir`, `ls`, `cp`, `df`, `bc`; всего их насчитывается 160. Обратите внимание: эти программы должны быть написаны поставщиком Unix-системы и входить в итоговый установочный пакет.

  Эти утилиты в основном применяются в терминале или скриптах командной оболочки и нечасто вызываются другими программами на C. Они, как правило, задействуют те же системные интерфейсы, которые доступны обычным программам, написанным для кольца пользовательских приложений.

  В качестве примера ниже приводится ссылка на исходный код утилиты `mkdir`, написанной для системы macOS High Sierra 10.13.6, которая является разновидностью дистрибутива Berkeley Software Distribution (BSD), основанного на Unix. Исходный код опубликован на сайте Apple Open Source в разделе macOS High Sierra (10.13.6) и доступен по адресу https://opensource.apple.com/source/file_cmds/file_cmds-272/mkdir/mkdir.c.

  Если пройти по этой ссылке и просмотреть исходник, то можно заметить, что в нем используются функции `mkdir` и `umask`, объявленные в рамках системных интерфейсов.
- *Сценарный интерфейс* — язык, который используется для написания *скриптов командной оболочки*. С его помощью в основном автоматизируют задачи, в которых применяются консольные утилиты. Этот интерфейс обычно называют *языком командной оболочки* (или *командной строки*).
- *Интерфейсы XCURSES* — набор интерфейсов, которые позволяют программе, написанной на C, взаимодействовать с пользователем с помощью минималистичного текстового GUI.

  На рис. 10.3 можно видеть пример GUI, написанного на реализации XCURSES под названием `ncurses`.

  В SUS v4 интерфейсу XCURSES отводится 379 функций, размещенных в трех заголовках, а также четыре утилиты.

  Многие программы до сих пор применяют XCURSES для более удобного взаимодействия с пользователем. Следует отметить, что интерфейсы на основе XCURSES не нуждаются в графической подсистеме. Благодаря этому с ними можно работать удаленно, по *SSH* (Secure Shell).

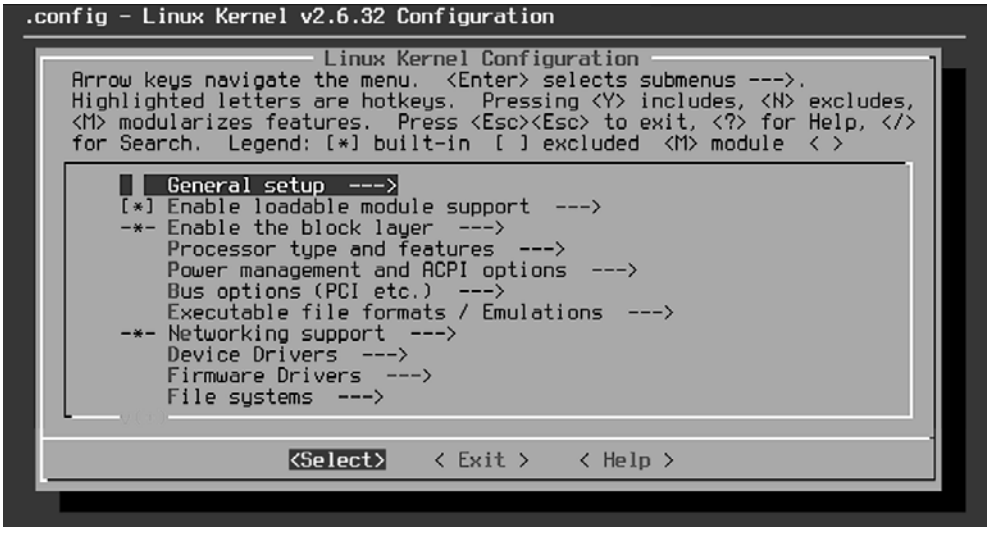

Рис. 10.3. Конфигурационное меню на основе ncurses («Википедия»)

Как видите, в SUS не описывается иерархия файловой системы и то, где можно найти те или иные заголовочные файлы. Этот стандарт лишь указывает на то, какие заголовки должны присутствовать и быть доступны в системе. Согласно широко распространенному соглашению о заголовочных файлах они должны находиться либо в `/usr/include`, либо `/usr/local/include`, но окончательное решение по-прежнему остается за операционной системой и пользователем. Это пути по умолчанию, и в конфигурации системы их можно изменить.

Если объединим системные и заголовочные интерфейсы, а также реализацию доступных функций, которая варьируется в зависимости от разновидности Unix, то получится *стандартная библиотека C*, или *libc*. Иными словами, libc — это набор функций, размещенных в определенных заголовочных файлах в соответствии с SUS, плюс статические и динамические библиотеки с реализациями доступных функций.

Определение libc тесно связано с процессом стандартизации систем Unix. Любая программа на языке C, разработанная в системе Unix, использует libc для взаимодействия с более низкими уровнями ядра и аппаратного обеспечения.

Следует помнить, что не все операционные системы полностью совместимы с Unix. Это касается, например, Microsoft Windows и ОС с ядром Linux, таких как Android. Эти операционные системы Unix-подобны, но не Unix-совместимы. Я использовал оба термина в предыдущих главах, не объясняя их настоящего значения. Теперь пришло время определить их должным образом.

Unix-совместимая система полностью соответствует стандартам SUS, чего нельзя сказать о Unix-подобной системе, которая имеет лишь частичную совместимость. Это значит, Unix-подобные системы соответствуют только определенному подмножеству стандартов SUS. Следовательно, программы, разработанные для одной Unix-совместимой системы, теоретически можно перенести на другую, но перенос на Unix-подобную ОС не гарантирован. Это особенно касается программ, которые переносятся с Linux на другие Unix-совместимые системы или наоборот.

Появление множества Unix-подобных операционных систем, особенно после рождения Linux, создало необходимость в классификации этого конкретного подмножества SUS. Данный стандарт был назван *POSIX* (Portable Operating System Interface — переносимый интерфейс операционных систем). Можно сказать, что POSIX — подмножество SUS, с которым совместимы Unix-подобные системы.

Пройдя по следующей ссылке, можно найти все интерфейсы, которые должны быть доступны в соответствии с POSIX: http://pubs.opengroup.org/onlinepubs/9699919799/.

Как видите, в POSIX и SUS есть похожие интерфейсы. Эти спецификации имеют довольно много общего, однако появление POSIX позволило применить стандарты Unix к более широкому спектру операционных систем.

Unix-подобные ОС, включая большинство дистрибутивов Linux, изначально POSIX-совместимы. Вот почему с Ubuntu можно работать так же, как с FreeBSD.

Но этого нельзя сказать о некоторых других ОС. Например, система Microsoft Windows не POSIX-совместима, но это можно исправить, установив дополнительные инструменты наподобие *Cygwin*¸ POSIX-совместимую среду, которая работает прямо в Windows.

Вышесказанное еще раз подтверждает, что совместимость с POSIX достигается за счет наличия стандартного кольца командной оболочки, а не ядра.

К слову, в 1990-х годах система Microsoft Windows стала совместимой с POSIX, чем наделала много шума. Но со временем эта поддержка устарела.

И SUS, и POSIX описывают необходимые интерфейсы. Оба стандарта определяют, что должно быть доступно, однако не уточняют, как это нужно реализовать. У каждой системы Unix есть своя реализация POSIX или SUS, воплощенная в библиотеках libc, которые являются частью кольца командной оболочки. Иными словами, кольцо командной оболочки Unix-системы содержит реализацию libc, которая предоставляется стандартным образом. Получив запрос, кольцо направляет его ядру для дальнейшей обработки.

Интерфейс ядра для кольца командной оболочки

В предыдущем разделе мы объяснили, что кольцо командной оболочки в Unix предоставляет доступ к интерфейсам, описанным в стандарте SUS или POSIX. Выполнить программную логику в этом кольце можно двумя основными способами: либо через libc, либо с помощью консольных программ. Пользовательское приложение должно быть скомпоновано с библиотеками libc для выполнения процедур командной оболочки или вызывать существующие утилиты, доступные в системе.

Обратите внимание: имеющиеся утилиты сами используют библиотеки libc. Следовательно, мы можем обобщить вышесказанное и утверждать, что все процедуры командной оболочки находятся в библиотеках libc. Это делает стандартную библиотеку C еще более значимой. Если вы хотите создать новую Unix-систему с нуля, то после ядра вам придется написать собственную версию libc.

Если вы читали эту книгу последовательно и уже знакомы с предыдущими главами, то у вас начнет вырисовываться общая картина. Нам нужен был процесс компиляции и механизм компоновки, чтобы спроектировать операционную систему, которая предоставляет интерфейс и реализует набор библиотечных файлов. Вы уже должны заметить, что каждая возможность языка C играет на руку Unix. Чем лучше вы будете понимать отношения между C и Unix, тем более очевидной будет для вас их тесная связь.

Итак, мы разобрались с отношениями между пользовательскими приложениями и кольцом командной оболочки. Теперь покажем, как это кольцо (libc) взаимодействует с кольцом ядра. Но прежде, чем продолжать, следует отметить, что в этом разделе мы не станем объяснять, что такое ядро. Мы будем считать его своеобразным черным ящиком, который предоставляет доступ к определенным функциям.

Для работы с ядром libc (или функции в кольце командной оболочки) в основном используют *системные вызовы*. Чтобы объяснить этот механизм и показать, в каком месте многослойной модели применяются системные вызовы, нам нужен реальный пример.

Нам также следует выбрать конкретную реализацию libc, чтобы иметь возможность проанализировать исходники и найти соответствующие системные вызовы. Я остановился на FreeBSD. Это Unix-подобная операционная система, которая является ответвлением от BSD Unix.

Git-репозиторий FreeBSD находится по адресу https://github.com/freebsd/freebsd. В нем содержатся исходные коды колец ядра и командной оболочки. Исходники libc этой системы можно найти в директории lib/libc.

Начнем с примера 10.1. В нем показана программа, которая просто ждет одну секунду и завершается. Она находится в прикладном кольце, что, несмотря на ее необычную простоту, делает ее пользовательским приложением.

Для начала рассмотрим исходный код примера 10.1 (листинг 10.1).

Листинг 10.1. Пример 10.1, в котором подключается функция sleep из кольца ядра (ExtremeC_examples_chapter10_1.c)

```
#include <unistd.h>

int main(int argc, char** argv) {
  sleep(1);
  return 0;
}
```

Как видите, код подключает заголовочный файл `unistd.h` и вызывает функцию `sleep`; и то и другое — часть интерфейсов, доступных в SUS. Но что происходит дальше, особенно в функции `sleep`? Вы, как программист на C, могли никогда не задаваться этим вопросом, но знание ответа улучшит ваше понимание системы Unix.

Мы всегда используем такие функции `sleep`, `printf` и `malloc`, не зная, как они работают внутри, но сделаем кое-что необычное и попробуем исследовать механизм, с помощью которого libc общается с ядром.

Мы уже знаем, что системные вызовы инициируются кодом, написанным в реализации libc. На самом деле именно так вызываются процедуры ядра. В SUS и впоследствии в POSIX-совместимых системах была предусмотрена программа, которая позволяла отслеживать системные вызовы во время выполнения кода.

С большой долей уверенности можно утверждать, что программа, не делающая никаких системных вызовов, фактически бесполезна. Из этого следует: любой код, который мы пишем, должен использовать системные вызовы, обращаясь к функциям libc.

Скомпилируем предыдущий пример и посмотрим, какие системные вызовы он использует. Для начала выполним команды, показанные в терминале 10.1.

Терминал 10.1. Сборка и запуск примера 10.1 с помощью утилиты truss для отслеживания выполняемых системных вызовов

```
$ cc ExtremeC_examples_chapter10_1.c -lc -o ex10_1.out
$ truss ./ex10_1.out
...
$
```

Как видите, мы воспользовались утилитой `truss`. Ниже приведен отрывок из тематического практического руководства, взятый из документации FreeBSD.

> *Утилита truss отслеживает системные вызовы, выполняемые заданным процессом или программой. Результат по умолчанию записывается в заданный файл или stderr. Для этого отслеживаемый процесс останавливается и перезапускается с помощью ptrace(2).*

Как следует из описания, `truss` — программа для просмотра всех системных вызовов, которые код делает во время выполнения. Аналогичные утилиты доступны в большинстве Unix-подобных систем. Например, в Linux для этого можно использовать `strace`.

В терминале 10.2 показан вывод `truss` в ходе мониторинга системных вызовов, выполняемых кодом из предыдущего примера.

Терминал 10.2. Вывод утилиты truss с системными вызовами, инициированными примером 10.1

```
$ truss ./ex10_1.out
mmap(0x0,32768,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANON,-1,0x0) = 34366160896
(0x800620000)
issetugid()                                     = 0 (0x0)
lstat("/etc",{ mode=drwxr-xr-x ,inode=3129984,size=2560,blksize=32768 }) =
0 (0x0)
lstat("/etc/libmap.conf",{ mode=-rw-r--r-- ,in
ode=3129991,size=109,blksize=32768 }) = 0 (0x0)
openat(AT_FDCWD,"/etc/libmap.conf",O_RDONLY|O_CLOEXEC,00) = 3 (0x3)
fstat(3,{ mode=-rw-r--r-- ,inode=3129991,size=109,blksize=32768 }) = 0 (0x0)
...
openat(AT_FDCWD,"/var/run/ld-elf.
so.hints",O_RDONLY|O_CLOEXEC,00) = 3 (0x3)
read(3,"Ehnt\^A\0\0\0\M^@\0\0\0Q\0\0\0\0"...,128) = 128 (0x80)
fstat(3,{ mode=-r--r--r-- ,inode=7705382,size=209,blksize=32768 }) = 0 (0x0)
lseek(3,0x80,SEEK_SET)                          = 128 (0x80)
read(3,"/lib:/usr/lib:/usr/lib/compat:/u"...,81) = 81 (0x51)
close(3)                                        = 0 (0x0)
access("/lib/libc.so.7",F_OK)                   = 0 (0x0)
openat(AT_FDCWD,"/lib/libc.so.7",O_RDONLY|O_CLOEXEC|O_VERIFY,00) = 3 (0x3)
...
sigprocmask(SIG_BLOCK,{ SIGHUP|SIGINT|SIGQUIT|SIGKILL|SIGPIPE|SIGALRM|
SIGTERM|SIGURG|SIGSTOP|SIGTSTP|SIGCONT|SIGCHLD|SIGTTIN|SIGTTOU|SIGIO|
SIGXCPU|SIGXFSZ|SIGVTALRM|SIGPROF|SIGWINCH|SIGINFO|SIGUSR1|
SIGUSR2 },{ }) = 0 (0x0)
sigprocmask(SIG_SETMASK,{ },0x0)                = 0 (0x0)
sigprocmask(SIG_BLOCK,{ SIGHUP|SIGINT|SIGQUIT|SIGKILL|SIGPIPE|SIGALRM|
SIGTERM|SIGURG|SIGSTOP|SIGTSTP|SIGCONT|SIGCHLD|SIGTTIN|SIGTTOU|SIGIO|
SIGXCPU|SIGXFSZ|SIGVTALRM|SIGPROF|SIGWINCH|SIGINFO|SIGUSR1|
SIGUSR2 },{ }) = 0 (0x0)
sigprocmask(SIG_SETMASK,{ },0x0)                = 0 (0x0)
nanosleep({ 1.000000000 })                      = 0 (0x0)
sigprocmask(SIG_BLOCK,{ SIGHUP|SIGINT|SIGQUIT|SIGKILL|SIGPIPE|SIGALRM|
SIGTERM|SIGURG|SIGSTOP|SIGTSTP|SIGCONT|SIGCHLD|SIGTTIN|SIGTTOU|SIGIO|
SIGXCPU|SIGXFSZ|SIGVTALRM|SIGPROF|SIGWINCH|SIGINFO|SIGUSR1|
SIGUSR2 },{ }) = 0 (0x0)
...
sigprocmask(SIG_SETMASK,{ },0x0)                = 0 (0x0)
```

```
exit(0x0)
process exit, rval = 0
$
```

В этом простом примере сделано много системных вызовов. Некоторые из них относятся к загрузке разделяемых объектных библиотек, особенно на этапе инициализации процесса. Первый системный вызов, выделенный полужирным шрифтом, открывает разделяемый объектный файл `libc.so.7`, содержащий непосредственную реализацию libc для FreeBSD.

В этом же терминале мы видим, что программа делает системный вызов `nanosleep`. Значение, которое она ему передает, равно `1000000000` наносекунд, что эквивалентно 1 секунде.

Системные вызовы подобны вызовам функций. Обратите внимание: у каждого из них есть предопределенный, фиксированный номер, а также имя и список аргументов. Каждый системный вызов выполняет определенное действие. В данном случае `nanosleep` замораживает вызывающий поток выполнения на заданное количество наносекунд.

Больше информации на эту можно получить в *справочнике по системным вызовам* в FreeBSD. В терминале 10.3 показана страница данного справочника, посвященная системному вызову `nanosleep`.

Терминал 10.3. Страница справочника, посвященная системному вызову nanosleep

```
$ man nanosleep
NANOSLEEP(2)             FreeBSD System Calls Manual
NANOSLEEP(2)

NAME
    nanosleep - high resolution sleep

LIBRARY
    Standard C Library (libc, -lc)

SYNOPSIS
    #include <time.h>

    Int
    clock_nanosleep(clockid_t clock_id, int flags,
        const struct timespec *rqtp, struct timespec *rmtp);

    int
    nanosleep(const struct timespec *rqtp, struct timespec *rmtp);

DESCRIPTION
    If the TIMER_ABSTIME flag is not set in the flags argument, then
    clock_nanosleep() suspends execution of the calling thread until either
```

```
    the time interval specified by the rqtp argument has elapsed, or a signal
    is delivered to the calling process and its action is to invoke
    a signalcatching function or to terminate the process. The clock
    used to measure the time is specified by the clock_id argument
...
...
$
```

На приведенной выше странице справочника говорится следующее.

- Вызов `nanosleep` является системным.
- Системный вызов доступен в виде функций `nanosleep` и `clock_nanosleep`, определенных в файле `time.h` и принадлежащих кольцу командной оболочки. Обратите внимание: мы воспользовались функцией `sleep` из `unitsd.h`. Но могли также вызвать две вышеупомянутые функции из `time.h`. Вдобавок следует отметить, что оба заголовочных файла и все приведенные здесь функции, включая те, которые использовались в примере, являются частью SUS и POSIX.
- Если вы хотите вызывать эти функции, то должны скомпоновать ваш исполняемый файл с libc; для этого компоновщику необходимо передать параметр `-lc`. Это может касаться только FreeBSD.
- На данной справочной странице речь идет не о самом системном вызове, а о стандартном API языка C, который доступен в кольце командной оболочки. Эти справочники рассчитаны на разработчиков приложений, и потому в них мало информации о системных вызовах и внутренностях ядра. Вместо этого основной акцент в них делается на API, доступные в кольце командной оболочки.

Теперь найдем то место в libc, где инициируется системный вызов. Мы будем использовать исходники FreeBSD на GitHub. В данном случае взята фиксация с хешем `bf78455d496` в ветви `master`. Чтобы клонировать из репозитория и применить подходящую фиксацию, выполните команды, показанные в терминале 10.4.

Терминал 10.4. Клонирование проекта FreeBSD и переход к определенной фиксации

```
$ git clone https://github.com/freebsd/freebsd
...
$ cd freebsd
$ git reset --hard bf78455d496
...
$
```

Мы также можем открыть проект FreeBSD на самом сайте GitHub с помощью ссылки https://github.com/freebsd/freebsd/tree/bf78455d496. В любом случае у вас должна быть возможность найти следующие строчки кода.

Если зайти в каталог lib/libc и выполнить `grep` для `sys_nanosleep`, то можно увидеть следующие файлы (терминал 10.5).

Терминал 10.5. Поиск файлов, относящихся к системному вызову nanosleep в libc для FreeBSD

```
$ cd lib/libc
$ grep sys_nanosleep . -R
./include/libc_private.h:int    __sys_nanosleep(const struct
timespec *, struct timespec *);
./sys/Symbol.map:      __sys_nanosleep;
./sys/nanosleep.c:__weak_reference(__sys_nanosleep, __nanosleep);
./sys/interposing_table.c:    SLOT(nanosleep, __sys_nanosleep),
$
```

Как можно видеть в файле `lib/libc/sys/interposing_table.c`, функция `nanosleep` привязана к вызову `__sys_nanosleep`. Следовательно, при выполнении `nanosleep` всегда вызывается `__sys_nanosleep`.

В FreeBSD названия всех функций, которые являются системными вызовами, принято начинать с `__sys`. Обратите внимание: это часть реализации libc, и потому соглашение об именовании и другие детали реализации относятся исключительно к FreeBSD.

В приведенном выше терминале есть еще один интересный участок. Файл `lib/libc/include/libc_private.h` содержит объявления приватных и внутренних функций-оберток вокруг системных вызовов.

Итак, мы увидели, каким образом кольцо командной оболочки направляет вызовы функций, принадлежащих libc, во внутренние кольца с помощью системных вызовов. Но для чего это делается? Если взять обычную функцию в пользовательском приложении или libc, то чем она отличается от системного вызова в кольце ядра? В главе 11 мы обсудим это подробнее, познакомившись с более конкретным определением системного вызова.

Следующий раздел посвящен кольцу ядра и его внутренним компонентам, которые являются общими для ядер в большинстве Unix-совместимых и Unix-подобных систем.

Ядро

Основное назначение кольца ядра состоит в управлении оборудованием, подключенным к системе, и предоставлении доступа к своим возможностям с помощью системных вызовов. На рис. 10.4 показано, каким образом пользовательское приложение обращается к определенной аппаратной функции.

Это общая схема того, о чем мы говорили ранее. В данном разделе мы сосредоточимся на самом ядре и поговорим о том, что оно собой представляет. Ядро, как и любой другой известный нам процесс, выполняет последовательность инструкций. Однако *процесс ядра* фундаментально отличается от обычных *пользовательских процессов*.

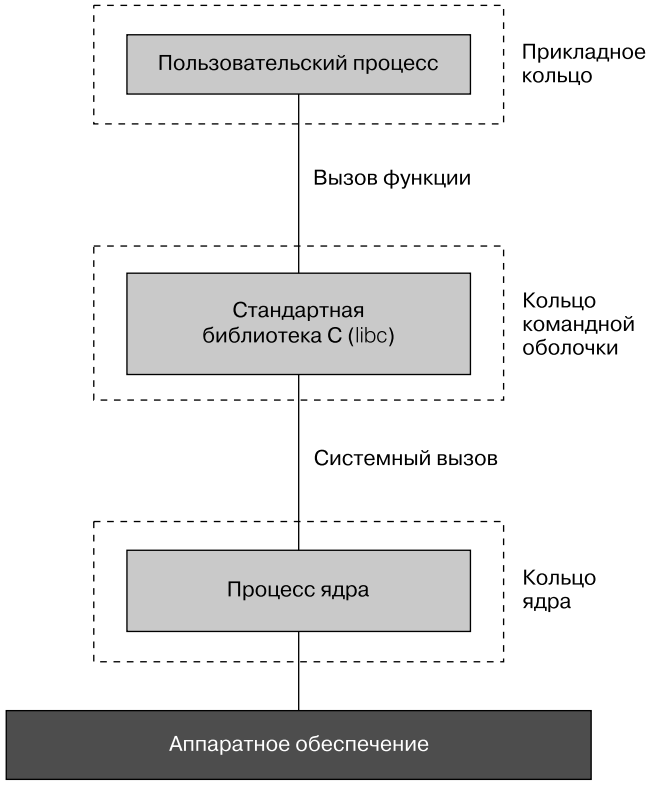

Рис. 10.4. Вызовы функций и системные вызовы, которые выполняются между разными кольцами Unix при обращении к оборудованию

В перечне ниже сравниваются процесс ядра и пользовательский процесс. Обратите внимание: речь идет в основном о монолитных ядрах, таких как Linux. Разные виды ядер будут рассмотрены в следующей главе.

- Процесс ядра — первое, что загружается и выполняется в системе. Только вслед за ним можно запускать пользовательские процессы.
- Процесс ядра существует в единственном экземпляре, при этом мы можем запустить сразу несколько пользовательских процессов.
- Процесс ядра создается в результате того, что загрузчик копирует его образ в основную память, а для создания пользовательского процесса применяются системные вызовы `exec` или `fork`, присутствующие в большинстве Unix-систем.
- Процесс ядра обрабатывает и выполняет системные вызовы, в то время как пользовательский их инициирует и ждет их выполнения. Это значит, что, когда пользовательский процесс запрашивает системный вызов, поток выполнения переходит к процессу ядра; то есть логику системного вызова от имени пользо-

вателя выполняет само ядро. Мы подробно разберем то, как это происходит, во второй части нашего знакомства с Unix, в главе 11.

- Процесс ядра имеет *привилегированный* доступ к физической памяти и всему подключенному оборудованию, в то время как пользовательский работает с виртуальной памятью, которая является отражением части физической памяти, и ничего не знает о ее физической структуре. Пользовательский процесс имеет управляемый и наблюдаемый доступ к ресурсам и оборудованию. Можно сказать, что он выполняется в изолированной среде, которая симулируется операционной системой. Это также означает, что пользовательские процессы не могут видеть память друг друга.

Из этого сравнения следует, что среда выполнения операционной системы имеет два разных режима. Один предназначен для процесса ядра, а другой — для пользовательских процессов.

Первый режим выполнения называется *пространством ядра*, а второй — *пользовательским пространством*. Оба пространства взаимодействуют с помощью предусмотренных системных вызовов. В целом причиной появления системных вызовов послужила необходимость в изоляции между пространствами ядра и пользователя. Пространство ядра обладает максимально привилегированным доступом к системным ресурсам, а доступ пользовательского пространства максимально ограниченный и контролируемый.

Внутреннюю структуру типичного Unix-ядра можно разделить по выполняемым задачам. В действительности обязанности ядра не ограничиваются управлением аппаратным обеспечением. Ниже перечислены задачи, которые возложены на ядро Unix. Стоит отметить, что одним из пунктов идет управление оборудованием.

- *Управление процессами*. Ядро создает пользовательские процессы с помощью системного вызова. Среди прочих операций перед запуском нового процесса необходимо сначала выделить память и загрузить его инструкции.
- *Межпроцессное взаимодействие (Inter-Process Communication, IPC)*. Пользовательские процессы, запущенные на одном компьютере, могут задействовать разные методы обмена данными, включая разделяемую память, каналы и доменные сокеты Unix. Эти методы должны обеспечиваться ядром, и некоторые из них применяют ядро для обмена информацией. Мы обсудим их в главе 19, посвященной методикам IPC.
- *Планирование*. Система Unix всегда славилась своей многозадачностью. Ядро управляет доступом к центральному процессору, пытаясь его балансировать. Планирование — процедура разделения процессорного времени между разными процессами в зависимости от их приоритета и значимости. В следующих главах мы более подробно поговорим о многозадачности, многопоточности и многопроцессности.

- *Управление памятью.* Это, несомненно, одна из ключевых обязанностей ядра. Ядро — единственный процесс, который видит всю физическую память и имеет к ней неограниченный доступ. Поэтому на него ложится разбиение ее на выделяемые страницы, назначение новых страниц процессам (при выделении кучи), освобождение памяти и многие другие сопутствующие процедуры.
- *Начальная загрузка системы.* После загрузки образа ядра в основную память его процесс должен запуститься и инициализировать пользовательское пространство. Обычно для этого создается первый пользовательский процесс с *PID* (Process Identifier — идентификатор процесса) 1. В некоторых разновидностях Unix, таких как Linux, этот процесс называется *init*. Вслед за ним запускаются другие сервисы и демоны.
- *Управление устройствами.* Ядро должно иметь возможность управлять не только процессором и памятью, но и аппаратным обеспечением. Для этого предусмотрен слой абстракции. В качестве *устройства* может выступать реальное или виртуальное оборудование, подключенное к системе. В Unix устройства привязываются к файлам, которые обычно хранятся в каталоге /dev. Это касается всех подключенных жестких дисков, сетевых адаптеров, USB-устройств и т. д. Эти файлы могут применяться пользовательскими процессами для взаимодействия с аппаратным обеспечением.

На рис. 10.5 показана внутренняя структура, присущая большинству ядер Unix и основанная на приведенном выше перечне.

Это подробная иллюстрация колец Unix. Здесь явно видно, что в кольце командной оболочки пользовательским приложениям доступно три компонента. Вы также можете видеть подробную внутреннюю структуру кольца ядра.

В верхней части кольца ядра мы имеем интерфейс системных вызовов. Как видите, все предыдущие компоненты в пользовательском пространстве могут взаимодействовать с нижними компонентами только через данный интерфейс. Это словно шлюз или барьер между пространствами ядра и пользователя.

В ядре размещены различные компоненты, такие как *блок управления памятью* (Memory Management Unit, MMU), который отвечает за доступную физическую память. *Модуль управления процессами* создает процессы в пользовательском пространстве, выделяет для них ресурсы и позволяет им взаимодействовать. На рис. 10.5 также показаны *символьные* и *блочные* устройства, которые открывают доступ к различным функциям ввода/вывода с помощью *драйверов*. *Файловая система* является неотъемлемой частью ядра и служит абстракцией над блочными и символьными устройствами, позволяя процессам и самому ядру задействовать общую файловую иерархию.

В следующем разделе мы поговорим об аппаратном обеспечении.

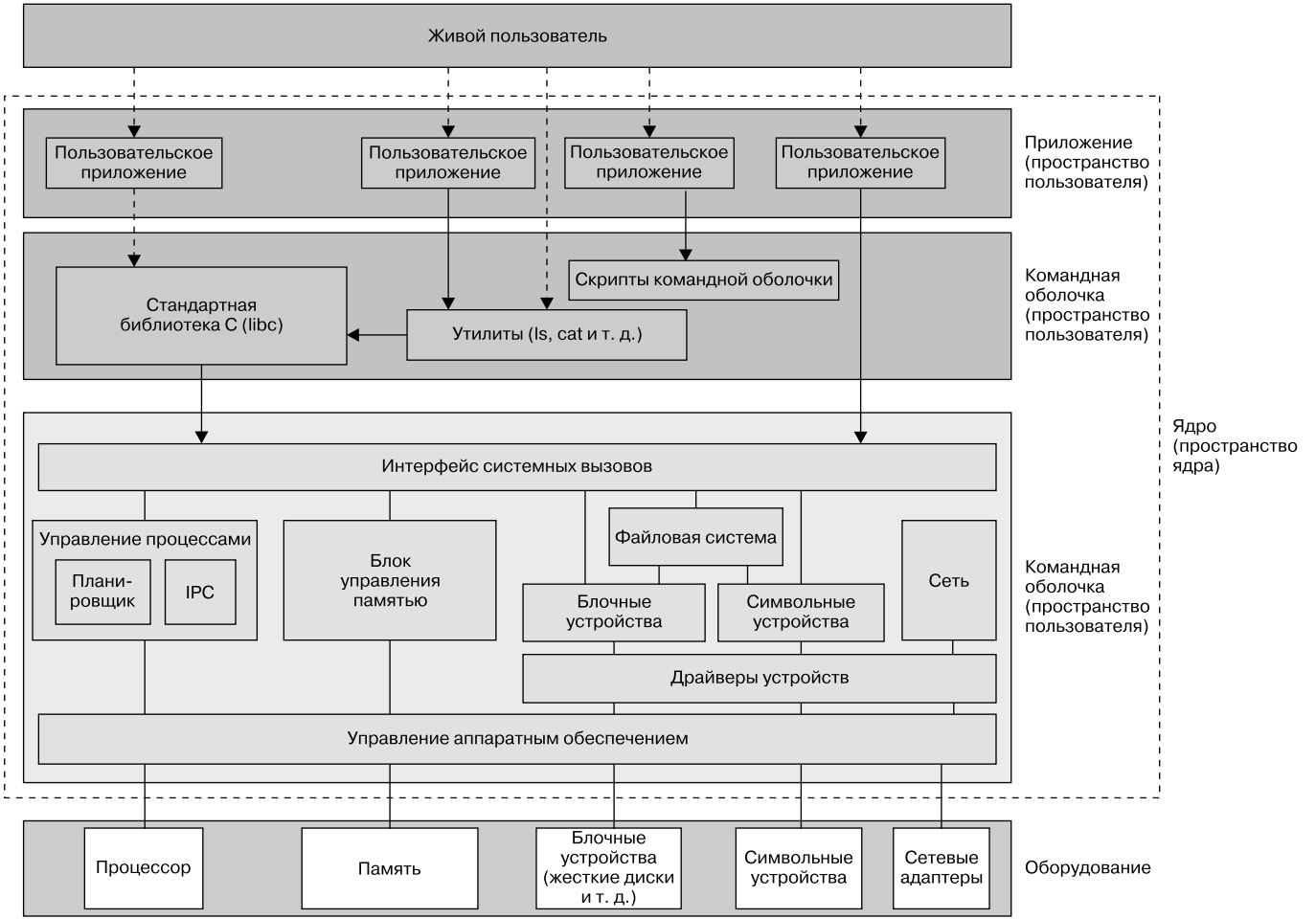

Рис. 10.5. Внутренняя структура разных колец в архитектуре Unix

Аппаратное обеспечение

Конечная цель любой операционной системы — дать пользователю и приложениям возможность работать с аппаратным обеспечением. Unix тоже пытается предоставить доступ к подключенному оборудованию абстрактным и прозрачным путем, используя один и тот же набор утилит и команд на всех существующих и будущих платформах.

Используя эту абстрактность и прозрачность, Unix инкапсулирует всевозможное оборудование в виде ряда устройств, подключенных к системе. Термин *«устройство»* — одно из ключевых в Unix, и любое имеющееся аппаратное обеспечение считается устройством, подключенным к системе Unix.

Оборудование, подключенное к компьютеру, можно разделить на две категории: *основное* и *периферийное*. К основным устройствам относятся центральный процессор и память. Все остальное, включая жесткие диски, сетевые адаптеры, мыши, мониторы, графические карты и адаптеры Wi-Fi, — периферийное оборудование.

Компьютер под управлением Unix не может работать без основных устройств, но существуют системы, которые обходятся без жесткого диска или сетевого адаптера. Обратите внимание: файловая система, без которой не может работать ядро Unix, может не требовать наличия жесткого диска!

Ядро Unix полностью инкапсулирует центральный процессор и физическую память, управляя ими напрямую и не давая обращаться к ним из пространства пользователя. За работу с физической памятью и процессором в ядре Unix отвечают соответственно *блок управления памятью* и *планировщик*.

С периферийным оборудованием все иначе. Доступ к нему осуществляется через *файлы устройств*. В системе Unix эти файлы находятся в каталоге /dev.

В терминале 10.6 показан список файлов, которые можно найти на обычном компьютере с Linux.

Терминал 10.6. Содержимое каталога /dev на компьютере с Linux

```
$ ls -l /dev
total 0
crw-r--r--  1 root   root     10, 235 Oct 14 16:55 autofs
drwxr-xr-x  2 root   root         280 Oct 14 16:55 block
drwxr-xr-x  2 root   root          80 Oct 14 16:55 bsg
crw-rw----  1 root   disk     10, 234 Oct 14 16:55 btrfs-control
drwxr-xr-x  3 root   root          60 Oct 14 17:02 bus
lrwxrwxrwx  1 root   root           3 Oct 14 16:55 cdrom -> sr0
drwxr-xr-x  2 root   root        3500 Oct 14 16:55 char
crw-------  1 root   root      5,   1 Oct 14 16:55 console
lrwxrwxrwx  1 root   root          11 Oct 14 16:55 core -> /proc/kcore
crw-------  1 root   root     10,  59 Oct 14 16:55 cpu_dma_latency
```

```
crw-------  1 root   root     10, 203 Oct 14 16:55 cuse
drwxr-xr-x  6 root   root         120 Oct 14 16:55 disk
drwxr-xr-x  3 root   root          80 Oct 14 16:55 dri
lrwxrwxrwx  1 root   root           3 Oct 14 16:55 dvd -> sr0
crw-------  1 root   root     10,  61 Oct 14 16:55 ecryptfs
crw-rw----  1 root   video    29,   0 Oct 14 16:55 fb0
lrwxrwxrwx  1 root   root          13 Oct 14 16:55 fd -> /proc/self/fd
crw-rw-rw-  1 root   root      1,   7 Oct 14 16:55 full
crw-rw-rw-  1 root   root     10, 229 Oct 14 16:55 fuse
crw-------  1 root   root    245,   0 Oct 14 16:55 hidraw0
crw-------  1 root   root     10, 228 Oct 14 16:55 hpet
drwxr-xr-x  2 root   root           0 Oct 14 16:55 hugepages
crw-------  1 root   root     10, 183 Oct 14 16:55 hwrng
crw-------  1 root   root     89,   0 Oct 14 16:55 i2c-0
...
crw-rw-r--  1 root   root     10,  62 Oct 14 16:55 rfkill
lrwxrwxrwx  1 root   root           4 Oct 14 16:55 rtc -> rtc0
crw-------  1 root   root    249,   0 Oct 14 16:55 rtc0
brw-rw----  1 root   disk      8,   0 Oct 14 16:55 sda
brw-rw----  1 root   disk      8,   1 Oct 14 16:55 sda1
brw-rw----  1 root   disk      8,   2 Oct 14 16:55 sda2
crw-rw----+ 1 root   cdrom    21,   0 Oct 14 16:55 sg0
crw-rw----  1 root   disk     21,   1 Oct 14 16:55 sg1
drwxrwxrwt  2 root   root          40 Oct 14 16:55 shm
crw-------  1 root   root     10, 231 Oct 14 16:55 snapshot
drwxr-xr-x  3 root   root         180 Oct 14 16:55 snd
brw-rw----+ 1 root   cdrom    11,   0 Oct 14 16:55 sr0
lrwxrwxrwx  1 root   root          15 Oct 14 16:55 stderr -> /proc/self/fd/2
lrwxrwxrwx  1 root   root          15 Oct 14 16:55 stdin -> /proc/self/fd/0
lrwxrwxrwx  1 root   root          15 Oct 14 16:55 stdout -> /proc/self/fd/1
crw-rw-rw-  1 root   tty       5,   0 Oct 14 16:55 tty
crw--w----  1 root   tty       4,   0 Oct 14 16:55 tty0
...
$
```

Как видите, к компьютеру подключено довольно много устройств. Конечно, не все они физические. Благодаря слою абстракции поверх аппаратного обеспечения система Unix поддерживает *виртуальные устройства*.

Например, у вас может быть виртуальный сетевой адаптер, который физически не существует, но может производить дополнительные операции с сетевыми данными. Это один из способов организации VPN в средах, основанных на Unix. Физический сетевой адаптер предоставляет реальные сетевые функции, а виртуальный — позволяет пропускать данные через безопасный туннель.

Как видно в показанном выше выводе, у каждого устройства есть собственный файл в каталоге `/dev`. Строчки, начинающиеся с `a` и `b`, представляют соответственно символьные и блочные устройства. Первые отправляют и принимают данные побайтно (как это, к примеру, делают последовательный и параллельный порты), а последние работают с блоками информации размером больше 1 байта (как,

например, жесткие диски, сетевые адаптеры, видеокамеры и т. д.). В приведенном выше терминале 10.6 строчки, начинающиеся с `'l'`, обозначают символьные ссылки на другие устройства, а строчки с `d` в начале — это каталоги, которые могут содержать другие файлы устройств.

Пользовательские процессы применяют эти файлы, чтобы обращаться к соответствующему оборудованию. Обмен данными с устройством осуществляется путем чтения и записи этих файлов.

Мы не станем углубляться в дальнейшие детали, но если хотите узнать больше об устройствах и их драйверах, то вам стоит почитать дополнительный материал на эту тему. В следующей главе мы более подробно обсудим системные вызовы и добавим собственный вызов в имеющееся ядро Unix.

Резюме

В этой главе мы начали разговор о системе Unix и о том, какое отношение она имеет к C. Следы архитектуры Unix можно обнаружить даже в операционных системах других семейств.

В ходе главы мы обсудили события, произошедшие в начале 1970-х годов, и узнали, как из Multics возникла система Unix и язык программирования B стал прародителем C. Вслед за этим мы поговорили о многослойной архитектуре Unix, состоящей из четырех колец: пользовательских приложений, командной оболочки, ядра и аппаратного обеспечения.

Мы кратко прошлись по различным кольцам в этой многослойной модели, отдельно остановившись на командной оболочке. Была представлена стандартная библиотека C и описано, как она в сочетании со стандартами POSIX и SUS позволяет программистам писать программы, которые можно собирать в разных Unix-системах.

Во второй части этого обсуждения, в главе 11, мы продолжим рассматривать систему Unix и ее архитектуру. Вы узнаете больше о том, как устроены ядро и окружающий его интерфейс системных вызовов.

11 Системные вызовы и ядра

В предыдущей главе мы обсудили историю системы Unix и ее многослойную архитектуру. Вы также познакомились со стандартами POSIX и SUS, которые описывают кольцо командной оболочки Unix, и увидели, как стандартная библиотека C предоставляет часто используемые функции, доступные в Unix-совместимых ОС.

В данной главе мы продолжим обсуждать *интерфейс системных вызовов* и *ядро* Unix. Это даст нам полную картину того, как работает система Unix.

Прочитав данную главу, вы будете способны проанализировать системные вызовы, инициируемые программой, и объяснить, как ее процесс работает и изменяется в среде Unix; вы также сможете использовать системные вызовы напрямую или через libc. Вдобавок мы поговорим о разработке ядра Unix и увидим, как добавлять в него новые системные вызовы и как их можно инициировать из кольца командной оболочки.

В последней части главы речь пойдет о *монолитных* ядрах, *микроядрах* и их отличиях. Мы познакомимся с монолитным ядром Linux и напишем для него *модуль*, который можно динамически загружать и выгружать.

Начнем с разговора о системных вызовах.

Системные вызовы

В предыдущей главе я коротко объяснил, что такое системный вызов. В этом разделе мы остановимся на данной теме более подробно и рассмотрим механизм, с помощью которого системные вызовы передают поток выполнения от пользовательского процесса процессу ядра.

Но сначала нам следует еще немного обсудить пространства ядра и пользователя, поскольку это поможет нам лучше понять принцип работы системных вызовов. Мы также напишем простой системный вызов, чтобы получить общее представление о разработке ядра.

Материал, представленный ниже, будет незаменимым для тех, кто хочет научиться писать новые системные вызовы и расширять возможности ядра. Это также позволит вам лучше понять пространство ядра и то, чем оно отличается от пользовательского пространства, поскольку в реальности они очень разнятся.

Тщательное исследование системных вызовов

Как уже упоминалось в предыдущей главе, кольца командной оболочки и ядра разделены. Все, что находится в первых двух кольцах (пользовательских приложениях и командной оболочке), принадлежит к пространству пользователя. Точно так же все, что мы видим в кольцах ядра и аппаратного обеспечения, принадлежит к пространству ядра.

Это разделение следует одному правилу: содержимое двух внутренних колец, ядра и аппаратного обеспечения не должно быть доступно напрямую из пространства пользователя. Иными словами, никакой пользовательский процесс не должен иметь прямого доступа к оборудованию, а также к внутренним структурам данных и алгоритмам ядра. Обращение к ним должно происходить через системные вызовы.

Вам может показаться, будто это слегка противоречит тому, что вы уже знаете о Unix-подобных операционных системах, таких как Linux, и вашему опыту работы с ними. Но если вы не видите никакой проблемы, то позвольте мне объяснить, о чем речь. Нелогичность возникает, к примеру, когда программа считывает байты из сетевого сокета, поскольку в действительности эти байты читает ядро и только потом они копируются в пользовательское пространство, где с ними уже может работать программа.

Чтобы прояснить ситуацию, можно рассмотреть пример того, как данные проходят от пространства пользователя к пространству ядра и обратно. Когда вам нужно прочитать файл с жесткого диска, вы пишете программу для кольца пользовательских приложений. Она задействует функцию ввода/вывода из libc под названием `fread` (или ее аналог) и в итоге выполняется в виде процесса в пространстве пользователя. Когда программа вызывает функцию `fread`, срабатывает ее реализация внутри libc.

Пока все это происходит в пользовательском процессе. В конечном счете реализация `fread` инициирует системный вызов, принимая три аргумента: *дескриптор* уже открытого файла, адрес буфера, выделенного в памяти процесса (который находится в пространстве пользователя) и длину данного буфера.

Когда реализация libc инициирует системный вызов, поток выполнения пользовательского процесса переходит к ядру, которое получает аргументы из пользовательского пространства и размещает их в пространстве ядра. Затем ядро читает файл,

обращаясь к своему модулю файловой системы (как можно видеть на рис. 10.5 в предыдущей главе).

После завершения операции `read` в кольце ядра прочитанные данные копируются в буфер в пространстве пользователя, указанный во втором аргументе функции `fread`; дальше поток выполнения переходит от системного вызова к пользовательскому процессу. Пока системный вызов делает свою работу, пользовательский процесс обычно находится в ожидании. В таком случае системный вызов называют блокирующим.

Этот сценарий имеет несколько важных аспектов.

- Выполнением всей логики системных вызовов занимается одно ядро.
- Если системный вызов *блокирующий*, то вызывающая сторона должна дождаться завершения его работы. *Неблокирующие* системные вызовы очень быстро возвращаются, но пользовательскому процессу приходится выполнять дополнительные действия, чтобы проверить, доступны ли результаты.
- Аргументы вместе с входными и выходными данными копируются в пользовательское пространство и из него. Ввиду этого системные вызовы должны быть рассчитаны на прием крошечных значений и указателей в качестве входных аргументов.
- Ядро обладает полным привилегированным доступом ко всем ресурсам системы. Следовательно, должен существовать такой механизм, который проверяет, может ли пользовательский процесс выполнять тот или иной системный вызов. В нашем случае, если пользователь не является владельцем файла, то операция `fread` должна завершиться ошибкой, связанной с нехваткой прав доступа.
- Похожее разделение существует между участками памяти, выделенными для пространств пользователя и ядра. Пользовательский процесс может обращаться только к памяти пространства пользователя. В ходе работы некоторых системных вызовов поток выполнения передается несколько раз.

Прежде чем идти дальше, я хочу спросить вас кое о чем. Как системный вызов передает поток выполнения ядру? Задумайтесь об этом на минуту, поскольку в следующем подразделе мы попытаемся найти ответ.

Выполнение системного вызова напрямую, в обход стандартной библиотеки C

Прежде чем отвечать на поставленный выше вопрос, рассмотрим пример, в котором системный вызов инициируется напрямую, в обход стандартной библиотеки C. Иными словами, программа выполняет системный вызов, не обращаясь к кольцу командной оболочки. Как уже отмечалось прежде, это считается плохим подходом,

однако некоторые системные вызовы недоступны в libc и их пользовательское приложение может инициировать напрямую.

Любая Unix-система предоставляет определенный метод для прямого выполнения системных вызовов. Например, в Linux для этого предусмотрена функция под названием `syscall`, размещенная в заголовочном файле `<sys/syscall.h>`.

В примере 11.1 (листинг 11.1) показана еще одна разновидность программы Hello World, в которой для передачи текста в стандартный вывод не используется libc. То есть не применяется функция `printf`, которая является частью кольца командной оболочки и стандарта POSIX. Вместо этого программа инициирует системный вызов напрямую, из-за чего данный код совместим только с Linux, но не с другими Unix-системами. Иными словами, данный код нельзя перенести между разными вариациями Unix.

Листинг 11.1. Еще одна версия примера Hello World, которая выполняет системный вызов write напрямую (ExtremeC_examples_chapter11_1.c)

```
// Это нужно, чтобы использовать элементы не из состава POSIX
#define _GNU_SOURCE

#include <unistd.h>

// Это не является частью POSIX!
#include <sys/syscall.h>

int main(int argc, char** argv) {
  char message[20] = "Hello World!\n";
  // Инициирует системный вызов 'write', который записывает
  // некоторые байты в стандартный вывод.
  syscall(__NR_write, 1, message, 13);
  return 0;
}
```

В начале данного листинга мы определили `_GNU_SOURCE`, сигнализируя тем самым, что в нем будут использоваться возможности *GNU C Library* (glibc), которые не входят в стандарты POSIX и SUS. Это нарушает переносимость программы, что не позволит вам скомпилировать свой код в других системах Unix. Дальше идет инструкция `include`, которая подключает один из уникальных заголовочных файлов glibs; эти файлы не существуют в других POSIX-системах, которые в качестве стандартной библиотеки C используют не glibc, а нечто иное.

Внутри `main` с помощью функции `syscall` выполняется системный вызов. Прежде всего мы должны указать целочисленный номер вызова. В Linux каждый системный вызов имеет уникальный *номер*.

В нашем коде вместо номера системного вызова передается константа `__R_write`, числовое значение которой нам неизвестно. Если заглянуть в заголовочный файл

unistd.h, то можно обнаружить, что системный вызов write, по всей видимости, имеет номер 64. Вслед за этим номером мы должны передать аргументы, необходимые системному вызову.

Несмотря на простоту представленного выше кода, вы должны понимать, что syscall — не обычная функция. Это ассемблерная процедура, которая заполняет подходящие регистры процессора и передает поток выполнения от пользовательского пространства пространству ядра. Мы вернемся к этому чуть позже.

Вызову write нужно передать три аргумента: дескриптор файла (1 обозначает стандартный вывод), *указатель на буфер*, размещенный в пространстве пользователя, и, наконец, *количество байтов*, которое нужно скопировать из этого буфера.

В терминале 11.1 показан вывод примера 11.1 после его компиляции с помощью gcc и запуска в Ubuntu 18.04.1.

Терминал 11.1. Вывод примера 11.1

```
$ gcc ExtremeC_examples_chapter11_1.c -o ex11_1.out
$ ./ex11_1.out
Hello World!
$
```

Теперь пришло время воспользоваться утилитой strace, с которой мы познакомились в предыдущей главе, чтобы посмотреть, какие системные вызовы на самом деле инициирует пример 11.1. Вывод strace, представленный в терминале 11.2, показывает, что наша программа выполнила нужный нам системный вызов.

Терминал 11.2. Вывод strace в ходе выполнения примера 11.1

```
$ strace ./ex11_1.out
execve("./ex11_1.out", ["./ex11_1.out"], 0x7ffcb94306b0 /* 22 vars */) = 0
brk(NULL)                               = 0x55ebc30fb000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
...
...
arch_prctl(ARCH_SET_FS, 0x7f24aa5624c0) = 0
mprotect(0x7f24aa339000, 16384, PROT_READ) = 0
mprotect(0x55ebc1e04000, 4096, PROT_READ) = 0
mprotect(0x7f24aa56a000, 4096, PROT_READ) = 0
munmap(0x7f24aa563000, 26144)           = 0
write(1, "Hello World!\n", 13Hello World!)          = 13
exit_group(0)                           = ?
+++ exited with 0 +++
$
```

Здесь полужирным шрифтом выделено то место, где утилита `strace` записала системный вызов. Взгляните на возвращаемое значение (`13`). Оно означает, что системный вызов успешно записал 13 байт в заданный файл (в нашем случае это стандартный вывод).

Пользовательское приложение никогда не должно пытаться инициировать системные вызовы напрямую. Обычно перед системным вызовом и после него необходимо выполнять определенные действия, которые реализованы в libc. Если пойти в обход стандартной библиотеки, то указанные действия придется выполнять самостоятельно; при этом они могут отличаться в зависимости от вашей Unix-системы.

Внутри функции syscall

Что же происходит внутри функции `syscall`? Обратите внимание: все, что будет сказано ниже, относится только к glibc, но не к другим реализациям libc. Определение `syscall` находится по ссылке https://github.com/lattera/glibc/blob/master/sysdeps/unix/sysv/linux/x86_64/syscall.S.

Если открыть эту ссылку в браузере, то можно увидеть, что данная функция написана на ассемблере.

Ассемблер можно использовать в исходных файлах языка C совместно с инструкциями последнего. На самом деле это одна из превосходных особенностей языка C, которая позволяет применять его для написания операционной системы. Что касается функции syscall, то ее объявление написано на C, а определение — на ассемблере.

В листинге 11.2 представлена часть исходного кода в файле `syscall.S`.

Листинг 11.2. Определение функции syscall в glibc

```
/* Copyright (C) 2001-2018 Free Software Foundation, Inc.
   This file is part of the GNU C Library.
...
   <http://www.gnu.org/licenses/>. */

#include <sysdep.h>

/* Please consult the file sysdeps/unix/sysv/linux/x86-64/sysdep.h
For more information about the value -4095 used below. */

/* Usage: long syscall (syscall_number, arg1, arg2, arg3, arg4, arg5, arg6)
We need to do some arg shifting, the syscall_number will be in rax. */

   .text
ENTRY (syscall)
```

```
    movq %rdi, %rax           /* Syscall number -> rax. */
    movq %rsi, %rdi           /* shift arg1 - arg5. */
    movq %rdx, %rsi
    movq %rcx, %rdx
    movq %r8, %r10
    movq %r9, %r8
    movq 8(%rsp),%r9          /* arg6 is on the stack. */
    syscall                   /* Do the system call. */
    cmpq $-4095, %rax         /* Check %rax for error. */
    jae SYSCALL_ERROR_LABEL   /* Jump to error handler if error.*/
    ret                       /* Return to caller. */

PSEUDO_END (syscall)
```

Этот метод выполнения системного вызова более сложен, но сами инструкции выглядят лаконично и просто. В комментариях объясняется, что каждому системному вызову в glibc можно предоставить до шести аргументов.

Это значит, glibc не позволяет использовать определенные возможности ядер, которые поддерживают системные вызовы с более чем шестью аргументами и добавление такой поддержки потребует изменения данной библиотеки. К счастью, шести аргументов достаточно в большинстве случаев; если системному вызову этого мало, то мы можем передать ему указатели на структурные переменные, выделенные в памяти пользовательского пространства.

В листинге выше вслед за инструкцией `movq` ассемблерный код вызывает процедуру `syscall`. Он просто генерирует *прерывание*, которое активирует определенный участок ядра, предназначенный для его обработки.

Как можно видеть в первой строчке процедуры `syscall`, номер системного вызова заносится в регистр `%rax`. В следующих строчках мы копируем в регистры другие аргументы. Когда генерируется прерывание, его обработчик в ядре подхватывает системный вызов вместе с номером и аргументами. Затем он обращается к *таблице системных вызовов*, чтобы найти подходящую функцию, которая должна быть выполнена на стороне ядра.

Интересно, что к моменту выполнения обработчика прерывания пользовательский код, который инициировал системный вызов, уже покинул центральный процессор, уступив место ядру. Это основной механизм, лежащий в основе системных вызовов. Когда такой вызов инициируется, процессор переключается в другой режим и принимает инструкции от ядра, тогда как приложение в пользовательском пространстве перестает выполняться. Вот почему мы говорим, что ядро выполняет логику системного вызова от имени пользовательского приложения.

В следующем подразделе я продемонстрирую сказанное выше на примере написания системного вызова, который выводит приветственное сообщение. Это можно считать модифицированной версией примера 11.1, которая принимает и возвращает строку с приветствием.

Добавление системного вызова в Linux

Здесь мы добавим новую запись в таблицу системных вызовов имеющегося Unix-подобного ядра. Для многих читателей это может стать первой возможностью написания кода на C, предназначенного для выполнения в пространстве ядра. Все примеры, которые мы разбирали в предыдущих главах, и почти весь код, приводимый на оставшихся страницах, работает в пользовательском пространстве.

На самом деле большинство написанных нами программ предназначены для выполнения в пространстве пользователя. Это то, что мы называем *программированием* или *разработкой на C*. Для написания кода, рассчитанного на работу в пространстве ядра, есть другое название: *разработка ядра*.

Прежде чем переходить к примеру 11.2, необходимо исследовать среду ядра и понять, чем оно отличается от пользовательского пространства.

Разработка ядра

Этот участок текста будет полезен тем из вас, кто планирует стать разработчиком ядра или исследователем безопасности в области операционных систем. Прежде чем переходить к самому системному вызову, мы объясним разницу между разработкой ядра и обычных программ на языке C.

Разработка ядер имеет ряд отличий от того, как разрабатываются обычные программы на C. И прежде, чем рассматривать эти отличия, следует отметить, что код на C обычно пишется для пространства пользователя.

Ниже перечислены шесть ключевых отличий в процессе разработки для ядра и для пользовательского пространства.

- Процесс ядра находится во главе системы и существует в единственном экземпляре. Это просто означает, что если ваш код приведет к сбою в ядре, то вам, скорее всего, придется перезагрузить компьютер и позволить ядру заново выполнить инициализацию. Следовательно, цена ошибки в таком коде очень высока, и для проведения экспериментов в нем требуется перезагрузка компьютера; в программах пользовательского пространства без этого можно легко обойтись. В случае ошибки генерируется *дамп сбоя ядра*, который помогает выявить причину.
- В кольце ядра нет никакой стандартной библиотеки C наподобие glibc! Иными словами, стандарты SUS и POSIX здесь не действуют. Поэтому вы не можете подключать заголовочные файлы libc, такие как `stdio.h` или `string.h`. Вы получаете в свое распоряжение отдельный набор функций, которые необходимо использовать для выполнения различных операций. Эти функции обычно находятся в *заголовках ядра* и могут отличаться в зависимости от разновид-

ности Unix, поскольку в данной области еще не проводилось никакой стандартизации.

Например, занимаясь разработкой ядра в Linux, вы можете выполнять запись в *буфер сообщений* ядра с помощью функции `printk`. Но в FreeBSD для этого предусмотрено семейство функций `printf`, которые отличаются от своих аналогов из libc; они находятся в заголовочном файле `<sys/system.h>`. Их эквивалент в ядре XNU, которое используется в macOS, — функция `os_log`.

- В ядре тоже можно читать и изменять файлы, но без использования функций из libc. У каждой Unix-системы есть собственный способ доступа к файлам из кольца ядра. То же самое относится ко всем другим возможностям, доступным в libc.
- В кольце ядра вы имеете полный доступ к физической памяти и многим другим возможностям. Поэтому очень важно писать безопасный и надежный код.
- В ядре нет механизма системных вызовов. Системные вызовы — основной метод взаимодействия с кольцом ядра из пользовательского пространства. Поэтому в самом ядре в нем нет необходимости.
- Процесс ядра создается путем копирования образа ядра в физическую память. Этим занимается *загрузчик*. Вы не можете добавить новые системные вызовы, не прибегнув к созданию совершенно нового образа и инициализации его в ходе перезагрузки системы. Кое-какие ядра поддерживают *модули*, которые можно динамически добавлять и удалять, но с системными вызовами этого делать нельзя.

Все это показывает, что процесс разработки ядра отличается от обычной разработки на языке C. Тестирование написанной логики затруднено, и некачественный код может вывести из строя всю систему.

Далее мы попробуем свои силы в разработке ядра на примере создания нового системного вызова. Мы это делаем вовсе не потому, что добавление системных вызовов — распространенный способ расширения возможностей ядра; это просто попытка познакомиться с данным видом разработки.

Написание демонстрационного системного вызова для Linux

Здесь мы напишем новый системный вызов для Linux. В Интернете есть много материала на эту тему, но мы возьмем за основу статью *Adding a Hello World System Call to Linux Kernel*, доступную по адресу https://medium.com/anubhav-shrimal/adding-a-hello-world-system-call-to-linux-kernel-dad32875872.

Пример 11.2 — углубленная версия примера 11.1; в ней используется другой, нестандартный системный вызов, который мы напишем здесь. Наш новый системный

вызов принимает четыре аргумента: первые два служат для ввода имени, а два последних — для получения выходной строки. Имя состоит из двух аргументов, один из которых является указателем на буфер, заранее выделенный в пользовательском пространстве, а второй обозначает длину этого буфера. Строка с приветствием возвращается с помощью двух других аргументов: указателя на другой буфер, который тоже находится в пространстве пользователя, и его длины в виде целого числа.

ОСТОРОЖНО

Пожалуйста, не проводите этот эксперимент в системе Linux, которую вы используете на домашнем или рабочем компьютере. Выполняйте эти команды на отдельном экспериментальном устройстве, в качестве которого я настоятельно рекомендую использовать виртуальную машину. Вы можете легко создавать виртуальные машины с помощью таких эмуляторов, как VirtualBox или VMware.

Следующие инструкции, будучи примененными неправильно или не в том порядке, могут теоретически повредить вашу систему и привести к частичной или полной потере данных. Если вы собираетесь выполнять их не на экспериментальном компьютере, то не забудьте воспользоваться каким-нибудь средством резервного копирования, чтобы сохранить свои данные.

Прежде всего мы должны загрузить самый свежий исходный код ядра Linux. Мы клонируем его из репозитория на GitHub и укажем нужный нам выпуск. Версия 5.3 была выпущена 15 сентября 2019 года, и в этом примере мы будем использовать именно ее.

Linux — это ядро. То есть оно может быть установлено только в кольце ядра Unix-подобной операционной системы. А вот дистрибутив Linux — другое дело. Он содержит определенные версии ядра Linux, библиотеки GNU libc и командной оболочки Bash (или GNU Shell).

Дистрибутивы Linux обычно поставляются с набором пользовательских приложений во внешних кольцах, поэтому их можно считать полноценными операционными системами. Обратите внимание: дистрибутив, «дистр» и разновидность Linux — одно и то же.

В этом примере я буду использовать дистрибутив Linux Ubuntu 18.04.1 на 64-битном компьютере.

Прежде чем мы начнем, очень важно убедиться в том, что у нас установлены и готовы к работе необходимые пакеты. Выполните для этого команды, показанные в терминале 11.3.

Терминал 11.3. Установка пакетов, необходимых для примера 11.2

```
$ sudo apt-get update
$ sudo apt-get install -y build-essential autoconf libncurses5-dev
libssl-dev bison flex libelf-dev git
...
...
$
```

Несколько замечаний по поводу этих инструкций: `apt` — основной диспетчер пакетов в дистрибутивах Linux, основанных на Debian, а `sudo` — утилита, которая используется для запуска команд от имени *администратора*. Они доступны почти во всех Unix-подобных операционных системах.

Следующими шагами будут клонирование репозитория Linux на GitHub и переход к версии 5.3. Как показано на примере следующих команд (терминал 11.4), версию можно выбрать по тегу выпуска.

Терминал 11.4. Клонирование ядра Linux и переход к версии 5.3

```
$ git clone https://github.com/torvalds/linux
$ cd linux
$ git checkout v5.3
$
```

Теперь, взглянув на корневой каталог, можно увидеть множество файлов и каталогов, которые вместе составляют кодовую базу ядра Linux (терминал 11.5).

Терминал 11.5. Содержимое кодовой базы ядра Linux

```
$ ls
total 760K
drwxrwxr-x  33 kamran kamran 4.0K Jan 28  2018 arch
drwxrwxr-x   3 kamran kamran 4.0K Oct 16 22:11 block
drwxrwxr-x   2 kamran kamran 4.0K Oct 16 22:11 certs
...
drwxrwxr-x 125 kamran kamran  12K Oct 16 22:11 Documentation
drwxrwxr-x 132 kamran kamran 4.0K Oct 16 22:11 drivers
-rw-rw-r--   1 kamran kamran 3.4K Oct 16 22:11 dropped.txt
drwxrwxr-x   2 kamran kamran 4.0K Jan 28  2018 firmare
drwxrwxr-x  75 kamraln kamran 4.0K Oct 16 22:11 fs
drwxrwxr-x  27 kamran kamran 4.0K Jan 28  2018 include
...
-rw-rw-r--   1 kamran kamran  287 Jan 28  2018 Kconfig
drwxrwxr-x  17 kamran kamran 4.0K Oct 16 22:11 kernel
drwxrwxr-x  13 kamran Kamran  12K Oct 16 22:11 lib
-rw-rw-r--   1 kamran kamran 429K Oct 16 22:11 MAINTAINERS
-rw-rw-r--   1 kamran kamran  61K Oct 16 22:11 Makefile
drwxrwxr-x   3 kamran kamran 4.0K Oct 16 22:11 mm
```

```
drwxrwxr-x  69 kamran kamran 4.0K Jan 28  2018 net
-rw-rw-r--   1 kamran kamran  722 Jan 28  2018 README
drwxrwxr-x  28 kamran kamran 4.0K Jan 28  2018 samples
drwxrwxr-x  14 kamran kamran 4.0K Oct 16 22:11 scripts
...
drwxrwxr-x   4 kamran kamran 4.0K Jan 28  2018 virt
drwxrwxr-x   5 kamran kamran 4.0K Oct 16 22:11 zfs
$
```

Некоторые из этих каталогов могут показаться знакомыми: `fs`, `mm`, `net`, `arch` и т. д. Мы не станем подробно рассматривать каждый из них, поскольку они могут существенно отличаться в зависимости от ядра; смысл в том, что большинство ядер имеют одну и ту же внутреннюю структуру.

Итак, получив исходники ядра, мы можем приступить к созданию нашего нового системного вызова. Но сначала нужно выбрать для него уникальный числовой идентификатор; в данном случае я назову его `hello_world` и назначу ему номер `999`.

Первым делом необходимо добавить объявление функции системного вызова в конец заголовочного файла `include/linux/syscalls.h`. После внесения этого изменения файл должен выглядеть так (листинг 11.3).

Листинг 11.3. Объявление нового системного вызова Hello World (include/linux/syscalls.h)

```
/*
 * syscalls.h - Linux syscall interfaces (non-arch-specific)
 *
 * Copyright (c) 2004 Randy Dunlap
 * Copyright (c) 2004 Open Source Development Labs
 *
 * This file is released under the GPLv2.
 * See the file COPYING for more details.
 */

#ifndef _LINUX_SYSCALLS_H
#define _LINUX_SYSCALLS_H

struct epoll_event;
struct iattr;
struct inode;

...

asmlinkage long sys_statx(int dfd, const char __user *path, unsigned flags,
                          unsigned mask, struct statx __user *buffer);

asmlinkage long sys_hello_world(const char __user *str,
                                const size_t str_len,
                                char __user *buf,
                                size_t buf_len);
#endif
```

Согласно описанию, которое приводится вверху, это заголовочный файл с интерфейсами `syscall`, *не зависящими от архитектуры*. Это значит, Linux предоставляет один и тот же набор системных вызовов на всех архитектурах.

В конце файла мы объявили функцию нашего системного вызова, которая принимает четыре аргумента. Как уже объяснялось ранее, эти две пары аргументов отводятся для содержимого и длины входной и выходной строк соответственно.

Обратите внимание: входные аргументы обозначены как `const`, а выходные — нет. Кроме того, идентификатор `__user` означает, что указатели ссылаются на адреса в пользовательском пространстве. Как видите, каждый системный вызов имеет целочисленное значение, которое они возвращают в рамках сигнатуры функции и которое будет служить результатом ее выполнения. Диапазон возвращаемых значений и их смысл варьируются от одного системного вызова к другому. В нашем системном вызове `0` означает успех, а любое другое число — провал.

Теперь нам нужно определить наш системный вызов. Для этого необходимо сначала создать папку с именем `hello_world` в корневом каталоге, используя команды, показанные в терминале 11.6.

Терминал 11.6. Создание каталога hello_world

```
$ mkdir hello_world
$ cd hello_world
$
```

Затем внутри `hello_world` нужно создать файл с именем `sys_hello_world.c`. Он должен иметь следующее содержимое (листинг 11.4).

Листинг 11.4. Определение системного вызова Hello World

```
#include <linux/kernel.h>   // Для printk
#include <linux/string.h>   // Для strcpy, strcat, strlen
#include <linux/slab.h>     // Для kmalloc, kfree
#include <linux/uaccess.h>  // Для copy_from_user, copy_to_user
#include <linux/syscalls.h> // Для SYSCALL_DEFINE4

// Определение системного вызова
SYSCALL_DEFINE4(hello_world,
          const char __user *, str,    // Вводимое имя
          const unsigned int, str_len, // Длина вводимого имени
          char __user *, buf,          // Выходной буфер
          unsigned int, buf_len) {     // Длина выходного буфера
  // Переменная в стеке ядра должна хранить содержимое
  // входного буфера
  char name[64];
  // Переменная в стеке ядра должна хранить итоговое
  // выходное сообщение.
```

```
  char message[96];
  printk("System call fired!\n");
  if (str_len >= 64) {
    printk("Too long input string.\n");
    return -1;
  }

  // Копируем данные из пространства ядра в пространство пользователя
  if (copy_from_user(name, str, str_len)) {
    printk("Copy from user space failed.\n");
    return -2;
  }

  // Формируем итоговое сообщение
  strcpy(message, "Hello ");
  strcat(message, name);
  strcat(message, "!");

  // Проверяем, помещается ли итоговое сообщение в выходной двоичный буфер
  if (strlen(message) >= (buf_len - 1)) {
    printk("Too small output buffer.\n");
    return -3;
  }

  // Копируем сообщение обратно из пространства ядра
  // в пространство пользователя
  if (copy_to_user(buf, message, strlen(message) + 1)) {
    printk("Copy to user space failed.\n");
    return -4;
  }

  // Записываем отправленное сообщение в журнал ядра
  printk("Message: %s\n", message);
  return 0;
}
```

В этом листинге мы использовали макрос `SYSCALL_DEFINE4` для определения нашей функции. Суффикс `DEFINE4` означает, что она принимает четыре аргумента.

Тело функции начинается с объявления двух символьных массивов на вершине стека ядра. У процесса ядра, как и у обычного процесса, есть адресное пространство со стеком. Дальше мы копируем данные из пользовательского пространства в пространство ядра и создаем приветственное сообщение, объединяя несколько строк. Полученная строка все еще находится в памяти ядра. В конце мы копируем сообщение обратно в пространство пользователя и делаем его доступным вызывающему процессу.

В случае ошибки возвращается соответствующий номер, чтобы вызывающий процесс знал о результате выполнения системного вызова.

Следующим шагом будет обновление еще одной таблицы. В архитектурах x86 и x64 предусмотрена лишь одна таблица для системных вызовов; мы должны добавить в нее наш новый системный вызов, чтобы сделать его доступным.

Только после этого системный вызов можно будет использовать на компьютерах с архитектурами x86 и x64. Итак, мы должны добавить в таблицу системный вызов `hello_word` и имя его функции, `sys_hello_world`.

Откройте для этого файл `arch/x86/entry/syscalls/syscall_64.tbl` и добавьте в его конец следующую строчку (листинг 11.5).

Листинг 11.5. Добавление нового системного вызова Hello World в таблицу системных вызовов

```
999     64     hello_world          __x64_sys_hello_world
```

После внесения этого изменения файл должен выглядеть так (терминал 11.7).

Терминал 11.7. Системный вызов Hello World был добавлен в таблицу системных вызовов

```
$ cat arch/x86/entry/syscalls/syscall_64.tbl
...
...
546     x32     preadv2                 __x32_compat_sys_preadv64v2
547     x32     pwritev2                __x32_compat_sys_pwritev64v2
999      64     hello_world             __x64_sys_hello_world
$
```

Обратите внимание на префикс `__x64_` в имени системного вызова. Он говорит о том, что системный вызов доступен только в системах с архитектурой x64.

Для компиляции всех исходных файлов и создания итогового образа ядра Linux используется система сборки Make. Вы должны создать файл `Makefile` в каталоге `hello_world`. Этот файл должен содержать одну строчку следующего вида (листинг 11.6).

Листинг 11.6. Файл Makefile для системного вызова Hello World

```
obj-y := sys_hello_world.o
```

Затем вам нужно добавить каталог `hello_world` в главный файл `Makefile`, который находится в корневом каталоге. Перейдите в корневой каталог ядра, откройте `Makefile` и найдите следующую строчку (листинг 11.7).

Листинг 11.7. Строчка, которую необходимо изменить в корневом файле Makefile

```
core-y += kernel/certs/mm/fs/ipc/security/crypto/block/
```

Добавьте в данный список `hello_world/`. Это всего лишь перечень каталогов, которые должны собираться вместе с ядром.

Нам нужно добавить каталог системного вызова Hello World, чтобы он был включен в процесс сборки и стал частью итогового образа ядра. После редактирования эта строчка должна выглядеть следующим образом (листинг 11.8).

Листинг 11.8. Строчка после редактирования

```
core-y += kernel/certs/mm/fs/hello_world/ipc/security/crypto/block/
```

Следующим шагом будет сборка ядра.

Сборка ядра

Чтобы собрать ядро, нам нужно сначала вернуться в его корневой каталог и предоставить конфигурацию со списком возможностей и модулей, которые будут скомпилированы в процессе сборки.

Команда, показанная в терминале 11.8, пытается сгенерировать нужную нам конфигурацию на основе включенной в текущее ядро Linux. Она берет имеющуюся конфигурацию и при наличии в нашем ядре, которое мы пытаемся собрать, новых значений просит вас их подтвердить. Для подтверждения достаточно нажать клавишу `Enter`.

Терминал 11.8. Создание новой конфигурации на основе текущего активного ядра

```
$ make localmodconfig
...
...
#
# configuration written to .config
#
$
```

Теперь можно начать процесс сборки (терминал 11.9). Поскольку ядро Linux содержит много исходных файлов, сборка может занять несколько часов. Поэтому компиляцию нужно выполнять параллельно.

Если вы используете виртуальную машину, то, пожалуйста, сделайте ее процессор многоядерным. Это существенно ускорит процесс сборки.

Терминал 11.9. Вывод процесса сборки ядра. Обратите внимание на строчку, относящуюся к компиляции системного вызова Hello World

```
$ make -j4
SYSHDR  arch/x86/include/generated/asm/unistd_32_ia32.h
SYSTBL  arch/x86/include/generated/asm/syscalls_32.h
HOSTCC  scripts/basic/bin2c
SYSHDR  arch/x86/include/generated/asm/unistd_64_x32.h
...
```

```
...
UPD     include/generated/compile.h
CC      init/main.o
CC      hello_world/sys_hello_world.o
CC      arch/x86/crypto/crc32c-intel_glue.o
...
...
LD [M]  net/netfilter/x_tables.ko
LD [M]  net/netfilter/xt_tcpudp.ko
LD [M]  net/sched/sch_fq_codel.ko
LD [M]  sound/ac97_bus.ko
LD [M]  sound/core/snd-pcm.ko
LD [M]  sound/core/snd.ko
LD [M]  sound/core/snd-timer.ko
LD [M]  sound/pci/ac97/snd-ac97-codec.ko
LD [M]  sound/pci/snd-intel8x0.ko
LD [M]  sound/soundcore.ko
$
```

Не забудьте установить все необходимые пакеты, представленные в первой части этого раздела, иначе получите ошибки компиляции.

Как видите, начался процесс сборки, состоящий из четырех заданий, которые пытаются скомпилировать файлы C в параллель. Дождитесь завершения этого процесса. После этого вы сможете легко установить свое новое ядро и перезагрузить компьютер (терминал 11.10).

Терминал 11.10. Создание и установка нового образа ядра

```
$ sudo make modules_install install
INSTALL arch/x86/crypto/aes-x86_64.ko
INSTALL arch/x86/crypto/aesni-intel.ko
INSTALL arch/x86/crypto/crc32-pclmul.ko
INSTALL arch/x86/crypto/crct10dif-pclmul.ko
...
...
run-parts: executing /et/knel/postinst.d/initam-tools 5.3.0+ /
boot/vmlinuz-5.3.0+
update-iniras: Generating /boot/initrd.img-5.3.0+
run-parts: executing /etc/keneostinst.d/unattende-urades 5.3.0+ /
boot/vmlinuz-5.3.0+
...
...
Found initrd image: /boot/initrd.img-4.15.0-36-generic
Found linux image: /boot/vmlinuz-4.15.0-29-generic
Found initrd image: /boot/initrd.img-4.15.0-29-generic
done.
$
```

Новый образ ядра версии 5.3.0 был создан и установлен. Теперь мы можем перезагрузить компьютер. Но прежде не забудьте проверить версию текущего ядра, если не знаете ее. В моем случае это `4.15.0-36-generic`. Для проверки я использовал команду, показанную в терминале 11.11.

Терминал 11.11. Проверка версии текущего ядра

```
$ uname -r
4.15.0-36-generic
$
```

Теперь перезагрузим систему (терминал 11.12).

Терминал 11.12. Перезагрузка системы

```
$ sudo reboot
```

В ходе загрузки системы будет выбрано и задействовано новое ядро. Стоит отметить, что загрузчик не подхватит старые ядра; следовательно, если у вас было ядро версии 5.3 и выше, то вам придется загрузить собранный вами образ вручную. В этом вам может помочь следующая ссылка: https://askubuntu.com/questions/82140/how-can-i-boot-with-an-older-kernel-version.

Когда операционная система завершит загрузку, у вас должно быть новое ядро. В этом можно убедиться следующим образом (терминал 11.13).

Терминал 11.13. Проверка версии ядра после перезагрузки

```
$ uname -r
5.3.0+
$
```

Если все прошло как следует, то у вас должно быть загружено новое ядро. Теперь мы можем вернуться к написанию программы на C, которая использует наш только что добавленный системный вызов Hello World. Она будет очень похожа на пример 11.1, в котором фигурировал системный вызов `write`. В листинге 11.9 показан код примера 11.2.

Листинг 11.9. Пример 11.2, инициирующий только что добавленный системный вызов Hello World (ExtremeC_examples_chapter11_2.c)

```
// Это нужно для использования элементов, не входящих в состав POSIX
#define _GNU_SOURCE

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

```
// Это не является частью POSIX!
#include <sys/syscall.h>

int main(int argc, char** argv) {
  char str[20] = "Kam";
  char message[64] = "";

  // Инициируем системный вызов hello world
  int ret_val = syscall(999, str, 4, message, 64);
  if (ret_val < 0) {
    printf("[ERR] Ret val: %d\n", ret_val);
    return 1;
  }
  printf("Message: %s\n", message);
  return 0;
}
```

Как видите, мы инициировали системный вызов под номер `999`. Мы подали на вход строку `Kam` и ожидаем получить в качестве приветствия `Hello Kam!`. Программа ожидает получения результата и выводит буфер с сообщением, записанный системным вызовом в пространстве ядра.

В терминале 11.14 показано, как собрать и запустить наш пример.

Терминал 11.14. Компиляция и запуск примера 11.2

```
$ gcc ExtremeC_examples_chapter11_2.c -o ex11_2.out
$ ./ex11_2.out
Message: Hello Kam!
$
```

Если выполнить этот пример и заглянуть в журнал ядра с помощью команды `dmesg`, то можно увидеть следующие записи, сгенерированные инструкцией `printk` (терминал 11.15).

Терминал 11.15. Использование команды dmesg для просмотра журнальных записей, сгенерированных системным вызовом Hello World

```
$ dmesg
...
...
[  112.273783] System call fired!
[  112.273786] Message: Hello Kam!
$
```

Если запустить пример 11.2 с помощью `strace`, то можно заметить, что он на самом деле использует вызов `999` (терминал 11.16). Это видно в строчке, которая

начинается с `syscall_0x3e7(...)`. Стоит отметить, что `0x3e7` — это 999 в шестнадцатеричной системе.

Терминал 11.16. Мониторинг системных вызовов, инициированных примером 11.2

```
$ strace ./ex11_2.out
...
...
mprotect(0x557266020000, 4096, PROT_READ) = 0
mprotect(0x7f8dd6d2d000, 4096, PROT_READ) = 0
munmap(0x7f8dd6d26000, 27048)           = 0
syscall_0x3e7(0x7fffe7d2af30, 0x4, 0x7fffe7d2af50, 0x40,
0x7f8dd6b01d80, 0x7fffe7d2b088) = 0
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 0), ...}) = 0
brk(NULL)                               = 0x5572674f2000
brk(0x557267513000)
...
...
exit_group(0)                           = ?
+++ exited with 0 +++
$
```

Как видите, был выполнен вызов `syscall_0x3e7`, который вернул `0`. Если отредактировать пример 11.2 так, чтобы передаваемое имя состояло из более чем 64 байт, то получится ошибка. Изменим наш пример и запустим его еще раз (листинг 11.10).

Листинг 11.10. Передача длинного сообщения (больше 64 байт) нашему системному вызову Hello World

```
int main(int argc, char** argv) {
  char name[84] = "A very very long message! It is really hard to
                   produce a big string!";
  char message[64] = "";
  ...
  return 0;
}
```

Снова скомпилируем и запустим пример (терминал 11.17).

Терминал 11.17. Компиляция и запуск отредактированного примера 11.2

```
$ gcc ExtremeC_examples_chapter11_2.c -o ex11_2.out
$ ./ex11_2.out
[ERR] Ret val: -1
$
```

Руководствуясь написанной нами логикой, системный вызов возвращает `-1`. Это также подтверждает выполнение примера с помощью `strace` (терминал 11.18).

Терминал 11.18. Мониторинг системных вызовов, инициированных отредактированным примером 11.2

```
$ strace ./ex11_2.out
...
...
munmap(0x7f1a900a5000, 27048) = 0
syscall_0x3e7(0x7ffdf74e10f0, 0x54, 0x7ffdf74e1110, 0x40,
0x7f1a8fe80d80, 0x7ffdf74e1248) = -1 (errno 1)
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 0), ...}) = 0
brk(NULL)                               = 0x5646802e2000
...
...
exit_group(1)                           = ?
+++ exited with 1 +++
$
```

В следующем разделе мы поговорим о разных подходах к проектированию ядер. В ходе этого обсуждения мы познакомимся с модулями ядра и увидим, как они используются в процессе разработки.

Ядра Unix

В этом разделе речь пойдет об архитектурах, которые были разработаны в ядрах Unix за последние 30 лет. Но прежде, чем обсуждать разные виды ядер, которых, к слову, не так уж и много, следует отметить, что процесс проектирования ядра никак не стандартизован.

Представленные здесь рекомендации опираются на наш многолетний опыт. На их основе была сформирована общая схема внутренних компонентов ядра Unix, один из вариантов которой был показан на рис. 10.5 в предыдущей главе. Таким образом, каждое ядро по-своему уникально. Но все они имеют общую черту: предоставляют доступ к своим возможностям через интерфейс системных вызовов. Однако каждое ядро имеет собственное ви́дение того, как нужно обрабатывать эти вызовы.

Такое разнообразие и споры вокруг него стали одной из самых горячих тем в области компьютерных архитектур в 1990-х годах. В этих обсуждениях принимали участие большие группы людей. Самым знаменитым примером считаются дебаты между *Таненбаумом* и *Торвальдсом*.

Мы не станем углубляться в суть этих дискуссий, но затронем два основных направления в проектировании ядер Unix: *монолит* и *микроядро*. Существуют и другие архитектуры, такие как *гибридные ядра*, *наноядра* и *экзоядра*, и каждая из них имеет определенное назначение.

Но мы сосредоточимся на монолитных ядрах и микроядрах. Проведем их сравнение, чтобы получше познакомиться с их характеристиками.

Монолитные ядра и микроядра

В предыдущей главе, посвященной архитектуре Unix, мы определили ядро как единый процесс, состоящий из множества компонентов. На самом деле это определение относится к монолитному ядру.

Монолитное ядро представляет собой один процесс с единым адресным пространством, содержащий более мелкие компоненты. Микроядра используют противоположный подход. Микроядро — минимальный процесс, из которого в пользовательское пространство вынесены такие компоненты, как файловая система, драйверы устройств и управление процессами; благодаря этому процесс ядра получается более компактным.

Обе архитектуры имеют свои преимущества и недостатки, благодаря чему стали темой одного из самых знаменитых обсуждений в истории операционных систем. Все началось еще в 1992 году, сразу после выпуска первой версии Linux. Начало дискуссии положил *Эндрю С. Таненбаум* публикацией в сети *Usenet*. Более подробно об этом можно почитать на странице «Википедии» https://ru.wikipedia.org/wiki/Спор_Таненбаума_—_Торвальдса.

Публикация породила непримиримое противостояние между Таненбаумом, создателем Linux, *Линусом Торвальдсом*, и многими другими энтузиастами, которые позже стали одними из первых разработчиков Linux. Споры велись о природе монолитных ядер и микроядер. Они касались таких тем, как архитектура ядра и влияние на нее аппаратной платформы.

Споры были продолжительными и сложными, поэтому мы не станем углубляться в них. Сравним оба подхода, чтобы вы получили представление о преимуществах и недостатках каждого из них.

Ниже перечислены различия между монолитными ядрами и микроядрами.

- Монолитное ядро представляет собой единый процесс, содержащий все возможности, предоставляемые этим ядром. Так разрабатывалось большинство ранних ядер Unix, и это считается старым подходом. У микроядер все иначе: каждую возможность они предоставляют в виде отдельного процесса.
- Процесс монолитного ядра находится в пространстве ядра, тогда как в микроядре *серверные процессы* обычно вынесены в пространство пользователя. Серверными называют процессы, которые предоставляют доступ к компонентам ядра, таким как диспетчер памяти, файловая система и т. д. Микроядра отличаются тем, что позволяют размещать серверные процессы в пользовательском пространстве. Это значит, одни операционные системы больше похожи на микроядро, чем другие.

- Монолитные ядра, как правило, быстрее. Это объясняется тем, что все их функции выполняются внутри одного процесса. Микроядрам же нужно организовывать *обмен сообщениями* между пространствами пользователя и ядра, что приводит к большему количеству системных вызовов и переключений контекста.
- В монолитной архитектуре все драйверы устройств загружаются прямо в ядро. То есть в рамках ядра может выполняться код, написанный сторонними поставщиками. Любой изъян в драйвере устройства или любом другом компоненте ядра может привести к системному сбою. С микроядрами все иначе, поскольку все драйверы устройств и многие другие компоненты выполняются в пространстве пользователя. Можно предположить, что это одна из причин, почему монолитные ядра не применяются в критически важных проектах.
- Для взлома монолитного ядра и, следовательно, всей системы достаточно внедрить в него небольшой фрагмент вредоносного кода. С микроядром так сделать не получится, поскольку многие серверные процессы вынесены в пользовательское пространство, а в пространстве ядра остается минимальный набор важнейших функций.
- В монолитной архитектуре малейшее изменение исходного кода требует перекомпиляции всего ядра и создания его нового образа. А для загрузки этого образа необходимо перезагрузить компьютер. Изменения в микроядре могут потребовать перекомпиляции лишь отдельного серверного процесса, без перезагрузки всей системы. В монолитных ядрах похожего эффекта можно достичь с помощью модулей.

Один из самых известных примеров микроядра — MINIX. Этот проект, написанный Эндрю С. Таненбаумом, изначально создавался в образовательных целях. В 1991 году Линус Торвальдс использовал MINIX в качестве среды для разработки собственного ядра в микропроцессоре 80386.

На протяжении уже почти 30 лет Линус является самым известным и успешным сторонником монолитных ядер, поэтому в следующем разделе речь пойдет о Linux.

Linux

Вы уже познакомились с ядром Linux в предыдущем разделе, в котором мы разрабатывали новый системный вызов. Здесь же мы сделаем акцент на монолитности Linux и на том факте, что в процессе ядра находятся все его возможности.

Однако должен существовать способ добавить в ядро новую функциональность, не прибегая к его перекомпиляции. Этого нельзя достичь за счет новых системных вызовов, поскольку, как вы уже видели, для их добавления нужно отредактировать много важных файлов, а это означает компиляцию ядра заново.

Но есть другой подход. Он предусматривает написание и динамическое подключение к ядру модулей. Именно о модулях пойдет речь в следующем подразделе, а затем мы попробуем написать собственный модуль для Linux.

Модули ядра

Монолитные ядра обычно предусматривают механизм, который позволяет динамически подключать к активному ядру новые компоненты. Эти подключаемые компоненты называются модулями ядра.

В отличие от серверных процессов в микроядрах, которые фактически являются отдельными процессами, взаимодействующими между собой по IPC, модули представляют собой *объектные файлы ядра*, скомпилированные и готовые к загрузке в его процесс. Эти файлы можно собрать статически и сделать их частью образа ядра; но вы также можете загружать их динамически, когда ядро уже работает.

Обратите внимание: по своей сути объектные файлы ядра аналогичны объектным файлам, которые создаются в процессе разработки на языке C.

Стоит еще раз подчеркнуть, что плохое поведение модуля может привести к *сбою всего ядра*.

Взаимодействие с модулями ядра происходит без участия системных вызовов, и с этими модулями нельзя работать путем вызова функций или использования определенного API. В целом общение с модулями ядра в Linux и некоторых похожих операционных системах происходит тремя способами.

- *Файлы устройств в каталоге /dev.* Модули ядра в основном разрабатываются для использования в качестве драйверов устройств. Вот почему самый распространенный способ взаимодействия с ними — использование файлов внутри каталога /dev, которые мы обсуждали в предыдущей главе. Читая и записывая эти файлы, вы можете обмениваться информацией с модулями.
- *Содержимое procfs.* Файлы в каталоге /proc можно использовать для чтения метаинформации об отдельных модулях ядра. Эти файлы также позволяют отправлять модулям метаинформацию или управляющие команды. Применение procfs будет показано чуть ниже, в примере 11.3.
- *Содержимое sysfs.* Это еще одна файловая система в Linux, которая позволяет управлять пользовательскими процессами и другими компонентами, относящимися к ядру (включая модули ядра), с помощью скриптов или вручную. Ее можно считать новой версией procfs.

Чтобы поближе познакомиться с модулями ядра, лучше всего написать собственный модуль, чем мы и займемся далее. Мы напишем модуль Hello World для Linux. Стоит отметить, что модули ядра существуют не только в Linux, но и в других монолитных ядрах. Например, FreeBSD использует похожий механизм.

Создание нового модуля ядра для Linux

Здесь мы напишем новый модуль ядра для Linux под названием Hello World. Он будет создавать запись в procfs. Затем мы используем эту запись, чтобы прочитать строку с приветствием.

Вы научитесь писать, компилировать и загружать модули в ядро, выгружать их оттуда и читать данные из записи в procfs. Основное назначение этого примера — дать вам возможность набраться опыта и стать более самостоятельным разработчиком.

Модули ядра компилируются в объектные файлы, которые можно загружать непосредственно в активное ядро. Для этого не нужно перезагружать систему. Главное, что7бы ваш модуль не натворил беды в ядре и не вывел его из строя. То же самое можно сказать о выгрузке модуля.

Первым делом нужно создать каталог, в котором будут находиться все файлы, относящиеся к модулю ядра. Это третий пример в данной главе, и потому назовем данный каталог `ex11_3` (терминал 11.19).

Терминал 11.19. Создание корневого каталога для примера 11.3

```
$ mkdir ex11_3
$ cd ex11_3
$
```

Затем создадим файл под названием `hwkm.c`, сокращенно от Hello World Kernel Module, со следующим содержимым (листинг 11.11).

Листинг 11.11. Модуль ядра Hello World (ex11_3/hwkm.c)

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/proc_fs.h>

// Структура, ссылающаяся на файл proc
struct proc_dir_entry *proc_file;

// Функция обратного вызова для чтения файла
ssize_t proc_file_read(struct file *file, char __user *ubuf, size_t count,
loff_t *ppos) {
  int copied = 0;
  if (*ppos > 0) {
    return 0;
  }
  copied = sprintf(ubuf, "Hello World From Kernel Module!\n");
  *ppos = copied;
  return copied;
}
```

```
static const struct file_operations proc_file_fops = {
 .owner = THIS_MODULE,
 .read  = proc_file_read
};

// Обратный вызов для инициализации модуля
static int __init hwkm_init(void) {
  proc_file = proc_create("hwkm", 0, NULL, &proc_file_fops);
  if (!proc_file) {
    return -ENOMEM;
  }
  printk("Hello World module is loaded.\n");
  return 0;
}

// Обратный вызов для выхода из модуля
static void __exit hkwm_exit(void) {
  proc_remove(proc_file);
  printk("Goodbye World!\n");
}

// Определение обратных вызовов для работы с модулем
module_init(hwkm_init);
module_exit(hkwm_exit);
```

С помощью двух последних выражений мы зарегистрировали обратные вызовы нашего модуля, которые отвечают за инициализацию и завершение работы. Эти функции вызываются соответственно при загрузке и выгрузке модуля. Выполнение кода начинается с обратного вызова для инициализации.

Заглянув внутрь функции `hwkm_init`, вы увидите, что она создает файл `hwkm` в каталоге /proc. Рядом находится обратный вызов для завершения работы, `hwkm_exit`; он удаляет файл `hwkm` из этого каталога. Файл `/proc/hwkm` позволяет взаимодействовать с модулем ядра из пользовательского пространства.

Функция `proc_file_read` — это обратный вызов, который дает возможность процессу из пользовательского пространства читать файл `/proc/hwkm`. Как вы вскоре увидите, для чтения файла будет использоваться утилита `cat`. С ее помощью мы просто будем копировать строку `Hello World From Kernel Module!` в пространство пользователя.

Обратите внимание: на данном этапе код нашего модуля имеет неограниченный доступ к почти любым компонентам ядра и из него в пользовательское пространство может утечь какая угодно информация. Это серьезная дыра в безопасности, вследствие чего вам следует поближе познакомиться с рекомендациями по написанию безопасных модулей ядра.

Чтобы скомпилировать приведенный выше код, нам нужен подходящий компилятор, с помощью которого мы потенциально можем выполнить компоновку с нужными нам библиотеками. Чтобы упростить себе жизнь, создадим файл с именем

`Makefile`, который вызовет все необходимые инструменты для сборки нашего модуля ядра.

Содержимое этого файла показано в листинге 11.12.

Листинг 11.12. Содержимое файла Makefile для модуля ядра Hello World

```
obj-m += hwkm.o

all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Затем можно выполнить команду `make` (терминал 11.20).

Терминал 11.20. Сборка модуля ядра Hello World

```
$ make
make -C /lib/modules/54.318.0+/build M=/home/kamran/extreme_c/ch11/codes/ex11_3
modules
make[1]: Entering directory '/home/kamran/linux'
  CC [M]  /home/kamran/extreme_c/ch11/codes/ex11_3/hwkm.o
  Building modules, stage 2.
  MODPOST 1 modules
WARNING: modpost: missing MODULE_LICENSE() in /home/kamran/extreme_c/ch11/codes/
ex11_3/hwkm.o
see include/linux/module.h for more information
  CC      /home/kamran/extreme_c/ch11/codes/ex11_3/hwkm.mod.o
  LD [M]  /home/kamran/extreme_c/ch11/codes/ex11_3/hwkm.ko
make[1]: Leaving directory '/home/kamran/linux'
$
```

Как видите, в результате компиляции кода получается объектный файл, который затем компонуется с другими библиотеками в файл `.ko`. Среди сгенерированных результатов должен находиться файл с именем `hwkm.ko`.

Обратите внимание на расширение `.ko`. Оно означает, что выходной файл является объектным файлом ядра. Это нечто вроде разделяемой библиотеки, которую можно динамически загружать в ядро, где она будет выполняться.

Пожалуйста, обратите внимание: процесс сборки, представленный в предыдущем терминале, выдал предупреждение о том, что у модуля нет лицензии. Крайне желательно, чтобы в ходе разработки или развертывания в тестовых и промышленных средах генерировались лицензированные модули ядра.

В терминале 11.21 показан список файлов, которые можно получить в результате сборки модуля ядра.

Терминал 11.21. Список существующих файлов после сборки модуля ядра Hello World

```
$ ls -l
total 556
-rw-rw-r-- 1 kamran kamran    154 Oct 19 00:36 Makefile
-rw-rw-r-- 1 kamran kamran      0 Oct 19 08:15 Module.symvers
-rw-rw-r-- 1 kamran kamran   1104 Oct 19 08:05 hwkm.c
-rw-rw-r-- 1 kamran kamran 272280 Oct 19 08:15 hwkm.ko
-rw-rw-r-- 1 kamran kamran    596 Oct 19 08:15 hwkm.mod.c
-rw-rw-r-- 1 kamran kamran 104488 Oct 19 08:15 hwkm.mod.o
-rw-rw-r-- 1 kamran kamran 169272 Oct 19 08:15 hwkm.o
-rw-rw-r-- 1 kamran Kamran     54 Oct 19 08:15 modules.order
$
```

Мы использовали инструменты сборки модулей из ядра Linux версии 5.3.0. Вы можете получить ошибку, если попытаетесь скомпилировать этот пример с помощью ядра, версия которого ниже 3.10.

Для загрузки и установки модуля ядра `hwkm` мы используем команду Linux `insmod`, как показано в терминале 11.22.

Терминал 11.22. Загрузка и установка модуля ядра Hello World

```
$ sudo insmod hwkm.ko
$
```

Теперь, заглянув в журнал ядра, можно увидеть строчки, сгенерированные функцией инициализации. Для просмотра самых последних журнальных записей ядра используйте команду `dmesg`, как показано в терминале 11.23.

Терминал 11.23. Проверка журнальных сообщений ядра после установки модуля

```
$ dmesg
...
...
[ 7411.519575] Hello World module is loaded.
$
```

Итак, модуль загружен, и уже должен быть создан файл `/proc/hwkm`. Мы можем прочитать его с помощью команды `cat` (терминал 11.24).

Терминал 11.24. Чтение файла /proc/hwkm с использованием команды cat

```
$ cat /proc/hwkm
Hello World From Kernel Module!
$ cat /proc/hwkm
Hello World From Kernel Module!
$
```

Здесь видно, что обе операции чтения файла возвращают одну и ту же строку: `Hello World From Kernel Module!`. Обратите внимание: эта строка копируется в пользовательское пространство модулем ядра и команда `cat` направляет ее в стандартный вывод.

В Linux для выгрузки модуля предусмотрена команда `rmmod` (терминал 11.25).

Терминал 11.25. Выгрузка модуля ядра Hello World

```
$ sudo rmmod hwkm
$
```

После выгрузки модуля опять заглянем в журнал ядра, чтобы найти прощальное сообщение (терминал 11.26).

Терминал 11.26. Проверка журнальных сообщений ядра после выгрузки модуля

```
$ dmesg
...
...
[ 7411.519575] Hello World module is loaded.
[ 7648.950639] Goodbye World!
$
```

Как вы сами можете убедиться, такие модули очень удобны для написания кода ядра.

В завершение этой главы, мне кажется, будет полезно перечислить особенности модулей ядра, с которыми мы успели познакомиться:

- модули ядра можно загружать и выгружать, не прибегая к перезагрузке компьютера;
- после загрузки модули становятся частью ядра и получают доступ ко всем его компонентам и структурам. Это можно считать уязвимостью, но ядро Linux имеет средства защиты от загрузки нежелательных модулей;
- модули ядра можно компилировать отдельно. А вот системные вызовы требуют перекомпиляции всего ядра, на что может легко уйти несколько часов.

Наконец, модули ядра могут пригодиться при разработке кода, который необходимо выполнять в самом ядре и инициировать с помощью системных вызовов. Сначала вы можете реализовать свою логику в виде модуля, а затем, протестировав ее должным образом, сделать доступной через настоящий системный вызов.

Процесс разработки системных вызовов с нуля может быть довольно трудоемким, поскольку требует регулярной перезагрузки компьютера. Но если сначала написать и протестировать логику в виде модуля, то это упростит разработку ядра. Стоит отметить: если ваш код нестабильный, то неважно, где он находится — в модуле

или в системном вызове; он может привести к сбою в ядре, что потребует перезагрузки системы.

В этом разделе мы обсудили различные виды ядер. Мы также рассмотрели модуль, который можно использовать внутри монолитного ядра, загружая и выгружая его динамически.

Резюме

На этом завершается наше обсуждение Unix, начатое в предыдущей главе. Вы узнали:

- что такое системный вызов и как он предоставляет доступ к определенным функциям;
- что происходит внутри ядра, когда вы инициируете системный вызов;
- каким образом некоторые системные вызовы можно инициировать прямо из кода на C;
- как добавить новый системный вызов в существующее Unix-подобное ядро (Linux) и как перекомпилировать это ядро;
- что такое монолитное ядро и чем оно отличается от микроядра;
- как работают модули монолитного ядра и как написать новый модуль для Linux.

В следующей главе мы поговорим о стандартах языка C и его последней версии, C18. Вы познакомитесь с новыми возможностями, которые появились в ней.

12 Последние нововведения в C

Прогресс не остановить. Язык программирования C является стандартом ISO, постоянно пересматриваемым в попытках сделать его лучше и привнести в него новые возможности. Но это не значит, что C становится проще; по мере развития языка в нем появляются новые и сложные концепции.

В этой главе я проведу краткий обзор новшеств C11. Как вы, наверное, знаете, стандарт C11 пришел на смену C99 и позже был заменен стандартом C18. Иными словами, C18 — самая последняя версия языка C, а C11 — предыдущая.

Интересно, что в C18 не появилось никаких новых возможностей; эта версия содержит лишь исправления ошибок, найденных в C11. Таким образом, все, что мы говорим о C11, фактически относится и к C18 — то есть к самому последнему стандарту C. Как видите, в C наблюдаются постоянные улучшения... вопреки мнению о том, что этот язык давно умер!

В данной главе будет представлен краткий обзор следующих тем:

- способы определения версии C и написания кода, совместимого с разными версиями этого языка;
- новые средства оптимизации и защиты исходного кода, такие как *невозвращаемые* функции и функции *с проверкой диапазона*;
- новые типы данных и методы компоновки памяти;
- функции с обобщенными типами;
- поддержка Unicode в C11, которой не хватало в предыдущих стандартах этого языка;
- анонимные структуры и объединения;
- встроенная поддержка многопоточности и методов синхронизации в C11.

Начнем эту главу с обсуждения стандарта C11 и его нововведений.

C11

Разработка нового стандарта для технологии, которая используется на протяжении более 30 лет, — непростая задача. На C написаны миллионы (если не миллиарды!) строчек кода, и если вы хотите добавить новые возможности, то это нужно делать так, чтобы не затронуть существующий код. Новшества не должны создавать новые проблемы для имеющихся программ и не должны содержать ошибки. Такой взгляд на вещи может показаться идеалистическим, но это то, к чему нам следует стремиться.

Приведенный ниже PDF-документ находится на сайте *Open Standards* и выражает обеспокоенность и мысли участников сообщества C перед началом работы над C11: http://www.open-std.org/JTC1/SC22/wg14/www/docs/n1250.pdf. Его полезно почитать, поскольку в нем собран опыт разработки нового стандарта для языка, на котором уже было написано несколько тысяч проектов.

Мы будем рассматривать выпуск C11 с учетом всего вышесказанного. Будучи опубликованным впервые, стандарт C11 был далек от идеала и имел некоторые серьезные дефекты, со списком которых можно ознакомиться по адресу http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2244.htm.

Через семь лет после выхода C11 был представлен стандарт C18, который должен был исправить недостатки предшественника. Стоит отметить, что C18 также *неофициально* называют C17, но это один и тот же стандарт. На странице, приведенной в ссылке выше, можно просмотреть перечень дефектов и их текущее состояние. Если состояние дефекта помечено как C17, то это значит, он был исправлен в рамках C18. Это показывает, насколько сложным и щепетильным может быть процесс формирования стандарта с таким большим количеством пользователей, как у языка C.

В следующих разделах речь пойдет о новых возможностях C11. Но прежде, чем мы по ним пройдемся, необходимо убедиться в том, что у нас есть компилятор, совместимый с данным стандартом. Об этом мы позаботимся в следующем разделе.

Определение поддерживаемой версии стандарта C

На момент написания этих строк с момента выхода стандарта C11 прошло почти восемь лет. И потому было бы логично ожидать, что он уже поддерживается многими компиляторами. И это действительно так. Открытые компиляторы, такие как `gcc` и `clang`, имеют полную поддержку C11, но при необходимости могут переключаться на C99 и даже более ранние версии C. В данном разделе я покажу, как с помощью специального макроса определить версию C и в зависимости от нее использовать поддерживаемые возможности языка.

Если ваш компилятор поддерживает разные версии стандарта C, то первым делом нужно проверить, какая версия является текущей. Каждый стандарт C определяет

специальный макрос, позволяющий сделать это. До сих пор мы использовали `gcc` в Linux и `clang` в macOS. В `gcc` 4.1 C11 предоставляется в качестве одного из поддерживаемых стандартов.

Рассмотрим следующий пример, чтобы понять, как на этапе выполнения узнать текущую версию стандарта C, используя уже определенный макрос (листинг 12.1).

Листинг 12.1. Определение версии стандарта C (ExtremeC_examples_chapter12_1.c)

```
#include <stdio.h>

int main(int argc, char** argv) {
#if __STDC_VERSION__ >=  201710L
  printf("Hello World from C18!\n");
#elif __STDC_VERSION__ >= 201112L
  printf("Hello World from C11!\n");
#elif __STDC_VERSION__ >= 199901L
  printf("Hello World from C99!\n");
#else
  printf("Hello World from C89/C90!\n");
#endif
  return 0;
}
```

Как видите, данный код может различать разные версии стандарта C. Чтобы продемонстрировать, как разные версии приводят к разному выводу, скомпилируем этот исходный код несколько раз с применением разных стандартов C, поддерживаемых компилятором.

Чтобы заставить компилятор использовать определенный стандарт C, ему нужно передать параметр `-std=CXX`. Взгляните на следующую команду и на вывод, который она генерирует (терминал 12.1).

Терминал 12.1. Компиляция примера 12.1 с помощью разных версий стандарта C

```
$ gcc ExtremeC_examples_chapter12_1.c -o ex12_1.out
$ ./ex12_1.out
Hello World from C11!
$ gcc ExtremeC_examples_chapter12_1.c -o ex12_1.out -std=c11
$ ./ex12_1.out
Hello World from C11!
$ gcc ExtremeC_examples_chapter12_1.c -o ex12_1.out -std=c99
$ ./ex12_1.out
Hello World from C99!
$ gcc ExtremeC_examples_chapter12_1.c -o ex12_1.out -std=c90
$ ./ex12_1.out
Hello World from C89/C90!
$ gcc ExtremeC_examples_chapter12_1.c -o ex12_1.out -std=c89
$ ./ex12_1.out
Hello World from C89/C90!
$
```

Как видите, в новых компиляторах по умолчанию используется C11. В более старых версиях для включения C11 может понадобиться параметр `-std`. Обратите внимание на комментарии в начале файла. Я использовал многострочный формат, `/* ... */`, вместо однострочного, `//`. Дело в том, что однострочные комментарии не поддерживались в стандартах, предшествовавших C99. Поэтому пришлось сделать комментарии многострочными, чтобы код компилировался со всеми версиями C.

Удаление функции gets

Из C11 была убрана знаменитая функция `gets`. Она была подвержена атакам с *переполнением буфера*, и в предыдущих версиях ее решили сделать *нерекомендуемой*. Позже она была удалена в рамках стандарта C11. Следовательно, старый исходный код, в котором используется эта функция, нельзя скомпилировать с помощью C11.

Вместо `gets` можно использовать функцию `fgets`. Вот отрывок из справочной страницы `gets` в macOS.

Соображения безопасности

Функция gets() не подходит для безопасного использования. Ввиду отсутствия проверки диапазона и неспособности вызывающей программы надежно определить длину следующей входной строчки, применение этой функции позволяет недобросовестным пользователям вносить произвольные изменения в функциональность запущенной программы с помощью атаки переполнения буфера. В любых ситуациях настоятельно рекомендуется задействовать функцию fgets() (см. FSA).

Изменения в функции fopen

Функция `fopen` обычно используется для открытия файла и возвращения его дескриптора. Понятие *файла* в Unix очень абстрактно и может не иметь ничего общего с файловой системой. Функция `fopen` имеет следующие сигнатуры (листинг 12.2).

Листинг 12.2. Различные сигнатуры функций семейства fopen

```
FILE* fopen(const char *pathname, const char *mode);
FILE* fdopen(int fd, const char *mode);
FILE* freopen(const char *pathname, const char *mode, FILE *stream);
```

Все сигнатуры, приведенные выше, принимают входной параметр `mode`. Это строка, которая определяет режим открытия файла. В терминале 12.2 приведено описание,

взятое из справочной страницы `fopen` в FreeBSD. В нем объясняется, как следует использовать `mode`.

Терминал 12.2. Отрывок из справочной страницы fopen в FreeBSD

```
$ man 3 fopen
...
The argument mode points to a string beginning with one of the following letters:

     "r"     Open for reading. The stream is positioned at the beginning
             of the file. Fail if the file does not exist.

     "w"     Open for writing. The stream is positioned at the beginning
             of the file. Create the file if it does not exist.

     "a"     Open for writing. The stream is positioned at the end of
             the file. Subsequent writes to the file will always end up
             at the then current end of file, irrespective of
             any intervening fseek(3) or similar. Create the file
             if it does not exist.

     An optional "+" following "r", "w", or "a" opens the file
     for both reading and writing. An optional "x" following "w" or
     "w+" causes the fopen() call to fail if the file already exists.
     An optional "e" following the above causes the fopen() call to set
     the FD_CLOEXEC flag on the underlying file descriptor.
     The mode string can also include the letter "b" after either
     the "+" or the first letter.
...
$
```

Режим `x`, описанный в данном отрывке, был представлен вместе со стандартом C11. Чтобы открыть файл для записи, функции `fopen` нужно передать режим `w` или `w+`. Но проблема вот в чем: если файл уже существует, то режимы `w` и `w+` сделают его пустым.

Поэтому если программист хочет добавить что-то в файл, не стирая имеющееся содержимое, то должен задействовать другой режим, `a`. Следовательно, перед вызовом `fopen` ему нужно проверить существование файла, используя API файловой системы, такой как `stat`, и затем выбрать подходящий режим в зависимости от результата. Но теперь программист может сначала попробовать режим `wx` или `wx+`, и если файл уже существует, то `fopen` вернет ошибку. После этого можно продолжить, применяя режим `a`.

Таким образом, открытие файла требует меньше шаблонного кода, поскольку нам больше не нужно проверять его существование с помощью API файловой системы. Теперь файл можно открыть в любом режиме, используя одну лишь функцию `fopen`.

В C11 также появился API `fopen_s`. Это безопасная версия `fopen`. Согласно документации, которая находится по ссылке https://en.cppreference.com/w/c/io/fopen, функция `fopen_s` выполняет дополнительную проверку предоставленных ей буферов и их границ, что позволяет обнаружить любые несоответствия.

Функции с проверкой диапазона

Программам на языке C, которые работают с массивами строк и байтов, присуща одна серьезная проблема: они могут легко выйти за пределы диапазона, определенного для буфера или байтового массива.

Напомню, буфер — область памяти, которая служит для хранения массива байтов или строковой переменной. Выход за ее границы приводит к *переполнению буфера*, чем могут воспользоваться злоумышленники, чтобы организовать атаку (которую обычно называют *атакой переполнения буфера*). Это приводит либо к *отказу в обслуживании* (denial of service, DoS), либо к *эксплуатации* атакуемой программы.

Большинство таких атак обычно начинаются с функции, которая работает с массивами символов или байтов. В число *уязвимых* попадают функции обработки строк наподобие `strcpy` и `strcat`, находящиеся в `string.h`. У них нет механизмов проверки границ, которые могли бы предотвратить атаки переполнения буфера.

Однако в C11 появился новый набор функций *с проверкой диапазона*. Они имеют те же имена, что и функции для работы со строками, но с суффиксом `_s` в конце. Он означает, что они являются *безопасной* (secure) разновидностью традиционных функций и проводят дополнительные проверки на этапе выполнения, защищаясь от уязвимостей. Среди функций с проверкой диапазона, появившихся в C11, можно выделить `strcpy_s` и `strcat_s`.

Эти функции принимают дополнительные аргументы для входных буферов, которые исключают выполнение опасных операций. Например, функция `strcpy_s` имеет следующую сигнатуру (листинг 12.3).

Листинг 12.3. Сигнатура функции strcpy_s

```
errno_t strcpy_s(char *restrict dest, rsize_t destsz, const char *restrict src);
```

Как видите, второй аргумент — длина буфера `dest`. С его помощью функция проводит определенные проверки на этапе выполнения; например, она убеждается в том, что строка `src` не длиннее буфера `dest`, предотвращая тем самым запись в невыделенную память.

Невозвращаемые функции

Вызов функции можно завершить либо с помощью ключевого слова `return`, либо при достижении конца ее блока. Бывают также ситуации, в которых вызов функции никогда не завершается, и обычно это делается намеренно. Взгляните на следующий пример кода, представленный в листинге 12.4.

Листинг 12.4. Пример функции, которая никогда не возвращается

```
void main_loop() {
  while (1) {
    ...
  }
}

int main(int argc, char** argv) {
  ...
  main_loop();
  return 0;
}
```

Основную работу в этой программе выполняет функция `main_loop`. После возвращения из нее программу можно считать завершенной. В таких исключительных случаях компилятор может провести дополнительную оптимизацию, но для этого он должен знать, что функция `main_loop` никогда не возвращается.

В C11 вы можете указать, что функция является *невозвращаемой* и никогда не прекращает работу. Для этого можно использовать ключевое слово `_Noreturn` из заголовочного файла `stdnoreturn.h`. Таким образом, код из листинга 12.4 можно изменить в соответствии с C11 (листинг 12.5).

Листинг 12.5. Использование ключевого слова _Noreturn, чтобы пометить функцию main_loop как невозвращаемую

```
_Noreturn void main_loop() {
  while (true) {
    ...
  }
}
```

Существуют и другие функции, которые считаются невозвращаемыми: `exit`, `quick_exit` (были добавлены недавно в рамках C11 для быстрого завершения программы) и `abort`. Кроме того, зная о невозвращаемых функциях, компилятор может распознать вызовы, которые не возвращаются по ошибке, и сгенерировать соответствующее предупреждение, поскольку это может быть признаком некорректной логики. Обратите внимание: ситуация, когда функция, помеченная как `_Noreturn`, возвращается, является примером *неопределенного поведения* и ни в коем случае не приветствуется.

Макрос для обобщенных типов

В C11 появилось новое ключевое слово: `_Generic`. С его помощью можно писать макросы, которые получают информацию о типе на этапе выполнения. Иными словами, вы можете написать макрос, который меняет свое значение в зависимости от типов своих аргументов. Это обычно называют *обобщающим селектором* (generic selection). Взгляните на следующий пример кода в листинге 12.6.

Листинг 12.6. Пример макроса _Generic

```
#include <stdio.h>
#define abs(x) _Generic((x), \
                        int: absi, \
                        double: absd)(x)
int absi(int a) {
  return a > 0 ? a : -a;
}

double absd(double a) {
  return a > 0 ? a : -a;
}

int main(int argc, char** argv) {
  printf("abs(-2): %d\n", abs(-2));
  printf("abs(2.5): %f\n", abs(2.5));;
  return 0;
}
```

В определении макроса есть два разных выражения, которые выбираются в зависимости от типа аргумента `x`. Для значений `int` используется `absi`, а для значений `double` — `absd`. Эта возможность появилась еще до C11, и ее можно встретить в более старых компиляторах, но раньше она не была частью стандарта. С выходом C11 данный синтаксис вошел в спецификацию языка C, и вы можете создавать с его помощью макросы, которые учитывают тип.

Unicode

Одним из самых важных нововведений в стандарте C11 стала поддержка Unicode (в виде кодировок UTF-8, UTF-16 и UTF-32). В языке C она долгое время отсутствовала, и программистам приходилось использовать сторонние библиотеки, такие как IBM *International Components for Unicode* (ICU).

До выхода C11 нам были доступны только восьмибитные типы `char` и `unsigned char`, предназначенные для хранения символов кодировок ASCII и Extended ASCII. Строки имели вид массивов этих ASCII-символов.

Стандарт ASCII состоит из 128 символов, которые занимают 7 бит. У него есть расширенная версия, Extended ASCII, содержащая дополнительные 128 символов (в общей сложности получается 256). Для их хранения достаточно восьмибитной или однобайтной переменной. В тексте, представленном ниже, ASCII может означать как обычную, так и расширенную версию кодировки.

Обратите внимание: поддержка символов и строк ASCII является фундаментальной и из языка C никогда не исчезнет. Поэтому можно быть уверенными в том, что в C мы всегда сможем работать с ASCII. Но в C11 появилась поддержка новых символов и, следовательно, новых строк, которые используют разное количество байтов (а не один байт для каждого символа).

Если более подробно, то в ASCII каждому символу выделяется один байт. Следовательно, все байты и символы взаимозаменяемы, но в целом это довольно нетипичный подход. В других кодировках применяются новые методы хранения широкого диапазона символов размером больше одного байта.

В ASCII в общей сложности имеется 256 символов. Поэтому все они могут поместиться в одном однобайтном (восьмибитном) символе. Если нам нужны дополнительные символы, то их числовые значения после 255 будут занимать больше одного байта. Символы, которые не влезают в один байт, обычно называют *широкими*. Символы ASCII не являются таковыми по определению.

Стандарт Unicode предусматривает различные методы использования более одного байта для кодирования всех символов в ASCII и Extended ASCII, а также широких символов. Эти методы называются *кодировки*. Среди этих кодировок в Unicode самыми известными являются UTF-8, UTF-16 и UTF-32. В UTF-8 первый байт используется для хранения первой половины таблицы ASCII, а остальные (обычно не более четырех) — для второй половины символов ASCII со всеми другими широкими символами. Таким образом, UTF-8 считается кодировкой переменной длины. Она использует биты в первом байте символа в целях обозначения количества байтов, которые необходимо прочитать, чтобы извлечь этот символ полностью. Кодировка UTF-8 считается надмножеством ASCII, поскольку ASCII-символы в ней представлены точно таким же образом (это не относится к символам из Extended ASCII).

UTF-16, как и UTF-8, задействует одно или два *слова* (каждое слово содержит 16 бит) для хранения всех символов; следовательно, это тоже кодировка переменной длины. В UTF-32 для значений всех символов используются ровно 4 байта; следовательно, это кодировка фиксированной длины. Для приложений, в которых для хранения символов, применяемых не так часто, используется меньшее количество байтов, подходит в первую очередь UTF-8 и затем UTF-16.

В UTF-32 всегда используется фиксированное число байтов, даже для символов ASCII. Поэтому для хранения строк данная кодировка требует больше памяти

и места на диске по сравнение с аналогами. UTF-8 и UTF-16 можно считать сжатыми кодировками, но они требуют дополнительных вычислений, чтобы вернуть значение символа.

Больше информации о строках UTF-8, UTF-16 и UTF-32, а также о том, как их декодировать, можно найти в «Википедии» и других источниках, включая следующие:

- https://unicodebook.readthedocs.io/unicode_encodings.html;
- https://javarevisited.blogspot.com/2015/02/difference-between-utf-8-utf-16-and-utf.html.

В C11 имеется поддержка всех кодировок Unicode, перечисленных выше. Взгляните на следующий пример, под номером 12.3. Он определяет различные строки в форматах ASCII, UTF-8, UTF-16 и UTF-32 и подсчитывает количество байтов, которое используется для их хранения, а также количество символов в каждой из них. Код будет представлен поэтапно, с предоставлением дополнительных комментариев. В листинге 12.7 показаны все необходимые операции подключения и объявления.

Листинг 12.7. Подключения и объявления, необходимые для сборки примера 12.3 (ExtremeC_examples_chapter12_3.c)

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#ifdef __APPLE__

#include <stdint.h>

typedef uint16_t char16_t;
typedef uint32_t char32_t;

#else
#include <uchar.h> // Нужно для char16_t и char32_t
#endif
```

Этот код состоит из инструкций `include` для примера 12.3. Как видите, в macOS нет заголовка `uchar.h`, и нам пришлось объявить новые типы для `char16_t` и `char32_t`. Тем не менее мы имеем полную поддержку строк Unicode. В Linux нет вообще никаких проблем с Unicode в C11.

Следующий участок кода (листинг 12.8) демонстрирует функции, которые используются для подсчета количества байтов и символов в разных видах строк Unicode. Обратите внимание: в C11 для работы со строками Unicode не предусмотрено никаких вспомогательных функций, поэтому нам приходится самостоятельно писать

для них `strlen`. На самом деле наша версия функций `strlen` возвращает не только количество символов, но и число потребленных байтов. Я не стану углубляться в детали реализации, но настоятельно рекомендую с ними ознакомиться.

Листинг 12.8. Определения функций, используемых в примере 12.3 (ExtremeC_examples_chapter12_3.c)

```
typedef struct {
  long num_chars;
  long num_bytes;
} unicode_len_t;

unicode_len_t strlen_ascii(char* str) {
  unicode_len_t res;
  res.num_chars = 0;
  res.num_bytes = 0;
  if (!str) {
    return res;
  }
  res.num_chars = strlen(str) + 1;
  res.num_bytes = strlen(str) + 1;
  return res;
}

unicode_len_t strlen_u8(char* str) {
  unicode_len_t res;
  res.num_chars = 0;
  res.num_bytes = 0;
  if (!str) {
    return res;
  }
  // Последний нулевой символ
  res.num_chars = 1;
  res.num_bytes = 1;
  while (*str) {
    if ((*str | 0x7f) == 0x7f) { // 0x7f = 0b01111111
      res.num_chars++;
      res.num_bytes++;
      str++;
    } else if ((*str & 0xf0) == 0xf0) { // 0xf0 = 0b11110000
      res.num_chars++;
      res.num_bytes += 4;
      str += 4;
    } else if ((*str & 0xe0) == 0xe0) { // 0xe0 = 0b11100000
      res.num_chars++;
      res.num_bytes += 3;
      str += 3;
    } else if ((*str & 0xc0) == 0xc0) { // 0xc0 = 0b11000000
      res.num_chars++;
      res.num_bytes += 2;
      str += 2;
```

```
    } else {
      fprintf(stderr, "UTF-8 string is not valid!\n");
      exit(1);
    }
  }
  return res;
}

unicode_len_t strlen_u16(char16_t* str) {
  unicode_len_t res;
  res.num_chars = 0;
  res.num_bytes = 0;
  if (!str) {
    return res;
  }
  // Последний нулевой символ
  res.num_chars = 1;
  res.num_bytes = 2;
  while (*str) {
    if (*str < 0xdc00 || *str > 0xdfff) {
      res.num_chars++;
      res.num_bytes += 2;
      str++;
    } else {
      res.num_chars++;
      res.num_bytes += 4;
      str += 2;
    }
  }
  return res;
}

unicode_len_t strlen_u32(char32_t* str) {
  unicode_len_t res;
  res.num_chars = 0;
  res.num_bytes = 0;
  if (!str) {
    return res;
  }
  // Последний нулевой символ
  res.num_chars = 1;
  res.num_bytes = 4;
  while (*str) {
      res.num_chars++;
      res.num_bytes += 4;
      str++;
  }
  return res;
}
```

Остается только функция `main`. Чтобы проверить работу приведенных выше функций, в ней объявляются разные строки с текстом на английском, персидском и инопланетном языках (листинг 12.9).

Листинг 12.9. Главная функция в примере 12.3 (ExtremeC_examples_chapter12_3.c)

```
int main(int argc, char** argv) {

  char ascii_string[32] = "Hello World!";

  char utf8_string[32] = u8"Hello World!";
  char utf8_string_2[32] = u8"ایند دورد!";

  char16_t utf16_string[32] = u"Hello World!";
  char16_t utf16_string_2[32] = u"ایند دورد!";
  char16_t utf16_string_3[32] = u"ჰჰჰ!";

  char32_t utf32_string[32] = U"Hello World!";
  char32_t utf32_string_2[32] = U"ایند دورد!";
  char32_t utf32_string_3[32] = U"ჰჰჰ!";

  unicode_len_t len = strlen_ascii(ascii_string);
  printf("Length of ASCII string:\t\t\t %ld chars, %ld bytes\n\n",
      len.num_chars, len.num_bytes);

  len = strlen_u8(utf8_string);
  printf("Length of UTF-8 english string:\t\t %ld chars, %ld bytes\n",
      len.num_chars, len.num_bytes);
  len = strlen_u16(utf16_string);
  printf("Length of UTF-16 english string:\t %ld chars, %ld bytes\n",
      len.num_chars, len.num_bytes);
  len = strlen_u32(utf32_string);
  printf("Length of UTF-32 english string:\t %ld chars, %ld bytes\n\n",
      len.num_chars, len.num_bytes);

  len = strlen_u8(utf8_string_2);
  printf("Length of UTF-8 persian string:\t\t %ld chars, %ld bytes\n",
      len.num_chars, len.num_bytes);
  len = strlen_u16(utf16_string_2);
  printf("Length of UTF-16 persian string:\t %ld chars, %ld bytes\n",
      len.num_chars, len.num_bytes);
  len = strlen_u32(utf32_string_2);
  printf("Length of UTF-32 persian string:\t %ld chars, %ld bytes\n\n",
      len.num_chars, len.num_bytes);

  len = strlen_u16(utf16_string_3);
  printf("Length of UTF-16 alien string:\t\t %ld chars, %ld bytes\n",
      len.num_chars, len.num_bytes);
  len = strlen_u32(utf32_string_3);
  printf("Length of UTF-32 alien string:\t\t %ld chars, %ld bytes\n",
      len.num_chars, len.num_bytes);

  return 0;
}
```

Теперь мы должны скомпилировать данный пример. Заметьте, что это можно сделать только с помощью компилятора, совместимого с C11. Вы можете попробовать

более старые компиляторы и посмотреть на ошибки, которые получатся в результате. Команды, показанные в терминале 12.3, компилируют и запускают нашу программу.

Терминал 12.3. Компиляция и запуск примера 12.3

```
$ gcc ExtremeC_examples_chapter12_3.c -std=c11 -o ex12_3.out
$ ./ex12_3.out
Length of ASCII string:            13 chars, 13 bytes

Length of UTF-8 english string:      13 chars, 13 bytes
Length of UTF-16 english string:     13 chars, 26 bytes
Length of UTF-32 english string:     13 chars, 52 bytes

Length of UTF-8 persian string:      11 chars, 19 bytes
Length of UTF-16 persian string:     11 chars, 22 bytes
Length of UTF-32 persian string:     11 chars, 44 bytes

Length of UTF-16 alien string:        5 chars, 14 bytes
Length of UTF-32 alien string:        5 chars, 20 bytes
$
```

Как видите, кодирование и хранение одной и той же строки с одним и тем же количеством символов требует разного количества байтов. Меньше всего места занимает UTF-8, особенно если существенная часть текста состоит из символов ASCII, просто потому, что большинство из этих символов занимают всего один байт.

Но когда вы имеете дело с символами не из латинского алфавита, которые, например, используются в азиатских языках, UTF-16 показывает лучший баланс между длиной строки и количеством примененных байтов, поскольку большинство символов будут двухбайтными.

Кодировка UTF-32 используется редко, но может пригодиться в системах, в которых предпочтителен *код для вывода* символов фиксированной длины; это касается систем с низкой вычислительной мощью или механизмами, рассчитанными на параллельную обработку. Следовательно, UTF-32 можно задействовать в качестве ключей для привязки символов к любого рода данным. Иными словами, с помощью этой кодировки можно строить индексы для быстрого поиска информации.

Анонимные структуры и анонимные объединения

Анонимные структуры и анонимные объединения — определения типов без имени; обычно они используются в качестве вложенных типов. Проще объяснить это на примере. В листинге 12.10 вы можете видеть тип, который содержит как анонимную структуру, так и анонимное объединение.

Листинг 12.10. Пример анонимной структуры и анонимного объединения

```
typedef struct {
  union {
    struct {
      int x;
      int y;
    };
    int data[2];
  };
} point_t;
```

Этот тип использует одну и ту же память для анонимной структуры и поля с массивом байтов `data`. Его применение демонстрируется в листинге 12.11.

Листинг 12.11. Главная функция с использованием анонимной структуры и анонимного объединения (ExtremeC_examples_chapter12_4.c)

```
#include <stdio.h>

typedef struct {
  union {
    struct {
      int x;
      int y;
    };
    int data[2];
  };
} point_t;

int main(int argc, char** argv) {
  point_t p;
  p.x = 10;
  p.data[1] = -5;
  printf("Point (%d, %d) using anonymous structure.\n", p.x, p.y);
  printf("Point (%d, %d) using anonymous byte array.\n",
      p.data[0], p.data[1]);
  return 0;
}
```

В этом примере мы создаем анонимное объединение с анонимной структурой внутри. Таким образом, одна и та же область памяти используется для хранения экземпляра анонимной структуры и двухэлементного целочисленного массива. В терминале 12.4 показан вывод данной программы.

Терминал 12.4. Компиляция и запуск примера 12.4

```
$ gcc ExtremeC_examples_chapter12_4.c -std=c11 -o ex12_4.out
$ ./ex12_4.out
Point (10, -5) using anonymous structure.
Point (10, -5) using anonymous byte array.
$
```

Как видите, любые изменения в двухэлементном целочисленном массиве отражаются на структурной переменной и наоборот.

Многопоточность

Поддержка многопоточности в C существует уже давно в виде поточных функций POSIX или библиотеки `pthreads`. Эта тема подробно рассматривается в главах 15 и 16.

Поточная библиотека POSIX, как можно понять по названию, доступна только в POSIX-совместимых системах, таких как Linux и других Unix-подобных ОС. Поэтому если вы имеете дело с ОС, которые несовместимы с POSIX (например, с Microsoft Windows), то вам следует использовать библиотеку, предоставляемую операционной системой. В рамках C11 доступна стандартная библиотека для работы с потоками, подходящая для всех систем, поддерживающих стандарт C, независимо от их совместимости с POSIX. Это самое большое изменение, которое можно наблюдать в C11.

К сожалению, данная возможность C11 не реализована в Linux и macOS. Поэтому на момент написания книги мы не можем показать вам рабочие примеры.

Немного о C18

Как уже упоминалось в предыдущих разделах, стандарт C18 содержит все исправления, внесенные в C11, но в нем нет никаких нововведений. По ссылке, приведенной ниже, можно ознакомиться с заявками, созданными для C11, и их обсуждением: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2244.htm.

Резюме

В этой главе мы прошлись по C11, C18 и самым последним стандартам языка C, исследовав различные нововведения в C11. Среди самых важных возможностей, появившихся в современной версии C, можно выделить поддержку Unicode, анонимных структур и объединений, а также новую стандартную библиотеку для работы с потоками (хотя последняя все еще недоступна в актуальных версиях компиляторов и платформ). Мы с нетерпением ждем выхода будущих стандартов C.

В следующей главе мы начнем обсуждение конкурентности и теории, лежащей в основе конкурентных систем. Это первая из шести глав, в которых будут рассмотрены такие темы, как многопоточность и многопроцессность, понимание которых поможет вам писать конкурентный код.

13 Конкурентность

В следующих двух главах мы обсудим *конкурентность* и теоретические основы, необходимые для разработки конкурентных программ не только на C, но и на других языках. Поэтому обе главы не будут содержать никакого кода на C. Вместо этого для представления конкурентных систем и характерных им свойств будет использоваться псевдокод.

Тема конкурентности ввиду ее обширности была разделена на две главы. Здесь мы поговорим об основных ее концепциях, а в главе 14 речь пойдет о сопутствующих проблемах и механизмах *синхронизации*, с помощью которых эти проблемы решаются в конкурентных программах. Общая цель обеих глав состоит в предоставлении теоретических знаний, достаточных для того, чтобы приступить к дальнейшему обсуждению многопоточности и многопроцессности.

Базовый материал, содержащийся в данной главе, также будет полезен при работе с *поточной библиотекой POSIX*.

Для начала мы попытаемся ответить на следующие вопросы:

- чем параллельные системы отличаются от конкурентных;
- когда нам нужна конкурентность;
- что такое *планировщик заданий* и какие алгоритмы планирования нашли широкое применение;
- как выполняется конкурентная программа и что такое чередование;
- что такое разделяемое состояние и как к нему могут обращаться различные задания.

Начнем наше обсуждение со знакомства с концепцией конкурентности и попробуем понять, какое значение она имеет для нас.

Введение в конкурентность

Конкурентность означает, что несколько участков логики программы выполняются одновременно. Многие современные программные системы конкурентны, поскольку программам необходимо совершать сразу несколько разных действий.

Таким образом, конкурентность в той или иной степени присутствует в любом современном ПО.

Можно сказать, это мощный инструмент, позволяющий писать код, который выполняет несколько заданий одновременно, и его поддержка обычно находится в сердце операционной системы — в ядре.

Существует множество примеров того, как обычной программе приходится выполнять сразу несколько действий. Например, вы можете перемещаться по вебстраницам, загружая при этом какие-то файлы. В данном случае задания выполняются конкурентно в контексте процесса браузера. Еще одна распространенная ситуация — *потоковая загрузка видео*, когда вы смотрите что-то на YouTube. Пока вы просматриваете уже загруженные куски видеоролика, проигрыватель занимается получением следующих.

Даже простой текстовый процессор выполняет в фоне несколько заданий. Когда я пишу эту главу в Microsoft Word, мне помогают механизмы проверки правописания и форматирования, выполняющиеся в параллель. Если вы читаете данную книгу в приложении Kindle на iPad, то какие, по вашему мнению, задания могут выполняться в ходе его работы?

Одновременное выполнение нескольких программ звучит чудесно, но, как часто бывает с технологиями, конкурентность несет с собой не только преимущества, но и определенные проблемы. Действительно, мало какая технология в истории компьютерных наук вызывала столько головной боли! Эти проблемы, о которых мы поговорим позже, могут оставаться незамеченными на протяжении долгого времени после выпуска (вплоть до нескольких месяцев), поскольку их обычно сложно выявлять, воспроизводить и устранять.

В начале этого раздела мы определили конкурентность как одновременное (или конкурентное) выполнение нескольких заданий. Подобное описание подразумевает, что задания работают параллельно, но все, строго говоря, не так. Это упрощение, к тому же не совсем точное, поскольку *конкурентная* и *параллельная* работа — разные вещи, и мы еще не объяснили, чем они отличаются. Две конкурентные программы отличаются от двух параллельных, и одна из целей данной главы — пролить свет на их отличия и предоставить определения, которые используются в официальной литературе в этой области.

В следующем разделе я объясню некоторые базовые концепции, относящиеся к конкурентности, такие как *задания*, *планирование*, *чередование*, *состояние* и *разделяемое состояние*; вы будете регулярно встречать данные термины на страницах книги. Вдобавок стоит отметить: большинство из этих концепций являются абстрактными и применимы к любой конкурентной системе, а не только к C.

Чтобы понять разницу между конкурентностью и параллелизмом, проведу краткий обзор параллельных систем.

Нужно сказать, что в текущей главе я буду придерживаться простых определений. Я всего лишь хочу дать вам общее представление о том, как работают конкурентные системы, поскольку более глубокие темы выходят за рамки этой книги.

Параллелизм

Параллелизм означает, что два задания выполняются одновременно, или *в параллель*. Именно словосочетание «в параллель» — ключевое отличие между параллелизмом и конкурентностью. Почему? Потому что параллельность подразумевает протекание двух событий в один и тот же момент. В случае с конкурентной системой это не так; прежде чем выполнять другое задание, она должна остановить текущее. Обратите внимание: в контексте современных конкурентных систем это определение может быть слишком простым и неполным, но чтобы понять общую идею, его должно быть достаточно.

Параллелизм регулярно встречается в повседневной жизни. Когда вы со своим другом одновременно выполняете разные задачи, это означает, что задачи выполняются в параллель. Для параллельного выполнения необходимо разделить и изолировать *вычислительные блоки*, каждый из которых назначается определенному заданию. Например, в компьютерной системе таким блоком выступает *ядро процессора*, которое может выполнять по одной инструкции за раз.

Остановитесь на минуту и подумайте о себе как об отдельно взятом читателе этой книги. Вы не можете читать две книги одновременно; чтобы взяться за одну книгу, вам пришлось бы отложить другую. Но скооперировавшись с другом, вы можете читать две книги параллельно.

Но что, если вам нужно прочитать три книги? Поскольку вы не владеете мастерством параллельного чтения, одному из вас пришлось бы отложить текущую книгу и взяться за новую. Проще говоря, вам или вашему другу нужно выделить достаточное количество времени для прочтения всех трех книг.

Для параллельного выполнения двух задач компьютерной системе нужно по меньшей мере два отдельных и независимых вычислительных блока. Современные процессоры содержат внутри несколько *ядер*, которые являются этими самыми блоками. Например, четырехъядерный процессор имеет четыре вычислительных блока; следовательно, может выполнять сразу четыре параллельных задания. Для простоты предположим, что в этой главе мы имеем дело с воображаемым процессором, имеющим всего одно ядро, вследствие чего лишен возможности параллельного выполнения. Мы еще поговорим о многоядерных процессорах в соответствующих разделах.

Представьте, что у вас есть два ноутбука с нашим воображаемым процессором внутри: один воспроизводит музыку, а другой ищет решение дифференциального уравнения. Оба они работают параллельно, но если вы решите выполнить эти

задачи на одном из них, используя один процессор с одним ядром, то это *нельзя* будет сделать параллельно. На самом деле выполнение будет конкурентным.

Параллелизм возможен только в задачах, которые можно распараллелить. Это означает, что алгоритм можно разделить на части и запустить на разных вычислительных блоках. Однако на сегодня большинство алгоритмов, которые мы пишем, по своей природе *последовательны*, а не параллельны. Даже в многопоточности у каждого потока есть ряд последовательных инструкций, не подлежащих делению на параллельные *потоки выполнения*.

Иными словами, операционной системе сложно автоматически разделить последовательный алгоритм на параллельные потоки выполнения; следовательно, этим должен заниматься программист. Поэтому, даже имея многоядерный процессор, вы все равно должны назначить каждый поток выполнения определенному ядру. И если одно ядро получит несколько потоков, то их нельзя будет выполнить параллельно. В результате вы сможете наблюдать конкурентное поведение.

Если коротко, то наличие двух потоков, назначенных двум разным ядрам, несомненно, может привести к созданию двух параллельных потоков. Но если назначить их одному ядру, то получатся два конкурентных потока. В многоядерном процессоре мы фактически наблюдаем смешанное поведение: как параллелизм между ядрами, так и конкурентность в рамках одного ядра.

Несмотря на простое определение и бесчисленные примеры из повседневной жизни, параллелизм — сложный аспект компьютерной архитектуры. На самом деле это отдельная дисциплина, с собственными теориями, книгами и учебной литературой. Проектирование операционной системы, которая может разделить последовательный алгоритм на какие-то параллельные потоки выполнения, — активное направление исследований, и текущие ОС не умеют этого делать.

Как уже отмечалось, цель текущей главы заключается не в сколь-нибудь глубоком обсуждении параллелизма, а лишь в том, чтобы дать вам базовое определение этой концепции. Поскольку любые дальнейшие подробности выходят за рамки нашей книги, мы можем переходить к понятию конкурентности.

Для начала поговорим о конкурентных системах и посмотрим, что они собой представляют в сравнении с параллельными.

Конкурентность

Вы, наверное, уже слышали о *многозадачности* — что ж, конкурентность зиждется на той же идее. Если ваша система выполняет сразу несколько задач, то это вовсе не означает их параллельную работу. В данном процессе может участвовать *планировщик заданий*; он просто очень быстро переключается между разными за-

даниями, выполняя некую крошечную часть каждого из них за довольно короткий промежуток времени.

Это, несомненно, происходит на компьютере с одним вычислительным блоком. В оставшейся части данного раздела предполагается, что мы имеем дело именно с таким компьютером.

Если планировщик достаточно *быстрый* и *беспристрастный*, то вы не заметите *переключения* между заданиями; вам будет казаться, что они работают параллельно. В этом вся магия конкурентности, и именно поэтому она применяется в большинстве широко известных операционных систем, включая Linux, macOS и Microsoft Windows.

Конкурентность можно считать имитацией параллельного выполнения заданий на одном вычислительном блоке. На самом деле это своего рода искусственный параллелизм. Для старых систем с одним одноядерным процессором это было большим шагом вперед, который позволил людям использовать одно ядро в многозадачной манере.

К слову, *Multics* была одной из первых операционных систем с поддержкой многозадачности и управления одновременными процессами. Как вы можете помнить из главы 10, проект Unix был основан на идеях, почерпнутых из Multics.

Как уже объяснялось ранее, почти все ОС могут выполнять конкурентные задания, используя многозадачность. Особенно это относится к POSIX-совместимым системам, поскольку эта возможность явно обозначена в стандарте POSIX.

Планировщик заданий

Как уже отмечалось, у всех многозадачных операционных систем в ядре должен быть *планировщик заданий*, или просто *планировщик*. В этом разделе вы увидите, как работает данный компонент и какую роль играет в беспрепятственном выполнении конкурентных заданий.

Ниже перечислены несколько фактов о планировщике.

- У планировщика есть *очередь* заданий, ждущих, когда их выполнят. *Задание* — просто наборы операций, которые должны быть выполнены в отдельных потоках.
- Очередь обычно поддерживает приоритеты. Высокоприоритетные задания выполняются в первую очередь.
- Планировщик управляет вычислительными ресурсами и разделяет их между всеми заданиями. Когда вычислительный блок свободен (его не использует ни одно задание), планировщик должен выбрать из очереди следующее задание

и назначить его этому блоку. Когда задание заканчивает работу, оно освобождает вычислительный блок и снова делает его доступным, после чего планировщик выбирает очередное задание. Описанное продолжается в непрерывном цикле и называется *планированием заданий*, и это единственная обязанность планировщика.

- Существует много разных *алгоритмов планирования*, которые может использовать планировщик, но все они должны соответствовать определенным требованиям. Например, им следует быть *беспристрастными*, и ни одно задание в очереди не должно оставаться без внимания на протяжении долгого времени.
- В зависимости от выбранной *стратегии* планировщик должен либо выделить определенный временной интервал, на протяжении которого задание может использовать вычислительный блок, либо подождать, пока блок не будет освобожден.
- Если стратегия планирования *вытесняющая*, то у планировщика должна быть возможность принудительно освобождать ядро процессора, чтобы отдать его следующему заданию. Это называется *вытесняющим планированием*. Есть еще одна стратегия, в которой задание добровольно освобождает процессор, — так называемое *кооперативное планирование*.
- Алгоритмы вытесняющего планирования пытаются разделять *временные интервалы* равномерно и беспристрастно между разными заданиями. Задания с повышенным приоритетом могут выбираться чаще других или даже получать более длинные интервалы в зависимости от реализации планировщика.

Задание — абстрактная концепция общего характера, которая может означать любой набор инструкций, предназначенный для выполнения в конкурентной системе, не обязательно компьютерной. О каких именно некомпьютерных системах идет речь, мы поговорим чуть позже. Точно так же процессор не единственный ресурс, который можно разделять между заданиями. Люди всегда занимались планированием и приоритизацией заданий, непригодных для одновременного выполнения. В следующих нескольких абзацах мы рассмотрим такие ситуации в качестве наглядных примеров планирования.

Перенесемся в начало XX века и представим, что на улице стоит всего одна телефонная будка, к которой растянулась очередь из десяти человек. В этом случае очередь должна соблюдать алгоритм планирования, чтобы время пользования телефоном справедливо разделялось между всеми ее участниками.

Прежде всего мы должны сформировать очередь. Первое, что приходит на ум цивилизованному человеку в подобной ситуации, — подождать своей очереди. Но этого недостаточно; организация такого подхода требует неких правил. Вы не можете разговаривать по телефону сколько вздумается, когда за вами еще девять человек. Через какое-то время вы должны покинуть будку и уступить место следующему человеку в очереди.

В редких случаях, когда время вышло, но человек не успел завершить разговор, он покидает будку и становится в конец очереди. Затем он должен подождать, чтобы иметь возможность вернуться к тому, на чем его прервали. Таким образом, все десять человек станут по очереди заходить в телефонную будку, пока не наговорятся вдосталь.

Это всего лишь иллюстрация. Мы ежедневно встречаем примеры разделения ресурсов между несколькими потребителями, и люди придумали много способов, как делать это по справедливости, — насколько позволяет человеческая природа! В следующем разделе мы поговорим о планировании в контексте компьютерных систем.

Процессы и потоки

В этой книге нас в основном интересует планирование в компьютерных системах. В ОС роль заданий играют либо *процессы*, либо *потоки*. Мы рассмотрим их и разницу между ними в следующих главах, но пока вам следует знать, что большинство операционных систем обращаются с ними практически одинаковым образом: как с некими заданиями, которые необходимо выполнять конкурентно.

Операционная система должна использовать планировщик в целях разделения ядер процессора между множеством заданий, будь то процессы или потоки, которым для выполнения нужно процессорное время. В момент создания новый процесс или поток поступает в очередь планировщика в качестве нового задания и, прежде чем начинать работу, ждет, когда ему выделят процессорное ядро.

Если у вас установлен *вытесняющий* планировщик *(с разделением времени)* и задание не успевает закончить работу в отведенное ему время, то он принудительно освобождает ядро процессора, а задание опять попадает в очередь — точно так же, как в примере с телефонной будкой.

В данном случае задание будет находиться в очереди, пока снова не получит доступ к ядру процессора, после чего сможет продолжить работу. Если ему не удается завершить свою логику со второй попытки, то процедура повторяется, пока задание не закончится.

Ситуация, когда вытесняющий планировщик останавливает процесс посреди выполнения и запускает вместо него другой, называется *переключением контекста*. Чем быстрее переключается контекст, тем сильнее ощущение параллельной работы заданий. Интересно, что большинство ОС на сегодня используют вытесняющий планировщик, и именно на нем мы сосредоточим свое внимание в оставшейся части этой главы.

С этого момента мы будем подразумевать, что все упоминаемые мной планировщики являются вытесняющими. Иные ситуации будут оговорены отдельно.

В ходе выполнения задание может испытывать сотни и даже тысячи переключений контекста. Однако эти переключения имеют одно причудливое и уникальное свойство — они *непредсказуемы*. Иными словами, мы не знаем, когда или даже на какой инструкции контекст будет переключен. Даже в двух максимально приближенных прогонах программы на одной и той же платформе переключения контекста будут отличаться.

Значимость данного факта и его последствий сложно переоценить; переключение контекста невозможно предсказать! Чуть позже вы увидите примеры, которые показывают, что из этого следует.

Переключения контекста крайне непредсказуемы, поэтому лучше всего предполагать, что они с одинаковой вероятностью могут произойти на любой инструкции. То есть вы должны быть готовы к тому, что при каждом выполнении программы все инструкции подвержены переключению контекста. Это просто означает возможный разрыв между выполнением двух соседних инструкций.

Учитывая все вышесказанное, пойдем дальше и рассмотрим единственные аспекты конкурентной среды, в которых можно быть уверенными.

Порядок выполнения инструкций

В предыдущем разделе мы определились с тем, что переключения контекста непредсказуемы; мы не знаем, в какие моменты они могут возникнуть в наших программах. Несмотря на это, об инструкциях, выполняющихся конкурентно, можно сказать кое-что определенное.

Рассмотрим простой пример. Мы будем исходить из того, что у нас есть задание с пятью инструкциями, наподобие показанного в листинге 13.1. Обратите внимание: эти инструкции абстрактны и никак не связаны с машинными инструкциями или инструкциями языка C.

Листинг 13.1. Простое задание с пятью инструкциями

```
Task P {
    1. num = 5
    2. num++
    3. num = num - 2
    4. x = 10
    5. num = num + x
}
```

Как видите, инструкции пронумерованы; это значит, они *должны* быть выполнены в заданном порядке, чтобы соответствовать назначению задания. Мы в этом уверены. Говоря техническим языком, каждая следующая инструкция может начинать работу только после завершения предыдущей. Инструкция `num++` должна быть

выполнена перед `num = num – 2`, и это ограничение должно соблюдаться независимо от того, как переключается контекст.

Обратите внимание: мы по-прежнему не знаем, когда будут происходить переключения контекста; необходимо помнить, что это может случиться между любыми двумя инструкциями.

В листинге 13.2 представлены два возможных варианта выполнения нашего задания с разными переключениями контекста.

Листинг 13.2. Один потенциальный прогон задания с переключениями контекста

```
Run 1:
  1. num = 5
  2. num++
>>>>> Context Switch <<<<<
  3. num = num – 2
  4. x = 10
>>>>> Context Switch <<<<<
  5. num = num + x
```

Второй прогон выглядит так (листинг 13.3).

Листинг 13.3. Другой потенциальный прогон задания с переключениями контекста

```
Run 2:
  num = 5
  >> Context Switch <<
  num++
  num = num – 2
  >> Context Switch <<
  x = 10
  >> Context Switch <<
  num = num + x
```

В листинге 13.2 можно видеть, что количество переключений контекста и места, где они происходят, могут меняться при каждом выполнении. Но, как отмечалось ранее, существует определенность относительно порядка выполнения инструкций.

Именно по этой причине общее поведение отдельно взятого задания может быть детерминированным. Какими бы ни были переключения контекста в разных прогонах, *общее состояние* задания остается неизменным. Под общим состоянием имеется в виду набор переменных и их значений после выполнения последней инструкции в задании. Если взять наш пример, то, независимо от переключений контекста, в итоговом состоянии переменные `num` и `x` будут равны соответственно `14` и `10`.

Тот факт, что общее состояние отдельного задания не меняется в разных прогонах, подталкивает нас к выводу: благодаря необходимости соблюдать порядок выполнения инструкций конкурентность не может повлиять на общее состояние задания. Но с этим выводом нужно быть осторожными.

Предположим, у нас есть система конкурентных заданий, каждое из которых может читать и изменять некий *разделяемый ресурс* (скажем, переменную). Если все задания занимаются исключительно чтением и ни одно из них не меняет значение переменной, то можно утверждать: независимо от того, как переключается контекст и сколько раз выполняются задания, результаты всегда будут одинаковыми. То же самое, к слову, относится к системе конкурентных заданий, у которых вообще нет разделяемой переменной.

Тем не менее, если хотя бы одно задание изменит разделяемую переменную, переключения контекста, производимые планировщиком, повлияют на общее состояние всех заданий. Это значит, результаты могут меняться от одного прогона к другому! Следовательно, во избежание нежелательных последствий необходимо использовать подходящий управляющий механизм. А все из-за того, что переключения контекста нельзя предсказать и *промежуточные состояния* заданий могут варьироваться между разными прогонами. Промежуточное состояние, в отличие от итогового, — набор переменных и их значений на определенной инструкции. Каждое задание имеет всего одно итоговое состояние, которое определяется после завершения работы, однако промежуточных состояний может быть много, и все они относятся к переменным и их значениям после выполнения определенной инструкции.

Если подытожить, то в конкурентной системе с несколькими заданиями, каждое из которых может читать и записывать разделяемый ресурс, разные прогоны будут давать разные результаты. Поэтому необходимо применять подходящие методы *синхронизации*, чтобы нивелировать последствия от переключения контекста и получать одинаковые детерминированные результаты при каждом выполнении.

Итак, мы получили общее представление о конкурентности, которая является основной темой данной главы. Понятия, описанные в этом разделе, послужат фундаментом для понимания многих других тем и на последующих страницах будут встречаться вам снова и снова.

Мы уже упоминали о том, что конкурентность может быть проблематичной и усложнять нам жизнь. Поэтому вы можете спросить, в каких ситуациях она может пригодиться? Ответ на этот вопрос будет дан в следующем разделе.

Когда следует использовать конкурентность

Учитывая все, о чем мы говорили выше, работать с одним заданием не так трудно, как с несколькими, которые выполняют одну и ту же работу конкурентно. Это правда; если вы можете написать программу, которая приемлемо работает, не прибегая к конкурентности, то я настоятельно рекомендую именно этот вариант. Существует несколько методов определения того, нужна ли конкурентность.

В данном разделе мы пройдемся по этим общим методам и посмотрим, как они приводят к разбиению программы на конкурентные потоки.

Любая программа, независимо от языка программирования, представляет собой набор инструкций, которые должные выполняться последовательно. Иными словами, следующая инструкция может быть выполнена только после предыдущей. Это называется *последовательным выполнением*. Неважно, сколько времени занимает выполнение текущей инструкции; пока она не завершится, следующая должна ждать своей очереди. Обычно говорят, что текущая инструкция *блокирует* следующую; в таком случае текущую инструкцию иногда называют *блокирующей*.

В любой программе все инструкции — блокирующие, а каждый поток выполнения — последователен. Можно лишь сказать, что программа будет работать быстро, если каждая инструкция блокирует выполнение на относительно короткий промежуток времени в пределах нескольких миллисекунд. Но что, если этот промежуток затягивается (например, до 2 секунд или 2000 миллисекунд) или его продолжительность нельзя предсказать? Чтобы определить, требуется ли программе конкурентность, можно использовать два существующих метода.

Если говорить более подробно, то каждой блокирующей инструкции требуется какое-то время на завершение работы. С нашей точки зрения, лучше всего, когда заданная инструкция завершается относительно быстро, не задерживая поток выполнения. Но нам не всегда так везет.

В некоторых ситуациях невозможно определить, сколько времени понадобится блокирующей инструкции. Обычно это происходит, когда она ожидает какого-то события или доступа к неким данным.

Рассмотрим еще один пример. Представьте, что у вас есть серверная программа, обслуживающая ряд клиентских приложений. Она содержит инструкцию, которая ждет подключения со стороны клиента. Серверная программа не может предугадать, когда к ней подключатся. Поэтому следующая инструкция не может быть выполнена на серверной стороне, ведь мы не знаем, когда завершится текущая. Это полностью зависит от того, когда к нам решит подключиться новый клиент.

Более простым примером будет чтение строки, которую вводит пользователь. Программа не может знать, когда пользователь выполнит ввод, поэтому следующие инструкции не могут быть выполнены. Это *первая ситуация*, в которой необходима конкурентная система заданий.

В целом речь идет о ситуации, когда у вас есть инструкция, которая может заблокировать поток выполнения на неопределенный промежуток времени. В этом случае программу следует разделить на два отдельных задания или потока выполнения. Это делается, если вам нужно выполнить следующие инструкции, не дожидаясь завершения текущей. Что еще более важно в данном контексте, мы исходим из того, что результат выполнения текущей инструкции не влияет на работу следующих.

Разделив наш изначальный поток выполнения на две задачи, мы можем сделать так, чтобы одна задача ждала завершения блокирующей инструкции, а другая продолжала выполнять следующие инструкции, которые блокировались в нашей предыдущей неконкурентной конфигурации.

В следующем примере показано, как в этой первой ситуации можно создать систему с конкурентными заданиями. Для представления инструкций в каждом задании будет использоваться псевдокод.

Для понимания следующего примера не требуется никаких предварительных знаний о компьютерных сетях.

Пример, на котором мы сконцентрируемся, посвящен серверной программе, которая имеет три обязанности:

- получает от клиента два числа и возвращает ему их сумму;
- регулярно записывает в файл количество обслуженных клиентов, независимо от того, занимается она обслуживанием клиента в этот момент или нет;
- вдобавок должна уметь обслуживать сразу несколько клиентов.

Прежде чем переходить к обсуждению итоговой конкурентной системы, удовлетворяющей перечисленным выше требованиям, представим, что можем использовать здесь лишь одно задание (или один поток). Это поможет продемонстрировать, что такую программу нельзя реализовать с помощью лишь одного задания. В листинге 13.4 показана однозадачная версия нашей серверной программы.

Листинг 13.4. Серверная программа, работающая в рамках одного задания

```
Calculator Server {
    Task T1 {
        1. N = 0
        2. Prepare Server
        3. Do Forever {
        4.     Wait for a client C
        5.     N = N + 1
        6.     Read the first number from C and store it in X
        7.     Read the second number from C and store it in Y
        8.     Z = X + Y
        9.     Write Z to C
        10.    Close the connection to C
        11.    Write N to file
           }
    }
}
```

Как видите, наш единственный поток ждет, пока к нему по сети не подключится клиент. Затем он принимает от клиента два числа, считает их сумму и возвращает результат обратно клиенту. В конце он закрывает соединение, записывает количество обслуженных клиентов в файл и продолжает ждать, пока к нему не присоединится следующий клиент. Чуть ниже я покажу, что этот код не может удовлетворить всем требованиям, перечисленным выше.

Данный псевдокод состоит всего из одного задания, `T1`. Он содержит 12 строчек с инструкциями, которые, как уже упоминалось ранее, выполняются последовательно. То есть все они блокирующие. Так о чем же нам говорит этот код? Пройдемся по порядку.

- *Первая* инструкция, `N = 0`, довольно простая и завершается очень быстро.
- *Вторая* инструкция, `Prepare Server`, должна завершиться за разумный период времени и не будет блокировать выполнение серверной программы.
- *Третья* инструкция просто запускает главный цикл и, как только мы в него войдем, должна быстро завершиться.
- *Четвертая* команда, `Wait for a client C`, является блокирующей инструкцией с неизвестным временем работы. Таким образом, команды *5, 6* и остальные выполнены не будут. Похоже, они должны подождать, пока не подключится новый клиент, и только после этого придет их очередь.

Как уже отмечалось ранее, инструкции с *5-й* по *10-ю* обязаны ждать подключения нового клиента. Иными словами, они зависят от вывода инструкции *4* и не могут быть выполнены, пока клиент не подключится. Однако инструкция `11`, `Write N to file`, должна выполняться независимо от того, есть у нас клиент или нет. Это продиктовано вторым требованием, которое мы определили для данного примера. В текущей конфигурации мы записываем `N` в файл, только если у нас есть клиент, что противоречит нашему изначальному требованию, согласно которому запись в файл происходит *вне зависимости* от наличия клиента.

Поток выполнения данного кода имеет еще одну проблему: его потенциально могут заблокировать инструкции *6* и *7*. Они ждут, когда клиент введет два числа, и поэтому мы не можем предсказать, когда они завершат свою работу. Это не дает программе продолжить выполнение.

Более того, эти инструкции блокируют прием данных от других потенциальных клиентов. Причина в том, что в случае затянувшейся работы инструкций *6* и *7* мы больше не вернемся к команде *4*. Следовательно, серверная программа не может обслуживать сразу несколько клиентов; и это опять не соответствует заданным целям.

Чтобы решить перечисленные выше проблемы, нам нужно разбить наше единое задание на три конкурентных потока, которые в совокупности будут удовлетворять требованиям, стоящим перед серверной программой.

В листинге 13.5 показан псевдокод с тремя потоками выполнения, `T1`, `T2` и `T3`, которые позволяют достичь поставленных целей за счет конкурентного решения.

Листинг 13.5. Серверная программа с тремя конкурентными заданиями

```
Calculator Server {
    Shared Variable: N

    Task T1 {
        1. N = 0
        2. Prepare Server
        3. Spawn task T2
        4. Do Forever {
        5. Write N to file
        6. Wait for 30 seconds
          }
    }

    Task T2 {
        1. Do Forever {
        2. Wait for a client C
        3. N = N + 1
        4. Spawn task T3 for C
          }
    }

    Task T3 {
        1. Read first number from C and store it in X
        2. Read first number from C and store it in Y
        3. Z = X + Y
        4. Write Z to C
        5. Close the connection to C
    }
}
```

Программа начинается с выполнения задания `T1`. Мы считаем его главным, поскольку с него все начинается. Заметьте, что в любой программе всегда есть одно задание, которое инициирует все остальные — либо напрямую, либо опосредованно.

В этом листинге мы видим два других задания, которые инициируются из главного, `T1`. У нас также есть разделяемая переменная, `N`, хранящая количество обслуженных клиентов; ее могут читать и изменять все задания.

Программа начинается с первой инструкции в задании `T1`, которая присваивает переменной `N` ноль. Вторая инструкция подготавливает сервер. В ходе ее выполнения нужно проделать некоторые предварительные шаги, чтобы серверная программа могла принимать входящие соединения. Обратите внимание: на этот момент вместе с `T1` не выполняется ни одно конкурентное задание.

Третья инструкция в T1 создает новый *экземпляр* задания T2. Создание нового задания обычно происходит мгновенно. Поэтому сразу после данной инструкции T1 входит в бесконечный цикл и продолжает записывать в файл значение, разделяемый переменной каждые 30 секунд. Это было нашим первым требованием, которое мы теперь удовлетворили. Таким образом, задание T1 периодически сохраняет значение N в файл, пока не завершится программа, без каких-либо прерываний или блокировок со стороны других инструкций.

Теперь поговорим о только что созданном задании, T2. Его единственное назначение — немедленно принимать входящие соединения при поступлении клиентских запросов. Вдобавок стоит помнить: все инструкции в задании T2 выполняются в бесконечном цикле. Вторая его команда ожидает подключения нового клиента. Она блокирует выполнение остальных инструкций, но лишь тех, которые находятся в T2. Если бы мы создали два экземпляра этого задания вместо одного, то инструкции одного из них не блокировали бы инструкции другого.

Другие конкурентные задания (в данном случае только T1) продолжают выполнять свои инструкции без какой-либо блокировки. Это стало возможным благодаря конкурентности: пока одни задания заблокированы в ожидании определенного события, другие могут продолжать работать без всяких прерываний. Как уже говорилось ранее, такова суть одного важного принципа проектирования: *если у вас есть блокирующая операция, время выполнения которой либо слишком длинное, либо непредсказуемое, то вы должны разбить задание на два конкурентных потока*.

Теперь представим, что к нам подключается новый клиент. Этот процесс может заблокировать операция чтения, как мы уже видели в неконкурентной версии серверной программы в листинге 13.4 (см. выше). Отталкиваясь от изложенного выше принципа проектирования и учитывая то, что инструкции чтения блокирующие, мы должны разделить логику на два конкурентных потока выполнения. Вот почему мы создали задание T3.

Для взаимодействия с только что подключенным клиентом задание T2 создает новый экземпляр T3. Это происходит в инструкции *4* задания T2, которая, напомню, выглядит так (листинг 13.6).

Листинг 13.6. Инструкция 4 в задании T2

```
4.     Spawn task T3 for C
```

Необходимо отметить: прежде чем создавать новое задание, T2 инкрементирует значение разделяемой переменной N, тем самым сигнализируя об обслуживании нового клиента. Опять же показанная выше инструкция довольно быстрая и не блокирует принятие соединений от новых клиентов.

Когда завершится инструкция *4* в задании T2, цикл продолжится, пока не дойдет до инструкции *2*, которая ждет подключения следующего клиента. Заметьте, что в нашем псевдокоде у нас имеется по одному экземпляру T1 и T2, но T3 создается для каждого клиента.

Единственное назначение T3 — связаться с клиентом и прочитать входные числа. Дальше это задание вычисляет сумму и отправляет ее обратно клиенту. Как уже отмечалось ранее, блокирующие инструкции внутри T3 не могут заблокировать выполнение в других заданиях и их поведение влияет только на текущий экземпляр T3. У инструкций в одном экземпляре T3 нет возможности заблокировать работу другого экземпляра T3. Таким образом, благодаря конкурентности серверная программа удовлетворяет всем нашим требованиям.

Дальше может возникнуть вопрос: когда завершаются задания? Мы знаем, что задание заканчивает работу, выполнив все свои инструкции. Тем не менее если у нас есть бесконечный цикл, включающий в себя все инструкции, то задание не останавливается и время его жизни зависит от *родительского задания*, которое его породило. В будущих главах мы подробно обсудим это в контексте процессов и потоков. Что касается нашего примера, то в предыдущей конкурентной программе родительским заданием для всех экземпляров T3 является один и тот же экземпляр T2. Конкретный экземпляр T3 завершает работу либо при закрытии соединения с клиентом после передачи двух блокирующих инструкций чтения, либо когда завершается единственный экземпляр T2.

Редко, но иногда все же возникает ситуация, когда на выполнение всех операций чтения уходит слишком много времени (намеренно или случайно), а количество новых клиентов быстро растет, и в какой-то момент получается множество экземпляров задания T3, которые ждут от своих клиентов входные числа. Это привело бы к существенному потреблению ресурсов. Затем, по мере создания все новых и новых соединений, серверная программа либо будет принудительно завершена операционной системой, либо просто не сможет больше обслуживать никаких клиентов.

Описанная выше ситуация, когда серверная программа перестает принимать соединения от клиентов, называется *отказом в обслуживании* (denial of service, DoS). Системы с конкурентными заданиями должны уметь справляться с такими проблемами, чтобы обслуживание клиентов прекращалось в разумной манере.

В процессе DoS-атаки оказывается чрезмерная нагрузка на серверный компьютер, чтобы он вышел из строя и прекратил отвечать на запросы. DoS-атаки принадлежат к группе сетевых атак, которые пытаются блокировать определенные сервисы с целью сделать их недоступными для клиентов. В их число входит широкий спектр вредоносных методов, нацеленных на остановку обслуживания, включая эксплойты. Это даже может выражаться в флудинге сети в попытке вывести из строя ее инфраструктуру.

В предыдущем примере серверной программы мы описали ситуацию с блокирующей инструкцией, время завершения которой было неопределенным. Это было первым условием применения конкурентности. Есть еще одно подобное условие, имеющее небольшое отличие.

Если инструкции или группе инструкций требуется слишком много времени на выполнение, то мы можем поместить их в отдельное конкурентное задание, которое будет выполняться отдельно от главного. Отличие от первого случая в том, что мы можем себе представить (хоть и с малой степенью точности) время завершения; нам известно, что оно наступит нескоро.

Последнее, на что следует обратить внимание в предыдущем примере, — это разделяемая переменная *N* и тот факт, что одно из заданий (а именно, экземпляр `T2`) может менять ее значение. Исходя из того, о чем говорилось ранее, это может привести к появлению в нашей системе проблем с конкурентностью, поскольку у нее есть разделяемая переменная, доступная для изменения со стороны одного из заданий.

Необходимо отметить: решение, которое мы предложили для серверной программы, далеко не идеальное. В следующей главе вы познакомитесь с проблемами, относящимися к конкурентности, и увидите, что из-за разделяемой переменной `N` этот пример подвержен *состоянию гонки*. В результате для устранения проблем, связанных с конкурентностью, необходимо внедрить подходящий контрольный механизм.

В следующем, заключительном разделе этой главы мы поговорим о *состояниях*, разделяемых между конкурентными заданиями. Мы также познакомимся с концепцией *чередования* и важными последствиями, которые она имеет для конкурентных систем с изменяемым разделяемым состоянием.

Разделяемые состояния

В предыдущем разделе мы обсуждали ситуации, в которых необходима система конкурентных заданий. До этого я уже упоминал, что неопределенность переключения контекста во время выполнения ряда конкурентных потоков в сочетании с наличием изменяемого разделяемого состояния может привести к непредсказуемым итоговым результатам работы всех заданий. В данном разделе приводится пример, демонстрирующий проблемы, которые могут возникнуть в простой программе из-за такого недетерминированного поведения.

Мы продолжим наше обсуждение и рассмотрим *разделяемые состояния*, чтобы увидеть, какую роль они играют в получении упомянутых выше непредсказуемых результатов. Для программиста термин *«состояние»* означает набор переменных и их соответствующих значений в определенный момент времени. Следовательно, когда речь идет об *итоговом состоянии*, определение которого было дано в первом

разделе, имеется в виду набор всех имеющихся неразделяемых переменных и их значений непосредственно после выполнения последней инструкции задания.

По аналогии *промежуточное состояние* задания — набор всех существующих неразделяемых переменных и их значений в момент выполнения определенной инструкции. Следовательно, на каждой инструкции промежуточное состояние задания меняется и количество таких состояний равно количеству инструкций. Согласно нашим определениям, последнее промежуточное состояние задания совпадает с итоговым.

Разделяемое состояние — тоже набор переменных вместе с соответствующими значениями в определенный момент времени, но эти значения могут быть прочитаны и изменены системой конкурентных заданий. Разделяемое состояние не принадлежит конкретному заданию (не является его локальным состоянием); любое задание, запущенное в системе, может прочитать и перезаписать его в любой момент.

В целом нас не интересуют разделяемые состояния, доступные только для чтения. Обращение к ним со стороны множества конкурентных заданий обычно является безопасным и не создает никаких проблем. Но если должным образом не защитить разделяемое состояние, которое можно изменять, то это может иметь серьезные отрицательные последствия. Поэтому все разделяемые состояния, рассматриваемые в этом разделе, доступны для записи как минимум одному заданию.

Подумайте, что может пойти не так, если одно из конкурентных заданий в системе изменит разделяемое состояние? Чтобы ответить на этот вопрос, мы для начала рассмотрим пример системы с двумя конкурентными заданиями, которые обращаются к одной разделяемой переменной (в данном случае это обычное целое число).

Представим, что у нас есть система, описанная в листинге 13.7.

Листинг 13.7. Система с двумя конкурентными заданиями и изменяемым разделяемым состоянием

```
Concurrent System {

    Shared State {
        X : Integer = 0
    }

    Task P {
        A : Integer
            1. A = X
            2. A = A + 1
            3. X = A
            4. print X
    }
```

```
    Task Q {
        B : Integer
            1. B = X
            2. B = B + 2
            3. X = B
            4. print X
    }
}
```

Допустим, в этой системе задания `P` и `Q` работают неконкурентно — то есть последовательно. Пусть сначала выполняются инструкции в `P`, а затем в `Q`. В таком случае итоговое состояние всей системы, независимо от итогового состояния отдельных заданий, будет состоять из переменной `X`, равной 3.

Если выполнить данную систему в обратном порядке (сначала инструкции в `Q`, а затем в `P`), то получится то же самое итоговое состояние. Однако это редкий случай; обычно выполнение разных заданий в обратном порядке дает другое итоговое состояние.

Как видите, выполнение этих заданий имеет детерминированный результат и нам не нужно беспокоиться о переключениях контекста.

Теперь предположим, что эти задания работают конкурентно на одном ядре процессора. Существует множество сценариев выполнения `P` и `Q` с переключениями контекста на разных инструкциях.

В листинге 13.8 представлен один потенциальный вариант.

Листинг 13.8. Возможное чередование заданий P и Q в ходе конкурентного выполнения

```
   Задание P  | Планировщик заданий    | Задание Q
--------------------------------------------------
              | Переключение контекста |
              |                        | B = X
              |                        | B = B + 2
              | Переключение контекста |
A = X         |                        |
              | Переключение контекста |
              |                        | X = B
              | Переключение контекста |
A = A + 1     |                        |
X = A         |                        |
              | Переключение контекста |
              |                        | print X
              | Переключение контекста |
print X       |                        |
              | Переключение контекста |
```

Это лишь один из возможных сценариев с переключениями контекста, происходящими в определенных местах. Каждый такой сценарий называется *чередованием*. Таким образом, в системе с конкурентными заданиями существует целый ряд потенциальных чередований, которые зависят от того, где могут происходить переключения контекста, и при каждом прогоне реализуется лишь одно из этих чередований. В результате чередования становятся непредсказуемыми.

В первом и последнем столбцах данного сценария видно, что порядок следования инструкций сохраняется, но в ходе выполнения могут возникать *разрывы*. Их нельзя предсказать, и, проследив выполнение представленного выше чередования, можно получить неожиданный результат. Процесс `P` выводит значение `1`, а процесс `Q` выводит `2`, хотя мы полагали, что оба они должны вывести в качестве итогового результата число `3`.

Обратите внимание: в этом примере было определено ограничение на принятие итогового результата: программа должна вывести два числа `3`. Здесь могло быть другое ограничение, не относящееся к видимому выводу программы. Кроме того, существуют другие важные ограничения, которые должны оставаться *неизменными* в условиях непредсказуемых переключений контекста. Сказанное, к примеру, касается отсутствия каких-либо *состояний гонки*, утечек памяти или даже сбоев. Все это куда важнее вывода, возвращаемого программой. Во многих реальных приложениях вывод отсутствует как таковой.

В листинге 13.9 показан еще один вариант чередования с другими результатами.

Листинг 13.9. Еще одно возможное чередование заданий `P` и `Q` в ходе конкурентного выполнения

```
   Задание P  | Планировщик заданий    | Задание Q
----------------------------------------------------
              | Переключение контекста |
              |                        | B = X
              |                        | B = B + 2
              |                        | X = B
              | Переключение контекста |
A = X         |                        |
A = A + 1     |                        |
              | Переключение контекста |
              |                        | print X
              | Переключение контекста |
X = A         |                        |
print X       |                        |
              | Переключение контекста |
```

В данном сценарии задание `P` выводит `3`, а задание `Q` — `2`. Так происходит из-за того, что заданию `P` не повезло и оно не успело обновить значение разделяемой переменной `X` до переключения контекста. Поэтому задание `Q` вывело значение `X`,

которое в тот момент было равно 2. Такая ситуация называется *состоянием гонки* за переменной X; более подробно об этом — в следующей главе.

В настоящей программе на языке C мы обычно пишем X++ или X = X + 1, вместо того чтобы сначала копировать X в A, затем инкрементировать A и, наконец, записывать результат обратно в X. Пример этого будет показан в главе 15.

Представленное выше наглядно показывает, что простое выражение X++ на языке C состоит из трех более мелких инструкций, которые выполняются не одномоментно. Иными словами, это выражение не *атомарно*, но включает в себя три атомарные инструкции. Атомарную инструкцию нельзя разбить на разные операции или прервать переключением контекста. Мы еще вернемся к этому в следующих главах, где речь пойдет о многопоточности.

В приведенном примере следует обратить внимание еще на один аспект. P и Q были не единственными заданиями, выполняемыми в системе; конкурентно с ними выполнялись другие задания, но мы не стали их учитывать и сосредоточились только на P и Q. Почему?

Ответ на данный вопрос связан с тем фактом, что разные виды чередования между любым из этих двух заданий и другими заданиями в системе не могли повлиять на промежуточное состояние P и Q. То есть другие задания не разделяли с P и Q никаких состояний, поэтому, как уже объяснялось ранее, в отсутствие разделяемого состояния чередования не имеют никакого значения. В данном случае это наглядно показано. Следовательно, мы могли исходить из того, что в нашей гипотетической системе нет других заданий, кроме P и Q.

Другие задания могут повлиять на P и Q, только если их будет слишком много и они могут замедлять работу системы. Это просто выльется в более длинные разрывы между соседними инструкциями в P и Q. Иными словами, ядра процессора придется разделять между большим количеством заданий. Поэтому задания P и Q будут находиться в очереди чаще обычного, что задержит их выполнение.

Пример данного сценария показывает, что даже одно состояние, разделяемое между всего двумя конкурентными заданиями, может сделать итоговый результат недетерминированным. Мы рассмотрели проблемы, связанные с нехваткой детерминизма; нам не хочется, чтобы наша программа выдавала разные результаты при каждом выполнении. Задания в нашем примере были относительно простыми и состояли из четырех тривиальных инструкций. Однако настоящие приложения, которые применяются в промышленной среде, выглядят намного сложнее.

Более того, существуют разного рода разделяемые ресурсы, которые могут храниться вне оперативной памяти, — например, файлы или сервисы, доступные в сети.

С другой стороны, к общему ресурсу может обращаться большое количество заданий, поэтому нам нужно более основательно изучить проблемы конкурентности

и подобрать механизмы, которые позволят вернуть детерминизм. В следующей главе мы продолжим обсуждать проблемы, связанные с конкурентностью, и способы их решения.

Напоследок кратко поговорим о планировщике и принципе его работы. Если у нас есть всего одно процессорное ядро, то в любой отдельно взятый момент оно может выполнять только одно задание.

Мы также знаем, что планировщик заданий сам по себе является компонентом программы, которому для выполнения тоже нужно занимать ядро процессора. Каким же образом он управляет разными заданиями, если ядро уже занято? Представьте, что планировщику выделено процессорное время. Вначале он выбирает задание из очереди, затем устанавливает таймер, по которому происходит *прерывание*, и освобождает ядро процессора, передавая ресурсы выбранному заданию.

Мы исходим из того, что планировщик выделяет каждому заданию определенный промежуток времени, по истечении которого произойдет прерывание. В результате ядро процессора перестанет выполнять текущее задание и немедленно загрузит планировщик, который сохранит последнее состояние предыдущего задания и загрузит следующее из очереди. Все это происходит, пока не загрузится операционная система. На компьютере с многоядерным процессором ситуация может быть другой: операционная система может планировать задания не на тех ядрах, на которых выполняется сама.

В этом разделе мы кратко рассмотрели концепцию разделяемых состояний и их роль в конкурентных системах. Продолжим наше обсуждение в следующей главе, в которой речь пойдет о проблемах конкурентности и методиках синхронизации.

Резюме

В этой главе были рассмотрены основы конкурентности и базовые понятия и термины, которые необходимо знать, прежде чем переходить к таким темам, как многопоточность и многопроцессность.

В частности, мы обсудили ряд аспектов.

- Определения конкурентности и параллелизма — тот факт, что каждому параллельному заданию нужен отдельный вычислительный блок, тогда как конкурентные задания могут работать на одном процессоре.
- Конкурентные задания используют один вычислительный блок, а планировщик распределяет между ними процессорное время. Это приводит к ряду переключений контекста и разным видам чередования между заданиями.

- Введение в блокирующие инструкции. Мы выяснили, в каких ситуациях требуется конкурентность, и узнали, как разбить одно задание на два или три конкурентных.
- Было дано определение разделяемому состоянию. Кроме того, показано, как разделяемое состояние может вызвать серьезные проблемы с конкурентностью, такие как состояние гонки, когда несколько заданий пытаются прочитать и записать одно разделяемое значение.

В следующей главе мы завершим рассмотрение конкурентности и разберем несколько видов проблем, с которыми вы столкнетесь в конкурентной среде. Вдобавок мы поговорим о том, как решить эти проблемы.

14 Синхронизация

В предыдущей главе мы прошлись по базовым концепциям и распространенным терминам, относящимся к конкурентности. Здесь же сосредоточимся на проблемах, которые могут возникнуть в программе с конкурентными заданиями. Как и прежде, мы обойдемся без исходного кода на языке C; наше внимание будет направлено исключительно на идеи и теорию, лежащие в основе проблем с конкурентностью и их решений.

В ходе этой главы мы:

- поговорим о *проблемах, связанных с конкурентностью, таких как состояние гонки и гонка данных,* — обсудим последствия разделения состояния между несколькими заданиями и то, как одновременное обращение к разделяемой переменной может вызвать проблемы;
- обсудим *методы управления конкурентностью, направленные на синхронизацию доступа к разделяемому состоянию,* — они будут рассмотрены в основном в теоретической плоскости. Мы выясним, какие подходы можно использовать для преодоления проблем, относящихся к конкурентности;
- рассмотрим *конкурентность в POSIX* — в рамках этой темы обсудим, как POSIX может унифицировать процесс разработки конкурентных программ. Мы также затронем и сравним многопоточные и многопроцессные программы.

В первом разделе мы продолжим разговор, начатый в предыдущей главе, и более подробно обсудим, как недетерминированная природа конкурентных сред может привести к проблемам с конкурентностью. Мы также попытаемся разделить эти проблемы на категории.

Проблемы с конкурентностью

В предыдущей главе вы видели, что изменяемое разделяемое состояние может вызвать проблемы, когда его значение могут менять конкурентные задания. Мы можем пойти дальше и спросить, о проблемах какого рода идет речь? Какова основная причина их возникновения? Здесь будут даны ответы на все эти вопросы.

Прежде всего необходимо выделить разные виды проблем, которые могут возникнуть применительно к конкурентности. Некоторые из них возможны только при отсутствии механизмов управления конкурентными заданиями, а другие являются следствием использования этих механизмов.

Когда случаются проблемы из первой категории, разные варианты чередования инструкций приводят к разным *итоговым состояниям*. После выявления одной из них следующим логичным шагом будет поиск подходящего решения.

Что касается второй категории, то такие проблемы возникают в результате применения решения, упомянутого выше. Это значит, при исправлении одних проблем можно создать новые, которые имеют совершенно другую природу и причины возникновения; вот что делает конкурентное программирование проблематичным.

Допустим, к примеру, у вас есть ряд заданий с возможностью чтения/записи одного разделяемого источника данных. После многократного выполнения этих заданий обнаруживается, что их алгоритмы работают не так, как вы ожидали. Это приводит к случайным сбоям и логическим ошибкам, которые возникают произвольным образом. Поскольку эти сбои и некорректные результаты невозможно предсказать, вы можете резонно предположить, что причина кроется в конкурентности.

Вы начинаете анализировать алгоритмы снова и снова. Наконец проблема найдена: *состояние гонки* за разделяемым источником данных. Теперь вам нужно выработать решение, которое будет управлять доступом к этому источнику. Реализовав решение и повторно запустив систему, вы с удивлением обнаруживаете, что некоторые источники не получают ни одного шанса обратиться к источнику данных. Говоря формальным языком, эти задания лишены ресурсов. Ваши изменения породили новую проблему совершенно иного рода!

Таким образом, мы имеем дело с двумя группами проблем, относящихся к конкурентности, а именно:

- с проблемами, существующими в конкурентной системе, в которой нет механизма управления (синхронизации). Мы называем такие проблемы *естественными*;
- с проблемами, возникшими в результате попытки исправления проблем из первой группы. Мы называем их *постсинхронизационными*.

Проблемы из первой группы называются *естественными*, поскольку присущи всем конкурентным системам. Их нельзя избежать, и для их решения нужны управляющие механизмы. В некотором смысле их можно считать даже не проблемами конкурентных систем, а их свойством. Несмотря на это, их нужно исправлять, так как их непредсказуемость мешает разработке детерминированных программ.

Проблемы из второй группы случаются только при неправильном использовании управляющих механизмов. Стоит отметить: сами по себе эти механизмы непроблематичны; они необходимы для того, чтобы снова сделать нашу программу детерминированной. Но если применять их неправильно, то они могут приводить к возникновению вторичных проблем с конкурентностью. Эти вторичные, или постконкурентные, проблемы можно считать программными ошибками, допущенными разработчиком, а не неотъемлемым свойством конкурентных систем.

В следующих разделах мы познакомимся с проблемами из обеих групп. Начнем с естественных проблем и обсудим основные причины их неотвратимости в конкурентных средах.

Естественные проблемы с конкурентностью

Любая конкурентная система с несколькими заданиями имеет ряд возможных вариантов чередования. Это можно считать ее неотъемлемым свойством. По предыдущему материалу нам известно, что данное свойство недетерминированное по своей природе и при каждом прогоне инструкции разных заданий могут выполняться в разном порядке, хотя в рамках отдельно взятого задания они остаются *последовательными*. Мы уже рассматривали это в предыдущей главе.

Чередования сами по себе нельзя назвать проблемными. Как уже объяснялось ранее, они являются неотъемлемым свойством конкурентной системы. Но в некоторых случаях оно нарушает определенные ограничения. Именно в таких ситуациях чередования вызывают проблемы.

Мы уже знаем, что при конкурентном выполнении ряда заданий количество чередований может быть большим. Однако проблемы возникают, только когда ограничения системы, которые должны оставаться неизменными, варьируются между разными прогонами. Поэтому наша цель — внедрить такие управляющие механизмы (их иногда называют *механизмами синхронизации*), которые бы сохраняли ограничения в их исходном виде.

Ограничение обычно выражается в виде списка условий и критериев, которые мы будем в дальнейшем называть *инвариантными ограничениями*. Эти ограничения могут относиться практически к любым аспектам системы.

Инвариантные ограничения иногда могут быть очень простыми, как в примере, рассмотренном нами в предыдущей главе, в котором программа должна была вывести два числа 3. Они также бывают весьма сложными, как в случае с сохра-

нением *целостности данных* всех внешних источников в огромной конкурентной программе.

Сгенерировать каждый возможный вариант чередований крайне непросто. В некоторых случаях конкретное чередование происходит с очень низкой долей вероятности. Оно может встречаться один раз на миллион попыток или вообще никогда.

Это еще один рискованный аспект конкурентной разработки. Некоторые варианты чередования попадаются очень редко, но притом вызывают большие проблемы. Они могут, к примеру, привести к авиакатастрофе или серьезной неисправности в медицинском оборудовании во время операции на головном мозге!

У любой конкурентной системы есть по крайней мере некоторые четко определенные инвариантные ограничения. Они имеются во всех примерах, которые мы будем рассматривать в этой главе. Мы обсудим каждое из них, поскольку нам необходимо спроектировать конкретную конкурентную систему, которая их не нарушает и сохраняет в неизменном виде.

Чередования, происходящие в конкурентной системе, должны соответствовать заранее определенным инвариантным ограничениям. В противном случае с системой что-то не так. Именно в этой ситуации подобные ограничения становятся очень важными. Если чередование нарушает инвариантное ограничение системы, то это значит, что мы имеем дело с *состоянием гонки*.

Состояние гонки — следствие неотъемлемого свойства конкурентной системы: чередований. При его возникновении всегда возникает опасность нарушения инвариантного ограничения.

Последствия подобных нарушений могут выражаться как в логических ошибках, так и в сбоях. Существует множество примеров того, как значения, хранящиеся в разделяемых переменных, не отражают реальное состояние системы. Основная причина этого связана с существованием разных вариантов чередований, которые нарушают *целостность данных* разделяемой переменной.

Проблемы, связанные с целостностью данных, будут рассмотрены позже в этой главе, а пока обратимся к следующему примеру. Как уже отмечалось, прежде чем разрабатывать систему, для нее необходимо определить инвариантные ограничения. Пример 14.1, показанный в листинге 14.1, содержит всего одно такое ограничение, согласно которому итоговое значение переменной `Counter` должно быть равно `3`. В этом примере есть три конкурентных задания. Каждое из них должно увеличивать `Counter` на единицу. Мы пытаемся реализовать данную логику в следующем коде.

Листинг 14.1. Система с тремя конкурентными заданиями, которые работают с одной разделяемой переменной

```
Concurrent System {

    Shared State {
      Counter : Integer = 0
    }

    Task T1 {
      A : Integer
        1.1. A = Counter
        1.2. A = A + 1
        1.2. Counter = A
    }

    Task T2 {
      B : Integer
        2.1. B = Counter
        2.2. B = B + 1
        2.2. Counter = B
    }

    Task T3 {
      A : Integer
        3.1. A = Counter
        3.2. A = A + 1
        3.2. Counter = A
    }
}
```

В этом листинге вы можете видеть конкурентную систему, написанную на псевдокоде. Она состоит из трех заданий. У нее также есть участок с разделяемыми состояниями. Единственная разделяемая переменная, которая доступна всем трем заданиям, — `Counter`.

У каждого задания может быть ряд локальных переменных, которые принадлежат только ему и не видны другим заданиям. Благодаря этому мы можем иметь две разные локальные переменные с одинаковым именем `A`, принадлежащие разным заданиям.

Обратите внимание: задания не могут работать с разделяемыми переменными напрямую и каждое из них может выполнять лишь какую-то одну операцию — чтение или запись. Вот, собственно, зачем нам нужны локальные переменные. Задания могут инкрементировать напрямую только локальную переменную, но не разделяемую. Похожую ситуацию можно наблюдать в многопоточных и многопроцессных системах, и именно поэтому мы выбрали для нашей конкурентной системы описанную выше конфигурацию.

Пример, показанный в листинге 14.1, демонстрирует возникновение логической ошибки в результате состояния гонки. Мы можем легко найти вариант чередований, после которых переменная `Counter` будет равна `2`. Взгляните на чередования в листинге 14.2.

Листинг 14.2. Чередования, которые нарушают инвариантное ограничение, определенное в листинге 14.1

```
 Планировщик заданий    | Задание T1   | Задание T2   | Задание T3
------------------------------------------------------------------
 Переключение контекста |              |              |
                        | A = Counter  |              |
                        | A = A + 1    |              |
                        | Counter = A  |              |
 Переключение контекста |              |              |
                        |              | B = Counter  |
                        |              | B = B + 1    |
 Переключение контекста |              |              |
                        |              |              | A = Counter
 Переключение контекста |              |              |
                        |              | Counter = B  |
 Переключение контекста |              |              |
                        |              |              | A = A + 1
                        |              |              | Counter = A
```

Здесь можно легко отследить чередования. Инструкции `2.3` и `3.3` (см. листинг 14.1) сохраняют в переменную `Counter` значение `2`. Эта ситуация называется *гонкой данных*, и мы еще вернемся к ней в текущем разделе.

В следующем примере, показанном в листинге 14.3, демонстрируется, как состояние гонки может привести к сбою.

Ниже используется пример с псевдокодом, поскольку мы все еще не познакомились с API POSIX, необходимым для написания программ на языке C с поддержкой создания и управления потоками или процессами.

Ниже показан пример, который мог бы завершиться ошибкой сегментации, будь он написан на C.

Листинг 14.3. Чередование, нарушающее инвариантное ограничение, описанное в листинге 14.1

```
Concurrent System {

    Shared State {
      char *ptr = NULL; // Разделяемый указатель типа char
                        // должен указывать на адрес
                        // в пространстве кучи. Он становится
                        // равным null по умолчанию.
    }

    Task P {
        1.1. ptr = (char*)malloc(10 * sizeof(char));
        1.2. strcpy(ptr, "Hello!");
        1.3. printf("%s\n", ptr);
    }
```

```
    Task Q {
        2.1. free(ptr);
        2.2. ptr = NULL;
    }
}
```

Одно из очевидных инвариантных ограничений, которое интересует нас в этом примере и не было определено явно, состоит в том, чтобы не допустить сбой задания. Если задание не может доработать до конца, то наличие инвариантного ограничения само по себе является противоречивым.

Существует несколько чередований, которые могут привести к сбою задания. Рассмотрим два из них.

- Представьте, что первой выполняется инструкция `2.1`. Поскольку указатель `ptr` равен `null`, задание `Q` аварийно завершится, а задание `P` продолжит работу. В итоге, если мы имеем дело с многопоточным кодом, в котором оба задания (потока) принадлежат одному процессу, выйдет из строя вся программа. Основная причина состоит в удалении нулевого указателя.
- Еще один вариант чередования происходит, когда инструкция `2.2` выполняется до `1.2`, но после `1.1`. В этом случае сбой произойдет в задании `P`, а задание `Q` продолжит работу без каких-либо проблем. Основная причина сбоя заключается в разыменовании нулевого указателя.

Как показывают предыдущие примеры, наличие состояния гонки в конкурентной системе может иметь разные последствия, такие как логические ошибки и неожиданные сбои. Очевидно, что обе проблемы нуждаются в адекватном решении.

Следует отдельно отметить: не все состояния гонки в конкурентной системе можно обнаружить легко. Некоторые из них проявляются только со временем. Вот почему я начал эту главу с утверждения о том, что конкурентные программы проблематичны.

Тем не менее обнаружить состояния гонки на участках кода, которые выполняются не так часто, иногда можно с помощью *специальных инструментов*. На самом деле они позволяют обнаружить чередования, приводящие к состоянию гонки.

Состояние гонки можно обнаружить с помощью отдельной категории программ, которые делятся на статические и динамические.

Статические инструменты анализируют исходный код, пытаясь сгенерировать из имеющихся инструкций все возможные чередования. Динамические же сначала запускают программу и ждут, когда она выполнит код, который может привести к состоянию гонки. Оба подхода используются совместно, что позволяет минимизировать риск возникновения подобных проблем.

Пришло время задать еще один вопрос: существует ли какая-то одна главная причина возникновения состояний гонки? Чтобы выработать решение подобного рода проблем, нам нужно как-то ответить на него. Мы знаем, что состояние гонки возникает каждый раз, когда чередование нарушает инвариантное ограничение. Поэтому чтобы дать ответ, нам нужно более детально проанализировать возможные инвариантные ограничения и подумать о том, как они могут быть нарушены.

Из увиденного в различных конкурентных системах можно сделать следующий вывод: чтобы удовлетворить инвариантные ограничения, некоторые инструкции, принадлежащие разным заданиям, должны быть выполнены в строго определенном порядке, независимо от варианта чередований.

Следовательно, чередования, соответствующие этому порядку, не нарушают инвариантное ограничение. Нас они устраивают, и мы можем наблюдать нужный вывод. Чередования, которые не обеспечивают строгий порядок выполнения, нарушают инвариантное ограничение и могут считаться проблемными.

На случай таких чередований нам нужны механизмы, которые позволят восстановить порядок выполнения и убедиться в том, что инвариантные ограничения всегда соблюдаются. В листинге 14.4 показан пример 14.2. Согласно ограничению, которое должно оставаться неизменным, *программа должна выводить 1*. Это довольно незрелое требование, которое вы вряд ли встретите в настоящих приложениях; мы используем его, чтобы продемонстрировать обсуждаемые здесь концепции.

Листинг 14.4. Очень простая конкурентная система, страдающая от состояния гонки

```
Concurrent System {

  Shared State {
    X : Integer = 0
  }

  Task P {
      1.1. X = 1
  }

  Task Q {
      2.1. print X
  }
}
```

Этот пример может иметь два разных вывода в зависимости от типа чередований. Если нам нужно вывести `1`, что соответствует инвариантному ограничению, то мы должны определить строгий порядок выполнения инструкций в двух разных заданиях.

В связи с этим инструкция `print` под номером `2.1` должна выполняться только после инструкции `1.1`. Существует еще одно чередование, которое легко нарушает

данный порядок (а значит, и инвариантное ограничение), и потому мы имеем состояние гонки. Нам нужно определить строгий порядок выполнения этих инструкций. Но сделать это не так-то просто. О том, как восстановить этот порядок, мы поговорим позже в данной главе.

Взгляните на пример 14.3, представленный в листинге 14.5. В коде описана система с тремя заданиями. Следует отметить, что у нее нет разделяемого состояния. Но, несмотря на это, мы получаем состояние гонки. Определим для данной системы инвариантное ограничение: *сначала должно выводиться число 1, затем 2 и в конце 3*.

Листинг 14.5. Еще одна очень простая система, которая страдает от состояния гонки, однако не имеет разделяемого состояния

```
Concurrent System {

  Shared State {
  }

  Task P {
      1.1. print 3
  }

  Task Q {
      2.1. print 1
  }

  Task R {
      3.1. print 2
  }
}
```

Даже в этой крайне простой системе нельзя предсказать, какое задание будет первым, вследствие чего мы имеем состояние гонки. Следовательно, чтобы не нарушить инвариантное ограничение, инструкции должны выполняться в таком порядке: `2.1`, `3.1`, `1.1`. Данный порядок должен сохраняться при всех возможных чередованиях.

Этот пример раскрывает важное свойство состояний гонки: их возникновение в конкурентной системе не требует разделяемого состояния. Чтобы их избежать, некоторые инструкции должны всегда выполняться в строго определенном порядке. Следует отметить: состояние гонки встречается только когда небольшая группа инструкций, известная под названием *«критический участок»*, выполняется не по порядку, в то время как другие инструкции могут выполняться в любом порядке.

Если у вас есть изменяемое разделяемое состояние с инвариантным ограничением, то операции *чтения* и *записи* в отношении данного состояния, возможно, придется выполнять в строгом порядке. Одно из самых важных ограничений, относящихся

к изменяемому разделяемому состоянию, касается целостности данных. Это просто означает, что у всех заданий всегда должна быть возможность прочитать последнее, самое свежее значение разделяемого состояния. Кроме того, прежде чем приступать к его изменению, они должны знать о любых обновлениях, произошедших с момента чтения.

Пример 14.4, показанный в листинге 14.6, демонстрирует ограничение целостности данных и, что еще важнее, ситуацию, в которой оно может быть легко нарушено.

Листинг 14.6. Конкурентная система, страдающая от состояния гонки за разделяемой переменной X

```
Concurrent System {

  Shared State {
    X : Integer = 2
  }

  Task P {
    A : Integer
      1.1. A = X
      1.2. A = A + 1
      1.3. X = A
  }

  Task Q {
    B : Integer
      2.1. B = X
      2.2. B = B + 3
      2.3. X = B
  }
}
```

Рассмотрим следующий вариант чередования. Сначала выполняется инструкция `1.1`. Следовательно, значение `X` копируется в локальную переменную `A`. Но заданию `P` повезло меньше, поэтому происходит переключение контекста, и процессор поступает в распоряжение задания `Q`. Затем выполняется инструкция `2.1`, после чего значение `X` копируется в локальную переменную `B`. Таким образом, переменные `A` и `B` содержат одно и то же значение `2`.

Теперь повезло заданию `Q`, и оно может продолжать работу. Выполняется инструкция `2.2`, в результате чего переменная `B` становится равна `5`. Задание `Q` продолжается и записывает значение `5` в разделяемое состояние `X`. Поэтому переменная `X` теперь равна `5`.

Происходит следующее переключение контекста, и доступ к процессору возвращается заданию `P`, которое продолжает работу с инструкции `1.2`. Вот где нарушается ограничение целостности.

Задание Q обновило состояние X, но задание P использует в оставшейся части своих вычислений старое значение 2. В конечном счете значение X будет сброшено к 3. Вряд ли нас удовлетворит подобный результат. Чтобы не нарушать ограничение целостности данных, мы должны сделать так, чтобы инструкция 1.1 выполнялась только после 2.3 или инструкция 2.1 выполнялась лишь после 1.3. В противном случае целостность данных может пострадать.

Вы можете спросить, зачем нам нужны переменные A и B, если мы могли легко записать X = X + 1 или X = X + 3.

Как уже объяснялось в предыдущей главе, выражение X = X + 1 (в C записывается как X++) не атомарно. То есть его нельзя уместить в одну инструкцию. Это связано с тем, что у нас не может быть прямого доступа к памяти, когда мы выполняем с ней некую операцию.

Мы всегда используем временную переменную (или регистр процессора) для хранения последнего значения. Поэтому операция выполняется с данной переменной или регистром, и только потом результат передается обратно в память. Следовательно, как бы вы это ни записывали, у вашего задания всегда будет временная локальная переменная.

Как вы сами убедитесь, в системах с несколькими ядрами процессора ситуация только усугубляется. Процессор также имеет кэш, в котором хранятся переменные, и значения из него попадают в основную память не мгновенно.

Поговорим еще об одном определении. Каждый раз, когда чередование нарушает ограничение целостности данных, относящееся к разделяемой переменной, эта переменная начинает участвовать в гонке данных.

Гонки данных очень похожи на состояния гонки, но их возникновение требует наличия переменной, разделяемой между несколькими заданиями, и как минимум одно из этих заданий должно иметь возможность ее изменять. Иными словами, разделяемая переменная должна быть доступна не только для чтения, и по меньшей мере одному заданию следует иметь логику, которая может что-то записать в эту переменную.

Как мы уже говорили, у переменной, доступной только для чтения, *не может* быть гонки данных. Целостность такой переменной невозможно нарушить, поскольку ее нельзя изменить.

В примере 14.5, показанном в листинге 14.7, продемонстрировано, как состояние гонки может возникнуть в ситуации, в которой невозможна гонка данных, относящаяся к переменной, доступной только для чтения.

Листинг 14.7. Конкурентная система с разделяемой переменной, доступной только для чтения

```
Concurrent System {

  Shared State {
    X : Integer (read-only) = 5
  }

  Task P {
    A : Integer
      1.1. A = X
      1.2. A = A + 1
      1.2. print A
  }

  Task Q {
      2.1. print X
  }

  Task R {
    B : Integer
      3.1. B = X + 1
      3.2. B = B + 1
      3.3. print B
  }
}
```

Представим, что в этом примере действует следующее инвариантное ограничение: *поддерживать целостность данных и выводить сначала 5, затем 6 и в конце 7*. Очевидно, мы имеем состояние гонки, так как должны обеспечить строгий порядок выполнения разных инструкций `print`.

Но, поскольку разделяемую переменную можно только читать, гонки данных не возникает. Обратите внимание: инструкции `1.2` и `3.2` изменяют лишь свое локальное состояние, что нельзя считать обновлением разделяемой переменной.

В заключение скажу следующее: не надейтесь легко устранить состояния гонки! Чтобы обеспечить определенный порядок выполнения инструкций из разных заданий, вам, несомненно, понадобятся некие механизмы синхронизации. Они позволят вам соблюдать заданный порядок во всех возможных вариантах чередований. На самом деле, как вы сами увидите в следующем разделе, для этого придется создавать новые чередования, которые соответствуют желаемому порядку.

Мы еще вернемся к этим механизмам позже в данной главе; но сначала необходимо поговорить о проблемах конкурентности, возникающих после применения методов синхронизации. В следующем разделе мы обсудим *постсинхронизационные* проблемы и то, чем они отличаются от естественных.

Постсинхронизационные проблемы

Далее мы поговорим о трех ключевых проблемах, которые можно ожидать в результате неправильного использования управляющих механизмов. Вы можете столкнуться с одним из них или со всеми сразу, поскольку они возникают по разным причинам.

- *Новые естественные проблемы.* Применение управляющих механизмов может привести к разным состояниям гонки или гонкам данных. Эти механизмы пытаются обеспечить определенный порядок следования инструкций, из-за чего могут возникнуть новые естественные проблемы. Это объясняется тем, что при внедрении управляющих механизмов добавляются новые чередования. А новые состояния гонки и гонки данных могут вызвать дополнительные логические ошибки и сбои. Чтобы это исправить, вам придется пройтись по использованным методикам синхронизации и адаптировать их к логике своей программы.
- *Недоступность ресурсов.* Эта ситуация возникает, когда задание в конкурентной системе не может обратиться к разделяемому ресурсу на протяжении длительного времени (что обычно случается из-за применения определенного управляющего механизма). Такое задание не имеет доступа к разделяемому ресурсу и, следовательно, не может эффективно выполнять свои функции. Если с ним взаимодействует другое задание, то данная проблема может причинить вред и ему.
- *Взаимные блокировки.* Когда все задания в конкурентной системе ждут друг друга и ни одно из них не продвигается вперед, мы называем подобную ситуацию взаимной блокировкой. Так происходит в основном в результате неправильного применения управляющего механизма, что заставляет задания войти в бесконечный цикл, в котором они ждут, когда одно из них освободит разделяемый ресурс или снимет блокировку. Обычно это называют *круговым ожиданием*. Пока задания ждут своей очереди, ни одно из них не может продолжить работу, в результате чего система перестает отвечать. Некоторые примеры взаимных блокировок можно найти на странице «Википедии» https://ru.wikipedia.org/wiki/Взаимная_блокировка.

  В такой ситуации все задания блокируются и ждут друг друга. Но зачастую это происходит лишь с несколькими из них, тогда как остальные могут продолжать выполнение. Это называется *частичной взаимной блокировкой*. Соответствующие примеры будут рассмотрены в следующих разделах.
- *Перестановка приоритетов.* Иногда после применения метода синхронизации задание с приоритетным доступом к разделяемому ресурсу блокируется заданием с низким приоритетом. То есть их приоритеты меняются местами. Это еще один побочный эффект, который можно наблюдать в случае неправильного использования методов синхронизации.

Изначально конкурентная система не страдает от недоступности ресурсов. Если к планировщику операционной системы не применять никаких методов синхро-

низации, то он будет вести себя беспристрастно и не оставит ни одно задание без доступа к разделяемому ресурсу. Эта проблема может возникнуть, только если программист внедрил некие управляющие механизмы. Точно так же без участия программиста в конкурентной системе не может произойти взаимная блокировка. Основная причина возникновения последней — использование таких блокировок, снятие которых ожидают все задания. В целом взаимная блокировка в конкурентных системах встречается чаще, чем недоступность ресурсов.

Теперь перейдем к обсуждению управляющих механизмов. В следующем разделе мы поговорим о различных методах синхронизации, с помощью которых можно избежать состояний гонки.

Методы синхронизации

В этом разделе речь пойдет о методах синхронизации, известных также как методы управления конкурентностью или конкурентные управляющие механизмы. Они предназначены для решения естественных проблем, присущих конкурентным системам. Если говорить в контексте всего того, что мы обсудили ранее, то управляющие механизмы пытаются нивелировать проблемы, которые могут возникнуть в системе из-за определенных чередований.

У любой конкурентной системы есть инвариантные ограничения, и не все чередования будут соблюдать каждое из них. Для таких чередований необходимо придумать механизм, который будет обеспечивать определенный порядок следования инструкций. Иными словами, нам нужно создать *новые* чередования, не нарушающие инвариантные ограничения, и подставить их вместо «плохих» чередований. После применения определенного метода синхронизации мы получим совершено новую конкурентную систему с некоторыми новыми чередованиями. Все это делается в надежде на то, что новая система будет соблюдать инвариантные ограничения и не вызовет никаких постсинхронизационных проблем.

Обратите внимание: чтобы использовать метод синхронизации, нам придется писать новый код и изменять имеющийся. В последнем случае вы фактически меняете порядок выполнения инструкций и, следовательно, чередования. Этим вы, в сущности, создаете новую конкурентную систему с новыми чередованиями.

Каким образом наличие новых чередований решает наши проблемы с конкурентностью? Добавляя новые чередования, мы ужесточаем порядок следования инструкций из разных заданий, что обеспечивает соответствие инвариантным ограничениям.

Стоит отметить, что две инструкции в одном задании всегда размещены в определенном порядке, чего нельзя сказать об инструкциях из разных заданий в конкурентной системе. Этот порядок мы устанавливаем за счет использования методов синхронизации.

Совершенно новая конкурентная система несет в себе новые проблемы, отличные от старых. Исходная конкурентная система носила естественный характер, и планировщик заданий был единственным ее компонентом, который инициировал переключения контекста. Но после внесения изменений мы получили искусственную конкурентную систему, помимо планировщика заданий имеющую и другие действующие элементы. Важную роль в ней играют механизмы управления конкурентностью, которые используются для сохранения ее инвариантных ограничений. В результате проявляются новые, *постсинхронизационные проблемы*, которые мы обсуждали в предыдущем разделе.

Выбор подходящих управляющих методик в целях такой синхронизации разных заданий, чтобы они соблюдали определенный порядок выполнения, зависит от исходного конкурентной среды. Например, управляющие механизмы, которые используются в многопроцессной программе, могут отличаться от методов, применяемых в многопоточном приложении.

Таким образом, подробное рассмотрение управляющих механизмов невозможно без использования настоящего кода на языке C. Поэтому мы будет обсуждать абстрактные концепции, применимые ко всем конкурентным системам независимо от того, как они реализованы. Это значит, что представленные ниже методики и концепции актуальны для всех конкурентных систем, но их реализация сильно зависит от истинной природы среды и от самой системы.

Холостые циклы и циклические блокировки

Чтобы заставить инструкцию из одного задания выполняться после инструкции из другого, первое задание должно подождать, пока второе не выполнит свою инструкцию. Тем временем если первое получит доступ к процессору в результате переключения контекста, то не должно продолжать работу. Иными словами, оно должно остановиться и подождать, пока не будет выполнена инструкция из второго задания. Это обобщенное решение.

После того как второе задание выполняет свою инструкцию, перед нами открываются две возможности. Первое задание может само проверить, завершилась ли эта инструкция, или же второе задание может уведомить первое о том, что оно может продолжать работу.

Этот сценарий похож на ситуацию, когда два человека пытаются что-то сделать в определенном порядке. Один из них должен ждать, пока второй не совершит свои действия, и только потом берется за работу. Можно сказать, что почти все управляющие механизмы используют аналогичный подход, но их реализации могут отличаться и зависеть от механизмов, доступных в определенной среде. В последнем разделе данной главы мы рассмотрим одну из таких сред, POSIX-совместимые системы, и доступные в нем механизмы.

Рассмотрим пример 14.6, чтобы проанализировать приведенную выше методику, на которой основаны все остальные управляющие механизмы. В листинге 14.8 показана система с двумя конкурентными заданиями, в которой мы хотим определить следующее инвариантное ограничение: *сначала должна выводиться буква A, а затем B*. Без использования каких-либо управляющих механизмов она выглядит следующим образом.

Листинг 14.8. Конкурентная система в примере 14.6 до применения управляющего механизма

```
Concurrent System {

  Task P {
    1.1. print 'A'
  }

  Task Q {
    2.1. print 'B'
  }
}
```

Очевидно, мы имеем состояние гонки, основанное на инвариантном ограничении. Чередование `{2.1, 1.1}` выводит сначала `A`, а затем `B`, а это противоречит данному ограничению. Следовательно, чтобы сохранить определенный порядок выполнения этих инструкций, нам нужен управляющий механизм.

Мы хотим, чтобы инструкция `2.1` выполнялась только после завершения инструкции `1.1`. Псевдокод, показанный в листинге 14.9, демонстрирует проектирование и применение ранее рассмотренного подхода в целях расстановки инструкций в нужном нам порядке:

Листинг 14.9. Решение для примера 14.6 с помощью холостого цикла

```
Concurrent System {

  Shared State {
    Done : Boolean = False
  }

  Task P {
    1.1. print 'A'
    1.2. Done = True
  }

  Task Q {
    2.1. While Not Done Do Nothing
    2.2. print 'B'
  }
}
```

Как видите, для синхронизации заданий нам пришлось добавить новые инструкции. Таким образом, у нас появилось множество новых чередований. Точнее, мы имеем дело с совершенно новой конкурентной системой, если сравнивать ее с исходной. У этой новой системы есть свой набор чередований, несравнимый с предыдущим.

Все эти новые чередования имеют общий аспект: тот факт, что инструкция `1.1` всегда выполняется до инструкции `2.2`. Именно этого мы хотели достичь, когда добавляли управляющий механизм. Независимо от варианта чередований и переключений контекста, мы установили строгий порядок выполнения инструкций `1.1` и `2.2`.

Как нам это удалось? В системе появилось новое разделяемое состояние `Done`, которое представляет собой булеву переменную с начальным значением `False`. Когда задание `P` выводит `A`, оно присваивает `Done` значение `True`. Затем задание `Q`, которое ждет, когда `Done` станет равно `True`, выходит из цикла `while` на инструкции `2.1` и выводит `B`. Иными словами, задание `Q` пребывает в цикле `while` до тех пор, пока переменная `Done` не получит значение `True`; это будет сигналом о том, что задание `P` выполнило свою команду `print`. Данное решение выглядит отлично и работает без проблем.

Представьте следующий вариант чередований. Когда задание `P` уступает ядро процессора заданию `Q` и если `Done` не равно `True`, то задание `Q` остается в цикле, пока не потеряет доступ к ядру. Это значит, что необходимое условие не выполняется. Как следствие, задание `Q`, хоть и работает в ядре процессора и не покидает ядро, в отведенный ему *интервал* не делает почти ничего, кроме как проверяет, выполнилось ли условие. Так продолжается до тех пор, пока ядро процессора не будет занято другим заданием. Иными словами, задание `Q` тратит процессорное время *впустую*, пока задание `P` не продолжит работу и не выведет `A`.

Если говорить техническим языком, то задание `Q` ждет в *холостом цикле*, когда выполнится определенное условие. Оно непрерывно отслеживает (или запрашивает) данное условие, пока то не станет равно `True`, и затем выходит из холостого цикла. Как бы мы это ни называли, задание `Q` тратит впустую драгоценное процессорное время, хотя приведенный выше метод полностью решает нашу проблему.

Холостой цикл — хоть и неэффективный, но зато простой способ ожидания событий. Поскольку внутри него задание не может делать ничего особенного, он расходует выделенный ему интервал без какой-либо пользы. Холостых циклов избегают при продолжительном ожидании. Потерянное процессорное время можно выделить другому заданию, чтобы оно могло проделать часть своей работы. Но в некоторых случаях, когда время ожидания должно быть коротким, применяются холостые циклы.

В настоящих программах на C и других языках программирования строгий порядок обычно соблюдается с использованием *блокировки*. Это просто объект или переменная, с помощью которой мы ждем выполнения условия или наступления события. Обратите внимание: `Done` в предыдущем примере — не блокировка, а флаг.

Чтобы лучше понять, как это работает, представьте, что перед выполнением инструкции `2.2` мы попытались получить блокировку. Только наличие блокировки позволит нам продолжить работу и выйти из цикла. Внутри цикла мы ждем, когда блокировка станет доступной. Блокировки бывают разных типов, и об этом мы поговорим несколько позже.

Далее мы попробуем реализовать пример, описанный выше, но с использованием более эффективного подхода, который не тратит впустую процессорное время. У него есть много разных названий, но мы будем называть его механизмом *ожидания/уведомления*.

Механизм ожидания/уведомления

Вместо использования холостого цикла, как было показано в предыдущем подразделе, можно представить другую ситуацию. Задание `Q` может не ждать флаг `Done`, а просто «уснуть», а задание `P` может его уведомить, когда этот флаг станет равен `True`.

Иными словами, задание `Q` «засыпает», как только обнаруживается, что флаг не равен `True`, позволяя заданию `P` быстрее получить процессорное время и выполнить свою логику. Задание `P`, в свою очередь, пробудит `Q` после того, как присвоит флагу `True`. На самом деле данный подход *фактически* реализован в большинстве операционных систем, и это позволяет избежать холостых циклов и применять управляющие механизмы более эффективно.

Псевдокод листинга 14.10 показывает, как с помощью этого подхода переписать решение примера, предложенное в предыдущем подразделе.

Листинг 14.10. Решение примера 14.6 методом ожидания/уведомления

```
Concurrent System {

  Task P {
      1.1. print 'A'
      1.2. Notify Task Q
  }

  Task Q {
      2.1. Go To Sleep Mode
      2.2. print 'B'
  }
}
```

Прежде чем анализировать этот код, следует обсудить некоторые новые понятия. Для начала, как задание может *«уснуть»*. В этом состоянии оно не получает никаких ресурсов процессора. Когда оно «засыпает», планировщик об этом узнает и не выделяеят ему никаких интервалов.

Каково преимущество этого подхода? «Спящее» задание не входит в холостой цикл и не тратит процессорное время. Вместо того чтобы запускать новый холостой цикл для проверки условия, оно «засыпает» и по выполнении условия получает уведомление. Это существенно повышает *эффективность использование процессора*, и задания, которым действительно нужны ресурсы, смогут их получить.

Когда задание переходит в режим сна, у вас должна быть возможность как-то его «пробудить». Для этого заданию обычно отправляются *уведомления* или *сигналы*. Когда уведомленное задание «просыпается», планировщик снова помещает его в очередь и выделяет ему ресурсы процессора. Затем задание продолжит работу непосредственно с того места, на котором «уснуло».

В написанном нами коде задание `Q` входит в спящий режим сразу после начала работы. После этого оно перестанет получать какие-либо ресурсы процессора, пока задание `P` не «пробудит» его с помощью уведомления. Задание `P` уведомляет `Q` только после вывода `A`. «Проснувшись», задание `Q` получает процессорное время и выводит `B`.

Благодаря этому методу мы не используем холостой цикл и не тратим время процессора впустую. Стоит отметить: для «засыпания» и уведомления «спящего» задания в большинстве операционных систем (особенно в POSIX-совместимых) предусмотрены специальные системные вызовы.

На первый взгляд может показаться, что вышеописанный сценарий решил нашу проблему, причем эффективно, — и это действительно так! Однако существует вариант чередования, который создает постсинхронизационные проблемы. Это происходит, когда наша система имеет порядок выполнения, показанный в листинге 14.11.

Листинг 14.11. Вариант чередования, вызывающий в конкурентной системе из листинга 14.10 частичную взаимную блокировку

```
1.1. print 'A'
1.2. Notify Task Q
2.1. Go To Sleep Mode
2.2. print 'B'
```

В этом сценарии задание `P` вывело `A` и затем уведомило задание `Q`, которое еще не успело «заснуть», поскольку ему не было выделено процессорное время. Получив ресурсы процессора, `Q` сразу же «засыпает». Однако у нас нет никаких других заданий, которые могли бы его уведомить. Следовательно, задание `Q` больше не будет выполняться ввиду того, что планировщик не выделяет ядра процессора

«спящим» заданиям. Это первый пример того, как в результате применения метода синхронизации возникла постсинхронизационная проблема.

Чтобы ее решить, нам снова нужно воспользоваться булевым флагом. Теперь, прежде чем «засыпать», задание Q должно проверить этот флаг. В листинге 14.12 показано наше итоговое решение.

Листинг 14.12. Улучшенное решение примера 14.6 на основе метода с ожиданием/уведомлением

```
Concurrent System {

  Shared State {
    Done : Boolean = False
  }

  Task P {
      1.1. print 'A'
      1.2. Done = True
      1.3. Notify Task Q
  }

  Task Q {
      2.1. While Not Done {
      2.2. Go To Sleep Mode If Done is False (Atomic)
      2.3. }
      2.4. print 'B'
  }
}
```

Как видите, задание Q «засыпает», если флагу Done еще не присвоили True. Инструкция 2.2 находится внутри цикла, который просто проверяет этот флаг и входит в спящий режим, только если Done равно False. Важное замечание касательно инструкции 2.2: она должна быть атомарной, иначе решение получится неполным и будет иметь ту же проблему.

Если вы уже работали с конкурентными системами, то можете удивиться тому, что инструкция объявлена атомарной. Основная причина — в нашем примере настоящая синхронизация происходит, только если мы четко определим важный участок кода и защитим его с помощью мьютекса. Позже это станет более очевидным, и, рассмотрев дополнительный теоретический материал, мы наконец сможем предложить настоящее практическое решение.

Цикл нужен для того, чтобы уведомление могло прийти не только от задания P, но и от любой другой части системы. В реальных проектах уведомление может также

отправить операционная система или другое задание, но здесь мы хотим, чтобы нас уведомляло лишь задание P.

Таким образом, когда задание получает уведомление и «просыпается», оно должно снова проверить флаг и вернуться в спящий режим, если тот еще не был установлен. Судя по нашему объяснению, это решение должно работать. Однако на самом деле оно неполное и тоже может приводить к частичной взаимной блокировке на компьютерах с многоядерными процессорами. Более подробно об этом будет сказано чуть ниже, в подразделе «Системы с несколькими вычислительными блоками» данного раздела.

Решения, основанные на механизме ожидания/уведомления, обычно реализуются с помощью условных переменных. Однако API POSIX предлагает альтернативные методы, теоретические аспекты которых будут рассмотрены чуть позже в отдельном подразделе.

Ни один механизм синхронизации не обходится без ожидания. Это единственный способ, с помощью которого задания могут оставаться синхронизированными. Одни задания ждут, а другие продолжают работу. Это хороший момент для знакомства с *семафорами* — стандартным механизмом, который останавливает и возобновляет выполнение заданий в конкурентной среде. Ему посвящен следующий подраздел.

Семафоры и мьютексы

Перенесемся в 1960-е годы, когда Эдсгер Дейкстра, известнейший датский математик и ученый в области информационных технологий, вместе со своей командой разработал новую операционную систему *THE Multiprogramming System* или *THE OS* для компьютера Electrologica X8. Система имела уникальную по тем временам архитектуру.

Это произошло менее чем за десять лет до того, как компания Bell Labs создала Unix и позже язык C. Для написания THE OS использовался ассемблер. Это была многозадачная операционная система и имела многоуровневую архитектуру. На самом верхнем уровне находился пользователь, а на самом нижнем — планировщик заданий. В терминах Unix это было эквивалентно совместному размещению *планировщика заданий* и *диспетчера управления процессами* в кольце ядра. Одной из идей, которую Дейкстра придумал вместе со своей командой, чтобы преодолеть некоторые сложности, связанные с конкурентностью, а также разделить ресурсы между разными заданиями, была концепция *семафоров*.

Семафоры — обычные переменные или объекты, которые используются для синхронизации доступа к разделяемому ресурсу. В этом разделе мы подробно их

обсудим и познакомимся со специальной разновидностью семафоров, мьютексами (mutex), которые широко используются в конкурентном программировании и на сегодняшний день существуют почти в любом языке программирования.

Когда задание хочет обратиться к разделяемому ресурсу (это может быть обычная переменная или файл), оно должно сначала проверить семафор и попросить разрешения продолжить работу. Чтобы лучше объяснить данный механизм и его роль, воспользуемся аналогией.

Представьте врача и пациентов, желающие попасть к нему на прием. Допустим, у нас нет системы предварительной записи и пациенты могут приходить когда им вздумается. У нашего врача есть секретарь, который занимается пациентами, размещает их в очереди и пропускает их в кабинет.

Мы исходим из того, что врач может принимать сразу несколько пациентов (не больше определенного количества). Это немного необычно, если судить по нашему повседневному опыту, и потому можете считать, что мы имеем дело с выдающимся врачом. Возможно, пациенты не против совместного приема. В ряде реальных ситуаций семафоры защищают ресурсы, которые могут использоваться множеством сторон. Поэтому попробуйте просто принять такую гипотетическую ситуацию.

Каждый новый пациент должен сначала подойти к секретарю, чтобы зарегистрироваться. У того есть обычный листок бумаги, на котором он записывает имена пациентов. Дальше пациент должен дождаться, когда секретарь его вызовет и пропустит в кабинет. А когда пациент выходит из него, секретарь вычеркивает его имя из своего списка.

По списку секретаря всегда можно понять, какие пациенты находятся на приеме у врача, а какие ждут своей очереди. Когда один пациент покидает кабинет, другой получает возможность войти. Этот процесс продолжается, пока врач не примет всех пациентов.

Теперь перенесем данную аналогию на конкурентную компьютерную систему и посмотрим, как семафоры делают то же самое, что и секретарь.

В нашем примере роль разделяемого ресурса играет врач. К нему может обращаться ряд пациентов, аналогично тому как задания обращаются к разделяемому ресурсу. Секретарь представлен семафором. Вместо бумажного списка у семафора есть очередь заданий, которые ждут разрешения обратиться к ресурсу. Кабинет врача можно считать *критическим участком*.

Критический участок — всего лишь набор инструкций, защищенных семафором. Задание не может в него войти без разрешения от семафора. С другой стороны, задача семафора — защищать критический участок. Каждый раз, пытаясь войти в него, задание должно уведомить об этом соответствующий семафор.

Точно так же этот семафор должен быть уведомлен о том, что задание завершило работу и хочет покинуть критический участок. Как видите, между семафорами и нашей аналогией с врачом есть прямое соответствие. Рассмотрим более программистский пример и попытаемся найти в нем семафоры и другие элементы.

Критический участок должен отвечать определенным критериям. Они будут описаны на оставшихся страницах этой главы.

Пример 14.7, показанный в листинге 14.13, снова состоит из двух заданий, которые пытаются инкрементировать разделяемый счетчик. Мы уже неоднократно обсуждали его в предыдущем подразделе, однако на этот раз будет предложено решение, основанное на семафорах.

Листинг 14.13. Использование семафоров для синхронизации заданий

```
Concurrent System {

  Shared State {
    S : Semaphore which allows only 1 task at a time
    Counter: Integer = 0
  }

  Task P {
    A : Local Integer
      1.1. EnterCriticalSection(S)
      1.2. A = Counter
      1.3. A = A + 1
      1.4. Counter = A
      1.5. LeaveCriticalSection(S)
  }

  Task Q {
    B : Local Integer
      2.1. EnterCriticalSection(S)
      2.2. B = Counter
      2.3. B = B + 2
      2.4. Counter = B
      2.5. LeaveCriticalSection(S)
  }
}
```

В этой системе есть два разных разделяемых состояния: разделяемый семафор `S`, который должен защищать доступ к другой разделяемой переменной `Counter`. В любой момент времени семафор позволяет находиться на защищаемом им критическом участке только одному заданию. Критические участки имеют вид инструкций, размещенных между `EnterCriticalSection(S)` и `EnterCriticalSection(S)`. Как видите, в каждом задании `S` защищает отдельную порцию кода.

Чтобы зайти на критический участок, задание должно выполнить `EnterCriticalSection(S)`. Если в этот момент другое задание находится на собственном критическом участке, то инструкция `EnterCriticalSection(S)` становится блокирующей и не завершается, и потому текущее задание ждет, пока семафор не позволит ему войти.

Инструкция `EnterCriticalSection(S)` может быть по-разному реализована в зависимости от ситуации: как всего лишь холостой цикл или же может перевести ожидающее задание в спящий режим. Второй подход встречается чаще, и задания, ожидающие входа в критический участок, обычно «засыпают».

В нашем примере семафор `S` используется таким образом, что на его критический участок может зайти только одно задание. Но семафоры куда более универсальны; они могут позволить находиться на критических участках сразу нескольким заданиям (количество определяется в момент создания семафора). Семафор, который разрешает входить на критический участок по одному, обычно называют *двоичным семафором* или *мьютексом*. Мьютексы куда более распространены, чем семафоры общего вида, и их всегда можно встретить в конкурентном коде. API POSIX предоставляет как семафоры, так и мьютексы, и вы можете использовать их в зависимости от ситуации.

Термин *«мьютекс»* означает *mutual exclusion* (взаимное исключение). Представьте, что у нас есть два задания, каждое из которых имеет критический участок с доступом к одному и тому же разделяемому ресурсу. Чтобы получить решение на основе взаимного исключения, которое делает невозможным состояние гонки, наши задания должны отвечать следующим требованиям:

- в любой момент времени на критическом участке может находиться только одно из них, а другое должно ждать, пока критический участок не освободится;
- решение не должно быть подвержено взаимной блокировке. Задание, которое останавливается перед критическим участком, должно в конечном счете в него войти. В ряде случаев ограничивается максимальное время ожидания *(период состязания)*;
- задание, находящееся на критическом участке, не может быть убрано с него *преждевременно*, чтобы туда могло войти другое задание. Иными словами, решение должно быть *совместным*, а не *вытесняющим*.

Мьютексы позволяют разработать решение на основе взаимного исключения. Обратите внимание: критические участки тоже должны удовлетворять похожим критериям. Они должны пропускать внутрь по одному заданию и не должны вызывать взаимную блокировку. Следует отметить, что семафоры тоже совместимы с последними двумя требованиями, но позволяют заходить на критический участок сразу нескольким заданиям.

Можно сказать, что взаимное исключение — самый важный аспект конкурентности и доминирующий фактор в различных управляющих механизмах, существующих

на сегодняшний день. Иными словами, в любом методе синхронизации, который вам известен, можно наблюдать последствия использования взаимного исключения на основе мьютексов (а иногда и семафоров).

Семафоры и мьютексы называют *блокируемыми* объектами. Если говорить более формальным языком, то ожидание семафора и вход на критический участок — это эквивалент *блокировки* семафора. С другой стороны, выход с критического участка и обновления семафора — это эквивалент *разблокировки* семафора.

Таким образом, блокировку и разблокировку семафоров можно считать двумя алгоритмами, которые используются в целях ожидания и получения доступа к критическому участку и его освобождения соответственно. Например, *циклическая блокировка* заключается в получении доступа к критическому участку с помощью холостого цикла вокруг семафора, но мы, несомненно, можем применять и другие алгоритмы блокировки и разблокировки. Мы поговорим о них в ходе разработки конкурентных программ с помощью API POSIX в главе 16.

Предыдущее решение, основанное на понятиях блокировки и разблокировки, будет выглядеть следующим образом (листинг 14.14).

Листинг 14.14. Использование операций блокировки и разблокировки для работы с семафорами

```
Concurrent System {

  Shared State {
    S : Semaphore which allows only 1 task at a time
    Counter: Integer = 0
  }

  Task P {
    A : Local Integer
      1.1. Lock(S)
      1.2. A = Counter
      1.3. A = A + 1
      1.4. Counter = A
      1.5. Unlock(S)
  }

  Task Q {
    B : Local Integer
      2.1. Lock(S)
      2.2. B = Counter
      2.3. B = B + 2
      2.4. Counter = B
      2.5. Unlock(S)
  }
}
```

Эта терминология используется в API POSIX, и мы будем применять ее в дальнейшем в наших фрагментах псевдокода.

В завершение этого подраздела дам итоговое определение. Когда несколько заданий хотят зайти на критический участок, они пытаются заблокировать семафор, но только определенное их количество (зависящее от семафора) сможет получить блокировку и выполнить вход. Другие задания будут ждать своей очереди. Процесс ожидания блокировки семафора называется *состязанием*. Чем больше заданий, тем жестче состязание, а его продолжительность определяет, насколько задания замедляют свое выполнение.

Очевидно, что на получение блокировки у заданий, участвующих в состязании, уходит какое-то время и чем больше заданий, тем дольше они ждут входа в критические участки. Время ожидания в этом состоянии обычно называют *периодом состязания*. Он может быть частью нефункциональных требований к конкурентной системе, соблюдение которых необходимо тщательно отслеживать во избежание снижения производительности.

В заключение следует сказать, что мьютексы — основной механизм для синхронизации некоторых конкурентных заданий. Они также присутствуют в API POSIX для работы с потоками и почти в любом языке программирования, поддерживающем конкурентность. Помимо мьютексов, важную роль в удовлетворении определенных критериев за счет ожидания на протяжении неопределенного периода времени играют *условные переменные*.

Условные переменные будут рассмотрены чуть позже, а перед этим необходимо поговорить о барьерах памяти и конкурентных средах с несколькими вычислительными блоками (о многопроцессорных системах или многоядерных процессорах). Этой теме посвящен следующий подраздел.

Системы с несколькими вычислительными блоками

Если ваша компьютерная система имеет всего один вычислительный блок (одноядерный процессор), то задания, пытающиеся обратиться к определенному адресу в основной памяти, всегда получают самое последнее и актуальное значение, даже будь этот адрес закэширован в ядре процессора. Сохранение определенных адресов памяти (даже вместе с изменениями, которые в них вносятся) внутри *локального кэша* ядра — распространенная практика. Она повышает производительность за счет уменьшения количества операций чтения и записи, направленных к основной памяти. При возникновении определенных событий ядро процессора синхронизирует внесенные изменения, сохраненные в кэше, с основной памятью.

Эти локальные кэши существуют, даже если у вас несколько вычислительных блоков. Под таковыми мы понимаем либо многоядерный процессор, либо несколько

процессоров с любым количеством ядер. Заметьте, что каждое ядро имеет собственный локальный кэш.

Таким образом, когда два разных задания выполняются на двух разных ядрах процессора и обращаются к одному и тому же адресу в основной памяти, каждое ядро заносит значение данного адреса в свой локальный кэш. Это значит, если одно из заданий попытается выполнить запись в разделяемый адрес, то изменения проявятся только в его локальном кэше, а не в основной памяти или в кэше других ядер процессора.

Это приводит к множеству разных проблем, и вот почему: когда задание, работающее на другом ядре, попытается прочитать значение из разделяемого адреса памяти, оно не сможет увидеть последние изменения, поскольку чтение будет выполнено из его локального кэша.

Чтобы решить проблему, которая возникает из-за наличия разных локальных кэшей в каждом ядре процессора, используется *протокол когерентности памяти*. В соответствии с ним все задания, работающие на разных ядрах, видят в своих локальных кэшах одно и то же значение, даже если одно из ядер изменило его. То есть можно сказать, что адрес памяти виден всем вычислительным блокам. Соблюдение протокола когерентности памяти делает память *видимой* для всех заданий, выполняющихся в разных вычислительных блоках. Когерентность кэша и видимость памяти — два важных фактора, которые необходимо учитывать в конкурентной системе с несколькими вычислительными блоками.

Вернемся к нашему первому решению примера 14.6, основанному на ожидании/уведомлении, которое мы обсуждали в предыдущих двух разделах. Инвариантное ограничение в этом примере было таким: *сначала выводится A, а затем B*.

В листинге 14.15 показан псевдокод нашего итогового решения, в котором мы использовали механизм ожидания/уведомления, чтобы обеспечить нужный нам порядок выполнения инструкций `print`. Я уже отмечал, что это решение имеет дефекты и может приводить к постсинхронизационным проблемам. Чуть ниже я объясню, как проявляется эта проблема.

Листинг 14.15. Решение, предложенное для примера 14.6 и основанное на методе ожидания/уведомления

```
Concurrent System {

  Shared State {
    Done : Boolean = False
  }

  Task P {
      1.1. print 'A'
```

```
      1.2. Done = True
      1.3. Notify Task Q
  }

  Task Q {
      2.1. While Not Done {
      2.2.     Go To Sleep Mode If Done is False (Atomic)
      2.3. }
      2.4. print 'B'
  }
}
```

Допустим, задания `P` и `Q` работают на разных ядрах процессора. В данном случае в локальном кэше каждого ядра есть запись для разделяемой переменной `Done`. Опять же обратите внимание на то, что инструкция `2.2` объявлена как атомарная. Это допущение необходимо, пока мы не выработаем подходящее решение на основе мьютексов. Представьте вариант чередований, когда `P` выполняет инструкцию `1.2` и уведомляет задание `Q`, которое в этот момент может «спать». Таким образом, задание `P` обновляет значение `Done` в своем локальном кэше, но данное изменение может не попасть в основную память или локальный кэш другого ядра.

Таким образом, у нас нет гарантии того, что мы увидим изменения в основной памяти и в локальном кэше задания `Q`. Следовательно, существует такая вероятность: задание `Q`, получив доступ к процессору и прочитав свой локальный кэш, увидит, что переменная `Done` равна `False`, и перейдет в спящий режим, а задание `P` завершит работу и больше не будет слать уведомления (поскольку уже давно это сделало). В конечном счете `Q` «засыпает» навсегда, в результате чего происходит частичная взаимная блокировка.

Чтобы решить эту проблему, необходимо использовать барьеры памяти. Это инструкции, которые ведут себя подобно барьерам; во время их выполнения (прохождения) все значения, измененные в одном локальном кэше, распространяются по основной памяти и других локальных кэшах. Они становятся видимыми для всех заданий, выполняющихся на других процессорных ядрах. Иными словами, барьеры памяти синхронизируют локальные кэши всех ядер процессора и основную память.

В листинге 14.16 представлено наше полноценное решение. И снова обратите внимание на то, что инструкция `2.3` объявлена как атомарная; это допущение необходимо, пока мы должным образом не решим эту проблему с помощью мьютексов.

Листинг 14.16. Улучшенное решение для примера 14.6 с использованием барьеров памяти

```
Concurrent System {

  Shared State {
    Done : Boolean = False
  }
```

```
  Task P {
      1.1. print 'A'
      1.2. Done = True
      1.3. Memory Barrier
      1.4. Notify Task Q
  }

  Task Q {
      2.1. Do {
      2.2.     Memory Barrier
      2.3.     Go To Sleep Mode If Done is False (Atomic)
      2.4. } While Not Done
      2.5. print 'B'
  }
}
```

Благодаря применению в этом псевдокоде барьеров памяти можно быть уверенными в том, что задание `Q` сможет видеть любые обновления разделяемой переменной `Done`. Можете самостоятельно пройтись по разным вариантам чередования и убедиться, что барьер памяти помогает сделать разделяемую переменную `Done` видимой для задания `Q` и предотвращает любые нежелательные ситуации с частичной взаимной блокировкой.

Стоит отметить, что создание задания, а также блокировка и разблокировка семафора — три разные операции, которые играют роль барьеров памяти и синхронизируют локальные кэши всех процессорных ядер и основную память, распространяя последние изменения, внесенные в разделяемое состояние.

Псевдокод, представленный в листинге 14.17, похож на предыдущее решение, но в нем используются мьютексы. Это поможет нам наконец избавиться от проблемы, из-за которой мы делали инструкцию `Go To Sleep Mode If Done is False` атомарной. Но имейте в виду: мьютексы — семафоры, которые позволяют находиться на критическом участке только одному заданию, и, как с любыми семафорами, их блокировка и разблокировка может служить барьером памяти.

Листинг 14.17. Улучшенное решение для примера 14.6 с использованием мьютексов

```
Concurrent System {

  Shared State {
      Done : Boolean = False
      M : Mutex
  }

  Task P {
      1.1. print 'A'
      1.2. Lock(M)
```

```
      1.3. Done = True
      1.4. Unlock(M)
      1.5. Notify Task Q
  }

  Task Q {
      2.1. Lock(M)
      2.2. While Not Done {
      2.3.     Go To Sleep Mode And Unlock(M) (Atomic)
      2.4.     Lock(M)
      2.5. }
      2.6. Unlock(M)
      2.7. print 'B'
  }
}
```

Инструкции `Lock(M)` и `Unlock(M)` выступают барьерами памяти, гарантируя видимость переменных во всех заданиях. Напомню, что инструкции между `Lock(M)` и `Unlock(M)` в каждом задании считаются критическим участком.

Обратите внимание: когда задание блокирует мьютекс (или семафор), автоматическая разблокировка может произойти в трех случаях:

- задание разблокирует мьютекс с помощью команды `Unlock`;
- когда задание завершается, все заблокированные мьютексы разблокируются;
- когда задание входит в спящий режим, заблокированные мьютексы становятся разблокированными.

Строго говоря, третий пункт этого списка не совсем верен. Если задание хочет «заснуть» на определенный промежуток времени, находясь на критическом участке, защищенном мьютексом, то переход в спящий режим, безусловно, не требует разблокировки данного мьютекса. Вот почему мы объявили инструкцию 2.3 как атомарную и добавили к ней Unlock(M). Чтобы завершить всестороннее рассмотрение этого примера, необходимо затронуть условные переменные, о которых мы поговорим в следующих подразделах.

Таким образом, когда инструкция `2.3` выполняется как единое целое, мьютекс `M`, заблокированный ранее, становится разблокированным. При повторном уведомлении задание заново получает блокировку, используя инструкцию `2.4`, и продолжает работать на критическом участке.

В заключение следует отметить, что задание не может заблокировать мьютекс, который уже подвергла данной операции. Попытка сделать это обычно приводит

к взаимной блокировке. Только *рекурсивный мьютекс* может многократно блокироваться одним и тем же заданием. И пока он заблокирован (неважно, сколько раз), все другие задания, пытающиеся получить его блокировку, будут сами блокироваться. Операции блокировки и разблокировки всегда парные, поэтому если задание дважды заблокировало рекурсивный мьютекс, то и разблокировать его должно два раза.

Итак, мы обсудили и использовали метод ожидания/уведомления в целом ряде примеров. Но, чтобы оценить этот метод по достоинству, необходимо познакомиться с новым понятием: условными переменными. В сочетании с мьютексами они являются основой для реализации управляющих механизмов, которые могут эффективно синхронизировать множество заданий, обращающихся к одному разделяемому ресурсу. Но сначала обсудим еще одно потенциальное решение для примера 14.6.

Циклические блокировки

Прежде чем начинать обсуждение условных переменных и того, как действительно следует реализовывать метод ожидания/уведомления, вернемся немного назад и напишем новое решение для примера 14.6 на основе холостого цикла и мьютексов. Напомню, что наш пример должен *подавать на стандартный вывод сначала A, а затем B*.

В листинге 14.18 показано предложенное решение, в котором используется мьютекс в сочетании с алгоритмом циклической блокировки. Мьютекс служит барьером памяти, поэтому у нас не будет никаких проблем с видимостью памяти; к тому же он фактически синхронизирует задания `P` и `Q` относительно разделяемого флага `Done`.

Листинг 14.18. Решение для примера 14.6 с использованием мьютекса и алгоритмов циклической блокировки

```
Concurrent System {

  Shared State {
      Done : Boolean = False
      M : Mutex
  }

  Task P {
      1.1. print 'A'
      1.2. SpinLock(M)
      1.2. Done = True
      1.3. SpinUnlock(M)
  }
```

```
  Task Q {
      2.1  SpinLock(M)
      2.2. While Not Done {
      2.3.   SpinUnlock(M)
      2.4.   SpinLock(M)
      2.5. }
      2.6. SpinUnlock(M)
      2.4. print 'B'
  }
}
```

Это первое решение, которое можно написать не только на псевдокоде, но и на обычном языке C, задействуя API POSIX для работы с потоками. Ни один из предыдущих листингов нельзя было реализовать в виде настоящей программы; они были либо слишком абстрактными, либо проблемными в определенных условиях, таких как работа в системе с несколькими вычислительными блоками. А данный псевдокод можно перевести на любой язык программирования, поддерживающий конкурентность.

В этом листинге мы используем *циклические блокировки*, которые представляют собой обычные алгоритмы холостого цикла. Каждый раз, когда вы блокируете такой мьютекс, задание входит в холостой цикл и ждет, пока тот не станет доступен, и только потом продолжает работу.

Мне кажется, в приведенном выше листинге нет ничего сложного, если не считать инструкций `2.3` и `2.4` внутри цикла, которые идут одна за другой и зачем-то выполняют блокировку и разблокировку! На самом деле это самая элегантная часть кода. Пока задание `Q` получает доступ к ядру процессора, мьютекс `M` циклически блокируется и разблокируется.

Что, если бы у нас не было инструкций `2.3` и `2.4`? В таком случае инструкция `2.1` блокировала бы мьютекс до выполнения инструкции `2.6`; это значит, задание `P` не имело бы возможности получить доступ к разделяемому флагу `Done`. Эти инструкции блокировки и разблокировки позволяют заданию `P` воспользоваться инструкцией `1.2` и обновить флаг `Done`. В противном случае мьютекс будет все время удерживаться заданием `Q`, и задание `P` никогда не доберется до инструкции `1.2`. Иными словами, в системе произойдет частичная взаимная блокировка. Данный псевдокод демонстрирует прекрасную гармонию операций блокировки/разблокировки, которая позволяет нам элегантно решить проблему с помощью циклических блокировок.

Обратите внимание: циклические блокировки очень распространены в высокопроизводительных системах, в которых переход задания в спящий режим занимает намного больше времени по сравнению с частотой событий, возникающих в системе. При использовании циклических блокировок задания должны быть написаны так, чтобы могли разблокировать мьютекс как можно раньше. Для этого критический

участок должен быть достаточно компактным. В нашем коде критический участок содержит всего одну булеву проверку (условие цикла).

В следующем подразделе мы исследуем условные переменные и их свойства.

Условные переменные

Решения для примера 14.6, представленные в предыдущих подразделах, нельзя реализовать на языке программирования, поскольку мы не знаем, как программно поместить в спящий режим одно задание и уведомить другое. В данном подразделе мы познакомимся с условными переменными. Это новая концепция, с помощью которой мы сможем сделать так, чтобы задание подождало и получило уведомление.

Условными называются обычные переменные (или объекты), которые позволяют перевести задание в спящий режим или оповестить другие задания, чтобы те «проснулись». Обратите внимание: под спящим режимом мы понимаем не задержку на какое-то определенное количество секунд или миллисекунд. Речь идет о том, что задание больше не хочет получать ресурсы процессора. По аналогии с мьютексами, которые защищают критический участок, условные переменные используются для организации *обмена сигналами* между разными заданиями.

Опять же, как и мьютексы, у которых есть операции *блокировки* и *разблокировки*, условные переменные умеют *уведомлять* задания и переводить их в *спящий режим*. У каждого языка программирования своя терминология, и потому спящий режим иногда называют *ожиданием*, а уведомления — *сигналами*. Но суть от этого не меняется.

Условную переменную необходимо использовать в связке с мьютексом. Если этого не сделать, то вашему решению попросту будет не хватать *взаимного исключения*. Помните, что условная переменная должна разделяться между несколькими заданиями, как разделяемый ресурс, иначе будет бесполезной. Вот почему доступ к ней нужно синхронизировать. Это зачастую достигается за счет применения мьютекса, который защищает критические участки. В псевдокоде листинга 14.19 показано, как с помощью условных переменных и мьютексов можно дождаться выполнения определенного условия или просто появления какого-то события. В данном случае мы ждем, когда в примере 14.6 разделяемый флаг `Done` станет равен `True`.

Листинг 14.19. Решение для примера 14.6 с использованием условной переменной

```
Concurrent System {

  Shared State {
      Done : Boolean = False
      CV   : Condition Variable
      M    : Mutex
  }
```

```
  Task P {
        1.1. print 'A'
        1.2. Lock(M)
        1.3. Done = True
        1.4. Notify(CV)
        1.5. Unlock(M)
  }

  Task Q {
        2.1. Lock(M)
        2.2. While Not Done {
        2.3.     Sleep(M, CV)
        2.4. }
        2.5. Unlock(M)
        2.6. print 'B'
  }
}
```

Это решение — пример того, как максимально правильно использовать условные переменные в целях обеспечения строгого порядка выполнения двух инструкций в конкурентной системе. Инструкции `1.4` и `2.3` применяют условную переменную `CV`. Как видите, операция `Sleep` должна знать как об этой переменной, так и о мьютексе `M`, поскольку должна разблокировать его, когда задание `Q` «засыпает», и заблокировать в момент поступления уведомления.

Обратите внимание: после получения уведомления задание `Q` продолжает выполнять свою логику внутри операции `Sleep`, включая повторную блокировку мьютекса `M`. Инструкция `1.4` тоже работает, только после получения блокировки для `M`, иначе получилось бы состояние гонки. Можете выполнить полезное упражнение: переберите все возможные варианты чередования и подумайте, как представленные выше условная переменная и мьютекс будут обеспечивать нужный порядок следования инструкций `1.1` и `2.6`.

И в качестве заключительного определения в этом подразделе отмечу, что связку из мьютекса и условной переменной обычно называют *монитором*. Кроме того, существует одноименный шаблон проектирования, относящийся к конкурентности; он описывает применение представленной выше методики в целях перестановки инструкций в некоторых конкурентных заданиях.

В предыдущих подразделах было показано, как семафоры, мьютексы и условные переменные вместе с алгоритмами блокировки, разблокировки, перевода в спящий режим и уведомления можно использовать для реализации управляющих механизмов, которые обеспечивают строгий порядок выполнения определенных инструкций в разных конкурентных задачах, а также для защиты критических участков. В следующих главах мы воспользуемся этими концепциями в целях написания многопоточных и многопроцессных программ на C. В следующем разделе речь пойдет о поддержке конкурентности в стандарте POSIX, реализованной и доступной во многих Unix-подобных системах.

Конкурентность в POSIX

Как уже объяснялось в предыдущих разделах, поддержку конкурентности или многозадачности предоставляет ядро операционной системы. Не все ядра были конкурентными с момента своего появления, но большинство из них уже поддерживают конкурентность. Приятно отметить, что система Unix обзавелась этой возможностью вскоре после выхода первой версии.

Если помните, в главе 10 я рассказывал, как в POSIX и единой спецификации Unix пытались стандартизировать API, предоставляемый кольцом командной оболочки в Unix-подобных ОС. Конкурентность уже давно является частью этих стандартов и позволяет многим разработчикам писать конкурентные программы для POSIX-совместимых операционных систем. Поддержка конкурентности в POSIX реализована во множестве ОС, таких как Linux и macOS, и получила широкое применение.

В POSIX-совместимых системах конкурентность обычно предоставляется двумя путями. Конкурентная программа может состоять либо из разных процессов (в этом случае ее называют *многопроцессной*), либо из разных потоков выполнения в рамках одного процесса (тогда ее именуют *многопоточной*). В текущем разделе мы обсудим оба метода и сравним их с точки зрения программиста. Но сначала нам нужно узнать больше о внутренностях ядра, поддерживающего конкурентность. Далее будет представлен краткий обзор того, что можно найти в таком ядре.

Ядра с поддержкой конкурентности

Почти все ядра, разрабатываемые и сопровождаемые на сегодняшний день, — многозадачные. Как мы уже знаем, у любого ядра есть *планировщик заданий*, распределяющий вычислительные блоки процессора между множеством активных процессов и потоков, которые в предыдущей главе фигурировали под общим названием «задания».

Прежде чем продолжать, необходимо поговорить о процессах и потоках, а также об их отличиях в контексте конкурентности. При каждом запуске программы создается новый процесс, в котором выполняется ее логика. Процессы изолированы друг от друга; то есть один не имеет доступа к содержимому (например, к памяти) другого.

Похожим образом работают и потоки выполнения, но они находятся в рамках определенного процесса. С их помощью в процессе реализуется конкурентность; она достигается за счет того, что несколько потоков совместно выполняют разные наборы инструкций в конкурентной манере. Отдельно взятый поток нельзя разделить между двумя процессами; он всегда работает локально в том из них, которому принадлежит. Все потоки процесса-владельца имеют доступ к его памяти, работая

с ней как с разделяемым ресурсом. В то же время каждый поток имеет собственный стек, доступный для других потоков того же процесса. Кроме того, и процессы, и потоки могут использовать разделяемые ресурсы центрального процессора, и планировщик заданий в большинстве операционных систем задействует один и тот же алгоритм для распределения между ними процессорных ядер.

Обратите внимание: ведя дискуссию в контексте ядра, я предпочитаю использовать термин *«задание»*, а не *«поток»* или *«процесс»*. Ядро имеет дело с очередью заданий, которые ждут возможности выполнить свои инструкции на вычислительном блоке процессора, и справедливое выделение этого ресурса каждому из них — обязанность планировщика.

В Unix-подобных ядрах термин «задание» обычно относится как к процессам, так и к потокам. На самом деле и те и другие — концепции пользовательского пространства, и они неприменимы в контексте ядра. Поэтому у Unix-подобных ядер есть планировщики, которые пытаются непредвзято управлять доступом к процессору со стороны различных заданий.

В разных ядрах планировщики используют разные стратегии и алгоритмы для управления заданиями. Но большинство из них можно отнести к двум основным категориям:

- совместное планирование;
- вытесняющее планирование.

Совместное (или кооперативное) планирование состоит в том, что задание, которому выделили ядро процессора, должно добровольно его освободить. Этот подход не является *вытесняющим* в том смысле, что в большинстве штатных ситуаций не применяется никаких мер по принудительному освобождению процессорного ядра от задания. Чтобы такие меры были приняты, планировщик должен получить высокоприоритетный *сигнал вытеснения*. В противном случае планировщик и все остальные задания в системе должны ждать, пока активное задание само не освободит ядро процессора. Большинство современных ОС работают по-другому, но иногда все еще встречаются системы, которые применяют совместное планирование в узкоспециализированных областях, таких как *вычисления в режиме реального времени*. Ранние версии macOS и Windows использовали совместное планирование, но в настоящее время задействуют вытесняющий подход.

Вытесняющее планирование — противоположность совместного. Оно позволяет заданию занимать ядро процессора до тех пор, пока его не освободит планировщик. В некоторых разновидностях вытесняющего планирования заданию позволено использовать ресурсы процессора на протяжении определенного времени. Такой подход называется *разделением времени* и является самой популярной стратегией

планирования в современных операционных системах. Промежуток времени, на протяжении которого задание может занимать процессор, имеет разные названия в разных академических источниках: *интервал*, *период* или *квант*.

Существуют разные виды планирования с разделением времени, которые отличаются используемыми в них алгоритмами. Один из самых широко распространенных алгоритмов — *циклический перебор*; он применяется в различных ОС (конечно, с кое-какими модификациями). Он обеспечивает честный доступ к разделяемому ресурсу (в нашем случае это ядро процессора) для всех заданий без исключения.

Несмотря на свою простоту, алгоритм циклического перебора можно модифицировать так, чтобы он поддерживал задания с несколькими уровнями приоритета. Это обычно требуется в современных операционных системах, поскольку существуют определенные виды заданий, которые инициируются самой ОС или ее важными компонентами, и эти задания должны обслуживаться в первую очередь.

Как уже говорилось прежде, конкурентность можно воплотить в программном обеспечении двумя способами. Первый — многопроцессность — заключается в применении *пользовательских процессов* для выполнения параллельных заданий в многозадачной среде. Второй — многопоточность — подразумевает разбиение заданий на параллельные потоки выполнения в рамках одного процесса с помощью *пользовательских потоков*. В крупных программных проектах очень часто можно встретить сочетание этих подходов. Несмотря на то что оба способа делают ПО конкурентным, они отличаются по своей природе на фундаментальном уровне.

В оставшихся двух разделах мы подробно обсудим многопроцессность и многопоточность. Две предстоящие главы посвящены многопоточной разработке в C, а в двух последующих будет рассмотрена многопроцессность.

Многопроцессность

Многопроцессность — просто использование процессов для выполнения конкурентных заданий. Очень хороший пример — стандарт *CGI* (Common Gateway Interface — общий интерфейс шлюза) в веб-серверах. С его помощью веб-серверы запускают новый *процесс интерпретатора* для каждого HTTP-запроса. Таким образом, они могут обслуживать несколько запросов одновременно.

На таких веб-серверах можно встретить множество процессов интерпретатора, запущенных параллельно, каждый из которых обрабатывает разные HTTP-запросы. Так делается для повышения пропускной способности. Поскольку это разные процессы, они изолированы и не могут видеть области памяти друг друга. К счастью, в случае с CGI процессам интерпретатора не нужно общаться или обмениваться данными между собой. Но в других ситуациях это не всегда так.

Существует множество примеров того, как ряду процессов, которые выполняют какие-то конкурентные задания, необходимо делиться важной информацией, чтобы программное обеспечение могло продолжать работу. В данном контексте можно привести как образец инфраструктуру Hadoop. У кластера Hadoop есть много узлов, на каждом из которых запущен ряд процессов, и все эти процессы обеспечивают работу кластера.

Данные процессы должны постоянно обмениваться фрагментами информации, чтобы кластер мог функционировать. Существует множество других распределенных многоузловых систем подобного рода, таких как Gluster, Kafka и криптовалютные сети. Все они опираются на внутреннее взаимодействие и обмен сообщениями между процессами, размещенными на разных узлах.

Когда процессы или потоки не имеют разделяемого состояния, между многопроцессностью и многопоточностью нет принципиальной разницы. Вы, скорее всего, сможете заменить потоки процессами и наоборот. Но при наличии разделяемого состояния разница между процессами, потоками и даже их сочетанием становится огромной. Одно из отличий состоит в том, какие методы синхронизации нам доступны. API для работы с этими механизмами более или менее одинаковые, но работать в многопроцессных средах куда сложнее, и их внутренняя реализация тоже различается, как и способы использования разделяемого состояния. В потоках можно задействовать все методики, которые доступны в процессах, но, помимо этого, им доступна такая роскошь, как возможность разделения состояния с помощью общей области памяти. Вы сами увидите в следующих главах, что это имеет большое значение.

Если немного углубиться в подробности, то у процесса есть приватная память, которая доступна для чтения и записи только ему, поэтому ее не так-то просто использовать для разделения чего бы то ни было между разными процессами. А вот с потоками все намного проще. Все потоки внутри одного процесса имеют доступ к общей памяти, поэтому могут применять ее для хранения разделяемого состояния.

Ниже перечислены механизмы, с помощью которых процессы могут обращаться к разделяемому состоянию. Больше информации о них будет дано в следующих главах.

- *Файловая система*. Это можно считать самым простым способом обмена данными между несколькими процессами. Данный подход очень старый, и его поддерживают почти все операционные системы. В качестве примера можно привести конфигурационные файлы, которые считываются многими процессами в программном проекте. Если один из процессов производит запись в файл, то необходимо использовать методы синхронизации, чтобы предотвратить гонку данных и другие проблемы, связанные с конкурентностью.

- *Отображение файла в память.* Во всех POSIX-совместимых операционных системах и в Microsoft Windows можно использовать области памяти, привязанные к файлам на диске. Эти области можно разделять между разными процессами, делая их доступными для чтения и записи.

  Этот подход очень похож на использование файловой системы, но создает меньше трудностей, связанных с потоковой передачей данных через файловые дескрипторы с помощью специального API. Если содержимое области памяти может быть модифицировано любым процессом, который имеет к нему доступ, то необходимо применять подходящие механизмы синхронизации.

- *Сеть.* Если процессы находятся на разных компьютерах, то их взаимодействие можно реализовать только с помощью сетевой инфраструктуры и API сокетов. Сокеты — замечательная часть стандартов SUS и POSIX, существующая почти в любой операционной системе.

  Эта методика включает в себя бесчисленное множество подробностей, и ей посвящены целые книги. В нее входят различные протоколы, архитектуры, методы обработки потоков данных и многие другие детали. Мы попытаемся частично ее рассмотреть в главе 20, но для разбора всех аспектов IPC поверх сети может понадобиться целая отдельная книга.

- *Сигналы.* Процессы, работающие внутри одной операционной системы, могут слать друг другу сигналы. Обычно эта возможность служит для передачи команд, но подходит и для разделения небольшого состояния (полезных данных). Значение разделяемого состояния можно передавать вместе с сигналом и интерпретировать на стороне процесса-получателя.

- *Разделяемая память.* В POSIX-совместимых операционных системах и в Microsoft Windows ряд процессов может иметь общую область разделяемой памяти. В ней они могут хранить переменные и обмениваться разными значениями. Разделяемая память не защищена от гонки данных, поэтому процессы, которые хотят задействовать ее для хранения своих изменяемых разделяемых состояний, должны применять подходящий механизм синхронизации, чтобы избежать проблем с конкурентностью. Область разделяемой памяти может использоваться сразу несколькими процессами.

- *Каналы.* В POSIX-совместимых операционных системах и в Microsoft Windows существуют однонаправленные каналы. С их помощью можно передавать разделяемое состояние между двумя процессами. Один процесс записывает в канал, а другой из него читает.

  Канал может быть либо именованным, либо анонимным. Обе разновидности имеют свои способы применения. Подробности и примеры будут рассмотрены в главе 19, в которой мы поговорим о различных методах межпроцессного взаимодействия на одном компьютере.

- *Сокеты домена Unix.* В POSIX-совместимых операционных системах, а также с недавних пор в Windows 10 есть поддержка конечной точки взаимодействия

под названием *«Unix-сокеты»*. Процессы, размещенные на одном компьютере и работающие внутри одной ОС, могут использовать *сокеты домена Unix* для передачи информации по полнодуплексному каналу. Они очень похожи на сетевые сокеты, но все данные проходят через ядро, что делает их передачу чрезвычайно быстрой. Несколько процессов могут задействовать один и тот же Unix-сокет для взаимодействия и обмена данными. Эти сокеты могут применяться и в особых ситуациях, таких как передача дескриптора файла между процессами на одном компьютере. Положительной стороной данного подхода является то, что для его использования предусмотрен такой же API, как и для сетевых сокетов.

- *Очередь сообщений*. Этот механизм существует почти в любой операционной системе. Очередь находится в ядре и может использоваться разными процессами для отправки и получения различных сообщений. Процессам не обязательно знать друг о друге; им достаточно лишь иметь доступ к очереди.

  Данный метод используется только для организации взаимодействия процессов на одном компьютере.

- *Переменные среды*. Unix-подобные операционные системы и Microsoft Windows предлагают набор переменных, которые принадлежат самой ОС. Они называются переменными среды, и доступ к ним имеют процессы в рамках одной системы.

Например, этот метод активно используется в реализациях CGI, о которых упоминалось в первом абзаце данного раздела, особенно когда главный процесс вебсервера хочет передать данные HTTP-запроса новому процессу интерпретатора.

Что касается синхронизации множества потоков/процессов, то вы сами увидите: в многопроцессных и многопоточных средах применяются очень похожие API, предоставляемые стандартом POSIX. Различия, скорее всего, заключаются во внутренней реализации мьютексов и условных переменных. Примеры этого будут показаны в следующих главах.

Многопоточность

Многопоточность — это когда для выполнения параллельных заданий в конкурентной среде применяются *пользовательские потоки*. Очень сложно найти реальную программу, состоящую из одного потока; почти все приложения, которые вам будут встречаться, являются многопоточными. Потоки могут существовать только внутри процессов; не бывает такого потока, у которого не было бы процесса-владельца. Каждый процесс содержит по меньшей мере один поток, обычно называемый *главным* или *основным*. Программа, которая выполняет все свои задания в одном потоке, называется *однопоточной*. Все потоки внутри одного процесса имеют доступ к одним и тем же областям памяти. Это значит, нам не нужно изобретать сложные механизмы для обмена данными, как в случае с многопроцессностью.

Ввиду похожести на процессы потоки могут использовать те же методики для разделения или передачи состояния. Поэтому все механизмы для обращения к разделяемому состоянию и обмена данными, перечисленные в предыдущем разделе, можно применять и для потоков. Однако в этом смысле потоки имеют одно преимущество по сравнению с процессами: доступ к общим областям памяти. Следовательно, один из распространенных методов разделения фрагмента данных между рядом потоков состоит в объявлении переменных.

Поскольку у всех потоков есть свой стек, его можно задействовать для хранения разделяемых состояний. Один поток может передать другому указатель на какой-то участок своего стека, и последний сможет легко к нему обратиться ввиду того, что все эти адреса принадлежат стеку процесса. Поток также может легко работать с кучей, принадлежащей процессу, используя ее для хранения своих разделяемых состояний. В следующей главе мы рассмотрим несколько примеров хранения разделяемых переменных в стеке и куче.

Механизмы синхронизации тоже очень напоминают те, которые применяются для процессов. Даже API POSIX ничем не отличается. Это, вероятно, объясняется тем фактом, что POSIX-совместимые операционные системы работают с процессами и потоками почти одним и тем же образом. В следующей главе мы выясним, как с помощью API POSIX объявлять семафоры, мьютексы, условные переменные и другое в многопоточной программе.

В заключение следует отметить: Microsoft Windows не поддерживает API POSIX для управления потоками (pthreads). Данная система имеет собственный API, который позволяет создавать потоки и управлять ими. Он входит в состав системной библиотеки Win32. Мы не станем ее рассматривать в этой книге, но в Интернете ей посвящено множество ресурсов.

Резюме

В этой главе мы обсудили проблемы, с которыми можно столкнуться при разработке конкурентной программы, а также методы, подходящие для их решения. Ниже перечислены основные рассмотренные нами темы.

- Мы обсудили проблемы конкурентности. Всем конкурентным системам присущи определенные недостатки, которые проявляются, когда различные варианты чередований не соответствуют инвариантным ограничениям.
- Мы рассмотрели постсинхронизационные проблемы, возникающие только после некорректного применения механизмов синхронизации.
- Мы исследовали управляющие механизмы, предназначенные для соблюдения инвариантных ограничений.

- Семафоры — ключевой элемент реализации управляющих механизмов. Мьютексы — особая разновидность семафоров, которые пропускают на критический участок только по одному заданию за раз, учитывая условия взаимного исключения.
- Шаблон проектирования «монитор» инкапсулирует мьютексы, а условные переменные можно использовать, когда задание ждет выполнения определенного условия.
- В заключение мы начали знакомство с конкурентной разработкой, кратко изложив суть многопроцессности и многозадачности в стандарте POSIX.

Следующие две главы посвящены многопоточной разработке в POSIX-совместимых операционных системах. В главе 15 речь в основном пойдет о потоках и их выполнении. В главе 16 мы рассмотрим доступные механизмы управления конкурентностью для многопоточных сред. Прочитав эти две главы, вы получите все необходимое для написания многопоточной программы.

15 Многопоточное выполнение

Как уже объяснялось в предыдущей главе, в POSIX-совместимой системе конкурентность можно реализовать в виде либо *многопоточности*, либо *многопроцессности*. Это обширные темы, и, чтобы рассмотреть их должным образом, мы разделили их на четыре главы:

- *многопоточный подход* обсуждается в этой и следующей главах;
- *многопроцессный подход* рассматривается в главах 17 и 18.

В данной главе мы проанализируем устройство потоков и API для их создания и управления. В следующей главе пройдемся по механизмам управления конкурентностью в многопоточной среде и посмотрим, как в них решаются проблемы конкурентности.

Основная идея многопроцессности состоит в том, что для реализации конкурентности логика программного обеспечения разбивается на конкурентные процессы, в результате чего получается многопроцессное ПО. Учитывая отличия между многопоточностью и многопроцессностью, мы решили вынести обсуждение последней в две отдельные главы.

Для сравнения, многопоточность, которой посвящены эта и следующая главы, действует в рамках одного процесса. Ввиду данной основополагающей особенности потоков мы начинаем именно с них.

В предыдущей главе мы кратко рассмотрели различия и сходства между многопоточностью и многопроцессностью. Здесь же сосредоточимся на многопоточности и на том, как ее следует использовать, чтобы организовать беспроблемную работу нескольких потоков выполнения в одном процессе.

- Мы начнем данную главу с раздела, в котором речь пойдет о потоках в пространствах *пользователя* и *ядра*, а также о самых важных свойствах потоков. Эти свойства помогут нам лучше понять многопоточную среду.
- Затем перейдем к следующему разделу, посвященному основам программирования с помощью библиотеки для работы с потоками POSIX (сокращенно *pthread*). Это основная стандартная библиотека, которая позволяет разрабатывать конкурентные программы в POSIX-системах, но это вовсе не значит, что

системы, несовместимы с POSIX, не поддерживают конкурентность. Последние, в число которых входит Microsoft Windows, предоставляют собственные API для написания конкурентных программ. Библиотека pthread поддерживает как потоки, так и процессы, хотя в данной главе мы сосредоточимся на потоках. Вы увидите, как эта библиотека позволяет создавать новые потоки и управлять ими в дальнейшем.

- В качестве следующего шага будет показан пример кода на языке C с применением библиотеки pthread, который демонстрирует состояние гонки и гонку данных. Он послужит основой для обсуждения *синхронизации потоков* в главе 16.

Чтобы вы могли должным образом ориентироваться в материале, который относится к многопоточному подходу, я настоятельно рекомендую дочитать данную главу до конца и только затем переходить к главе 16. Это вызвано тем, что темы, рассматриваемые ниже, будут постоянно упоминаться при обсуждении синхронизации потоков, о которой пойдет речь в следующей главе.

Прежде чем продолжать, обратите внимание на то, что в данной главе мы будем рассматривать только основы работы с библиотекой потоков POSIX. Многие ее захватывающие особенности выходят за рамки нашей книги, поэтому я рекомендую вам самостоятельно исследовать ее более детально и основательно попрактиковаться в написании кода, чтобы вам было комфортно работать с ней. Более углубленные примеры использования pthread будут продемонстрированы в оставшихся главах.

А пока подробно разберем концепцию потоков, начиная с краткого обзора всего того, что нам о них известно. Это послужит отправной точкой для дальнейшего знакомства с ключевыми аспектами потоков, которые мы будем рассматривать на оставшихся страницах данной главы.

Потоки

В предыдущей главе мы обсуждали потоки в контексте многопоточного подхода, который можно использовать при написании конкурентных программ в POSIX-совместимых операционных системах.

В данном разделе я проведу краткий обзор всего того, что вам уже должно быть известно о потоках. Прибавлю к этому новую информацию, относящуюся к темам, которые будут обсуждаться позже. Помните: данный материал послужит фундаментом для дальнейшей разработки многопоточных программ.

Инициатор и постоянный владелец любого потока — процесс. Невозможно создать разделяемый поток или передать владением им другому процессу. У каждого

процесса есть по меньше мере один поток, который называют *главным* или *основным*. В программе на языке C в рамках главного потока выполняется функция `main`.

Все потоки имеют общий *идентификатор процесса* (proces ID, PID). Если воспользоваться такой утилитой, как `top` или `htop`, то можно легко увидеть, что все потоки имеют один и тот же PID и все они под ним сгруппированы. Более того, каждый поток процесса наследует все его атрибуты, включая ID группы, пользователя, текущий рабочий каталог и обработчики сигналов. Например, поток имеет тот же текущий рабочий каталог, что и его процесс-владелец.

У каждого потока есть уникальный идентификатор (thread ID, TID). С его помощью можно отправлять сигналы конкретному потоку или отслеживать поток во время отладки. Как вы вскоре увидите, идентификатор POSIX-потока доступен в переменной `pthread_t`. Помимо этого, у каждого потока есть собственная выделенная ему маска сигналов, позволяющая фильтровать сигналы, которые он будет получать.

Все потоки внутри одного процесса имеют доступ ко всем *файловым дескрипторам*, открытым любым потоком в данном процессе. Следовательно, каждый поток может читать и изменять ресурсы, на которые ссылаются эти дескрипторы. То же самое касается *дескрипторов* открытых *сокетов*. Больше о дескрипторах сокетов и файлов вы узнаете в следующих главах.

К потокам можно применять все те методы разделения и передачи состояния, с которыми мы познакомились в главе 14. Обратите внимание: наличие разделяемого состояния в общедоступном месте (таком как база данных) — не то же самое, что, к примеру, передача его по сети; в обоих случаях используются разные виды методов межпроцессного взаимодействия. Мы еще вернемся к этому в будущих главах.

Ниже перечислены методы, с помощью которых потоки могут разделять или передавать состояние в POSIX-совместимой системе:

- память процесса-владельца (сегменты данных, стека и кучи). Этот метод применим *только* к потокам, но не к процессам;
- файловая система;
- отображение файлов в память;
- сеть (с использованием сетевых сокетов);
- передача сигналов между потоками;
- разделяемая память;
- POSIX-каналы;
- сокеты домена Unix;
- очереди сообщений POSIX;
- переменные среды.

Продолжая тему свойств потоков, следует сказать, что все потоки внутри одного процесса могут использовать его память для хранения разделяемого состояния. Это самый распространенный способ разделения ресурсов между несколькими потоками. Для этого обычно применяется сегмент кучи процесса.

Время жизни потока зависит от времени жизни владеющего им процесса. Когда процесс *принудительно завершается*, вместе с ним удаляются все его потоки.

Когда заканчивает работу главный поток, процесс немедленно завершается. Но если у него есть другие *отсоединенные* потоки, то он подождет, пока все они не доработают до конца. Отделенные потоки будут рассмотрены в рамках процесса создания потоков в стандарте POSIX.

В качестве процесса, создающего поток, может выступать ядро. Но это может быть и пользовательский процесс, созданный в пользовательском пространстве. Если речь идет о ядре, то мы получаем так называемый *поток уровня ядра* (или просто *поток ядра*); в противном случае создается *поток пользовательского уровня*. Потоки ядра обычно выполняют важную логику, и в связи с этим имеют повышенный приоритет по сравнению с пользовательскими. Например, драйвер устройства может применять поток ядра в целях ожидания сигнала от оборудования.

По аналогии с тем, как пользовательские потоки имеют доступ к одной и той же области памяти, потоки ядра могут обращаться к пространству памяти этого ядра — то есть ко всем его процедурам и модулям.

На страницах нашей книги основное внимание будет уделяться пользовательским потокам. Причина в том, что API для работы с ними предоставляется стандартом POSIX. А вот для создания потоков ядра и управления ими стандартного интерфейса нет; соответствующие механизмы реализуются конкретным ядром.

Создание и управление потоками ядра выходит за рамки данной книги. Таким образом, начиная с этого момента, термин *«поток»* будет относиться только к пользовательским потокам.

Пользователь не может создать поток напрямую. Он должен сначала породить процесс, и только потом в главном потоке этого процесса можно будет инициировать еще один поток. Стоит отметить, что потоки могут создаваться лишь другими потоками.

Что касается структуры памяти потоков, то у каждого из них есть стек, который можно считать приватной областью памяти, выделенной специально для него. Однако на практике к данной области могут обращаться и другие потоки внутри того же процесса; им для этого достаточно иметь указатель, ссылающийся на нее.

Следует помнить, что все эти сегменты стека являются частью общего пространства памяти процесса и доступны для всех его потоков.

Если говорить о методах синхронизации, то управляющие механизмы, с помощью которых синхронизируются процессы, можно использовать и для потоков. В число инструментов, позволяющих синхронизировать как потоки, так и процессы, входят семафоры, мьютексы и условные переменные.

Программу, в которой потоки синхронизированы и больше не наблюдается гонка данных или состояния гонки, называют *потокобезопасной*. Этот же термин применим к библиотеке или набору функций, которые можно легко использовать в многопоточной программе, не опасаясь появления новых проблем с конкурентностью. Мы как программисты должны стремиться к написанию потокобезопасного кода.

По ссылке http://man7.org/linux/man-pages/man7/pthreads.7.html вы можете найти информацию о POSIX-потоках и их общих характеристиках. Материал посвящен интерфейсу POSIX для работы с потоками в реализации NPTL. Обсуждение ведется в контексте среды Linux, но большая его часть применима и к другим Unix-подобным операционным системам.

Мы рассмотрели некоторые основополагающие идеи и свойства потоков выполнения, чтобы облегчить ориентирование в следующих разделах. Позже, когда мы будем обсуждать различные многопоточные примеры, вы увидите многие из описанных свойств в действии.

В следующем разделе представлен первый пример кода с созданием POSIX-потока. В нем показаны лишь основы работы с потоками в стандарте POSIX, поэтому у вас не должно возникнуть никаких трудностей. Позже эти основы послужат фундаментом для более сложных тем.

POSIX-потоки

Этот раздел посвящен API POSIX для работы с потоками, более известному как *библиотека pthread*. Данный API очень важен, поскольку служит основным механизмом создания потоков и управления ими в POSIX-совместимых операционных системах.

В таких ОС, как Microsoft Windows, которые несовместимы с POSIX, для этого обычно предусмотрен другой API, и информацию о нем можно найти в документации соответствующей системы. Например, в случае с Microsoft Windows работа с потоками реализована в Windows API (Win32 API). Вот ссылка на документацию от Microsoft, посвященную данному API: https://docs.microsoft.com/en-us/windows/desktop/procthread/process-and-thread-functions.

В C11 есть унифицированный API для работы с потоками, и в идеале он должен быть доступен для написания программ в любых системах, независимо от их совместимости с POSIX. Поддержка такого универсального API крайне желательна, однако на сегодняшний день отсутствует в различных реализациях стандарта C, таких как glibc.

Возвращаясь к нашей теме, отмечу, что библиотека pthread — всего лишь набор *заголовков* и *функций*, пригодных для написания многопоточных программ в POSIX-совместимых операционных системах. У каждой ОС есть своя реализация этой библиотеки, которая может отличаться от любых других, но в целом все реализации предоставляют доступ к одному и тому же интерфейсу (API).

В качестве общеизвестного примера можно привести *библиотеку потоков POSIX* (Native POSIX Threading Library, NPTL) — главную реализацию pthread для операционной системы Linux.

Согласно API pthread, чтобы получить доступ ко всем функциям для работы с потоками, достаточно подключить заголовок `pthread.h`. У этой библиотеки также есть некоторые расширения, доступные только при подключении `semaphore.h`. Например, одно из расширений предназначено сугубо для операций с семафорами, такими как создание, инициализация, удаление и т. д.

Библиотека потоков POSIX предоставляет следующие возможности (я уже подробно описывал их в предыдущих главах, поэтому они должны быть вам знакомы):

- управление потоками, включая их создание, присоединение и отсоединение;
- мьютексы;
- семафоры;
- условные переменные;
- различные виды блокировок, включая циклически и рекурсивные.

Объяснение перечисленных выше возможностей нужно начинать с префикса `pthread_`. Он присутствует во всех функциях в pthread, кроме семафоров, которые изначально не входили в библиотеку потоков POSIX и были добавлены в нее позже в качестве расширения. В этом случае используется префикс `sem_`.

В следующих разделах данной главы я продемонстрирую некоторые из этих возможностей в процессе написания многопоточной программы. Для начала научимся создавать POSIX-потоки, чтобы выполнять код конкурентно по отношению к главному потоку. Вы познакомитесь с функциями `pthread_create` и `pthread_join`, которые принадлежат к основному API и используются для *создания* и *присоединения* потоков соответственно.

Порождение POSIX-потоков

В предыдущих главах мы прошлись по всем фундаментальным понятиям, таким как чередования, блокировки, мьютексы и условные переменные, а в этой уже успели познакомиться с концепцией POSIX-потоков. Теперь можно приступать к написанию кода.

Первым делом нужно создать POSIX-поток. В этом разделе я покажу, как с помощью API POSIX создавать новые потоки внутри процесса. В примере 15.1 (листинг 15.1) создается поток, который выполняет такое простое задание, как вывод строки.

Листинг 15.1. Порождение нового POSIX-потока (ExtremeC_examples_chapter15_1.c)

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

// Стандартный заголовок POSIX для использования библиотеки pthread
#include <pthread.h>

// Эта функция содержит логику, которая должна выполняться
// как тело отдельного потока
void* thread_body(void* arg) {
  printf("Hello from first thread!\n");
  return NULL;
}

int main(int argc, char** argv) {

  // Обработчик потоков
  pthread_t thread;

  // Создаем новый поток
  int result = pthread_create(&thread, NULL, thread_body, NULL);
  // Если создание потока завершилось неудачно
  if (result) {
    printf("Thread could not be created. Error number: %d\n",
           result);
    exit(1);
  }

  // Ждем, пока созданный поток не завершит работу
  result = pthread_join(thread, NULL);
  // Если присоединение потока оказалось неудачным
  if (result) {
    printf("The thread could not be joined. Error number: %d\n",
           result);
    exit(2);
  }
  return 0;
}
```

Приведенный выше код создает новый POSIX-поток. Это первый пример с двумя потоками в данной книге. Все предыдущие примеры были однопоточными, и весь код всегда работал внутри главного потока.

Разберем этот листинг. Вверху мы подключили новый заголовочный файл, `pthread.h`. Это стандартный заголовок, который дает доступ ко всем возможностям библиотеки pthread. Он нужен для того, чтобы мы могли воспользоваться объявлениями функций `pthread_create` и `pthread_join`.

Непосредственно перед `main` мы объявили новую функцию: `thread_body`. У нее специфическая сигнатура. Она принимает один указатель `void*` и возвращает другой. Напомню, что `void*` — обобщенный тип указателя, который может представлять указатели любого другого типа, такого как `int*` или `double*`.

Это самая общая сигнатура, которую только может иметь функция в C. Стандарт POSIX требует, чтобы ее соблюдали все функции, желающие быть *компаньонами* потока (то есть использоваться в качестве его логики). Вот почему мы объявили функцию `thread_body` таким образом.

Функция main — часть логики главного потока; она выполняется во время его создания. Это значит, перед функцией main может выполняться какой-то другой код.

Вернемся к нашему примеру. Первой инструкцией в функции `main` служит объявление переменной типа `pthread_t`. Это ссылка на поток, и на момент объявления она не указывает ни на что конкретное. Иными словами, данная переменная еще не содержит ID никакого действующего потока. Только после успешного создания нового потока корректная ссылка на него будет присвоена этой переменной.

На самом деле после создания потока ссылка содержит его идентификатор. Последний представляет поток на уровне операционной системы, а ссылка делает то же самое в программе. В большинстве случаев значение, хранящееся в ссылке на поток, совпадает с его ID. Чтобы получить свой идентификатор и сослаться на себя, поток может воспользоваться переменной `pthread_t`. Извлечь соответствующую ссылку можно с помощью функции `pthread_self`. Работа с этими функциями будет продемонстрирована в примерах ниже.

Поток создается при вызове функции `pthread_create`. Как видите, мы передали адрес переменной `thread`, чтобы ей присвоили ссылку на только что созданный поток (то есть его идентификатор).

Второй аргумент определяет атрибуты потока. Каждый поток обладает такими атрибутами, как *размер стека*, *адрес стека* и *состояние* (*присоединен* или *отсоединен*), которые можно подготовить перед его порождением.

Позже вы увидите другие примеры конфигурации этих атрибутов и узнаете, как они влияют на поведение потока. Если в качестве второго аргумента передать `NULL`, то новый поток будет использовать для своих атрибутов значения по умолчанию. Именно так мы поступили в приведенном выше коде.

Третьим аргументом, переданным в `pthread_create`, был указатель на *функцию-компаньон* потока, которая содержит его логику. В нашем коде логика потока определена в функции `thread_body`, поэтому мы передали ее адрес, чтобы привязаться к переменной-ссылке `thread`.

Четвертый и последний аргумент — ввод для логики потока, который в нашем случае равен `NULL`. Это значит, мы не хотим ничего передавать нашей функции. Следовательно, во время выполнения потока параметр `arg` в `thread_body` тоже будет равен `NULL`. В примерах, приведенных в следующем разделе, вы увидите, как данной функции можно передавать параметры, отличные от `NULL`.

Все функции в библиотеке pthread, включая `pthread_create`, в случае успешного выполнения должны возвращать ноль. Следовательно, любое ненулевое значение говорит о том, что функция отработала неудачно, и нам возвращается *номер ошибки*.

Обратите внимание: создание потока с помощью `pthread_create` вовсе не означает, что его логика начнет немедленно выполняться. Это зависит от планирования, и вы не можете предсказать, когда новый поток получит доступ к одному из ядер процессора и сможет начать выполнение.

Создав поток, мы его присоединяем. Но что именно это значит? Как уже объяснялось, каждый процесс изначально имеет всего один, *главный поток*, родителем которого выступает процесс-владелец. Сам этот поток играет роль родителя для всех остальных потоков. Обычно процесс завершается вместе с главным потоком, а вслед за ним немедленно прекращают работу все другие активные и «спящие» потоки.

Таким образом, если новый поток еще не начал выполняться (поскольку не получил доступа к процессору), а родительский процесс тем временем завершился (неважно, по какой причине), он будет удален еще до выполнения своей первой инструкции. Поэтому главный поток должен присоединить второй поток, чтобы подождать, пока тот не завершит работу.

Поток становится завершенным только после возвращения функции-компаньона. В предыдущем примере порожденный поток завершается, когда функция-компаньон `thread_body` возвращает `NULL`. Затем главный поток, заблокированный вызовом `pthread_join`, освобождается и получает возможность продолжить работу, что в конечном счете приводит к успешному завершению программы.

Не присоедини мы новый поток, он вряд ли получил бы шанс выполниться. Как уже объяснялось ранее, это вызвано тем фактом, что главный поток завершается еще до начала выполнения в порожденном нами потоке.

Кроме того, следует помнить, что для выполнения потока его недостаточно просто создать. До того как он получит доступ к ядру процессора и начнет работу, может пройти какое-то время. И если процесс успеет завершиться к этому моменту, то новый поток не получит возможности успешно выполниться.

Итак, мы прошлись по коду примера 15.1. В терминале 15.1 показан результат его выполнения.

Терминал 15.1. Сборка и запуск примера 15.1

```
$ gcc ExtremeC_examples_chapter15_1.c -o ex15_1.out -lpthread
$ ./ex15_1.out
Hello from first thread!
$
```

Как видите, для компиляции необходимо добавить параметр `-lpthread`. Это делается для того, чтобы скомпоновать нашу программу с имеющейся реализацией библиотеки pthread. На некоторых платформах, таких как macOS, при компоновке программы можно обойтись и без параметра `-lpthread`. Но я настоятельно рекомендую указывать его при компоновке любого кода, использующего библиотеку pthread. Это необходимо для того, чтобы ваши *сборочные скрипты* работали на любой платформе и при сборке проектов на языке C у вас не возникало никаких проблем с совместимостью.

Поток, который можно присоединить, называют *соединяемым* (joinable). Все потоки таковы по умолчанию. Их противоположность, *отсоединенные* потоки, невозможно присоединить.

В примере 15.1 мы могли не присоединять порожденный нами поток, а сделать его отсоединенным. Этим мы бы дали процессу понять, что, прежде чем завершаться, он должен подождать, пока не отработает отсоединенный поток. Обратите внимание: в таком случае процесс может продолжать работу после завершения главного потока.

В заключительном примере данного раздела мы попробуем переписать предыдущий код, используя отсоединенные потоки (листинг 15.2). Вместо того чтобы присоединять новый поток, функция `main` делает его отсоединенным. Благодаря этому процесс завершится только после окончания второго потока, несмотря на то что главный поток уже закончился.

Листинг 15.2. Пример 15.1 с порождением отсоединенного потока (ExtremeC_examples_chapter15_1_2.c)

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

// Стандартный заголовок POSIX для использования библиотеки pthread
#include <pthread.h>
```

```
// Эта функция содержит логику, которая должна выполняться
// как тело отдельного потока
void* thread_body(void* arg) {
  printf("Hello from first thread!\n");
  return NULL;
}

int main(int argc, char** argv) {

  // Обработчик потоков
  pthread_t thread;

  // Создаем новый поток
  int result = pthread_create(&thread, NULL, thread_body, NULL);
  // Если создание потока завершилось неудачно
  if (result) {
    printf("Thread could not be created. Error number: %d\n",
           result);
    exit(1);
  }

  // Отсоединяем поток
  result = pthread_detach(thread);
  // Если отсоединение потока оказалось неудачным
  if (result) {
    printf("Thread could not be detached. Error number: %d\n",
           result);
    exit(2);
  }

  // Выходим из главного потока
  pthread_exit(NULL);

  return 0;
}
```

Вывод этого кода идентичен тому, в котором использовались соединяемые потоки. Единственное отличие — в том, как мы управляем созданным нами потоком.

Сразу после создания новый поток отсоединяется от главного. Вслед за этим главный поток завершает работу. Инструкция `pthread_exit(NULL)` нужна для того, чтобы процесс дождался завершения другого, отсоединенного потока. Не выполни мы разъединение, процесс завершился бы вместе с главным потоком.

Отсоединенное состояние — один из атрибутов потока, который можно установить перед его созданием и который позволяет его отсоединить. Это еще один способ создания новых отсоединенных потоков в дополнении к вызову pthread_detach из соединяемого потока. Разница в том, что данный атрибут позволяет сделать поток отсоединенным с самого начала.

В следующем разделе мы рассмотрим первый пример, демонстрирующий состояние гонки. Для его написания мы будем использовать все функции, с которыми познакомились в данном разделе. Поэтому у вас будет шанс еще раз их обсудить в разных сценариях.

Пример состояния гонки

Для нашего второго примера мы возьмем более проблемную ситуацию. В примере 15.2, показанном в листинге 15.3, видно, как происходят чередования и что на практике невозможно достоверно предсказать итоговый вывод конкурентной системы ввиду ее недетерминированной природы. В этом примере представлена программа, которая почти одновременно создает три потока, выводящие на экран разные строки.

В результате выполнения кода, представленного ниже, три разных потока выводят строки, но делают это в непредсказуемом порядке. Будь у нас инвариантное ограничение (см. предыдущую главу), согласно которому строки в выводе должны быть упорядочены определенным образом, наш код нарушил бы его. Основная причина этого заключается в непредсказуемых чередованиях. Рассмотрим листинг 15.3.

Листинг 15.3. Пример 15.2, подающий на вывод три разные строки (ExtremeC_examples_chapter15_2.c)

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

// Стандартный заголовок POSIX для использования библиотеки pthread
#include <pthread.h>

void* thread_body(void* arg) {
  char* str = (char*)arg;
  printf("%s\n", str);
  return NULL;
}

int main(int argc, char** argv) {

  // Обработчики потоков
  pthread_t thread1;
  pthread_t thread2;
  pthread_t thread3;

  // Создаем новые потоки
  int result1 = pthread_create(&thread1, NULL,
          thread_body, "Apple");
  int result2 = pthread_create(&thread2, NULL,
          thread_body, "Orange");
```

```
  int result3 = pthread_create(&thread3, NULL,
          thread_body, "Lemon");

  if (result1 || result2 || result3) {
    printf("The threads could not be created.\n");
    exit(1);
  }

  // Ждем, пока потоки не завершат работу
  result1 = pthread_join(thread1, NULL);
  result2 = pthread_join(thread2, NULL);
  result3 = pthread_join(thread3, NULL);

  if (result1 || result2 || result3) {
    printf("The threads could not be joined.\n");
    exit(2);
  }
  return 0;
}
```

Данный код очень похож на пример 15.1, только вместо одного потока создает три. Здесь в каждом потоке используется одна и та же функция-компаньон.

Как видите, мы передали функции `pthread_create` четвертый аргумент, а не `NULL`, как в примере 15.1. Все наши аргументы доступны в функции-компаньоне потока, `thread_body`, через параметр `arg`, который является обобщенным указателем.

Внутри функции `thread_body` поток приводит обобщенный указатель `arg` к типу `char*` и выводит строку с помощью `printf`, начиная с этого адреса. Вот как потоку можно передавать аргументы. Их размер неважен, поскольку мы передаем лишь указатель.

Если у вас есть несколько значений, которые следует передать потоку в момент его создания, то вы можете разместить их в структуре и передать указатель на структурную переменную с нужными вам полями. Я покажу, как это делается, в следующей главе.

Потенциальная возможность передать потоку указатель говорит о том, что у новых потоков должен быть доступ к той же области памяти, к которой может обращаться главный поток. Однако этот доступ не ограничен определенным сегментом или областью памяти процесса-владельца; все потоки процесса могут свободно работать с его сегментами стека, кучи, исполняемого кода и данных.

Если взять пример 15.2 и выполнить его несколько раз, то окажется, что порядок, в котором выводятся строки, может варьироваться, хотя содержимое самих строк остается неизменным.

В терминале 15.2 показан процесс компиляции примера 15.2 и его вывод после трех последовательных запусков.

Терминал 15.2. Выполнение примера 15.2 три раза; мы можем наблюдать имеющееся состояние гонки и различные чередования

```
$ gcc ExtremeC_examples_chapter15_2.c -o ex15_2.out -lpthread
$ ./ex15_2.out
Apple
Orange
Lemon
$ ./ex15_2.out
Orange
Apple
Lemon
$ ./ex15_2.out
Apple
Orange
Lemon
$
```

Получить вариант чередований, в котором первый и второй потоки выводят свои строки раньше третьего, довольно легко. Намного сложнее получить чередования, при которых третий поток выводит свои строки вместе с первым или вторым. Конечно, это когда-нибудь случится, но с низкой долей вероятности. Чтобы получить такой вариант, вам, возможно, придется выполнить данный пример много раз. Это может потребовать терпения.

Приведенный выше код не является *потокобезопасным*. Это важный термин; многопоточную программу можно считать потокобезопасной, исключительно когда согласно имеющимся инвариантным ограничениям у нее нет состояния гонки. У нашего кода оно есть, поэтому он не потокобезопасный. В такой ситуации мы должны обеспечить потоковую безопасность за счет использования подходящих управляющих механизмов, которые мы рассмотрим в следующей главе.

Как видно в выводе нашего примера, между символами в строках `Apple` и `Orange` нет никаких чередований. Если бы они там были, то мы бы увидели вывод наподобие представленного в терминале 15.3.

Терминал 15.3. Гипотетический вывод, который нельзя получить в примере 15.2

```
$ ./ex15_2.out
AppOrle
Ange
Lemon
$
```

Это иллюстрирует тот факт, что функция `printf` является потокобезопасной. То есть независимо от варианта чередований, когда один поток занимается выводом строки, экземпляры `printf` в других потоках не могут ничего вывести.

Кроме того, в предыдущем коде функция-компаньон `thread_body` была выполнена три раза в контексте трех разных потоков. В предыдущих главах, еще до рассмотрения многопоточных примеров, все наши функции выполнялись в контексте главного потока. Начиная с этого момента, каждый вызов функции будет происходить в определенном потоке (не обязательно главном).

Два потока не могут инициировать один и тот же вызов функции. Причина этого очевидна: каждому вызову необходимо создать *стековый фрейм*, который должен быть помещен на вершину стека какого-то определенного потока, а два разных потока имеют два разных сегмента стека. Таким образом, отдельно взятый вызов функции может быть инициирован только одним потоком. Иными словами, два потока могут вызвать одну и ту же функцию по отдельности, и в результате получится два отдельных вызова, но они не могут разделять один и тот же вызов.

Следует отметить, что указатель, который передается потоку, не должен быть *висячим*, иначе могут возникнуть серьезные проблемы с памятью, которые будет сложно отследить. Напомню: висячий указатель ссылается на адрес в памяти, по которому не выделена переменная. Если точнее, то в прошлом там могли находиться переменная или массив, но в момент использования указателя этот участок памяти уже освобожден.

В представленном выше коде мы передали каждому из трех потоков по одному строковому литералу. Поскольку память, необходимая для хранения этих литералов, выделяется в сегменте данных, а не в куче или в стеке, их адреса никогда не освобождаются и потому указатель `arg` никогда не станет висячим.

Наш код можно легко переписать таким образом, чтобы указатели становились висячими. В листинге 15.4 показан тот же пример, но с висячими указателями. Вскоре вы увидите, что это приводит к нарушению работы памяти.

Листинг 15.4. Пример 15.2, в котором литералы выделяются из стека главного потока (ExtremeC_examples_chapter15_2_1.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Стандартный заголовок POSIX для использования библиотеки pthread
#include <pthread.h>

void* thread_body(void* arg) {
  char* str = (char*)arg;
  printf("%s\n", str);
```

```
  return NULL;
}

int main(int argc, char** argv) {

  // Обработчики потоков
  pthread_t thread1;
  pthread_t thread2;
  pthread_t thread3;

  char str1[8], str2[8], str3[8];
  strcpy(str1, "Apple");
  strcpy(str2, "Orange");
  strcpy(str3, "Lemon");

  // Создаем новые потоки
  int result1 = pthread_create(&thread1, NULL, thread_body, str1);
  int result2 = pthread_create(&thread2, NULL, thread_body, str2);
  int result3 = pthread_create(&thread3, NULL, thread_body, str3);

  if (result1 || result2 || result3) {
    printf("The threads could not be created.\n");
    exit(1);
  }

  // Отсоединяем потоки
  result1 = pthread_detach(thread1);
  result2 = pthread_detach(thread2);
  result3 = pthread_detach(thread3);

  if (result1 || result2 || result3) {
    printf("The threads could not be detached.\n");
    exit(2);
  }

  // Теперь строки становятся освобожденными
  pthread_exit(NULL);

  return 0;
}
```

Этот код почти идентичен примеру 15.2, если не считать двух отличий.

Прежде всего, указатели, передаваемые в потоки, ссылаются не на строковые литералы, размещенные в сегменте данных, а на массивы символов, выделенные из стека главного потока. Эти массивы были объявлены и затем инициализированы с помощью строковых литералов в рамках функции `main`.

Необходимо помнить, что строковые литералы по-прежнему находятся в сегменте данных, однако функция `strcpy` инициализирует объявленные нами массивы, используя те же значения.

Второе отличие связано с тем, как ведет себя главный поток. В предыдущем листинге он присоединяет новые потоки, а в этом отсоединяет их и сразу же завершается. В результате массивы, объявленные на вершине стека главного потока, будут удалены, и в некоторых вариантах чередований другие потоки могут обратиться к этим освобожденным участкам памяти. Следовательно, в ряде ситуаций указатели, передаваемые потокам, могут становиться висячими.

Отсутствие сбоев, висячих указателей и в целом проблем, связанных с памятью, всегда можно считать частью инвариантных ограничений программы. Поэтому конкурентная система, в некоторых вариантах чередований имеющая проблемы с висячими указателями, определенно страдает от серьезного состояния гонки.

Для выявления висячих указателей необходимо использовать *профилировщик памяти*. Если вы предпочитаете более простой подход, то можете выполнить свою программу несколько раз и дождаться возникновения сбоя. Но это потребует определенного везения; заметьте, что в данном примере неполадки так и не проявились.

Чтобы выявить некорректное поведение в этом примере, мы воспользуемся профилировщиком памяти `valgrind`. Как вы помните, мы применяли его в главах 4 и 5 для поиска *утечек памяти*. Здесь же мы с его помощью попытаемся найти место, в котором происходит некорректное обращение к памяти.

Необходимо помнить, что работа с висячим указателем и доступ к его содержимому не всегда приводит к сбою. Это особенно касается представленного выше кода, в котором строки размещаются на вершине стека, принадлежащего потоку.

Пока выполняются другие потоки, сегмент стека остается в том же состоянии, в котором пребывал в момент завершения главного потока, поэтому, несмотря на взаимную блокировку массивов `str1`, `str2` и `str3` при выходе из функции `main`, у вас все равно будет доступ к строкам. Иными словами, в C или C++ среда выполнения не следит за тем, является ли указатель висячим; она просто продвигается по цепочке инструкций.

Если указатель висячий, а память, на которую он ссылается, изменилась, то могут произойти такие нежелательные вещи, как сбои или логические ошибки. Но если эта память остается *нетронутой*, то использование висячих указателей может не вызывать никаких проблем. Это очень опасная ситуация, которую сложно отследить.

Скажу коротко: возможность доступа к области памяти через висячий указатель вовсе не означает, что нам позволено работать с этой областью. Вот почему мы должны использовать профилировщик `valgrind`, который сообщит нам о таком некорректном доступе к памяти.

Мы скомпилируем программу и дважды запустим ее с помощью `valgrind`. При первом выполнении не происходит ничего плохого, а во время второго `valgrind` сигнализирует о некорректном обращении к памяти.

В терминале 15.4 показан первый прогон.

Терминал 15.4. Первое выполнение примера 15.2 с помощью профилировщика valgrind

```
$ gcc -g ExtremeC_examples_chapter15_2_1.c -o ex15_2_1.out -lpthread
$ valgrind ./ex15_2_1.out
==1842== Memcheck, a memory error detector
==1842== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==1842== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==1842== Command: ./ex15_2_1.out
==1842==
Orange
Apple
Lemon
==1842==
==1842== HEAP SUMMARY:
==1842==     in use at exit: 0 bytes in 0 blocks
==1842==   total heap usage: 9 allocs, 9 frees, 3,534 bytes allocated
==1842==
==1842== All heap blocks were freed -- no leaks are possible
==1842==
==1842== For counts of detected and suppressed errors, rerun with: -v
==1842== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
$
```

В ходе второй попытки `valgrind` сообщает о некоторых проблемах с доступом к памяти (обратите внимание: здесь представлен неполный вывод; можете сами выполнить эту команду, чтобы просмотреть его целиком) (терминал 15.5).

Терминал 15.5. Второе выполнение примера 15.2 с помощью профилировщика valgrind

```
$ valgrind ./ex15_2_1.out
==1854== Memcheck, a memory error detector
==1854== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==1854== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==1854== Command: ./ex15_2_1.out
==1854==
Apple
Lemon
==1854== Thread 4:
==1854== Conditional jump or move depends on uninitialised value(s)
==1854==    at 0x50E6A65: _IO_file_xsputn@@GLIBC_2.2.5 (fileops.c:1241)
==1854==    by 0x50DBA8E: puts (ioputs.c:40)
==1854==    by 0x1087C9: thread_body (ExtremeC_examples_chapter15_2_1.c:17)
==1854==    by 0x4E436DA: start_thread (pthread_create.c:463)
==1854==    by 0x517C88E: clone (clone.S:95)
```

```
==1854==
...
==1854==
==1854== Syscall param write(buf) points to uninitialised byte(s)
==1854==    at 0x516B187: write (write.c:27)
==1854==    by 0x50E61BC: _IO_file_write@@GLIBC_2.2.5 (fileops.c:1203)
==1854==    by 0x50E7F50: new_do_write (fileops.c:457)
==1854==    by 0x50E7F50: _IO_do_write@@GLIBC_2.2.5 (fileops.c:433)
==1854==    by 0x50E8402: _IO_file_overflow@@GLIBC_2.2.5 (fileops.c:798)
==1854==    by 0x50DBB61: puts (ioputs.c:41)
==1854==    by 0x1087C9: thread_body (ExtremeC_examples_chapter15_2_1.c:17)
==1854==    by 0x4E436DA: start_thread (pthread_create.c:463)
==1854==    by 0x517C88E: clone (clone.S:95)
...
==1854==
Orange
==1854==
==1854== HEAP SUMMARY:
==1854==     in use at exit: 272 bytes in 1 blocks
==1854==   total heap usage: 9 allocs, 8 frees, 3,534 bytes allocated
==1854==
==1854== LEAK SUMMARY:
==1854==    definitely lost: 0 bytes in 0 blocks
==1854==    indirectly lost: 0 bytes in 0 blocks
==1854==      possibly lost: 272 bytes in 1 blocks
==1854==    still reachable: 0 bytes in 0 blocks
==1854==         suppressed: 0 bytes in 0 blocks
==1854== Rerun with --leak-check=full to see details of leaked memory
==1854==
==1854== For counts of detected and suppressed errors, rerun with: -v
==1854== Use --track-origins=yes to see where uninitialised values come from
==1854== ERROR SUMMARY: 13 errors from 3 contexts (suppressed: 0 from 0)
$
```

Как видите, первое выполнение прошло хорошо, без каких-либо проблем с доступом к памяти, хотя упомянутое выше состояние гонки по-прежнему присутствует. Однако в ходе второго выполнения, когда один из потоков пытается обратиться к строке `Orange`, на которую указывает `str2`, возникают затруднения.

А произошло следующее: указатель, переданный второму потоку, становится висячим. В выводе, представленном выше, можно явно видеть, что трассировка стека указывает на строчку внутри функции `thread_body`, в которой находится инструкция `printf`. Обратите внимание: в действительности трассировка стека указывает на функцию `puts`, поскольку наш компилятор C подставляет ее вместо `printf`. В этом выводе также можно заметить, что системный вызов `write` использует указатель с именем `buf`, ссылающийся на область памяти, которая еще *не выделена* и *не инициализирована*.

В этом примере профилировщик `valgrind` не делает вывод о том, является ли указатель висячим. Он просто сообщает о неправильном доступе к памяти.

Перед сообщением об ошибке, которое относится к неправильному доступу, выводится строка `Orange`, хотя попытка ее чтения является некорректной. Это лишь доказывает то, насколько легко можно усложнить код, выполняемый в конкурентной манере.

В этом разделе было наглядно показано, насколько просто написать код, не являющийся потокобезопасным. Далее рассмотрим еще один интересный пример, в котором возникает гонка данных. В нем будет проиллюстрирован более сложный способ использования библиотеки pthread и ее различных функций.

Пример гонки данных

В примере 15.3 (листинг 15.5) продемонстрирована гонка данных. В предыдущем примере у нас не было разделяемого состояния. Теперь же у нас будет переменная, общая для двух потоков.

Инвариантное ограничение в данном примере состоит в обеспечении *целостности* разделяемого состояния; прибавьте к этому все остальные очевидные ограничения, такие как отсутствие сбоев, некорректного доступа к памяти и т. д. Иными словами, способ вывода данных неважен. Главное, чтобы в ситуации, когда один поток изменил значение разделяемой переменной, другой поток, которому это значение неизвестно, не стал его изменять. Вот что мы имеем в виду под целостностью данных.

Листинг 15.5. Пример 15.3 с двумя потоками, которые работают с одной и той же разделяемой переменной (ExtremeC_examples_chapter15_3.c)

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

// Стандартный заголовок POSIX для использования библиотеки pthread
#include <pthread.h>

void* thread_body_1(void* arg) {
  // Получаем указатель на разделяемую переменную
  int* shared_var_ptr = (int*)arg;
  // Инкрементируем разделяемую переменную на 1, выполняя запись
  // непосредственно по ее адресу в памяти
  (*shared_var_ptr)++;
  printf("%d\n", *shared_var_ptr);
  return NULL;
}

void* thread_body_2(void* arg) {
  // Получаем указатель на разделяемую переменную
  int* shared_var_ptr = (int*)arg;
  // Инкрементируем разделяемую переменную на 2, выполняя запись
  // непосредственно по ее адресу в памяти
```

```
  *shared_var_ptr += 2;
  printf("%d\n", *shared_var_ptr);
  return NULL;
}

int main(int argc, char** argv) {

  // Разделяемая переменная
  int shared_var = 0;

  // Обработчики потоков
  pthread_t thread1;
  pthread_t thread2;

  // Создаем новые потоки
  int result1 = pthread_create(&thread1, NULL,
          thread_body_1, &shared_var);
  int result2 = pthread_create(&thread2, NULL,
          thread_body_2, &shared_var);

  if (result1 || result2) {
    printf("The threads could not be created.\n");
    exit(1);
  }

  // Ждем, пока потоки не завершат работу
  result1 = pthread_join(thread1, NULL);
  result2 = pthread_join(thread2, NULL);

  if (result1 || result2) {
    printf("The threads could not be joined.\n");
    exit(2);
  }
  return 0;
}
```

Разделяемое состояние было объявлено в первой строчке функции `main`. В этом примере мы имеем дело с одной целочисленной переменной, выделенной в стеке главного потока, однако в настоящих приложениях все может быть намного сложнее. Начальное значение данной переменной равно нулю, и каждый поток его увеличивает, выполняя запись в соответствующую область памяти.

В данном примере нет локальной переменной для хранения копии разделяемого значения в каждом потоке. Однако вы должны с осторожностью подходить к операциям инкрементирования, поскольку они не *атомарны* и, следовательно, могут испытывать разные варианты чередований. Мы уже рассматривали данную ситуацию в предыдущей главе.

Каждый поток может менять значение разделяемой переменной с помощью указателя, который он принимает внутри своей функции-компаньона через

аргумент `arg`. В обоих вызовах `pthread_create` видно, что четвертым аргументом передается переменная `shared_var`.

Стоит отметить, что в этих потоках указатель никогда не становится висячим, поскольку главный поток их присоединяет и не завершается, пока они не отработают.

В терминале 15.6 показан вывод нескольких прогонов предыдущего кода, которые приводят к разным вариантам чередований. Как вы помните, мы хотим обеспечить целостность разделяемой переменной `shared_var`.

Итак, исходя из логики, определенной в `thread_body_1` и `thread_body_2`, нам подходят только два варианта вывода: `1 3` и `2 3`.

Терминал 15.6. Несколько выполнений примера 15.3, в результате которых можно наблюдать нарушение целостности разделяемой переменной

```
$ gcc ExtremeC_examples_chapter15_3.c -o ex15_3.out -lpthread
$ ./ex15_3.out
1
3
$
...
...
...
$ ./ex15_3.out
3
1
$
...
...
...
$ ./ex15_3.out
1
2
$
```

Как видите, в последнем прогоне не выполняется условие, относящееся к целостности разделяемой переменной.

Первый поток, функцией-компаньоном которого выступает `thread_body_1`, читает значение разделяемой переменной, которое равно `0`.

Второй поток (с функцией-компаньоном `thread_body_2`) тоже читает данную переменную и также получает `0`. На этот момент оба потока пытаются инкрементировать значение разделяемой переменной и сразу же его вывести. Это нарушает целостность данных, поскольку в то время как один поток работает с разделяемым состоянием, у другого не должно быть возможности в него записывать.

Как уже объяснялось выше, это явный случай гонки данных, а именно по `shared_var`.

При самостоятельном выполнении примера 15.3 наберитесь терпения и дождитесь вывода 1 2. На это может потребоваться 100 прогонов! Я мог наблюдать гонку по данным как в macOS, так и в Linux.

Избавиться от гонки данных можно, воспользовавшись управляющим механизмом, таким как семафор или мьютекс, чтобы синхронизировать доступ к разделяемой переменной. В следующей главе мы реализуем это, добавив мьютекс в приведенный выше код.

Резюме

Эта глава стала нашим первым шагом на пути к написанию многопоточной программы на языке C с использованием библиотеки потоков POSIX. Здесь мы:

- прошлись по основам библиотеки потоков POSIX, которая является основным инструментом для написания многопоточных приложений в POSIX-совместимых системах;
- исследовали различные свойства потоков и структуру их памяти;
- затронули механизмы, с помощью которых потоки могут общаться и разделять состояние;
- выяснили, почему производить обмен данными и взаимодействовать лучше всего с помощью областей памяти, доступных всем потокам внутри одного процесса;
- поговорили о потоках ядра и пользовательских потоках, а также о различиях между ними;
- узнали, что такое соединяемые и отсоединенные потоки и чем они различаются с точки зрения работы;
- рассмотрели использование функций `pthread_create` и `pthread_join`, а также аргументы, которые они принимают;
- рассмотрели состояние гонки и гонку данных на примере реального кода на языке C; вдобавок увидели, что использование висячих указателей может привести к серьезным проблемам с памятью и в конечном счете стать причиной сбоя или логической ошибки.

В следующей главе мы продолжим обсуждать многопоточность и рассмотрим проблемы, связанные с конкурентностью, а также механизмы для их предотвращения и решения.

16 Синхронизация потоков

В предыдущей главе мы рассмотрели создание и управление POSIX-потоками. Кроме того, были продемонстрированы две самые распространенные проблемы с конкурентностью: состояние гонки и гонка данных.

В этой главе мы завершим обсуждение многопоточного программирования с использованием библиотеки потоков POSIX и научимся управлять несколькими потоками.

Как вы помните из главы 14, проблемы, связанные с конкурентностью, на самом деле таковыми не являются; скорее, это следствие неотъемлемых свойств конкурентной системы. Поэтому при работе с конкурентными системами вы, скорее всего, будете сталкиваться с ними.

В предыдущей главе вы могли убедиться в том, что эти проблемы возникают и при использовании библиотеки потоков POSIX. В примерах 15.2 и 15.3 были показаны состояние гонки и гонка данных. Мы будем применять их в качестве отправной точки для внедрения механизмов синхронизации, предоставляемых библиотекой pthread, чтобы синхронизировать несколько разных потоков.

Мы рассмотрим такие темы, как:

- использование POSIX-мьютексов для защиты критических участков, которые обращаются к разделяемому ресурсу;
- ожидание выполнения определенного условия с помощью условных переменных POSIX;
- применение различных видов блокировок в сочетании с мьютексами и условными переменными;
- использование барьеров POSIX и синхронизация с их помощью ряда потоков;
- концепция семафоров и ее аналог в библиотеке pthread: POSIX-семафор. Вы увидите, что мьютексы — это всего лишь двоичные семафоры;
- структура памяти потока и ее влияние на видимость памяти в многоядерной системе.

Начнем эту главу с общего обзора средств управления конкурентностью. В следующих разделах вы познакомитесь с инструментами и понятиями, необходимыми для написания предсказуемых многопоточных программ.

Управление конкурентностью в POSIX

В этом разделе мы рассмотрим управляющие механизмы, доступные в библиотеке pthread. Семафоры, мьютексы и условные переменные в сочетании с разного рода блокировками используются в разных комбинациях в целях обеспечения детерминированности в многопоточных программах. Начнем с POSIX-мьютексов.

POSIX-мьютексы

Мьютексы, доступные в библиотеке pthread, можно применять для синхронизации как процессов, так и потоков. В этом подразделе мы будем использовать их в многопоточной программе на языке C, чтобы синхронизировать ряд потоков.

Напомню: мьютекс — это семафор, который позволяет заходить на критический участок только одному потоку за раз. В целом семафоры допускают нахождение на критическом участке нескольких потоков.

Мьютексы также называют двоичными семафорами, поскольку они являются разновидностью семафора с двумя возможными состояниями.

Начнем этот подраздел с устранения гонки данных, которая наблюдалась в примере 15.3 предыдущей главы. Воспользуемся для этого POSIX-мьютексом. Мьютекс позволяет заходить на критический участок и выполнять там операции чтения и записи разделяемой переменной только одному потоку за раз. Таким образом, он гарантирует целостность разделяемой переменной. В коде листинга 16.1 представлено решение проблемы с гонкой данных.

Листинг 16.1. Использование POSIX-мьютекса для решения проблемы с гонкой данных, обнаруженной в примере 15.3 (ExtremeC_examples_chapter15_3_mutex.c)

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

// Стандартный заголовок POSIX для использования библиотеки pthread
#include <pthread.h>

// Объект мьютекса для синхронизации доступа
// к разделяемому состоянию
pthread_mutex_t mtx;
```

```
void* thread_body_1(void* arg) {
  // Получаем указатель на разделяемую переменную
  int* shared_var_ptr = (int*)arg;

  // Критический участок
  pthread_mutex_lock(&mtx);
  (*shared_var_ptr)++;
  printf("%d\n", *shared_var_ptr);
  pthread_mutex_unlock(&mtx);

  return NULL;
}

void* thread_body_2(void* arg) {
  int* shared_var_ptr = (int*)arg;

  // Критический участок
  pthread_mutex_lock(&mtx);
  *shared_var_ptr += 2;
  printf("%d\n", *shared_var_ptr);
  pthread_mutex_unlock(&mtx);

  return NULL;
}

int main(int argc, char** argv) {

  // Разделяемая переменная
  int shared_var = 0;

  // Обработчики потоков
  pthread_t thread1;
  pthread_t thread2;

  // Инициализирует мьютекс и его внутренние ресурсы
  pthread_mutex_init(&mtx, NULL);

  // Создаем новые потоки
  int result1 = pthread_create(&thread1, NULL,
          thread_body_1, &shared_var);
  int result2 = pthread_create(&thread2, NULL,
          thread_body_2, &shared_var);

  if (result1 || result2) {
    printf("The threads could not be created.\n");
    exit(1);
  }

  // Ждем, пока потоки не завершат работу
  result1 = pthread_join(thread1, NULL);
  result2 = pthread_join(thread2, NULL);
```

```
  if (result1 || result2) {
    printf("The threads could not be joined.\n");
    exit(2);
  }

  pthread_mutex_destroy(&mtx);

  return 0;
}
```

Если скомпилировать этот код и выполнить его какое угодно количество раз, то вывод всегда будет либо `1 3`, либо `2 3`. Это достигается за счет использования POSIX-мьютекса для синхронизации критических участков.

В начале файла мы объявили глобальный объект POSIX-мьютекса, `mtx`. Затем внутри функции `main` инициализировали его с помощью атрибутов по умолчанию, используя функцию `pthread_mutex_init`. Второй аргумент, равный `NULL`, мог бы содержать видоизмененные атрибуты, указанные программистом. В следующих подразделах вы увидите пример того, как их устанавливать.

В обоих потоках мьютекс используется для защиты критических участков, размещенных между выражениями `pthread_mutex_lock(&mtx)` и `pthread_mutex_unlock(&mtx)`.

Наконец, прежде чем покинуть функцию `main`, мы уничтожаем объект мьютекса.

Первая пара выражений `pthread_mutex_lock(&mtx)` и `pthread_mutex_unlock(&mtx)` в функции-компаньоне `thread_body_1` составляет критический участок первого потока. Соответственно, вторая пара в функции-компаньоне `thread_body_2` составляет критический участок второго потока. Оба участка защищены мьютексом, и в любой момент времени на каждом из них может находиться только один поток, а остальные должны ждать снаружи, пока участок не будет освобожден.

Заходя на критический участок, поток блокирует мьютекс, и остальные потоки должны остановиться перед выражением `pthread_mutex_lock(&mtx)` и подождать, пока мьютекс не будет разблокирован снова.

Поток, ожидающий разблокирования мьютекса, по умолчанию входит в спящий режим. Но что, если вместо этого нам хочется использовать *холостой цикл*? В таком случае пригодится *циклическая блокировка*. Ниже перечислены функции, которыми можно полностью заменить все операции по работе с мьютексами. К счастью, в pthread применяется согласованная схема именования функций.

Нам доступны следующие типы и функции, относящиеся к циклическим блокировкам:

- `pthread_spin_t` — тип, используемый для создания объекта циклической блокировки. Аналог `pthread_mutex_t`;

- `pthread_spin_init` — инициализирует объект циклической блокировки. Аналог функции `pthread_mutex_init`;
- `pthread_spin_destroy` — аналог функции `pthread_mutex_destory`;
- `pthread_spin_lock` — аналог функции `pthread_mutex_lock`;
- `pthread_spin_unlock` — аналог функции `pthread_mutex_unlock`.

Как видите, знакомые нам типы и функции, относящиеся к мьютексам, можно легко заменить типами и функциями для работы с циклическими блокировками. В результате при ожидании освобождения объекта мьютекса используется холостой цикл, что меняет поведение программы.

В этом подразделе вы познакомились с POSIX-мьютексами и увидели, как с их помощью можно избавиться от гонки данных. Далее мы рассмотрим процесс ожидания определенного события, использующий условную переменную. Мы попытаемся устранить состояние гонки, возникшее в примере 15.2, но при этом внесем некоторые изменения в оригинальный код.

Условные переменные POSIX

Как вы помните из предыдущей главы, в примере 15.2 возникло состояние гонки. Ниже мы рассмотрим пример 16.1, новый, очень похожий, но более подходящий для использования условной переменной. Он состоит не из трех (как в примере 15.2), а из двух потоков, которые должны выводить символы `A` и `B`. Мы хотим, чтобы они это делали в определенном порядке: сначала `A`, а потом `B`.

Наше инвариантное ограничение состоит в том, чтобы *увидеть в выводе сначала A, а затем B* (плюс целостность всех разделяемых состояний, отсутствие некорректного доступа к памяти, висячих указателей, сбоев и другие очевидные ограничения). В коде листинга 16.2 показано, как этот пример можно реализовать на языке C, задействуя условную переменную.

Листинг 16.2. Использование условной переменной POSIX для обеспечения определенного порядка выполнения двух потоков (ExtremeC_examples_chapter16_1_cv.c)

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

// Стандартный заголовок POSIX для использования библиотеки pthread
#include <pthread.h>

#define TRUE  1
#define FALSE 0

typedef unsigned int bool_t;
```

```
// Структура для хранения всех переменных, относящихся
// к разделяемому состоянию
typedef struct {
  // Флаг, показывающий, был выведен символ 'A' или нет
  bool_t          done;
  // Объект мьютекса для защиты критического участка
  pthread_mutex_t mtx;
  // Условная переменная для синхронизации двух потоков
  pthread_cond_t  cv;
} shared_state_t;

// Инициализирует члены объекта shared_state_t
void shared_state_init(shared_state_t *shared_state) {
  shared_state->done = FALSE;
  pthread_mutex_init(&shared_state->mtx, NULL);
  pthread_cond_init(&shared_state->cv, NULL);
}

// Уничтожает члены объекта shared_state_t
void shared_state_destroy(shared_state_t *shared_state) {
  pthread_mutex_destroy(&shared_state->mtx);
  pthread_cond_destroy(&shared_state->cv);
}

void* thread_body_1(void* arg) {
  shared_state_t* ss = (shared_state_t*)arg;
  pthread_mutex_lock(&ss->mtx);
  printf("A\n");
  ss->done = TRUE;
  // Шлем сигнал потокам, ожидающим условной переменной
  pthread_cond_signal(&ss->cv);
  pthread_mutex_unlock(&ss->mtx);
  return NULL;
}

void* thread_body_2(void* arg) {
  shared_state_t* ss = (shared_state_t*)arg;
  pthread_mutex_lock(&ss->mtx);
  // Ждем, пока флаг не станет равным TRUE
  while (!ss->done) {
    // Ждем условную переменную
    pthread_cond_wait(&ss->cv, &ss->mtx);
  }
  printf("B\n");
  pthread_mutex_unlock(&ss->mtx);
  return NULL;
}

int main(int argc, char** argv) {

  // Разделяемое состояние
  shared_state_t shared_state;
```

```
  // Инициализируем разделяемое состояние
  shared_state_init(&shared_state);

  // Обработчики потоков
  pthread_t thread1;
  pthread_t thread2;

  // Создаем новые потоки
  int result1 =
    pthread_create(&thread1, NULL, thread_body_1, &shared_state);
  int result2 =
    pthread_create(&thread2, NULL, thread_body_2, &shared_state);

  if (result1 || result2) {
    printf("The threads could not be created.\n");
    exit(1);
  }

  // Ждем, пока потоки не завершат работу
  result1 = pthread_join(thread1, NULL);
  result2 = pthread_join(thread2, NULL);

  if (result1 || result2) {
    printf("The threads could not be joined.\n");
    exit(2);
  }

  // Уничтожаем разделяемое состояние, освобождаем мьютекс
  // и объекты условных переменных
  shared_state_destroy(&shared_state);

  return 0;
}
```

Здесь используется удобная структура, в которую инкапсулируются разделяемый мьютекс, разделяемая условная переменная и разделяемый флаг. Заметьте, что в каждый поток можно передать только по одному указателю. Поэтому нам пришлось поместить все наши разделяемые ресурсы в одну структурную переменную.

В качестве второго типа (после `bool_t`) мы определили `shared_state_t` (листинг 16.3).

Листинг 16.3. Размещение всех разделяемых переменных из примера 16.1 в одной структуре

```
typedef struct {
  bool_t          done;
  pthread_mutex_t mtx;
  pthread_cond_t  cv;
} shared_state_t;
```

После типов мы определили две функции, предназначенные для инициализации и удаления экземпляров `shared_state_t`. Их можно считать *конструктором* и *деструктором* типа `shared_state_t` соответственно. Больше информации о конструкторах и деструкторах можно найти в главе 6.

Вот как мы используем условную переменную. Поток, который ее *ждет* (пребывая в состоянии *«сна»*), рано или поздно получает уведомление и «просыпается». Более того, поток может *уведомлять* (или *«пробуждать»*) любые другие потоки, ожидающие (в состоянии «сна») условную переменную. Все эти операции *должны* быть защищены мьютексом, и именно поэтому мьютексы всегда необходимо применять в сочетании с условными переменными.

Именно это мы и сделали в коде, приведенном выше. В нашем объекте разделяемого состояния есть условная переменная и сопутствующий мьютекс, который должен ее защищать. Следует еще раз подчеркнуть, что условную переменную нужно использовать только на критических участках, защищенных сопутствующим мьютексом.

Так что же происходит в этом коде? Поток, который должен вывести `A`, пытается заблокировать мьютекс `mtx`, используя указатель на объект разделяемого состояния. Получив блокировки, данный поток выводит `A`, устанавливает флаг `done` и вызывает функцию `pthread_cond_signal`, чтобы уведомить другой поток, который в это время может ждать условную переменную `cv`.

С другой стороны, если второй поток станет активным до того, как первый успеет вывести `A`, то сам попытается получить блокировку мьютекса `mtx` и в случае успеха проверит флаг `done`. Если флаг равен `false`, то это просто означает, что первый поток еще не зашел на критический участок (в противном случае был бы равен `true`). Следовательно, второй поток подождет условную переменную и немедленно освободит процессор, вызвав функцию `pthread_cond_wait`.

Обязательно обратите внимание на то, что во время ожидания условной переменной связанный с ней мьютекс освобождается и другой поток может продолжить работу. Вдобавок при активации и выходе из состояния ожидания необходимо снова получить мьютекс. Познакомиться с условными переменными поближе можно, проанализировав другие потенциальные варианты чередований.

Функцию pthread_cond_signal можно использовать для уведомления только единственного потока. Если вы хотите уведомить все потоки, которые ждут условную переменную, то для этого предусмотрена функция pthread_cond_broadcast. Вскоре вы увидите соответствующий пример.

Но почему мы использовали для проверки флага `done` цикл `while`, а не обычное выражение `if`? Дело в том, что второй поток могут уведомить другие источники, а не только первый поток. В таких случаях, если потоку при выходе из состояния ожидания и активации удается заблокировать мьютекс, то он может проверить

условие цикла и в случае невыполнения снова подождать. Ожидание условной переменной внутри цикла — приемлемый метод.

Приведенное выше решение соответствует и ограничению, связанному с видимостью памяти. Как уже объяснялось в предыдущих главах, все операции блокировки и разблокировки приводят к согласованию памяти между разными ядрами процессора; поэтому значения разных закэшированных флагов `done` всегда совпадают и являются самыми актуальными.

Проблему с состоянием гонки, которую мы наблюдали в примерах 15.2 и 16.1 (при отсутствии управляющих механизмов), также можно решить с помощью POSIX-барьеров. В следующем подразделе мы поговорим о них и перепишем пример 16.1, задействуя другой подход.

POSIX-барьеры

POSIX-барьеры используют другой метод синхронизации разных потоков. Представьте группу людей, которые планируют выполнять какие-то задачи параллельно; на определенных этапах им нужно встречаться, пересматривать планы и продолжать работу. То же самое может происходить с потоками (и даже процессами). Одни потоки выполняют задания быстрее, а другие — медленнее. Мы можем предусмотреть контрольную точку (или точку встречи), при достижении которой все потоки должны остановиться и подождать, пока к ним не присоединятся остальные. Эти контрольные точки можно имитировать с помощью *POSIX-барьеров*.

Код листинга 16.4 использует барьеры, чтобы решить проблемы, которые мы наблюдали в примере 16.1. Напомню, что там у нас было два потока. Один из них должен был выводить `A`, а другой — `B`, и мы хотели, чтобы, независимо от чередований, буква `A` всегда выводилась первой.

Листинг 16.4. Решение для примера 16.1 с применением POSIX-барьеров (ExtremeC_examples_chapter16_1_barrier.c)

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

#include <pthread.h>

// Объект барьера
pthread_barrier_t barrier;

void* thread_body_1(void* arg) {
  printf("A\n");
  // Ждем присоединения другого потока
  pthread_barrier_wait(&barrier);
  return NULL;
}
```

```
void* thread_body_2(void* arg) {
  // Ждем присоединения другого потока
  pthread_barrier_wait(&barrier);
  printf("B\n");
  return NULL;
}

int main(int argc, char** argv) {

  // Инициализируем объект барьера
  pthread_barrier_init(&barrier, NULL, 2);

  // Обработчики потоков
  pthread_t thread1;
  pthread_t thread2;

  // Создаем новые потоки
  int result1 = pthread_create(&thread1, NULL,
          thread_body_1, NULL);
  int result2 = pthread_create(&thread2, NULL,
          thread_body_2, NULL);

  if (result1 || result2) {
    printf("The threads could not be created.\n");
    exit(1);
  }

  // Ждем, пока потоки не завершат работу
  result1 = pthread_join(thread1, NULL);
  result2 = pthread_join(thread2, NULL);

  if (result1 || result2) {
    printf("The threads could not be joined.\n");
    exit(2);
  }

  // Уничтожаем объект барьера
  pthread_barrier_destroy(&barrier);

  return 0;
}
```

Как видите, код значительно уменьшился по сравнению с тем, в котором был основан на условных переменных. POSIX-барьеры позволяют легко синхронизировать разные потоки на определенных этапах выполнения.

Вначале мы объявили глобальный объект барьера с типом `pthread_barrier_t`. Затем внутри функции `main` инициализировали этот объект с помощью функции `pthread_barrier_init`.

Первый аргумент — указатель на объект барьера. Второй аргумент — пользовательские атрибуты данного объекта. Мы передаем `NULL`, поэтому атрибуты в объекте барьера будут инициализированы с помощью значений по умолчанию. Важную роль играет третий аргумент: это количество потоков, которые должны инициировать ожидание барьера путем вызова функции `pthread_barrier_wait`, и их выполнение продолжится только после их освобождения.

В приведенном выше примере данный аргумент равен 2. Поэтому, когда два потока начнут ждать объект барьера, оба они будут разблокированы и смогут продолжить работу. Оставшийся код мало чем отличается от примеров, которые уже обсуждались в предыдущей главе.

Объект барьера можно реализовать с помощью мьютекса и условной переменной, подобно тому что было показано в предыдущем разделе. На самом деле POSIX-совместимые операционные системы не предоставляют такого механизма, как барьер, в своих интерфейсах системных вызовов, и большинство реализаций основаны на мьютексах и условных переменных.

В сущности, вот почему некоторые ОС, такие как macOS, не предоставляют реализации POSIX-барьеров. Показанный выше код нельзя скомпилировать на компьютере с macOS ввиду отсутствия соответствующих функций. Код проверялся в Linux и FreeBSD; он работает в обеих системах. Поэтому будьте осторожны при использовании барьеров, поскольку их наличие делает ваш код менее переносимым.

Тот факт, что macOS не предоставляет функций для работы с POSIX-барьерами, означает: данная система лишь частично совместима с POSIX, и программы, в которых используются барьеры (являющиеся частью стандарта), нельзя скомпилировать на компьютерах с macOS. Это противоречит одному из принципов философии языка C: напиши один раз — компилируй где угодно.

В заключение отмечу, что POSIX-барьеры обеспечивают видимость памяти. Подобно операциям блокировки и разблокировки, ожидание барьеров гарантирует, что все закэшированные версии одной и той же переменной синхронизируются между разными потоками непосредственно перед выходом из барьера.

В следующем подразделе мы рассмотрим пример семафоров. Этот механизм нечасто используется в конкурентном программировании, но имеет специфические сферы применения.

Широкое распространение получили специальные двоичные семафоры (то же самое, что мьютексы), и выше мы неоднократно сталкивались с ними.

POSIX-семафоры

В большинстве случаев мьютексов (или *двоичных семафоров*) достаточно для синхронизации разных потоков, обращающихся к разделяемому ресурсу. Дело в том, что для последовательного выполнения операций чтения и записи на критическом участке должен находиться только один поток. Это называется *взаимным исключением* (mutual exclusion — отсюда и название «мьютекс»).

Но иногда у вас может возникнуть необходимость в том, чтобы сразу несколько потоков могло работать с разделяемым ресурсом на критическом участке. В таких случаях следует использовать *семафоры общего вида*.

Прежде чем переходить к обычным семафорам, рассмотрим пример, в котором используется двоичный семафор (или мьютекс). Вместо `pthread_mutex_*` мы будем применять функции `sem_*`, которые предоставляют возможности, относящиеся к семафорам.

Двоичные семафоры

В листинге 16.5 показано решение для примера 15.3 на основе семафоров. Напомню: этот пример состоял из двух потоков, каждый из которых инкрементировал разделяемое целочисленное значение на определенную величину. Наша задача состоит в том, чтобы обеспечить целостность разделяемой переменной. Обратите внимание: в коде мы не будем использовать POSIX-мьютексы.

Листинг 16.5. Решение для примера 15.3 с использованием POSIX-семафоров (ExtremeC_examples_chapter15_3_sem.c)

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

// Стандартный заголовок POSIX для использования библиотеки pthread
#include <pthread.h>

// Семафоры недоступны в заголовке pthread.h
#include <semaphore.h>

// Главный указатель на объект семафора, применяемый
// для синхронизации доступа к разделяемому состоянию.
sem_t *semaphore;

void* thread_body_1(void* arg) {
  // Получаем указатель на разделяемую переменную
  int* shared_var_ptr = (int*)arg;
  // Ждем семафор
  sem_wait(semaphore);
  // Инкрементируем разделяемую переменную на 1,
  // выполняя запись непосредственно по ее адресу в памяти
```

```
  (*shared_var_ptr)++;
  printf("%d\n", *shared_var_ptr);
  // Освобождаем семафор
  sem_post(semaphore);
  return NULL;
}

void* thread_body_2(void* arg) {
  // Получаем указатель на разделяемую переменную
  int* shared_var_ptr = (int*)arg;
  // Ждем семафор
  sem_wait(semaphore);
  // Инкрементируем разделяемую переменную на 1,
  // выполняя запись непосредственно по ее адресу в памяти
  (*shared_var_ptr) += 2;
  printf("%d\n", *shared_var_ptr);
  // Освобождаем семафор
  sem_post(semaphore);
  return NULL;
}

int main(int argc, char** argv) {

  // Разделяемая переменная
  int shared_var = 0;

  // Обработчики потоков
  pthread_t thread1;
  pthread_t thread2;

#ifdef __APPLE__
  // Неименованные семафоры не поддерживаются в OS/X. Поэтому
  // семафор нужно инициализировать как именованный, используя
  // функцию sem_open
  semaphore = sem_open("sem0", O_CREAT | O_EXCL, 0644, 1);
#else
  sem_t local_semaphore;
  semaphore = &local_semaphore;
  // Инициализируем семафор как мьютекс (двоичный семафор)
  sem_init(semaphore, 0, 1);
#endif

  // Создаем новые потоки
  int result1 = pthread_create(&thread1, NULL,
          thread_body_1, &shared_var);
  int result2 = pthread_create(&thread2, NULL,
          thread_body_2, &shared_var);

  if (result1 || result2) {
    printf("The threads could not be created.\n");
    exit(1);
  }
```

```
  // Ждем, пока потоки не завершат работу
  result1 = pthread_join(thread1, NULL);
  result2 = pthread_join(thread2, NULL);

  if (result1 || result2) {
    printf("The threads could not be joined.\n");
    exit(2);
  }

#ifdef __APPLE__
  sem_close(semaphore);
#else
  sem_destroy(semaphore);
#endif

  return 0;
}
```

Первое, что бросается в глаза в приведенном выше коде, — использование других функций для работы с семафорами в системах Apple. В этих ОС (macOS, OS X и iOS) не поддерживаются *неименованные семафоры*. Как следствие, мы не могли применить функции `sem_init` и `sem_destroy`. У неименованных семафоров нет имен (что неудивительно), и разные потоки могут работать с ними только внутри процесса. Для сравнения, именованные семафоры доступны на уровне системы и их могут видеть и использовать разные процессы.

В системах Apple функции, необходимые для создания неименованных семафоров, помечены как устаревшие, и объект семафора нельзя инициализировать с помощью `sem_init`. Таким образом, мы используем вместо этого функции `sem_open` и `sem_close`, чтобы определить именованные семафоры.

Именованные семафоры используются для синхронизации процессов, и мы обсудим их в главе 18. В других POSIX-совместимых операционных системах, в частности в Linux, мы по-прежнему можем применять неименованные семафоры и инициализировать/удалять их с помощью функций `sem_init` и `sem_destroy` соответственно.

В приведенном выше коде мы подключили дополнительный заголовочный файл, `semaphore.h`. Как уже объяснялось прежде, поддержка семафоров появилась в библиотеке потоков POSIX в виде расширения, поэтому они недоступны в заголовке `pthread.h`.

Подключив заголовки, мы объявили глобальный указатель на адрес, по которому находится объект семафора. Нам приходится это делать, поскольку в системах Apple необходимо использовать функцию `sem_open`, которая возвращает указатель.

Затем внутри функции `main` мы определили блок для систем Apple, в котором создается именованный семафор `sem0`. В POSIX-совместимых операционных системах

мы инициализируем семафор с помощью `sem_init`. Стоит отметить, что в данном случае указатель `semaphore` ссылается на переменную `local_sempahore`, которая находится на вершине стека главного потока. `semaphore` не может превратиться в висячий указатель, поскольку главный поток присоединяет два других потока и ждет их завершения.

Обратите внимание: макрос `__APPLE__` позволяет нам различать системы Apple и другие ОС. Он определен по умолчанию в препроцессорах языка C в системах Apple. Таким образом, с его помощью можно исключить код, который не должен компилироваться в этих системах.

Заглянем внутрь потоков. В функции-компаньоне критические участки защищены `sem_wait` и `sem_post` — аналогами функций `pthread_mutex_lock` и `pthread_mutex_unlock` соответственно в API для работы с POSIX-мьютексами. Заметьте, что `sem_wait` позволяет заходить на критический участок нескольким потокам.

Максимальное количество потоков, которым позволено находиться на критическом участке, определяется во время инициализации объекта семафора с помощью последнего аргумента функций `sem_open` и `sem_init`. Мы передали значение `1`, поэтому наш семафор должен вести себя подобно мьютексу.

Чтобы лучше понять, как работают семафоры, немного углубимся в детали. У каждого объекта семафора есть целочисленное значение. Если оно больше нуля и поток, который ожидает семафор, вызывает функцию `sem_wait`, то значение уменьшается на 1, а потоку разрешается зайти на критический участок. Если оно равно нулю, то поток должен подождать, пока оно снова не станет положительным. Каждый раз, когда поток выходит с критического участка, вызывая функцию `sem_post`, значение семафора увеличивается на 1. Следовательно, если с самого начала сделать его равным 1, то мы в конечном счете получим двоичный семафор.

В конце нашего кода вызывается функция `sem_destroy` (или `sem_close` в системах Apple), который фактически освобождает объект семафора со всеми его внутренними ресурсами. Что касается именованных семафоров, то они могут разделяться между разными процессами, поэтому при их закрытии могут возникать более сложные ситуации. Мы поговорим о них в главе 18.

Семафоры общего вида

Пришло время рассмотреть классический пример, в котором использованы семафоры общего вида. Синтаксис довольно похож на показанный в предыдущем листинге, но ситуация, когда на критический участок позволено заходить сразу нескольким потокам, может оказаться интересной.

В классическом примере речь идет о создании 50 молекул воды. Чтобы их получить, нам понадобится 50 атомов кислорода и 100 атомов водорода. Если представить

каждый атом в виде отдельного потока, то, чтобы сгенерировать молекулу воды, на свои критические участки должны зайти два потока для водорода и один для кислорода.

В следующем коде мы сначала создаем 50 потоков для кислорода и 100 для водорода. В первых критические участки будут защищены мьютексом, а во вторых — семафором общего вида, который позволяет находиться на критическом участке сразу двум потокам.

Для уведомления потоков мы используем POSIX-барьеры, но, поскольку они не поддерживаются в системах Apple, нам нужно реализовать их в виде мьютексов и условных переменных. Соответствующий код показан в листинге 16.6.

Листинг 16.6. Использование семафоров общего вида для имитации процесса создания 50 молекул воды из 50 атомов кислорода и 100 атомов водорода (ExtremeC_examples_chapter16_2.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>
#include <errno.h> // For errno and strerror function

// Стандартный заголовок POSIX для использования библиотеки pthread
#include <pthread.h>
// Семафоры недоступны в заголовке pthread.h
#include <semaphore.h>

#ifdef __APPLE__
// В системах Apple нужно имитировать возможности для работы с барьерами
pthread_mutex_t barrier_mutex;
pthread_cond_t  barrier_cv;
unsigned int    barrier_thread_count;
unsigned int    barrier_round;
unsigned int    barrier_thread_limit;

void barrier_wait() {
  pthread_mutex_lock(&barrier_mutex);
  barrier_thread_count++;
  if (barrier_thread_count >= barrier_thread_limit) {
    barrier_thread_count = 0;
    barrier_round++;
    pthread_cond_broadcast(&barrier_cv);
  } else {
    unsigned int my_round = barrier_round;
    do {
      pthread_cond_wait(&barrier_cv, &barrier_mutex);
    } while (my_round == barrier_round);
  }
  pthread_mutex_unlock(&barrier_mutex);
}
```

```
#else
// Барьер для синхронизации потоков водорода и кислорода
pthread_barrier_t water_barrier;
#endif

// Мьютекс для синхронизации потоков кислорода
pthread_mutex_t   oxygen_mutex;

// Общий семафор для синхронизации потоков водорода
sem_t*            hydrogen_sem;

// Разделяемое целое число для подсчета созданных молекул воды
unsigned int      num_of_water_molecules;

void* hydrogen_thread_body(void* arg) {
  // На этот критический участок может зайти два потока водорода
  sem_wait(hydrogen_sem);
  // Ждем присоединения другого потока водорода
#ifdef __APPLE__
  barrier_wait();
#else
  pthread_barrier_wait(&water_barrier);
#endif
  sem_post(hydrogen_sem);
  return NULL;
}

void* oxygen_thread_body(void* arg) {
  pthread_mutex_lock(&oxygen_mutex);
  // Ждем присоединения потоков водорода
#ifdef __APPLE__
  barrier_wait();
#else
  pthread_barrier_wait(&water_barrier);
#endif
  num_of_water_molecules++;
  pthread_mutex_unlock(&oxygen_mutex);
  return NULL;
}

int main(int argc, char** argv) {

  num_of_water_molecules = 0;

  // Инициализируем мьютекс кислорода
  pthread_mutex_init(&oxygen_mutex, NULL);

  // Инициализируем семафор водорода
#ifdef __APPLE__
  hydrogen_sem = sem_open("hydrogen_sem",
          O_CREAT | O_EXCL, 0644, 2);
#else
  sem_t local_sem;
```

```
  hydrogen_sem = &local_sem;
  sem_init(hydrogen_sem, 0, 2);
#endif

  // Инициализируем барьер воды
#ifdef __APPLE__
  pthread_mutex_init(&barrier_mutex, NULL);
  pthread_cond_init(&barrier_cv, NULL);
  barrier_thread_count = 0;
  barrier_thread_limit = 0;
  barrier_round = 0;
#else
  pthread_barrier_init(&water_barrier, NULL, 3);
#endif

  // Чтобы создать 50 молекул воды, нам понадобится 50 атомов кислорода
  // и 100 атомов водорода
  pthread_t thread[150];

  // Создаем потоки кислорода
  for (int i = 0; i < 50; i++) {
    if (pthread_create(thread + i, NULL,
                oxygen_thread_body, NULL)) {
      printf("Couldn't create an oxygen thread.\n");
      exit(1);
    }
  }

  // Создаем потоки водорода
  for (int i = 50; i < 150; i++) {
    if (pthread_create(thread + i, NULL,
                hydrogen_thread_body, NULL)) {
      printf("Couldn't create an hydrogen thread.\n");
      exit(2);
    }
  }

  printf("Waiting for hydrogen and oxygen atoms to react ...\n");
  // Ждем завершения всех потоков
  for (int i = 0; i < 150; i++) {
    if (pthread_join(thread[i], NULL)) {
      printf("The thread could not be joined.\n");
      exit(3);
    }
  }

  printf("Number of made water molecules: %d\n",
          num_of_water_molecules);

#ifdef __APPLE__
  sem_close(hydrogen_sem);
```

```
#else
  sem_destroy(hydrogen_sem);
#endif

  return 0;
}
```

Несколько начальных строчек этого кода находятся между `#ifdef __APPLE__` и `#endif`. Эти строчки компилируются только в системах Apple и в основном представляют собой реализацию и переменные, необходимые для имитации поведения POSIX-барьеров. В других POSIX-совместимых системах используются POSIX-барьеры. Мы не станем подробно останавливаться на том, как реализованы барьеры в системах Apple, но если хотите в этом разобраться, то можете почитать код.

Среди ряда глобальных переменных, определенных в этом коде, можно заметить мьютекс `oxygen_mutex`, который должен защищать критические участки потоков кислорода. В любой момент времени на критическом участке должен находиться только один такой поток.

Затем на самом критическом участке поток кислорода ждет, пока к нему не присоединятся два потока водорода, и продолжает работу, инкрементируя счетчик молекул воды. Операция инкремента происходит внутри критического участка кислорода.

Вы лучше поймете происходящее на критическом участке, когда мы рассмотрим роль, которую играет семафор общего вида. В нашем коде мы объявили такой семафор, `hydrogen_sem`, чтобы он защищал критический участок потоков водорода. На свои критические участки эти потоки могут заходить по двое, и они должны ждать перед объектом барьера, разделяющимся между потоками кислорода и водорода.

Когда количество потоков, ожидающих перед объектом барьера, достигает двух, это означает, что у нас имеется один атом кислорода и два атома водорода; вуаля — мы получаем молекулу воды, и все ожидающие потоки могут продолжить работу. Потоки водорода сразу же завершаются, а поток кислорода сначала инкрементирует счетчик молекул воды.

В заключение отмечу, что в примере 16.2 при реализации барьеров для систем Apple использовалась функция `pthread_cond_broadcast`. Она уведомляет все потоки, которые ждут условную переменную барьера и должны продолжить работу после присоединения к ним других потоков.

В следующем разделе речь пойдет о модели памяти, лежащей в основе POSIX-потоков, и о том, как эти потоки взаимодействуют с памятью процесса, который ими владеет. Кроме того, мы рассмотрим примеры использования сегментов стека и кучи и увидим, как из-за них могут возникать серьезные проблемы, относящиеся к памяти.

POSIX-потоки и память

Этот раздел посвящен взаимодействию потоков и памяти процесса. Вы уже знаете, что структура памяти процесса состоит из нескольких сегментов, таких как сегменты кода, стека, данных и кучи. Мы обсуждали их в главе 4. С каждым из этих сегментов потоки работают по-разному. В рамках данного раздела будут рассмотрены только стек и куча, поскольку при написании многопоточных программ эти области используются чаще других и вызывают больше всего проблем.

Кроме того, вы увидите, как синхронизация потоков и хорошее понимание их модели памяти может помочь в разработке более качественных конкурентных программ. Эти концепции особенно очевидны в контексте сегмента кучи, поскольку управление памятью в нем происходит вручную, а в конкурентных системах за выделение и освобождение блокировок в куче отвечают сами потоки. Элементарное состояние гонки может вызвать серьезные проблемы с памятью, поэтому во избежание подобных катастроф необходимо должным образом озаботиться о синхронизации.

В следующем подразделе я объясню, как разные потоки обращаются к сегменту стека и какие меры предосторожности при этом следует принимать.

Сегмент стека

У каждого потока есть собственный сегмент стека, который должен быть доступен только ему. Стек потока — часть общего стека процесса-владельца, в котором по умолчанию должны выделяться сегменты стека для всех потоков. Но стек может быть выделен и в куче. В будущих примерах я покажу, как это делается, но пока мы будем исходить из того, что стек потока принадлежит стеку процесса.

Поскольку все потоки внутри одного процесса могут читать и изменять его сегмент стека, они фактически имеют доступ к стекам друг друга. Но *не должны* пользоваться этим доступом. Стоит отметить: работа со стеками других потоков считается опасным поведением, так как переменные, определенные на вершине стека, могут быть уничтожены в любой момент, особенно при завершении потока или возвращении функции.

Вот почему мы пытаемся исходить из того, что каждый сегмент стека доступен только его потоку-владельцу, но не другим потокам. Таким образом, *локальные переменные* (объявленные на вершине стека) считаются приватными ресурсами потока и другие потоки не должны к ним обращаться.

В однопоточных приложениях есть всего один, главный поток. Поэтому мы используем его сегмент стека так, словно это стек процесса. Причина в том, что в однопоточной программе между главным потоком и самим процессом нет никакого разделения. Но в многопоточных программах все иначе. У каждого потока есть свой стек, который отличается от стеков других потоков.

При создании нового потока в стеке процесса выделяется блок памяти. Если программист самостоятельно не указал его размер, то будет использовано значение по умолчанию, которое зависит от платформы и варьируется от одной архитектуры к другой. В POSIX-совместимой системе можно использовать команду `ulimit -s`, чтобы узнать размер стека по умолчанию.

На моей текущей платформе (macOS на 64-битном компьютере с процессором Intel) стандартный размер стека равен 8 Мбайт (терминал 16.1).

Терминал 16.1. Вывод размера стека по умолчанию

```
$ ulimit -s
8192
$
```

API для работы с POSIX-потоками позволяет назначить новому потоку область стека. В примере 16.3 (листинг 16.7) у нас два потока. В одном из них мы используем параметры стека по умолчанию, а в другом выделяем буфер в сегменте кучи и назначаем его в качестве стека. Стоит отметить, что для выделяемого буфера следует указывать минимальный размер, иначе его нельзя будет использовать в качестве стека.

Листинг 16.7. Назначение блока кучи в качестве стека потока (ExtremeC_examples_chapter16_3.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

#include <pthread.h>

void* thread_body_1(void* arg) {
  int local_var = 0;
  printf("Thread1 > Stack Address: %p\n", (void*)&local_var);
  return 0;
}

void* thread_body_2(void* arg) {
  int local_var = 0;
  printf("Thread2 > Stack Address: %p\n", (void*)&local_var);
  return 0;
}

int main(int argc, char** argv) {

  size_t buffer_len = PTHREAD_STACK_MIN + 100;
  // Буфер, выделенный из кучи и используемый как
  // стек потока
  char *buffer = (char*)malloc(buffer_len * sizeof(char));
```

```
  // Обработчики потоков
  pthread_t thread1;
  pthread_t thread2;

  // Создаем новый поток с атрибутами по умолчанию
  int result1 = pthread_create(&thread1, NULL,
          thread_body_1, NULL);

  // Создаем новый поток с нашим собственным стеком
  pthread_attr_t attr;
  pthread_attr_init(&attr);
  // Задаем адрес и размер стека
  if (pthread_attr_setstack(&attr, buffer, buffer_len)) {
    printf("Failed while setting the stack attributes.\n");
    exit(1);
  }
  int result2 = pthread_create(&thread2, &attr,
          thread_body_2, NULL);

  if (result1 || result2) {
    printf("The threads could not be created.\n");
    exit(2);
  }

  printf("Main Thread > Heap Address: %p\n", (void*)buffer);
  printf("Main Thread > Stack Address: %p\n", (void*)&buffer_len);

  // Ждем, пока потоки не завершат работу
  result1 = pthread_join(thread1, NULL);
  result2 = pthread_join(thread2, NULL);

  if (result1 || result2) {
    printf("The threads could not be joined.\n");
    exit(3);
  }

  free(buffer);

  return 0;
}
```

В начале программы мы создаем первый поток со стандартными параметрами стека. Таким образом, стек должен быть выделен в одноименном сегменте процесса. После этого создается второй поток, для которого в качестве стека указывается адрес буфера.

Обратите внимание: указанное значение, 100 байт, превышает минимальный размер стека, представленный макросом `PTHREAD_STACK_MIN`. Эта константа имеет разные значения на разных платформах и подключается вместе с заголовочным файлом `limits.h`.

Если собрать и запустить эту программу на устройстве с Linux, то вы увидите примерно следующий результат (терминал 16.2).

Терминал 16.2. Сборка и выполнение примера 16.3

```
$ gcc ExtremeC_examples_chapter16_3.c -o ex16_3.out -lpthread
$ ./ex16_3.out
Main Thread > Heap Address: 0x55a86a251260
Main Thread > Stack Address: 0x7ffcb5794d50
Thread2 > Stack Address: 0x55a86a2541a4
Thread1 > Stack Address: 0x7fa3e9216ee4
$
```

Как следует из этого вывода, адрес локальной переменной `local_var`, выделенной на вершине стека второго потока, находится в другом диапазоне (диапазоне пространства кучи). Это значит, что область стека второго потока находится в куче, чего нельзя сказать о первом потоке.

В полученном выводе видно, что адрес локальной переменной в первом потоке находится в диапазоне сегмента стека, принадлежащего процессу. В результате мы могли легко выделить для нашего нового потока область стека в сегменте кучи.

В ряде ситуаций возможность назначать потоку область стека может оказаться незаменимой. Например, в средах с большими стеками, у которых общий объем памяти является низким, или в высокопроизводительных системах, где выделение стека для каждого потока было бы слишком расточительным, использование заранее выделенных буферов может иметь смысл. В целях предварительного выделения буфера в качестве области стека нового потока можно применить описанную выше процедуру.

Следующий пример, под номером 16.4, демонстрирует, как разделение адреса в стеке одного из потоков может привести к проблемам с памятью. Поток, которому принадлежит разделяемый адрес, должен оставаться активным, иначе все указатели с этим адресом станут висячими.

Код, представленный в листинге 16.8, не является потокобезопасным, поэтому можно ожидать, что при многократных прогонах время от времени будут возникать сбои. Кроме того, потоки имеют стандартные параметры стека; это значит, что их стеки выделяются в одноименном сегменте.

Листинг 16.8. Попытка прочитать переменную, выделенную в области стека другого потока (ExtremeC_examples_chapter16_4.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include <pthread.h>
```

```
int* shared_int;

void* t1_body(void* arg) {
  int local_var = 100;
  shared_int = &local_var;
  // Ждем, пока другой поток не выведет разделяемое целое значение
  usleep(10);
  return NULL;
}

void* t2_body(void* arg) {
  printf("%d\n", *shared_int);
  return NULL;
}

int main(int argc, char** argv) {

  shared_int = NULL;

  pthread_t t1;
  pthread_t t2;

  pthread_create(&t1, NULL, t1_body, NULL);
  pthread_create(&t2, NULL, t2_body, NULL);

  pthread_join(t1, NULL);
  pthread_join(t2, NULL);

  return 0;
}
```

Вначале мы объявили глобальный разделяемый указатель. Он может принимать любой адрес, независимо от того, в каком месте структуры памяти процесса тот находится: в сегменте стека, кучи или даже данных.

В представленном выше коде внутри функции-компаньона `t1_body` мы сохраняем адрес локальной переменной в разделяемый указатель. Эта переменная принадлежит первому потоку и выделена на вершине его стека.

С этого момента по завершении первого потока разделяемый указатель станет висячим, и любое его разыменование, скорее всего, проведет к сбою, логической ошибке или в лучшем случае к неочевидным проблемам с памятью. Это происходит в некоторых вариантах чередований, и в итоге, запустив программу несколько раз, время от времени можно наблюдать сбои.

Необходимо отметить: если один поток хочет использовать переменную, выделенную в стеке другого потока, то следует реализовать подходящие методы синхронизации. Поскольку время жизни переменной в стеке привязано к ее области видимости, синхронизация должна быть направлена на то, чтобы сохранять эту область видимости активной, пока поток не закончит работать с переменной.

Обратите внимание: мы не проверяем результаты функций pthread, чтобы не усложнять наш код, хотя проверка возвращаемых значений всегда приветствуется. Не все функции pthread ведут себя одинаково на всех платформах, и если что-то пойдет не так, то возвращаемые значения помогут вам об этом узнать.

В этом подразделе было в целом продемонстрировано, почему не следует разделять адреса, принадлежащие сегментам стека, и почему разделяемые состояния лучше не выделять в стеке. Дальше речь пойдет о самом распространенном месте для хранения разделяемых состояний — о куче. Как вы можете представить, работа с ней тоже имеет сложности и нужно остерегаться утечек памяти.

Сегмент кучи

Сегменты кучи и данных доступны всем потокам, однако первый является динамическим и разделяется во время выполнения, тогда как второй генерируется на этапе компиляции. Потоки могут читать и изменять содержимое кучи; оно существует на протяжении всего времени жизни процесса и не зависит от отдельных его потоков. Кроме того, в кучу можно поместить большие объекты. Благодаря совокупности всех этих факторов куча — отличное место для хранения состояний, которые должны разделяться между разными потоками.

Когда дело доходит до выделения кучи, управление памятью превращается в настоящий кошмар. Причиной тому факт, что выделяемые ресурсы в конечном счете должен освободить один из активных потоков, иначе можно столкнуться с утечками памяти.

Применительно к конкурентным средам чередования могут легко привести к появлению висячих указателей, в результате чего возникают сбои. Важнейшая цель синхронизации состоит в упорядочении выполнения таким образом, чтобы висячие указатели не могли появиться, и достичь этого непросто.

Взгляните на пример 16.5, представленный ниже. Он состоит из пяти потоков. Первый выделяет массив в куче, а второй и третий заполняют его следующим образом. Второй присваивает элементам с четными индексами прописные буквы, начиная с *Z* и двигаясь обратно к *A*; третий присваивает элементам с нечетными индексами строчные буквы, начиная с *a* и двигаясь вперед к *z*. Четвертый поток выводит полученный массив, а пятый его удаляет и освобождает память в куче.

В данном примере следует применить все методы управления конкурентностью с помощью средств POSIX, описанных в предыдущих разделах, чтобы не позволить этим потокам выйти из-под контроля при работе с кучей. Листинг 16.9 не содержит управляющего механизма и потому явно не является типобезопасным. Стоит отметить, что это не окончательная версия кода. Чуть позже мы добавим в нее механизмы управления конкурентностью.

Листинг 16.9. Пример 16.5 без каких-либо механизмов синхронизации (ExtremeC_examples_chapter16_5_raw.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include <pthread.h>

#define CHECK_RESULT(result) \
if (result) { \
  printf("A pthread error happened.\n"); \
  exit(1); \
}

int TRUE = 1;
int FALSE = 0;

// Указатель на разделяемый массив
char* shared_array;
// Размер разделяемого массива
unsigned int shared_array_len;

void* alloc_thread_body(void* arg) {
  shared_array_len = 20;
  shared_array = (char*)malloc(shared_array_len * sizeof(char*));
  return NULL;
}

void* filler_thread_body(void* arg) {
  int even = *((int*)arg);
  char c = 'a';
  size_t start_index = 1;
  if (even) {
    c = 'Z';
    start_index = 0;
  }
  for (size_t i = start_index; i < shared_array_len; i += 2) {
    shared_array[i] = even ? c-- : c++;
  }
  shared_array[shared_array_len - 1] = '\0';
  return NULL;
}

void* printer_thread_body(void* arg) {
  printf(">> %s\n", shared_array);
  return NULL;
}

void* dealloc_thread_body(void* arg) {
  free(shared_array);
```

```
  return NULL;
}

int main(int argc, char** argv) {
  ... Create threads ...
}
```

Можно легко заметить, что этому коду не хватает потоковой безопасности: когда пятый поток пытается освободить память массива, возникают серьезные сбои.

Каждый раз, получая процессорное время, пятый поток немедленно освобождает буфер, находящийся в куче, после чего указатель `shared_array` становится висячим и другие потоки начинают выходить из строя. Чтобы поток, освобождающий память, выполнялся последним и обеспечивался корректный порядок выполнения логики в других потоках, нужно использовать подходящие средства синхронизации.

В листинге 16.10 мы заворачиваем приведенный выше код в POSIX-объект в целях управления конкурентностью, чтобы сделать его потокобезопасным.

Листинг 16.10. Пример 16.5 с механизмами синхронизации (ExtremeC_examples_chapter16_5.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include <pthread.h>

#define CHECK_RESULT(result) \
if (result) { \
  printf("A pthread error happened.\n"); \
  exit(1); \
}

int TRUE = 1;
int FALSE = 0;

// Указатель на разделяемый массив
char* shared_array;
// Размер разделяемого массива
size_t shared_array_len;

pthread_barrier_t alloc_barrier;
pthread_barrier_t fill_barrier;
pthread_barrier_t done_barrier;

void* alloc_thread_body(void* arg) {
  shared_array_len = 20;
  shared_array = (char*)malloc(shared_array_len * sizeof(char*));
```

```
  pthread_barrier_wait(&alloc_barrier);
  return NULL;
}

void* filler_thread_body(void* arg) {
  pthread_barrier_wait(&alloc_barrier);
  int even = *((int*)arg);
  char c = 'a';
  size_t start_index = 1;
  if (even) {
    c = 'Z';
    start_index = 0;
  }
  for (size_t i = start_index; i < shared_array_len; i += 2) {
    shared_array[i] = even ? c-- : c++;
  }
  shared_array[shared_array_len - 1] = '\0';
  pthread_barrier_wait(&fill_barrier);
  return NULL;
}

void* printer_thread_body(void* arg) {
  pthread_barrier_wait(&fill_barrier);
  printf(">> %s\n", shared_array);
  pthread_barrier_wait(&done_barrier);
  return NULL;
}

void* dealloc_thread_body(void* arg) {
  pthread_barrier_wait(&done_barrier);
  free(shared_array);
  pthread_barrier_destroy(&alloc_barrier);
  pthread_barrier_destroy(&fill_barrier);
  pthread_barrier_destroy(&done_barrier);
  return NULL;
}

int main(int argc, char** argv) {

  shared_array = NULL;

  pthread_barrier_init(&alloc_barrier, NULL, 3);
  pthread_barrier_init(&fill_barrier, NULL, 3);
  pthread_barrier_init(&done_barrier, NULL, 2);

  pthread_t alloc_thread;
  pthread_t even_filler_thread;
  pthread_t odd_filler_thread;
  pthread_t printer_thread;
  pthread_t dealloc_thread;
```

```
  pthread_attr_t attr;
  pthread_attr_init(&attr);
  int res = pthread_attr_setdetachstate(&attr,
          PTHREAD_CREATE_DETACHED);
  CHECK_RESULT(res);

  res = pthread_create(&alloc_thread, &attr,
          alloc_thread_body, NULL);
  CHECK_RESULT(res);

  res = pthread_create(&even_filler_thread,
          &attr, filler_thread_body, &TRUE);
  CHECK_RESULT(res);

  res = pthread_create(&odd_filler_thread,
          &attr, filler_thread_body, &FALSE);
  CHECK_RESULT(res);

  res = pthread_create(&printer_thread, &attr,
          printer_thread_body, NULL);
  CHECK_RESULT(res);

  res = pthread_create(&dealloc_thread, &attr,
          dealloc_thread_body, NULL);
  CHECK_RESULT(res);

  pthread_exit(NULL);

  return 0;
}
```

Чтобы сделать код в данном листинге потокобезопасным, достаточно добавить в него POSIX-барьеры. Это самый простой способ гарантировать последовательное выполнения нескольких потоков.

Если сравнить листинги 16.9 и 16.10, то можно увидеть, как POSIX-барьеры используются для упорядочения разных потоков. Единственное исключение составляют два потока фильтрации. Они могут выполняться независимо, не блокируя друг друга, и, так как изменяют четные и нечетные индексы по отдельности, проблем с конкурентностью возникнуть не может. Обратите внимание: приведенный выше код нельзя скомпилировать в системах Apple. В таких ОС вам нужно имитировать поведение барьеров с помощью мьютексов и условных переменных (как мы делали в примере 16.2).

В терминале 16.3 показан вывод предыдущего кода. Независимо от того, сколько раз вы его запускаете, он никогда не испытывает сбоев. Иными словами, этот код защищен от различных чередований и является потокобезопасным.

Терминал 16.3. Сборка и выполнение примера 16.5

```
$ gcc ExtremeC_examples_chapter16_5.c -o ex16_5 -lpthread
$ ./ex16_5
>> ZaYbXcWdVeUfTgShRiQ
$ ./ex16_5
>> ZaYbXcWdVeUfTgShRiQ
$
```

В этом подразделе мы рассмотрели пример использования пространства кучи в качестве хранилища разделяемых состояний. В отличие от стека, в котором память выделяется автоматически, куча требует ручного выделения памяти. В противном случае неизбежным побочным эффектом будут утечки памяти.

Самое простое и зачастую оптимальное место для хранения разделяемых состояний (с точки зрения усилий, которые программист должен приложить, чтобы управлять памятью) — сегмент данных, в котором операции выделения и освобождения выполняются автоматически. Переменные, находящиеся в этом сегменте, считаются глобальными и имеют максимально продолжительное время жизни — с момента создания процесса до самого его завершения. Но в ряде ситуаций такое время жизни может быть нежелательным, особенно если вам нужно хранить в сегменте данных большой объект.

В следующем подразделе мы поговорим о видимости памяти и о том, как она гарантируется функциями POSIX.

Видимость памяти

Видимость памяти и *когерентность* уже обсуждались в предыдущих главах, когда мы рассматривали системы с несколькими процессорными ядрами. Здесь же будет показано, как видимость памяти гарантируется библиотекой pthread.

Как вы уже знаете, протокол когерентности кэша ядер процессора следит за тем, чтобы все закэшированные версии одного адреса памяти во всех ядрах оставались синхронизированными и соответствовали последним изменениям, внесенным в одно из ядер. Однако этот протокол должен как-то инициироваться.

Существуют системные вызовы, которые приводят к срабатыванию протокола когерентности кэша и делают память видимой для всех ядер процессора. В библиотеке pthread тоже есть ряд функций, перед выполнением которых гарантируется видимость памяти.

Кое-какие из них вы уже могли встречать ранее:

- `pthread_barrier_wait`;
- `pthread_cond_broadcast`;

- pthread_cond_signal;
- pthread_cond_timedwait;
- pthread_cond_wait;
- pthread_create;
- pthread_join;
- pthread_mutex_lock;
- pthread_mutex_timedlock;
- pthread_mutex_trylock;
- pthread_mutex_unlock;
- pthread_spin_lock;
- pthread_spin_trylock;
- pthread_spin_unlock;
- pthread_rwlock_rdlock;
- pthread_rwlock_timedrdlock;
- pthread_rwlock_timedwrlock;
- pthread_rwlock_tryrdlock;
- pthread_rwlock_trywrlock;
- pthread_rwlock_unlock;
- pthread_rwlock_wrlock;
- sem_post;
- sem_timedwait;
- sem_trywait;
- sem_wait;
- semctl;
- semop.

Помимо локальных кэшей в процессорных ядрах, компиляторы могут применять механизмы кэширования для часто используемых переменных. Для этого компилятор должен проанализировать код и оптимизировать его таким образом, чтобы часто используемые переменные записывались и считывались из его кэша. Это программные кэши, которые компилятор размещает в итоговом исполняемом файле, чтобы оптимизировать и ускорить работу программы.

При написании многопоточного кода такой подход может принести не только пользу, но и лишнюю головную боль, связанную с видимостью памяти. И потому иногда для некоторых переменных эти кэши отключаются.

Переменные, которые компилятор не должен оптимизировать с помощью кэширования, можно объявлять *изменчивыми*. Стоит отметить, что изменчивые переменные могут кэшироваться на уровне процессора, но компилятор не станет размещать их в своих кэшах. Изменчивая переменная объявляется с помощью ключевого слова `volatile`. В листинге 16.11 показан пример объявления целочисленной изменчивой переменной.

Листинг 16.11. Объявление целочисленной изменчивой переменной

```
volatile int number;
```

Важной особенностью изменчивых переменных является то, что они не избавляют вас от проблем видимости в многопоточных системах. Обеспечить видимость памяти можно с помощью перечисленных выше POSIX-функций, использованных на подходящих участках кода.

Резюме

В этой главе мы обсудили механизмы управления конкурентностью, которые предоставляются API POSIX для работы с потоками. Были рассмотрены следующие темы:

- POSIX-мьютексы и их применение;
- условные переменные и барьеры POSIX, а также их использование;
- POSIX-семафоры и то, чем двоичные семафоры отличаются от семафоров общего вида;
- взаимодействие потоков с областью стека;
- назначение потоку новой области стека, выделенной в куче;
- взаимодействие потоков с пространством кучи;
- видимость памяти и функции POSIX, которые ее гарантируют;
- изменчивые переменные и кэши компилятора.

В следующей главе мы продолжим обсуждение и рассмотрим второй подход к конкурентности в программных системах — многопроцессность. Мы поговорим о том, как выполнить процесс и чем он отличается от потока.

17 Процессы

Пришло время поговорить о системах, архитектура которых состоит из нескольких процессов. Такие системы обычно называются многопроцессными. В этой и следующей главах мы рассмотрим концепцию многопроцессности и провести анализ ее достоинств и недостатков в сравнении с многопоточностью, о которой речь шла в главах 15 и 16.

В данной главе мы сосредоточимся на API и методиках для запуска новых процессов, а также увидим, как на самом деле выполняется процесс. Следующая глава посвящена конкурентным средам, состоящим из нескольких процессов. Я объясню способы разделения различных состояний между разными процессами и то, как обычно происходит обращение к этим состояниям в многопроцессных средах.

Часть главы посвящена сравнению многопроцессных и многопоточных сред. Вдобавок мы затронем локальные и распределенные многопроцессные системы.

API для выполнения процессов

Любая программа выполняется в виде процесса. Но поначалу она представляет собой исполняемый двоичный файл, содержащий некоторые сегменты памяти и, скорее всего, множество инструкций машинного уровня. А вот процесс — это отдельный экземпляр программы на этапе ее выполнения. Таким образом, одну скомпилированную программу (или исполняемый двоичный файл) можно выполнять по нескольку раз в разных процессах. На самом деле это основная причина, почему в данной главе я уделяю внимание процессам, а не самим программам.

В двух предыдущих главах мы обсуждали потоки в однопроцессном программном обеспечении. Здесь же речь пойдет о ПО с несколькими процессами. Но сначала нужно разобраться с тем, как и с помощью какого API можно создать новый процесс.

Стоит отметить, что нас в основном интересует выполнение процессов в Unix-подобных операционных системах, поскольку все они имеют многослойную

архитектуру, свойственную Unix, и предоставляют широко известные и похожие API. У других ОС могут быть собственные механизмы для выполнения процессов, но большинство из них более или менее соответствуют многослойной архитектуре, поэтому от них можно ожидать наличия аналогичных методов.

В Unix-подобной операционной системе есть не так уж много способов выполнить процесс на уровне системных вызовов. Как вы, наверное, помните из главы 11, *кольцо ядра* находится почти в самом центре, сразу после *кольца аппаратного обеспечения*, и предоставляет внешним кольцам (*командной оболочке* и *прикладным программам*) различные возможности ядра в виде *интерфейса системных вызовов*. Два таких системных вызова предназначены для создания и выполнения процессов; речь идет о `fork` и `exec` (или `execve` в Linux) соответственно. Создание подразумевает порождение нового процесса, а при выполнении мы берем имеющийся процесс и подставляем в него новую программу; таким образом, во втором случае новый процесс не порождается.

В результате использования этих системных вызовов программа всегда выполняется в виде нового процесса, но данный процесс порождается не всегда! Вызов `fork` порождает новый процесс, а вызов `exec` заменяет вызывающий процесс новым. О различиях между `fork` и `exec` мы поговорим позже. А сначала посмотрим, как к этим системным вызовам обращаться из внешних колец.

Как уже объяснялось в главе 10, Unix-подобные операционные системы имеют два стандарта, которые среди прочего описывают интерфейс, предоставляемый кольцу командной оболочки. Эти стандарты — *единая спецификация Unix* (Single Unix Specification, SUS) и *POSIX*. Более подробно о них, а также об их сходствах и различиях можно почитать в главе 10.

Интерфейс, предоставляемый кольцом ядра, должен быть подробно описан в стандарте POSIX, и, действительно, некоторые аспекты этого стандарта посвящены выполнению и управлению процессами.

Поэтому, как следовало ожидать, в POSIX должны присутствовать заголовки и функции для создания и выполнения процессов. И они там действительно есть; соответствующие возможности обнаруживаются в разных заголовочных файлах. POSIX-функции, предусмотренные для создания и выполнения процессов, перечислены ниже:

- функция `fork`, которая находится в заголовочном файле `unistd.h`, отвечает за создание процесса;
- функции `posix_spawn` и `posix_spawnp`, находящиеся в заголовочном файле `spawn.h`, отвечают за создание процесса;
- функции семейства `exec*`, такие как `execl` и `execlp`, находятся в заголовочном файле `unistd.h` и отвечают за выполнение процесса.

Обратите внимание: перечисленные выше функции не следует путать с системными вызовами `fork` и `exec`. Эти функции входят в состав интерфейса POSIX, доступного в кольце командной оболочки, тогда как системные вызовы предоставляются кольцом ядра. Со стандартом POSIX совместимо большинство Unix-подобных систем и некоторые другие ОС. Последние тоже могут содержать функции, представленные выше, но их внутренние механизмы для порождения процесса могут отличаться на уровне системных вызовов.

В качестве реального примера можно привести использование Cygwin или MinGW в Microsoft Windows в целях обеспечения совместимости с POSIX. Установив эти программы, вы сможете писать и компилировать стандартный код на языке C, который использует интерфейс POSIX. Таким образом, система Microsoft Windows становится частично POSIX-совместимой, хотя не содержит системных вызовов `fork` или `exec`! Это очень важное обстоятельство может вызывать сильную путаницу, и потому вы должны знать: кольцо командной оболочки не всегда предоставляет тот же интерфейс, что и кольцо ядра.

Детали реализации функции fork в Cygwin можно найти по адресу https://github.com/openunix/cygwin/blob/master/winsup/cygwin/fork.cc. Стоит отметить, что она не использует внутри системный вызов fork, который обычно присутствует в Unix-подобных ядрах; вместо этого она подключает заголовки из Win32 API и вызывает общеизвестные функции, предназначенные для создания процессов и управления ими.

Согласно спецификации POSIX, кольцо командной оболочки в Unix-подобных системах предоставляет не только стандартную библиотеку C. В терминале есть предустановленные командные утилиты, которые позволяют использовать стандартные функции C в сложных сценариях. И каждый раз, когда пользователь вводит команду в терминале, создается новый процесс.

Даже такие простые команды, как `ls` или `sed`, порождают новые процессы, которые могут проработать меньше одной секунды. Вы должны понимать, что эти утилиты в основном написаны на языке C и обращаются к тому же интерфейсу POSIX, который вы можете использовать при написании собственных программ.

Скрипты командной оболочки тоже выполняются в отдельных процессах, но немного по-другому. Мы вернемся к ним в будущих разделах при обсуждении того, как процессы выполняются в Unix-подобных системах.

Создание процесса происходит в ядре, особенно если оно монолитно. Каждый раз, когда пользователь порождает новый процесс или даже поток, интерфейс системных вызовов получает запрос и передает его кольцу ядра. А там уже в ответ на этот запрос создается новое *задание*, будь то процесс или поток.

В монолитных системах, таких как Linux или FreeBSD, механизм отслеживания заданий (процессов и потоков) находится внутри ядра, поэтому логично, что процессы создаются в самом ядре.

Обратите внимание: любое новое задание, которое создается в ядре, помещается в очередь *планировщика заданий* и до того, как оно получит ресурсы процессора и сможет начать выполнение, может пройти какое-то время.

Создание новых процессов требует родительского процесса. Вот почему у любого процесса есть родитель. На самом деле он может быть только один. Цепочка родителей и прародителей растягивается до самого первого пользовательского процесса, который обычно называется *init*. Родителем init служит процесс ядра.

Это предок всех остальных процессов внутри Unix-подобной системы, и существует он вплоть до ее выключения. Когда процесс завершает работу, но у него остаются дочерние, *осиротевшие процессы*, его место в качестве родителя занимает init.

Такие отношения между родителями и потомками формируют большое дерево процессов, которое можно просмотреть с помощью командной утилиты `pstree`. О том, как ее использовать, вы узнаете в будущих примерах.

Итак, мы знаем, что для выполнения новых процессов предусмотрен специальный API. Теперь пришло время рассмотреть настоящие примеры на языке C, чтобы увидеть, как это работает на самом деле. Начнем с функции `fork`, которая в конечном счете использует одноименный системный вызов.

Создание процесса

Как уже упоминалось выше, функцию `fork` можно использовать для порождения новых процессов. Мы также выяснили, что новый процесс всегда является потомком какого-то другого процесса. Здесь мы рассмотрим примеры того, как породить новый дочерний процесс подобным образом.

Чтобы создать новый дочерний процесс, родительскому процессу необходимо вызвать функцию `fork`. Ее объявление можно подключить вместе с заголовочным файлом `unistd.h`, который входит в состав POSIX.

При вызове функции `fork` создается точная копия вызывающего (родительского) процесса. В результате оба процесса продолжают работать конкурентно, начиная с инструкции, которая идет за вызовом `fork`. Отмечу, что дочерний процесс наследует много атрибутов от своего родителя, включая все сегменты памяти и их содержимое. Следовательно, он имеет доступ ко всем тем же переменным в сегментах данных и кучи, а также содержит те же программные инструкции в сегменте кода.

О других наследуемых атрибутах мы поговорим чуть ниже. Но сначала рассмотрим пример.

Поскольку мы получаем два разных процесса, функция `fork` возвращается дважды: в родительском процессе и в дочернем. Кроме того, в каждом процессе `fork` возвращает разные значения: ноль в дочернем и PID нового процесса в родительском. В примере 17.1 показан один из простейших сценариев использования функции `fork` (листинг 17.1).

Листинг 17.1. Создание дочернего процесса с помощью API fork (ExtremeC_examples_chapter17_1.c)

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

int main(int argc, char** argv) {
  printf("This is the parent process with process ID: %d\n",
         getpid());
  printf("Before calling fork() ...\n");
  pid_t ret = fork();
  if (ret) {
    printf("The child process is spawned with PID: %d\n", ret);
  } else {
    printf("This is the child process with PID: %d\n", getpid());
  }
  printf("Type CTRL+C to exit ...\n");
  while (1);
  return 0;
}
```

Выше вы можете видеть, что использовалась инструкция `printf` для вывода некоторых журнальных записей с целью отслеживать работу процессов. Чтобы породить новый процесс, была вызвана функция `fork`. Она не принимает никаких аргументов, поэтому применять ее очень легко и просто.

В момент вызова функции `fork` из вызывающего процесса, который впоследствии становится родителем, создается (или клонируется) новый. После этого они продолжают работать конкурентно, как два разных процесса.

Естественно, функция `fork`, в свою очередь, инициирует дополнительные операции на уровне системных вызовов, и только после этого соответствующая логика в ядре может создать новую копию процесса.

Непосредственно перед инструкцией `return` находится бесконечный цикл, который позволяет работать обоим процессам и не дает им завершиться. Обратите внимание: процессы рано или поздно должны дойти до этого цикла, поскольку имеют совершенно одинаковые инструкции в своих сегментах кода.

Мы специально предотвращаем завершение процессов, чтобы увидеть их в выводе команд `pstree` и `top`. Но сначала нам нужно скомпилировать приведенный выше код и понаблюдать за тем, как создается новый процесс (терминал 17.1).

Терминал 17.1. Сборка и запуск примера 17.1

```
$ gcc ExtremeC_examples_chapter17_1.c -o ex17_1.out
$ ./ex17_1.out
This is the parent process with process ID: 10852
Before calling fork() …
The child process is spawned with PID: 10853
This is the child process with PID: 10853
Type CTRL+C to exit ...
$
```

Как видите, родительский процесс выводит свой PID, который равен `10852`. Заметьте: PID меняется при каждом выполнении. После создания дочернего процесса родитель выводит PID, который вернула функция `fork`; он равен `10853`.

В следующей строчке потомок выводит свой PID, тоже равный `10853`. Это соответствует тому, что получил родитель из функции `fork`. Наконец, оба процесса входят в бесконечный цикл, давая нам время понаблюдать за ними с помощью специальных утилит.

Как вы могли заметить, дочерний процесс наследует от родителя те же файловый дескриптор `stdout` и терминал. Следовательно, может возвращать данные в тот же вывод, что и родитель. Потомок наследует файловые дескрипторы, открытые на момент вызова функции `fork` в родительском процессе.

Кроме того, наследуются и другие атрибуты, о которых можно почитать на справочной странице функции `fork`. В случае с Linux она находятся по адресу http://man7.org/linux/man-pages/man2/fork.2.html.

Если пройти по этой ссылке и просмотреть атрибуты, то можно заметить, что некоторые из них являются общими для родителя и его потомков, а другие — уникальными для каждого процесса. Например, PID, PID родителя, потоки и т. д.

Отношения между родительскими и дочерними процессами можно наглядно продемонстрировать с помощью таких утилит, как `pstree`. У каждого процесса есть родитель, и все вместе они составляют большое дерево. Помните, что родитель у любого отдельно взятого процесса может быть только один.

Пока процессы в нашем примере находятся в своих бесконечных циклах, мы можем воспользоваться командной утилитой `pstree`, чтобы вывести древовидный список всех процессов в системе. В терминале 17.2 показан вывод `pstree` на компьютере с Linux. Обратите внимание: в системах Linux эта утилита установлена по умолчанию, но в других Unix-подобных ОС ее, возможно, придется устанавливать отдельно.

Терминал 17.2. Использование утилиты pstree для поиска процессов, порожденных в примере 17.1

```
$ pstree -p
systemd(1)─┬─accounts-daemon(877)─┬─{accounts-daemon}(960)
           │                      └─{accounts-daemon}(997)
...
...
...
           ├─systemd-logind(819)
           ├─systemd-network(673)
           ├─systemd-resolve(701)
           ├─systemd-timesyn(500)───{systemd-timesyn}(550)
           ├─systemd-udevd(446)
           └─tmux: server(2083)─┬─bash(2084)───pstree(13559)
                                └─bash(2337)───ex17_1.
out(10852)───ex17_1.out(10853)
$
```

В последней строчке этого терминала указано два процесса с PID `10852` и `10852`, которые имеют отношения вида «родитель — потомок». Интересно, что родитель процесса `10852` имеет PID `2337`, который принадлежит процессу *bash*.

Обратите внимание на предпоследнюю строчку, в которой указан сам процесс `pstree`; он является потомком процесса bash с PID `2084`. Оба процесса bash принадлежат одному и тому же эмулятору терминала *tmux* с PID `2083`.

В Linux самым первым процессом является *планировщик*, который входит в образ ядра и имеет PID `0`. Следующий процесс обычно называется *init*, и его PID равен `1`; это первый процесс, который создается планировщиком. Он существует с момента запуска системы и до ее выключения. Все остальные пользовательские процессы — прямые или непрямые потомки init. Процесс, потерявший родителя, становится сиротой, и init делает его своим прямым потомком.

Однако в более новых версиях почти всех известных дистрибутивов Linux процесс init был заменен демоном *systemd*. Вот почему в первой строчке терминала 17.2 значится `systemd(1)`. По приведенной ниже ссылке находится отличный материал об отличиях init и systemd и о том, почему разработчики дистрибутивов Linux решили выполнить этот переход: https://www.tecmint.com/systemd-replaces-init-in-linux.

При использовании функции `fork` родительский и дочерний процессы выполняются конкурентно. Это значит, что они должны проявлять некоторые свойства конкурентных систем.

Самое известное свойство, которое мы можем наблюдать, — разные варианты чередований. Если вы незнакомы с этим термином, то я настоятельно рекомендую почитать главы 13 и 14.

В примере 17.2, представленном ниже, видно, как родительский и дочерний процессы могут демонстрировать недетерминированные чередования. Мы выведем несколько строк и посмотрим, как меняются чередования в двух последовательных прогонах (листинг 17.2).

Листинг 17.2. Два процесса, которые записывают разные строки в стандартный вывод (ExtremeC_examples_chapter17_2.c)

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

int main(int argc, char** argv) {
  pid_t ret = fork();
  if (ret) {
    for (size_t i = 0; i < 5; i++) {
      printf("AAA\n");
      usleep(1);
    }
  } else {
    for (size_t i = 0; i < 5; i++) {
      printf("BBBBBB\n");
      usleep(1);
    }
  }
  return 0;
}
```

Здесь все очень похоже на код, написанный нами в примере 17.1. Мы создаем копию процесса, которая затем вместе со своим родителем записывает текстовые строки в стандартный вывод. Родительский и дочерний процессы выводят по пять раз `AAA` и `BBBBBB` соответственно. В терминале 17.3 показан вывод двух последовательных прогонов одного и того же скомпилированного исполняемого файла.

Терминал 17.3. Вывод двух последовательных прогонов примера 17.2

```
$ gcc ExtremeC_examples_chapter17_2.c -o ex17_2.out
$ ./ex17_2.out
AAA
AAA
AAA
AAA
AAA
BBBBBB
BBBBBB
BBBBBB
BBBBBB
BBBBBB
$ ./ex17_2.out
AAA
AAA
```

```
BBBBBB
AAA
AAA
BBBBBB
BBBBBB
BBBBBB
AAA
BBBBBB
$
```

В этом выводе явно видно, что мы имеем разные чередования. Это означает следующее: если определить наше инвариантное ограничение в соответствии с тем, что мы видим в стандартном выводе, то существует риск состояния гонки. Это рано или поздно приведет к проблемам, с которыми мы сталкивались при написании многопоточного кода, и для борьбы с ними следует использовать аналогичные методы. В следующей главе мы подробно обсудим такие решения.

Следующий подраздел посвящен выполнению процесса и тому, как его инициировать с помощью функций `exec*`.

Выполнение процесса

Еще один способ выполнить новый процесс заключается в использовании функций семейства `exec*`. В них применяется другой подход по сравнению с API `fork`. Философию, стоящую за функциями `exec*`, можно описать так: вначале создается простой базовый процесс, и затем в какой-то момент в него загружается нужный нам исполняемый файл, который становится новым *образом процесса*. Это загруженная версия исполняемого файла с выделенными сегментами памяти, готовая к выполнению. В будущих разделах мы обсудим разные этапы загрузки исполняемого файла и подробно узнаем, что собой представляет образ процесса.

Таким образом, использование функций `exec*` вместо создания нового процесса вызывает подмену уже существующего. Это самое важное отличие от функции `fork`. Базовый процесс не копируется, а полностью заменяется новым набором сегментов памяти и программных инструкций.

В листинге 17.3, содержащем пример 17.3, показано использование функции `execvp`, входящей в семейство функций `exec*`, для запуска процесса `echo`. Функция `execvp` — одна из функций группы `exec*`, которая наследует от родительского процесса переменную среды `PATH` и ищет исполняемые файлы так, как это делал родитель.

Листинг 17.3. Демонстрация работы функции execvp (ExtremeC_examples_chapter17_3.c)

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

```
#include <string.h>
#include <errno.h>

int main(int argc, char** argv) {
  char *args[] = {"echo", "Hello", "World!", 0};
  execvp("echo", args);
  printf("execvp() failed. Error: %s\n", strerror(errno));
  return 0;
}
```

Как видите, была вызвана функция `execvp`. Чуть выше уже упоминалось о том, что она наследует от базового процесса переменную среды `PATH` и точно так же ищет существующие исполняемые файлы. Эта функция принимает два аргумента: имя исполняемого файла или скрипта, который должен быть загружен и выполнен, и список аргументов, которые ему нужно передать.

Обратите внимание: мы указали `echo`, а не абсолютный путь. Таким образом, функция `execvp` должна сначала найти соответствующий исполняемый файл. Он может находиться в любом месте Unix-подобной операционной системы: в `/usr/bin`, `/usr/local/bin` или где-то еще. Реальное местонахождение `echo` можно определить путем перебора всех каталогов, указанных в переменной среды `PATH`.

Функции exec* могут запускать разного рода исполняемые файлы. Ниже перечислены все файловые форматы, которые поддерживаются этими функциями:

- исполняемые файлы ELF;
- скриптовые файлы с символами #! в самом начале, которые обозначают интерпретатор скрипта;
- традиционный формат двоичных файлов a.out;
- исполняемые файлы ELF FDPIC.

Обнаружив исполняемый файл `echo`, функция `execvp` делает все остальное. Она инициирует системный вызов `exec` (`execve` в Linux) с подготовленным набором аргументов, в результате чего ядро формирует из найденного исполняемого файла образ процесса. Когда все готово, ядро заменяет текущий образ процесса новым, и после этого базовый процесс теряется навсегда. Дальше управление переходит к новому процессу, который начинает работу с функции `main`, как при обычном выполнении.

В результате данной процедуры инструкция `printf` не может быть выполнена, если вызов функции `execvp`, за которой она идет, был успешным, поскольку мы теперь имеем совершенно новый процесс с новыми сегментами памяти и инструкциями. Если инструкция `printf` выполняется, то это признак того, что вызов функции `execvp` завершился неудачей.

Как уже говорилось ранее, `execvp` — это лишь одна из функций семейства `exec*`. И несмотря на похожее поведение, они все же слегка отличаются. Ниже приводится их сравнение.

- Функция `execl(const char* path, const char* arg0, …, NULL)` принимает абсолютный путь к исполняемому файлу вместе с набором аргументов, которые должны быть переданы новому процессу. Последним аргументом должна быть нулевая строка, `0` или `NULL`. Пример 17.3, переписанный с использованием `execl`, выглядел бы так: `execl("/usr/bin/echo", "echo", "Hello", "World", NULL)`.
- Функция `execlp(const char* file, const char* arg0, ..., NULL)` принимает относительный путь в качестве первого аргумента и может легко найти исполняемый файл благодаря доступу к переменной среды `PATH`. Эта функция также принимает набор аргументов, которые должны быть переданы новому процессу. Последним аргументом должна быть нулевая строка, `0` или `NULL`. Пример 17.3, переписанный с использованием `execlp`, выглядел бы так: `execlp("echo", "echo," "Hello," "World," NULL)`.
- Функция `excele(const char* path, const char* arg0, ..., NULL, const char* env0, ..., NULL)` принимает абсолютный путь к исполняемому файлу вместе с набором аргументов, которые должны быть переданы новому процессу. В конце этого набора должна быть нулевая строка. Затем должен идти список строк, представляющих переменные среды. Они тоже должны заканчиваться нулевой строкой. Пример 17.3, переписанный с использованием `execle`, выглядел бы так: `execle("/usr/bin/echo", "echo", "Hello", "World", NULL, "A=1", "B=2", NULL)`. Заметьте, что в этом вызове мы передали новому процессу две новые переменные среды: `A` и `B`.
- Функция `execv(const char* path, const char* args[])` принимает абсолютный путь к исполняемому файлу вместе с массивом аргументов, которые должны быть переданы новому процессу. Последним элементом массива должна быть нулевая строка, `0` или `NULL`. Пример 17.3, переписанный с использованием `execl`, выглядел бы так: `execl("/usr/bin/echo", args)`, где переменная `args` была бы объявлена как `char* args[] = {"echo", "Hello", "World", NULL}`.
- Функция `execvp(const char* file, const char* args[])` принимает относительный путь в качестве первого аргумента и может легко найти исполняемый файл благодаря доступу к переменной среды `PATH`. Данная функция также принимает массив аргументов, которые должны быть переданы новому процессу. Последним элементом массива должна быть нулевая строка, `0` или `NULL`. Именно эту функцию мы использовали в примере 17.3.

В случае успешного выполнения функций `exec*` предыдущий процесс исчезает, а его место занимает новый. Таким образом, создание второго процесса не происходит. По этой причине мы не можем продемонстрировать разные чередования, как в случае с функцией `fork`. В следующем подразделе мы сравним `fork` и функции `exec*` в контексте выполнения новой программы.

Разные методы создания и выполнения процессов

Учитывая все вышесказанное и примеры, приведенные выше, мы можем сравнить два метода выполнения новой программы.

- В результате успешного вызова `fork` получаются два отдельных процесса: родительский, который вызывал `fork`, и дочерний. Успешное выполнение `exec*` заключается в замене вызывающего процесса новым образом, в результате чего новый процесс не создается.
- Вызов `fork` дублирует все содержимое памяти родительского процесса, поэтому потомку доступны все те же адреса и переменные. Для сравнения, вызов `exec*` уничтожает структуру памяти базового процесса и создает новую структуру, основанную на загруженном исполняемом файле.
- У скопированного процесса есть доступ к определенным атрибутам родителя, таким как открытые файловые дескрипторы. А вот при использовании функций `exec*` новый процесс не знает об атрибутах старого и ничего от него не наследует.
- В обоих случаях новый процесс содержит всего один главный поток. Потоки родительного процесса не копируются при использовании `fork`.
- Функции `exec*` можно применять для выполнения скриптов и внешних исполняемых файлов, а `fork` подходит только для создания новых процессов, которые являются копией той же программы.

В следующем разделе мы поговорим о тех этапах, через которые проходит большинство ядер при загрузке и выполнении нового процесса. Перечень этапов и разные их аспекты могут отличаться от одного ядра к другому, но мы попробуем описать общую процедуру выполнения процессов, которой следуют большинство известных ядер.

Этапы выполнения процесса

Чтобы выполнить процесс из исполняемого файла, в большинстве операционных систем пространства ядра и пользователя проходят через ряд общих этапов. Как уже отмечалось в предыдущем разделе, исполняемыми обычно являются объектные файлы, такие как ELF и Mach, или скрипты, которым для работы нужен интерпретатор.

С точки зрения кольца прикладных приложений для этого следует инициировать системный вызов, такой как `exec`. Стоит отметить, что мы не рассматриваем здесь системный вызов `fork`, поскольку он в действительности занимается не выполнением, а, скорее, клонированием текущего активного процесса.

Когда пользовательское пространство инициирует системный вызов `exec`, ядро получает новый запрос на запуск исполняемого файла. Оно пытается найти подходящий обработчик для файлов этого типа и затем использует *загрузчик*, чтобы загрузить содержимое исполняемого файла.

Обратите внимание: в случае со скриптом загружается программа-интерпретатор, которая обычно указывается в первой строчке скрипта вслед за символами `#!`. Чтобы выполнить процесс, загрузчик обязан произвести следующие действия:

- проверить контекст выполнения и права доступа пользователя, который запросил выполнение;
- выделить для нового процесса адресное пространство в основной памяти;
- скопировать двоичное содержимое исполняемого файла в выделенное пространство. Это в основном касается сегментов данных и кода;
- выделить область памяти для сегмента стека и подготовить начальные привязки памяти;
- создать главный поток выполнения и его сегмент стека;
- скопировать аргументы командной строки в *стековый фрейм* и поместить его на вершину стека главного потока;
- инициализировать регистры, необходимые для выполнения;
- выполнить первую инструкцию точки входа в программу.

В случае со скриптом путь к файлу копируется в качестве аргумента командной строки для процесса интерпретатора. Описанные выше этапы присущи большинству ядер, но детали их реализации могут варьироваться.

Подробности о конкретных операционных системах можно найти в их документации или с помощью Google. Если вас интересует выполнение процессов в Linux, то можете начать с отличных статей на сайте LWN: https://lwn.net/Articles/631631/ и https://lwn.net/Articles/630727/.

Далее мы перейдем к обсуждению тем, относящихся к конкурентности, и подготовимся к следующей главе, посвященной углубленному анализу методов синхронизации в многопроцессных системах. Начнем с рассмотрения разделяемых состояний, которые можно использовать в многопроцессном коде.

Разделяемые состояния

Процессы, как и потоки, могут разделять между собой состояния. Единственное отличие в том, что потоки имеют доступ к общему пространству памяти, которое принадлежит их процесс-владельцу. Процессы лишены такой роскоши, поэтому для разделения состояний между ними необходимо использовать другие механизмы.

В данном разделе мы обсудим эти методики и уделим отдельное внимание тем из них, которые играют роль хранилища. В первом подразделе мы попытаемся разделить их на группы в зависимости от того, какова их природа.

Методы разделения ресурсов

Если проанализировать методы разделения состояния (переменной или массива) между двумя процессами, окажется, что их не так уж много. Теоретически существует два общих подхода к разделению состояния между разными процессами, но в реальных компьютерных системах у каждого из них есть несколько подкатегорий.

Состояние либо размещается в каком-то «месте», доступном ряду процессов, либо *отправляется* (или *передается*) другим процессам в виде сообщения, сигнала или события. Точно так же вы должны либо *извлечь* имеющееся состояние из заданного «места», либо *принять* его в виде сообщения, сигнала или события. Для первого подхода требуется хранилище (или *носитель*), такое как буфер памяти или файловая система, а для второго — механизм обмена сообщениями, или *канал*, между процессами.

В качестве примера первого подхода вполне уместен массив, который находится в разделяемой области памяти и может быть прочитан и изменен разными процессами. Что касается второго подхода, то это может быть компьютерная сеть, служащая каналом для передачи сообщений между процессами, размещенными на разных компьютерах в данной сети.

На самом деле наше обсуждение методов разделения состояния не ограничено одними лишь процессами; все сказанное применимо и к потокам. Вдобавок потоки могут разделять и распространять состояние, обмениваясь сигналами.

В другой терминологии методики, относящиеся к первой группе и требующие *носитель* или хранилище для разделения состояний, называются *активными*. Это объясняется тем, что процесс, который хочет прочитать состояние, должен извлечь его из хранилища.

Методики из второй группы, которым нужен *канал* для передачи состояния, называются *пассивными*, поскольку состояние доставляется принимающему процессу по каналу и тот не должен извлекать его из носителя. В дальнейшем я буду использовать эти термины для описания вышеупомянутых подходов.

Изобилие пассивных методик привело к появлению в современной индустрии разработки ПО различных распределенных архитектур. Активные методики считаются устаревшими по сравнению с пассивными, и вы можете встретить их во многих корпоративных приложениях, в которых для разделения состояния всей системы используется единая центральная база данных.

Но в настоящее время все более популярным становится пассивный подход. Он привел к появлению таких шаблонов проектирования, как *источник событий* (event sourcing) и других похожих методик, которые позволяют сохранять все элементы программной системы в согласованном состоянии, не прибегая к централизованному хранению всех данных.

В этой главе нас главным образом интересует первый подход, а о втором речь пойдет в главах 19 и 20. В них будут представлены различные каналы, доступные для обмена сообщениями между разными процессами в рамках *межпроцессного взаимодействия* (inter-process communication, IPC). Только после этого можно будет перейти к активным методикам и рассмотреть реальные примеры проблем, которые наблюдаются в конкурентных системах, и управляющих механизмов, предназначенных для их решения.

Ниже перечислены активные методики, которые поддерживаются в стандарте POSIX и могут свободно использоваться во всех POSIX-совместимых операционных системах.

- *Разделяемая память*. Это просто область главной памяти, которая разделяется между разными процессами и доступна каждому из них; они могут использовать ее как обычный блок памяти для хранения переменных и массивов. Объект разделяемой памяти — настоящая оперативная память, а не файл на диске. Он может существовать в виде отдельного объекта файловой системы, даже если не используется никакими процессами. Объект разделяемой памяти может быть удален либо самим процессом за ненадобностью, либо в результате перезагрузки системы. Учитывая то, что этот объект не может пережить перезагрузку, его можно считать временным.
- *Файловая система*. Процессы могут разделять состояние с помощью файловой системы. Это одна из старейших методик разделения состояния в программной системе между разными процессами. В какой-то момент трудности синхронизации доступа к общим файлам в сочетании со многими другими справедливыми причинами привели к изобретению *систем управления базами данных* (СУБД), но данный подход до сих пор используется в определенных ситуациях.
- *Сетевые сервисы*. Процессы могут использовать сетевые хранилища или сервисы для хранения и извлечения разделяемого состояния. В этом случае процессы не знают, что именно происходит внутри. Они просто обращаются к сетевым сервисам через строго определенный API, который позволяет им выполнять различные операции с разделяемым состоянием. В качестве примеров можно привести *NFS* (network filesystems — сетевые файловые системы) и СУБД. Они предоставляют сервисы для работы с состояниями с использованием четко определенной модели и набора сопутствующих операций. В частности, можно упомянуть о *реляционных СУБД*, которые позволяют сохранять состояния в реляционной модели с помощью SQL-команд.

В следующих подразделах мы проведем обзор каждого из перечисленных выше методов в контексте интерфейса POSIX. Начнем с разделяемой памяти и узнаем, как ее использование может вызывать уже знакомую нам гонку данных, с которой мы сталкивались в главе 16.

Разделяемая память в POSIX

Разделяемая память поддерживается стандартом POSIX и является одним из самых распространенных методов разделения информации между разными процессами. Процессы, в отличие от потоков, не имеют доступа к общему пространству памяти; он запрещен на уровне операционной системы. Следовательно, нам нужен какой-то механизм, который позволит разделять участки памяти между двумя процессами, и разделяемая память — именно такой механизм.

В следующих примерах мы обсудим подробности создания и использования объекта разделяемой памяти. Для начала создадим в памяти общую область. В листинге 17.4 показано, как создать и заполнить объект разделяемой памяти в POSIX-совместимой системе.

Листинг 17.4. Создание и инициализация объекта разделяемой памяти средствами POSIX (ExtremeC_examples_chapter17_4.c)

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <string.h>
#include <sys/mman.h>

#define SH_SIZE 16

int main(int argc, char** argv) {
  int shm_fd = shm_open("/shm0", O_CREAT | O_RDWR, 0600);
  if (shm_fd < 0) {
    fprintf(stderr, "ERROR: Failed to create shared memory: %s\n",
        strerror(errno));
    return 1;
  }
  fprintf(stdout, "Shared memory is created with fd: %d\n",
          shm_fd);
  if (ftruncate(shm_fd, SH_SIZE * sizeof(char)) < 0) {
    fprintf(stderr, "ERROR: Truncation failed: %s\n",
            strerror(errno));
    return 1;
  }
  fprintf(stdout, "The memory region is truncated.\n");
  void* map = mmap(0, SH_SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);
```

```
  if (map == MAP_FAILED) {
    fprintf(stderr, "ERROR: Mapping failed: %s\n",
            strerror(errno));
    return 1;
  }
  char* ptr = (char*)map;
  ptr[0] = 'A';
  ptr[1] = 'B';
  ptr[2] = 'C';
  ptr[3] = '\n';
  ptr[4] = '\0';
  while(1);
  fprintf(stdout, "Data is written to the shared memory.\n");
  if (munmap(ptr, SH_SIZE) < 0) {
    fprintf(stderr, "ERROR: Unmapping failed: %s\n",
            strerror(errno));
    return 1;
  }
  if (close(shm_fd) < 0) {
    fprintf(stderr, "ERROR: Closing shared memory failed: %s\n",
        strerror(errno));
    return 1;
  }
  return 0;
}
```

Данный код создает объект разделяемой памяти с именем `/shm0` и 16 байтами внутри. Затем он записывает туда литерал `ABC\n` и выходит, *отвязываясь* от разделяемой области памяти. Обратите внимание: эта область остается на месте даже после завершения программы. Будущие процессы могут снова его открыть и прочитать. Объект разделяемой памяти уничтожается либо во время перезагрузки системы, либо самим процессом.

В FreeBSD имена объектов разделяемой памяти должны начинаться с /. В Linux и macOS это не обязательно, но мы все равно указали косую черту, чтобы сделать код совместимым с FreeBSD.

В начале приведенного выше листинга мы используем функцию `shm_open` для открытия объекта разделяемой памяти. Она принимает имя объекта и его режимы. Элементы `O_CREAT` и `O_RDWR` означают, что разделяемая память должна быть создана и доступна как для чтения, так и для записи.

Обратите внимание: операция создания объекта завершится успешно, даже если он уже существует. Последний аргумент определяет права доступа к разделяемой памяти. Элемент `0600` означает, что она доступна для чтения и записи со стороны процессов, инициированных владельцем соответствующего объекта.

В следующих строчках мы определяем размер разделяемой области, усекая его с помощью функции `ftruncate`. Этот шаг необходим, если вы хотите создать новый объект разделяемой памяти. В нашем примере мы указали, что нужно выделить и затем усечь 16 байт.

Далее мы привязали наш объект к области, доступной процессу, используя функцию `mmap`. В результате получился указатель на привязанную память, который можно применять для обращения к соответствующей области. Это тоже обязательный шаг, который делает разделяемую память доступной для нашей программы на языке C.

Функция `mmap` обычно служит для отображения файла или области разделяемой памяти (изначально выделенной в адресном пространстве ядра) на память, доступную вызывающему процессу. Благодаря этому к такой области можно обращаться как к любой другой, используя обычные указатели.

Элемент `PROT_WRITE` указывает на то, что область привязывается с возможностью записи, а `MAP_SHARED` означает ее разделение между процессами. Наличие `MAP_SHARED` всего лишь говорит о том, что любые изменения, внесенные в привязанную область, будут доступны всем остальным процессам, которые к ней привязаны.

Вместо `MAP_SHARED` можно было бы указать `MAP_PRIVATE`; в этом случае изменения, вносимые в привязанную область, не распространялись бы на другие процессы, оставаясь локальными. Такой подход встречается нечасто и подходит в ситуациях, когда разделяемую память нужно использовать только в рамках одного процесса.

После привязки области разделяемой памяти приведенный выше код записывает в нее строку `ABC\n`; обратите внимание на символ перевода строки в конце. Затем процесс отвязывает область разделяемой памяти с помощью функции `munmap` и закрывает файловый дескриптор, назначенный объекту этой области.

Каждая операционная система предлагает собственный способ создания неименованных или анонимных объектов разделяемой памяти. В FreeBSD для этого достаточно передать функции shm_open в качестве пути к объекту разделяемой памяти параметр SHM_ANON. В Linux вместо объекта можно создать анонимный файл, используя функцию memfd_create, и затем передать его дескриптор для создания привязанной области. Анонимная разделяемая память принадлежит владеющему ей процессу и не может применяться для разделения состояний между несколькими процессами.

Наш код можно скомпилировать в системах macOS, FreeBSD и Linux. В Linux объекты разделяемой памяти доступны в каталоге /dev/shm. Обратите внимание: он находится не в обычной файловой системе, поэтому то, что вы в нем видите, не является файлами на диске. Для /dev/shm используется файловая система `shmfs`. Она доступна только в Linux и предназначена для предоставления доступа к временным объектам внутри памяти через подключенный каталог.

Скомпилируем пример 17.4 в Linux и исследуем содержимое каталога /dev/shm. Чтобы итоговому исполняемому файлу в Linux были доступны механизмы работы с разделяемой памятью, его необходимо скомпоновать с библиотекой `rt`. Вот почему вы можете видеть параметр `-lrt` в терминале 17.4.

Терминал 17.4. Сборка и выполнение примера 17.4 с последующей проверкой того, был ли создан объект разделяемой памяти

```
$ ls /dev/shm
$ gcc ExtremeC_examples_chapter17_4.c -lrt -o ex17_4.out
$ ./ex17_4.out
Shared memory is created with fd: 3
The memory region is truncated.
Data is written to the shared memory.
$ ls /dev/shm
shm0
$
```

В первой строчке видно, что в каталоге /dev/shm нет никаких объектов разделяемой памяти. Во второй строчке мы собираем пример 17.4, а в третьей — запускаем полученный исполняемый файл. Затем еще раз проверяем каталог /dev/shm и видим, что там появился новый объект разделяемой памяти, `shm0`.

Создание объекта разделяемой памяти подтверждается и выводом программы. Еще один аспект этого терминала, заслуживающий внимания, — дескриптор `3`, который назначается объекту разделяемой памяти.

Когда вы открываете какой-либо файл, в каждом процессе для него создается новый дескриптор. Данный файл может находиться не на диске; это может быть объект разделяемой памяти, стандартный вывод и т. д. В каждом процессе файловые дескрипторы начинаются с `0` и инкрементируются, пока не достигнут максимально разрешенного числа.

Следует отметить, что в каждом процессе номера файловых дескрипторов `0`, `1` и `2` заранее назначены потокам данных `stdout`, `stdin` и `stderr` соответственно. Эти дескрипторы открываются для каждого нового процесса перед началом выполнения функции `main`. Вот почему в нашем примере объект разделяемой памяти получает файловый дескриптор `3`.

В macOS для просмотра активных IPC-объектов, существующих в системе, можно использовать утилиту pics. Она умеет выводить очереди активных сообщений и объекты разделяемой памяти. С ее помощью также можно проверить активные семафоры.

У каталога /dev/shm есть еще одно любопытное свойство. Вы можете применить утилиту `cat`, чтобы просматривать содержимое объектов разделяемой памяти, хотя эта возможность доступна только в Linux. Воспользуемся ею для просмотра нашего

нового объекта `shm0`. Его содержимое показано в терминале 17.5. Это строка `ABC` и символ перевода строки.

Терминал 17.5. Использование программы cat для просмотра содержимого объекта разделяемой памяти, созданного в рамках примера 17.4

```
$ cat /dev/shm/shm0
ABC
$
```

Как уже объяснялось ранее, объект разделяемой памяти существует, пока его использует хотя бы один процесс. Даже попросив операционную систему его удалить, процесс продолжит существовать, пока кто-то им пользуется. Однако удаление произойдет при перезагрузке системы даже без запросов со стороны процессов. Объект разделяемой памяти не может пережить перезагрузку, и процессам, которые хотят взаимодействовать, приходится создавать его заново.

В следующем примере показано, как процесс может открыть и прочитать уже существующий объект разделяемой памяти и как этот объект можно удалить. Пример 17.5 выполняет чтение из объекта, созданного в примере 17.4 (листинг 17.5). Это можно считать дополнением к коду, который был представлен ранее.

Листинг 17.5. Чтение из объекта разделяемой памяти, созданного в примере 17.4 (ExtremeC_examples_chapter17_5.c)

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <string.h>
#include <sys/mman.h>

#define SH_SIZE 16

int main(int argc, char** argv) {
  int shm_fd = shm_open("/shm0", O_RDONLY, 0600);
  if (shm_fd < 0) {
    fprintf(stderr, "ERROR: Failed to open shared memory: %s\n",
        strerror(errno));
    return 1;
  }
  fprintf(stdout, "Shared memory is opened with fd: %d\n", shm_fd);
  void* map = mmap(0, SH_SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);
  if (map == MAP_FAILED) {
    fprintf(stderr, "ERROR: Mapping failed: %s\n",
            strerror(errno));
    return 1;
  }
  char* ptr = (char*)map;
  fprintf(stdout, "The contents of shared memory object: %s\n",
          ptr);
```

```
  if (munmap(ptr, SH_SIZE) < 0) {
    fprintf(stderr, "ERROR: Unmapping failed: %s\n",
            strerror(errno));
    return 1;
  }
  if (close(shm_fd) < 0) {
    fprintf(stderr, "ERROR: Closing shared memory fd filed: %s\n",
        strerror(errno));
    return 1;
  }
  if (shm_unlink("/shm0") < 0) {
    fprintf(stderr, "ERROR: Unlinking shared memory failed: %s\n",
        strerror(errno));
    return 1;
  }
  return 0;
}
```

Первое выражение в функции `main` открывает существующий объект разделяемой памяти с именем `/shm0`. Если такого объекта не существует, то мы генерируем ошибку. Как видите, объект открывается только для чтения; это значит, что мы не будем ничего записывать в разделяемую память.

В следующих строчках мы привязываем область разделяемой памяти. Опять же, передавая аргумент `PROT_READ`, мы указываем, что эта область предназначена только для чтения. После этого мы наконец получаем указатель на разделяемую память и используем его для вывода ее содержимого. Закончив, мы отвязываем область. Дальше закрывается назначенный файловый дескриптор, и в конце объект разделяемой памяти регистрируется для удаления с помощью функции `shm_unlink`.

Когда все процессы, которые задействуют одну и ту же разделяемую память, заканчивают с ней работать, ее объект удаляется из системы. Еще раз подчеркну: объект разделяемой памяти существует, пока им пользуется хотя бы один процесс.

В терминале 17.6 показан результат выполнения приведенного выше кода. Обратите внимание на содержимое /dev/shm до и после запуска примера 17.5.

Терминал 17.6. Чтение из объекта разделяемой памяти, созданного в примере 17.4, и его удаление

```
$ ls /dev/shm
shm0
$ gcc ExtremeC_examples_chapter17_5.c -lrt -o ex17_5.out
$ ./ex17_5.out
Shared memory is opened with fd: 3
The contents of the shared memory object: ABC

$ ls /dev/shm
$
```

Пример гонки данных с использованием разделяемой памяти

Теперь пришло время продемонстрировать гонку данных при использовании функции `fork` в сочетании с разделяемой памятью. Это будет аналог примера из главы 15, в котором гонка данных была показана в контексте нескольких потоков.

В примере 17.6 у нас есть переменная-счетчик, размещенная внутри области разделяемой памяти. Код создает копию главного текущего процесса, и затем каждый процесс пытается инкрементировать разделяемый счетчик. В итоговом выводе явно видно, что это приводит к гонке данных (листинг 17.6).

Листинг 17.6. Демонстрация гонки данных при использовании разделяемой памяти POSIX и функции fork (ExtremeC_examples_chapter17_6.c)

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/wait.h>

#define SH_SIZE 4

// Разделяемый файловый дескриптор, с помощью которого
// мы ссылаемся на объект в разделяемой памяти
int shared_fd = -1;

// Указатель на разделяемый счетчик
int32_t* counter = NULL;

void init_shared_resource() {
  // Открываем объект в разделяемой памяти
  shared_fd = shm_open("/shm0", O_CREAT | O_RDWR, 0600);
  if (shared_fd < 0) {
    fprintf(stderr, "ERROR: Failed to create shared memory: %s\n",
        strerror(errno));
    exit(1);
  }
  fprintf(stdout, "Shared memory is created with fd: %d\n",
          shared_fd);
}

void shutdown_shared_resource() {
  if (shm_unlink("/shm0") < 0) {
```

```
    fprintf(stderr, "ERROR: Unlinking shared memory failed: %s\n",
        strerror(errno));
    exit(1);
  }
}

void inc_counter() {
  usleep(1);
  int32_t temp = *counter;
  usleep(1);
  temp++;
  usleep(1);
  *counter = temp;
  usleep(1);
}

int main(int argc, char** argv) {

  // Родительский процесс должен инициализировать разделяемый ресурс
  init_shared_resource();

  // Выделяем и усекаем разделяемую область памяти
  if (ftruncate(shared_fd, SH_SIZE * sizeof(char)) < 0) {
    fprintf(stderr, "ERROR: Truncation failed: %s\n",
            strerror(errno));
    return 1;
  }
  fprintf(stdout, "The memory region is truncated.\n");

  // Отражаем разделяемую память и инициализируем счетчик
  void* map = mmap(0, SH_SIZE, PROT_WRITE,
          MAP_SHARED, shared_fd, 0);
  if (map == MAP_FAILED) {
    fprintf(stderr, "ERROR: Mapping failed: %s\n",
            strerror(errno));
    return 1;
  }
  counter = (int32_t*)map;
  *counter = 0;

  // Создаем новый процесс из текущего
  pid_t pid = fork();
  if (pid) { // Родительский процесс
    // Инкрементируем счетчик
    inc_counter();
    fprintf(stdout, "The parent process sees the counter as %d.\n",
        *counter);

    // Ждем завершения дочернего процесса
    int status = -1;
```

```
    wait(&status);
    fprintf(stdout, "The child process finished with status %d.\n",
        status);
  } else { // Дочерний процесс
    // Инкрементируем счетчик
    inc_counter();
    fprintf(stdout, "The child process sees the counter as %d.\n",
        *counter);
  }

  // Оба процесса должны уничтожить отражение в области памяти
  // и закрыть свои файловые дескрипторы
  if (munmap(counter, SH_SIZE) < 0) {
    fprintf(stderr, "ERROR: Unmapping failed: %s\n",
            strerror(errno));
    return 1;
  }
  if (close(shared_fd) < 0) {
    fprintf(stderr, "ERROR: Closing shared memory fd filed: %s\n",
        strerror(errno));
    return 1;
  }

  // Только родительскому процессу нужно закрывать разделяемый ресурс
  if (pid) {
    shutdown_shared_resource();
  }

  return 0;
}
```

В этом коде, помимо `main`, есть три функции. Функция `init_shared_resource` создает объект разделяемой памяти. Причина, по которой я назвал ее именно так, а не `init_shared_memory`, обусловлена тем, что в будущих примерах такое общее имя позволит нам использовать в ней другие активные методики, не меняя функцию `main`.

Функция `shutdown_shared_resource` уничтожает разделяемую память и удаляет ее из процесса. Функция `inc_counter` увеличивает разделяемый счетчик на 1.

Функция `main` усекает и привязывает область разделяемой памяти точно так же, как это делалось в примере 17.4. После этого начинается логика копирования процесса. Вызов функции `fork` порождает новый процесс, и затем оба процесса (родительский и дочерний) пытаются инкрементировать счетчик, вызывая функцию `inc_counter`.

Когда родительский процесс изменяет разделяемый счетчик, он ждет завершения своего потомка и только затем пытается отвязать, закрыть и удалить объект разделяемой памяти. Следует отметить, что отвязка и закрытие файлового дескриптора происходит в обоих процессах, но удаление выполняет только родитель.

В листинге 17.6 внутри функции `inc_counter` используются необычные вызовы `usleep`. Это делается для того, чтобы заставить планировщик заданий передать ядро процессора от одного процесса к другому. Без вызова `usleep` процессорное ядро обычно не передается между процессами, и потому результаты разных чередований проявлялись бы не так часто.

Одна из причин такого эффекта связана с небольшим количеством инструкций в каждом процессе. Будь их значительно больше, мы смогли бы увидеть недетерминированное поведение чередований даже без вызовов usleep. Например, наличие в каждом процессе цикла, который считает до 10 000 и инкрементирует счетчик на каждой итерации, с большой долей вероятности выявило бы гонку данных. Можете сами попробовать такой способ.

Напоследок можно отметить, что родительский процесс создает и открывает объект разделяемой памяти и назначает ему файловый дескриптор до вызова функции `fork`. Скопированный процесс не открывает объект разделяемой памяти, но может использовать тот же файловый дескриптор. Тот факт, что все файловые дескрипторы наследуются от родительского процесса, помогает потомку продолжить работу, ссылаясь на тот же объект разделяемой памяти.

В листинге 17.7 показан вывод примера 17.6 после нескольких запусков. Как видите, у нас получилась явная гонка данных, относящаяся к разделяемому счетчику. В некоторых ситуациях родительский или дочерний процесс обновляет счетчик, не зная последнего измененного значения, в результате чего оба процесса выводят `1`.

Терминал 17.7. Демонстрация того, как разделяемый счетчик в примере 17.6 приводит к гонке данных

```
$ gcc ExtremeC_examples_chapter17_6 -o ex17_6.out
$ ./ex17_6.out
Shared memory is created with fd: 3
The memory region is truncated.
The parent process sees the counter as 1.
The child process sees the counter as 2.
The child process finished with status 0.
$ ./ex17_6
...
...
...
$ ./ex17_6.out
Shared memory is created with fd: 3
The memory region is truncated.
The parent process sees the counter as 1.
The child process sees the counter as 1.
The child process finished with status 0.
$
```

Я показал, как создавать и использовать разделяемую память. Кроме того, продемонстрировал пример гонки данных и то, как ведет себя конкурентная система при доступе к области разделяемой памяти. В следующем подразделе мы поговорим еще об одном распространенном активном механизме разделения состояния между разными процессами — о файловой системе.

Файловая система

В стандарте POSIX есть аналогичный API для работы с объектами файловой системы. Он ничем не отличается от того, с помощью которого мы получали доступ к разделяемой памяти. Он позволяет ссылаться на различные системные объекты — главное, чтобы для этого использовались файловые дескрипторы.

Файловые дескрипторы дают возможность ссылаться на файлы, хранящиеся в таких файловых системах, как `ext4`, а также в разделяемой памяти, каналах и т. д. Более того, вам доступна та же семантика для открытия, чтения, записи и привязки дескрипторов к локальной области памяти. Таким образом, обсуждение файловой системы и, вероятно, код для работы с ней будут напоминать то, что мы уже видели в контексте разделяемой памяти. Это будет продемонстрировано в примере 17.7.

Файловые дескрипторы обычно привязываются (отображаются). Однако существуют определенные исключительные случаи, когда привязать можно дескрипторы сокетов. Дескрипторы сокетов похожи на файловые, но используются для сетевых или Unix-сокетов. По приведенной ниже ссылке находится интересный пример привязки буфера ядра к TCP-сокету, известный как механизм приема данных без копирования: https://lwn.net/Articles/752188/.

Похожесть API для работы с файловой системой и разделяемой памяти вполне закономерна, но это не значит, что они имеют похожую реализацию. На самом деле объекты файловой системы, находящиеся на жестком диске, фундаментально отличаются от объектов разделяемой памяти. Проведем краткий обзор этих различий.

- Объект разделяемой памяти — это, в сущности, адресное пространство процесса ядра, а объект файловой системы находится на диске. В крайнем случае файл имеет выделенные буферы для операций чтения и записи.
- Состояния, записанные в разделяемую память, стираются при перезагрузке системы, а состояния, сохраненные в разделяемый файл, остаются на месте, если этот файл находится на жестком диске или на другом постоянном накопителе.
- Доступ к разделяемой памяти обычно быстрее доступа к файловой системе.

В листинге 17.7 показан такой же пример гонки данных, который мы рассматривали выше в контексте разделяемой памяти. Поскольку API для работы с файловой системой и разделяемой памятью похожи, в примере 17.6 нужно отредактировать всего две функции: `init_shared_resource` и `shutdown_shared_resource`. Больше никаких изменений не будет. Это важное достижение, которое стало возможным благодаря использованию общего API POSIX для работы с файловыми дескрипторами. Перейдем к коду.

Листинг 17.7. Демонстрация гонки данных на примере обычных файлов и функции fork (ExtremeC_examples_chapter17_7.c)

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/wait.h>

#define SH_SIZE 4

// Разделяемый файловый дескриптор, с помощью которого
// мы ссылаемся на разделяемый файл
int shared_fd = -1;

// Указатель на разделяемый счетчик
int32_t* counter = NULL;

void init_shared_resource() {
  // Открываем файл
  shared_fd = open("data.bin", O_CREAT | O_RDWR, 0600);
  if (shared_fd < 0) {
    fprintf(stderr, "ERROR: Failed to create the file: %s\n",
        strerror(errno));
    exit(1);
  }
  fprintf(stdout, "File is created and opened with fd: %d\n",
          shared_fd);
}

void shutdown_shared_resource() {
  if (remove("data.bin") < 0) {
    fprintf(stderr, "ERROR: Removing the file failed: %s\n",
        strerror(errno));
    exit(1);
  }
}
```

```
void inc_counter() {
  ... как в примере 17.6 ...
}

int main(int argc, char** argv) {
  ... как в примере 17.6 ...
}
```

Как видите, большая часть кода взята из примера 17.6. Остальное — это код, в котором вместо `shm_open` и `shm_unlink` используются функции `open` и `remove`.

Обратите внимание: файл `data.bin` создается в текущем каталоге, поскольку мы не передавали функции `open` абсолютный путь. Выполнение приведенного выше кода приводит к такой же гонке данных, относящейся к разделяемому счетчику. Результат можно проанализировать с помощью того же подхода, который применялся в примере 17.6.

До сих пор демонстрировалось, как сохранять состояние в разделяемой памяти и разделяемых файлах и обращаться к нему из разных процессов в конкурентной манере. Теперь пришло время провести более детальное сравнение многозадачности и многопроцессности.

Сравнение многопоточности и многопроцессности

Благодаря обсуждению многопоточности и многопроцессности, проведенному в главе 14, а также концепциям, рассмотренным в предыдущих главах, мы готовы к тому, чтобы сравнить их и дать общее описание ситуаций, в которых следует использовать тот или иной подход. Представьте, что мы проектируем программное обеспечение для конкурентной обработки ряда входящих запросов. Мы рассмотрим три разных сценария. Начнем с первого.

Многопоточность

В первом сценарии мы можем написать приложение, состоящее из одного процесса, принимающего все запросы. Вся логика должна быть написана в рамках единого процесса, который реализует все возможности нашей системы и в итоге становится слишком раздутым. Данное ПО является однопроцессным, и если мы хотим конкурентно обрабатывать множество запросов, то это нужно делать многопоточным образом, создавая для обработки каждого запроса отдельные потоки. Кроме того, более удачным архитектурным решением здесь будет *пул потоков* с ограниченным количеством элементов.

Ниже перечислены некоторые аспекты конкурентности и синхронизации, о которых необходимо позаботиться. Отмечу: хоть речь здесь и не идет о циклах событий и асинхронном вводе/выводе, эти механизмы могут служить разумной альтернативой многозадачности.

Если количество запросов существенно возрастает, то для того, чтобы справиться с нагрузкой, следует увеличить количество потоков в пуле. Это буквально требует обновления аппаратного обеспечения и ресурсов компьютера, на котором выполняется главный процесс. Такой подход называется *вертикальным масштабированием*. Он подразумевает увеличение производительности отдельно взятого компьютера, чтобы тот мог обработать больше запросов. Помимо времени простоя, необходимого для установки нового оборудования (хотя его можно свести к нулю), такие обновления требуют денежных расходов, и их приходится выполнять каждый раз, когда запросов становится больше.

Если в ходе обработки запросов вносятся изменения в разделяемое состояние или хранилище данных, то мы можем легко применить методы синхронизации, зная о том, что потоки имеют доступ к общему пространству памяти. Конечно, это требуется как в случае с разделяемой структурой данных, которую нужно хранить, так и при доступе к удаленному хранилищу данных, которое не является транзакционным.

Все потоки выполняются на одном компьютере и могут использовать все те методики разделения состояния, которые мы рассматривали ранее (и которые подходят не только для потоков, но и для процессов). Эта прекрасная особенность существенно облегчает синхронизацию потоков.

Теперь обсудим второй сценарий, в котором мы имеем дело с несколькими процессами, размещенными на одном компьютере.

Локальная многопроцессность

В этом сценарии у нашего программного обеспечения есть несколько процессов, но все они развернуты на одном компьютере. Процессы могут быть однопоточными или содержать внутри себя пул потоков, который позволяет каждому из них обрабатывать сразу несколько запросов.

При увеличении количества запросов мы можем создавать новые процессы вместо новых потоков. Это обычно называется *горизонтальным масштабированием*. Однако при наличии всего одного компьютера масштабирование должно происходить вертикально, то есть за счет обновления аппаратного обеспечения. Это чревато теми же проблемами, которые упоминались в предыдущем подразделе в контексте вертикального масштабирования многопоточной программы.

Процессы выполняются в конкурентной среде. В целях разделения состояния и синхронизации они могут использовать только многопроцессные методики. Это, безусловно, не настолько удобно, как писать многопоточный код. Кроме того, для разделения состояний процессы могут применять как активные, так и пассивные механизмы.

Многопроцессность на одном компьютере не очень эффективна, и многопоточность, по всей видимости, более удобна с точки зрения усилий, необходимых для написания кода.

В следующем подразделе речь пойдет о распределенной многопроцессной среде, которая считается лучшим архитектурным решением для создания современного ПО.

Распределенная многопроцессность

В нашем заключительном сценарии мы написали программу, которая состоит из нескольких процессов, выполняемых на разных компьютерах. Все они соединены с помощью сети, и на каждом из них может находиться несколько процессов. Такая конфигурация имеет свои особенности.

При существенном увеличении количества запросов данная система может горизонтально масштабироваться без каких-либо ограничений. Это отличное свойство, позволяющее справляться с пиковыми нагрузками за счет добавления нового потребительского оборудования. Объединение данного оборудования в кластеры, вместо того чтобы использовать мощные серверы, было одной из тех идей, которые позволили компании Google выполнять свои алгоритмы *Page Rank* и *Map Reduce* на кластерах компьютеров.

Подходы, рассмотренные в текущей главе, мало чем помогают, поскольку у всех у них есть одно важное требование: все процессы должны находиться на одном компьютере. Следовательно, чтобы синхронизировать процессы и предоставить всем им доступ к разделяемым состояниям, нужен совершенно другой набор алгоритмов и методик. В таких распределенных системах следует анализировать и оптимизировать *латентность, отказоустойчивость, доступность, согласованность данных* и многие другие факторы.

Процессы, размещенные на разных компьютерах, взаимодействуют в пассивной манере с помощью сетевых сокетов, в то время как процессы, принадлежащие одному компьютеру, могут использовать для передачи данных и разделения состояния локальные методы IPC, такие как очереди сообщений, разделяемая память, каналы и т. д.

В заключение отмечу: в современной компьютерной индустрии предпочтение отдается не вертикальному, а горизонтальному масштабированию. Благодаря этому возникло множество новых идей и технологий хранения данных, синхронизации, обмена сообщениями и т. д. Существуют даже аппаратные архитектуры, разработанные специально для поддержки горизонтального масштабирования.

Резюме

В этой главе мы исследовали многопроцессные системы и различные методики, с помощью которых можно разделять состояние между разными процессами:

- познакомились с API POSIX для выполнения процессов и объяснили, как работают функции `fork` и `exec*`;
- рассмотрели этапы, через которые проходит ядро при выполнении процесса;
- обсудили разные способы разделения состояния между разными процессами;
- познакомились с двумя общими категориями, к которым можно отнести все методики синхронизации: активными и пассивными;
- узнали, что разделяемая память и разделяемые объекты файловой системы — одни из самых распространенных методов разделения состояния в активной манере;
- рассмотрели сходства и различия между многопоточными и многопроцессными конфигурациями, а также такие понятия, как вертикальное и горизонтальное масштабирование в распределенной программной системе.

В следующей главе речь пойдет о конкурентности в локальных многопроцессных средах. Мы поговорим о проблемах конкурентности и способах синхронизации разных процессов для защиты разделяемых ресурсов. Эти темы очень похожи на те, с которыми мы уже сталкивались в главе 16, однако на сей раз внимание будет уделяться не потокам, а процессам.

18 Синхронизация процессов

Здесь мы продолжаем обсуждение, начатое в предыдущей главе, и теперь в центре нашего внимания — синхронизация процессов. Управляющие механизмы в многопроцессных программах отличаются от тех методик, с которыми мы познакомились при обсуждении многопоточности. Отличается не только память; отдельные факторы существуют лишь в многопроцессных средах, и вы не встретите их в многопоточной программе.

В отличие от потоков, привязанных к одному процессу, сами процессы могут свободно выполняться на любых компьютерах с любыми операционными системами и в любого рода сети, даже в такой большой, как Интернет. Как можно себе представить, это все усложняет. Синхронизировать разные процессы в такой распределенной системе будет нелегко.

Эта глава посвящена синхронизации процессов, происходящей на одном компьютере. Иными словами, основное внимание будет уделено локальной синхронизации и сопутствующим методикам. Мы также затронем синхронизацию процессов в распределенных системах, однако не станем углубляться в подробности.

Мы рассмотрим следующие темы.

- Вначале будет описано многопроцессное программное обеспечение, в котором все процессы выполняются на одном компьютере. Мы познакомимся с методиками, доступными в таких средах, и задействуем знания, полученные в прошлой главе, чтобы показать примеры использования этих методик.
- В нашей первой попытке синхронизировать разные процессы будут использоваться именованные POSIX-семафоры. Мы выясним, как их следует применять, и затем рассмотрим примеры решения проблемы с состоянием гонки, с которой сталкивались в предыдущих главах.
- После этого речь пойдет об именованных POSIX-мьютексах; будет показано, как подготовить их к работе с помощью областей разделяемой памяти. В качестве примера попробуем избавиться от того же состояния гонки, которое было устранено благодаря семафорам, однако на сей раз применим именованные мьютексы.

- Последней методикой синхронизации процессов, которую мы рассмотрим, будут именованные условные переменные POSIX. Как и именованные мьютексы, они должны размещаться в области разделяемой памяти, чтобы к ним могли обращаться разные процессы. Я приведу подробный пример использования этого метода, в котором будет показана синхронизация многопроцессной системы с помощью условных переменных POSIX.
- В конце главы будет затронута тема многопроцессных систем, процессы которых распределены по сети. Мы обсудим их свойства и дополнительные проблемы, присущие им в сравнении с локальными многопроцессными системами.

Начну эту главу с краткого обзора механизмов управления конкурентностью на одном компьютере и методик, которые для них доступны.

Локальное управление конкурентностью

Довольно часто возникает ситуация, когда несколько процессов выполняются на одном компьютере и пытаются одновременно обратиться к разделяемому ресурсу. Поскольку все процессы работают в рамках одной операционной системы, они имеют доступ ко всем механизмам, которые она предоставляет.

В данном разделе я покажу, как с помощью некоторых из этих механизмов управлять синхронизацией процессов. Ключевую роль в большинстве этих механизмов играет разделяемая память, вследствие чего в дальнейшем мы будем в значительной мере опираться на материал, рассмотренный в предыдущей главе.

Ниже перечислены управляющие механизмы, предусмотренные в стандарте POSIX, которые можно применять, когда все процессы выполняются на одном POSIX-совместимом компьютере.

- *Именованные POSIX-семафоры*. Те же POSIX-семафоры, которые были рассмотрены в главе 16, но с одним отличием: теперь у них есть имя, благодаря чему их можно использовать глобально по всей системе. То есть это уже не *анонимные* или *приватные* семафоры.
- *Именованные мьютексы*. Те же мьютексы с теми же свойствами, о которых шла речь в главе 16, но с именами и возможностью использования по всей системе. Чтобы сделать эти мьютексы доступными разным процессам, их необходимо поместить в разделяемую память.
- *Именованные условные переменные*. Те же условные переменные POSIX, которые обсуждались в главе 16, но, как и в случае с мьютексами, их необходимо размещать в объекте разделяемой памяти, чтобы они были доступны разным процессам.

Далее мы поговорим обо всех этих методиках и рассмотрим их работу. Следующий раздел посвящен именованным POSIX-семафорам.

Именованные POSIX-семафоры

Как вы уже знаете из главы 16, семафоры — основное средство синхронизации разных конкурентных заданий. Они уже встречались нам в многопоточных программах, в которых позволяли преодолевать проблемы конкурентности.

В этом разделе я покажу, как задействовать их в контексте процессов. Пример 18.1 демонстрирует применение POSIX-семафора в целях устранения гонки данных, с которыми мы сталкивались в примерах 17.6 и 17.7 в предыдущей главе. Представленный ниже код очень похож на пример 17.6; в нем тоже используется область разделяемой памяти для хранения переменной с разделяемым счетчиком. Однако на сей раз для синхронизации доступа к счетчику применяются именованные семафоры.

В следующих листингах демонстрируется использование именованного семафора для синхронизации двух процессов, обращающихся к разделяемой переменной. Начнем с глобальных объявлений в примере 18.1 (листинг 18.1).

Листинг 18.1. Глобальные объявления в примере 18.1 (ExtremeC_examples_chapter18_1.c)

```
#include <stdio.h>
...
#include <semaphore.h>  // Для использования семафоров

#define SHARED_MEM_SIZE 4

// Разделяемый файловый дескриптор, с помощью которого
// мы ссылаемся на объект в разделяемой памяти
int shared_fd = -1;

// Указатель на разделяемый счетчик
int32_t* counter = NULL;

// Указатель на разделяемый семафор
sem_t* semaphore = NULL;
```

Здесь объявляются глобальный счетчик и глобальный указатель на объект семафора, который будет инициализирован позже. Этот указатель позволит родительскому и дочернему процессам иметь синхронизированный доступ к разделяемому счетчику через его указатель.

В коде листинга 18.2 показаны определения функций, которые будут заниматься синхронизацией процессов. Часть этих определений вы уже видели в примере 17.6, и потому они убраны из данного листинга.

Листинг 18.2. Определение функций синхронизации (ExtremeC_examples_chapter18_1.c)

```
void init_control_mechanism() {
  semaphore = sem_open("/sem0", O_CREAT | O_EXCL, 0600, 1);
  if (semaphore == SEM_FAILED) {
    fprintf(stderr, "ERROR: Opening the semaphore failed: %s\n",
        strerror(errno));
    exit(1);
  }
}

void shutdown_control_mechanism() {
  if (sem_close(semaphore) < 0) {
    fprintf(stderr, "ERROR: Closing the semaphore failed: %s\n",
        strerror(errno));
    exit(1);
  }
  if (sem_unlink("/sem0") < 0) {
    fprintf(stderr, "ERROR: Unlinking failed: %s\n",
        strerror(errno));
    exit(1);
  }
}

void init_shared_resource() {
  ... как в примере 17.6 ...
}

void shutdown_shared_resource() {
  ... как в примере 17.6 ...
}
```

Если сравнивать с примером 17.6, то у нас появились две новые функции: `init_control_mechanism` и `shutdown_control_mechanism`. Мы также внесли некоторые изменения в функцию `inc_counter` (показана в листинге 18.3, см. ниже), добавив в нее семафор и сформировав внутри критический участок.

Внутри функций `init_control_mechanism` и `shutdown_control_mechanism` используется API для открытия, закрытия и удаления семафора, подобный интерфейсу разделяемой памяти.

Функции `sem_open`, `sem_close` и `sem_unlink` можно считать аналогами `shm_open`, `shm_close` и `shm_unlink`. Но есть одно отличие: функция `sem_open` возвращает указатель на семафор, а не файловый дескриптор.

Обратите внимание: в данном примере используется тот же API для работы с семафорами, что и прежде, поэтому остальной код остается без изменений и не отличается от примера 17.6. Мы инициализируем семафор с помощью значения `1`, превращая его тем самым в мьютекс. В листинге 18.3 показаны критический

участок и процедура синхронизации операций чтения и записи разделяемого счетчика с помощью семафора.

Листинг 18.3. Критический участок, в котором инкрементируется разделяемый счетчик (ExtremeC_examples_chapter18_1.c)

```
void inc_counter() {
  usleep(1);
  sem_wait(semaphore); // Возвращаемое значение должно проверяться
  int32_t temp = *counter;
  usleep(1);
  temp++;
  usleep(1);
  *counter = temp;
  sem_post(semaphore); // Возвращаемое значение должно проверяться
  usleep(1);
}
```

Если сравнивать с функцией `inc_counter` из примера 17.6, то для входа на критический участок и выхода с него используются вызовы `sem_wait` и `sem_post` соответственно.

В листинге 18.4 представлена функция `main`. Она почти ничем не отличается от той, которую вы видели в примере 17.6; изменения наблюдаются только в начале и конце, где находится код двух новых функций, показанных в листинге 18.2.

Листинг 18.4. Главная функция примера 18.1 (ExtremeC_examples_chapter18_1.c)

```
int main(int argc, char** argv) {

  // Родительский процесс должен инициализировать разделяемый ресурс
  init_shared_resource();

  // Родительский процесс должен инициализировать механизм управления
  init_control_mechanism();

  ... как в примере 17.6 ...

  // Только родительский процесс должен закрывать разделяемый ресурс
  // и задействованный механизм управления
  if (pid) {
    shutdown_shared_resource();
    shutdown_control_mechanism();
  }

  return 0;
}
```

В терминале 18.1 можно видеть вывод после двух успешных выполнений примера 18.1.

Терминал 18.1. Сборка и два последовательных запуска примера 18.1 в Linux

```
$ gcc ExtremeC_examples_chapter18_1.c -lrt -lpthread -o ex18_1.out
$ ./ex18_1.out
Shared memory is created with fd: 3
The memory region is truncated.
The child process sees the counter as 1.
The parent process sees the counter as 2.
The child process finished with status 0.
$ ./ex18_1.out
Shared memory is created with fd: 3
The memory region is truncated.
The parent process sees the counter as 1.
The child process sees the counter as 2.
The child process finished with status 0.
$
```

Обратите внимание: этот код нужно скомпоновать с библиотекой `pthread`, поскольку мы используем POSIX-семафоры. В Linux также необходимо провести компоновку с библиотекой `rt`, чтобы иметь доступ к разделяемой памяти.

В показанном выше выводе все понятно. Иногда первым получает ресурсы процессора и инкрементирует счетчик дочерний процесс, а иногда — родительский. Они не могут зайти на критический участок одновременно, поэтому наш код соблюдает целостность данных в отношении разделяемого счетчика.

Отмечу, что для работы с именованными семафорами не обязательно использовать функцию `fork`. Один и тот же семафор могут открыть совершенно разные процессы, которые не являются родителем и потомком, — они всего лишь должны выполняться на одном компьютере и в одной и той же операционной системе. Это будет продемонстрировано в примере 18.3.

В заключение следует сказать, что в Unix-подобных операционных системах существует два вида именованных семафоров: *семафоры System V* и *POSIX-семафоры*. В этом разделе использовались последние, так как благодаря своей производительности и удобному API их репутация куда лучше. Ниже приводится вопрос на сайте Stack Overflow, в котором хорошо описаны различия между семафорами System V и POSIX-семафорами: https://stackoverflow.com/questions/368322/differences-between-system-v-and-posix-semaphores.

Если говорить о работе с семафорами, то Microsoft Windows не является POSIX-совместимой ОС. У нее есть собственный API для создания и управления семафорами.

В следующем разделе мы обсудим именованные мьютексы. Если коротко, то это обычные мьютексы, размещенные в области разделяемой памяти.

Именованные мьютексы

В многопоточных программах POSIX-мьютексы работают довольно просто; это было показано в главе 16. Однако в многопроцессных средах все немного иначе. Чтобы мьютекс можно было использовать в разных процессах, его необходимо определить там, где он будет им доступен.

Лучшее место подобного рода — область разделяемой памяти. Следовательно, чтобы получить мьютекс, который работает в многопроцессной среде, его нужно распределить именно в этой области.

Первый пример

Следующий пример, 18.2, — копия предыдущего, но для борьбы с потенциальным состоянием гонки в нем применяются именованные мьютексы вместо именованных семафоров. В нем также показано, как создать область разделяемой памяти и сохранить в ней разделяемый мьютекс.

Поскольку каждый объект разделяемой памяти имеет глобальное имя, мьютекс, размещаемый в этой области, можно считать *именованным*, и обращаться к нему могут другие процессы в системе.

В листинге 18.5 показаны объявления, необходимые для примера 18.2. Это то, что требуется для создания разделяемого мьютекса.

Листинг 18.5. Глобальные объявления в примере 18.2 (ExtremeC_examples_chapter18_2.c)

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/wait.h>
#include <pthread.h> // Для использования функций pthread_mutex_*

#define SHARED_MEM_SIZE 4

// Разделяемый файловый дескриптор для обращения к объекту
// в разделяемой памяти
int shared_fd = -1;

// Разделяемый файловый дескриптор для обращения
// к объекту в разделяемой памяти мьютекса
int mutex_shm_fd = -1;
```

```
// Указатель на разделяемый счетчик
int32_t* counter = NULL;

// Указатель на разделяемый мьютекс
pthread_mutex_t* mutex = NULL;
```

Здесь мы объявили:

- глобальный файловый дескриптор для обращения к области разделяемой памяти, в которой должна храниться переменная разделяемого счетчика;
- глобальный файловый дескриптор для области разделяемой памяти, в которой хранится разделяемый мьютекс;
- указатель на разделяемый счетчик;
- указатель на разделяемый мьютекс.

Ниже эти переменные будут инициализированы соответствующим образом.

В следующих листингах показаны все те функции, которые у нас были в примере 18.1, но, как видите, мы обновили их определения, чтобы они работали с именованным мьютексом вместо именованного семафора. Начнем с функции `init_control_mechanism` (листинг 18.6).

Листинг 18.6. Функция init_control_mechanism в примере 18.2 (ExtremeC_examples_chapter18_2.c)

```
void init_control_mechanism() {
  // Открываем разделяемую память мьютекса
  mutex_shm_fd = shm_open("/mutex0", O_CREAT | O_RDWR, 0600);
  if (mutex_shm_fd < 0) {
    fprintf(stderr, "ERROR: Failed to create shared memory: %s\n"
        , strerror(errno));
    exit(1);
  }
  // Выделяем и усекаем разделяемую область памяти мьютекса
  if (ftruncate(mutex_shm_fd, sizeof(pthread_mutex_t)) < 0) {
    fprintf(stderr, "ERROR: Truncation of mutex failed: %s\n",
        strerror(errno));
    exit(1);
  }
  // Отображаем разделяемую память мьютекса
  void* map = mmap(0, sizeof(pthread_mutex_t),
          PROT_READ | PROT_WRITE, MAP_SHARED, mutex_shm_fd, 0);
  if (map == MAP_FAILED) {
    fprintf(stderr, "ERROR: Mapping failed: %s\n",
            strerror(errno));
    exit(1);
  }
  mutex = (pthread_mutex_t*)map;
  // Инициализируем объект мьютекса
  int ret = -1;
```

```
  pthread_mutexattr_t attr;
  if ((ret = pthread_mutexattr_init(&attr))) {
    fprintf(stderr, "ERROR: Failed to init mutex attrs: %s\n",
        strerror(ret));
    exit(1);
  }
  if ((ret = pthread_mutexattr_setpshared(&attr,
                PTHREAD_PROCESS_SHARED))) {
    fprintf(stderr, "ERROR: Failed to set the mutex attr: %s\n",
        strerror(ret));
    exit(1);
  }
  if ((ret = pthread_mutex_init(mutex, &attr))) {
    fprintf(stderr, "ERROR: Initializing the mutex failed: %s\n",
        strerror(ret));
    exit(1);
  }
  if ((ret = pthread_mutexattr_destroy(&attr))) {
    fprintf(stderr, "ERROR: Failed to destroy mutex attrs : %s\n"
        , strerror(ret));
    exit(1);
  }
}
```

Внутри функции `init_control_mechanism` мы создали новый объект разделяемой памяти с именем `/mutex0`. В качестве размера области разделяемой памяти указано значение `sizeof(pthread_mutex_t)`; это говорит о том, что мы хотим разделить объект POSIX-мьютекса.

Далее мы получаем указатель на область разделяемой памяти. Теперь у нас есть мьютекс, выделенный в разделяемой памяти, однако нам все еще нужно его инициализировать. Для этого мы вызываем функцию `pthread_mutex_init` и передаем ей атрибуты, которые говорят о том, что объект мьютекса должен быть разделяемым и доступным для других процессов. Это особенно важно, поскольку в противном случае мьютекс не будет работать в многопроцессной среде, даже если поместить его в область разделяемой памяти. Как было показано в предыдущем листинге в функции `init_control_mechanism`, мы установили атрибут `PTHREAD_PROCESS_SHARED`, чтобы сделать мьютекс разделяемым. Рассмотрим следующую функцию (листинг 18.7)

Листинг 18.7. Функция destroy_control_mechanism в примере 18.2 (ExtremeC_examples_chapter18_2.c)

```
void shutdown_control_mechanism() {
  int ret = -1;
  if ((ret = pthread_mutex_destroy(mutex))) {
    fprintf(stderr, "ERROR: Failed to destroy mutex: %s\n",
        strerror(ret));
    exit(1);
  }
```

```
  if (munmap(mutex, sizeof(pthread_mutex_t)) < 0) {
    fprintf(stderr, "ERROR: Unmapping the mutex failed: %s\n",
        strerror(errno));
    exit(1);
  }
  if (close(mutex_shm_fd) < 0) {
    fprintf(stderr, "ERROR: Closing the mutex failed: %s\n",
        strerror(errno));
    exit(1);
  }
  if (shm_unlink("/mutex0") < 0) {
    fprintf(stderr, "ERROR: Unlinking the mutex failed: %s\n",
        strerror(errno));
    exit(1);
  }
}
```

В функции `destroy_control_mechanism` мы уничтожаем объект мьютекса и затем закрываем и удаляем соответствующую область разделяемой памяти. Это ничем не отличается от уничтожения обычного объекта разделяемой памяти. Перейдем к следующему фрагменту кода (листинг 18.8).

Листинг 18.8. Эти функции мы уже видели в примере 18.1 (ExtremeC_examples_chapter18_2.c)

```
void init_shared_resource() {
  ... как в примере 18.1 ...
}

void shutdown_shared_resource() {
  ... как в примере 18.1 ...
}
```

Показанные выше функции не претерпели никаких изменений по сравнению с примером 18.1. Взглянем на критический участок в функции `inc_counter`, которая теперь использует именованный мьютекс вместо именованного семафора (листинг 18.9).

Листинг 18.9. Для защиты разделяемого счетчика на критическом участке теперь используется именованный мьютекс (ExtremeC_examples_chapter18_2.c)

```
void inc_counter() {
  usleep(1);
  pthread_mutex_lock(mutex); // Нужно проверить возвращаемое значение
  int32_t temp = *counter;
  usleep(1);
  temp++;
  usleep(1);
  *counter = temp;
  pthread_mutex_unlock(mutex); // Нужно проверить возвращаемое значение
  usleep(1);
}
```

```
int main(int argc, char** argv) {
  ... как в примере 18.1 ...
}
```

В целом, как показывают эти листинги, мы изменили всего несколько участков по сравнению с примером 18.1, и лишь три функции поменялись существенно. Например, функция `main` осталась точно в таком же виде, в котором была в примере 18.1. Это объясняется тем, что мы заменили лишь управляющий механизм, не трогая остальную логику.

Завершая рассматривать листинг 18.9, отметим: в функции `inc_counter` объект мьютекса используется точно так же, как это делалось в многопоточной программе. API абсолютно аналогичен; он спроектирован таким образом, чтобы его можно было применять как в многопоточных, так и в многопроцессных средах. Эта замечательная особенность POSIX-мьютексов позволяет нам использовать один и тот же код, независимо от того, что работает с этими объектами: потоки или процессы. Хотя, конечно, инициализация и уничтожение могут отличаться.

Вывод представленного выше кода очень похож на тот, который мы наблюдали в примере 18.1. В данном случае разделяемый счетчик защищен мьютексом, а не семафором, как раньше. Семафор, который применялся в предыдущем примере, был двоичным, и, согласно объяснениям в главе 16, двоичные семафоры могут имитировать мьютексы. Таким образом, в примере 18.2 мало что изменилось.

Второй пример

Именованные объекты разделяемой памяти и мьютексы можно применять в любом процессе, запущенном в системе. И этот процесс не обязательно должен быть клонированным. В примере 18.3 мы рассмотрим совместное использование разделяемого мьютекса и разделяемой памяти для одновременного завершения нескольких процессов, которые работают параллельно. Мы хотим, чтобы при нажатии `Ctrl+C` в одном из процессов все они немедленно прекращали работу.

Обратите внимание: код будет представлен поэтапно. Вслед за каждым этапом идут комментарии. Начнем с глобальных определений.

Этап 1: глобальные определения

В этом примере мы будем работать с одним исходным файлом, который можно будет скомпилировать и многократно запустить для создания нескольких процессов. В целях синхронизации своего выполнения эти процессы используют области разделяемой памяти. Один из них выбирает владелец этих областей; он будет отвечать за их создание и уничтожение. Другие процессы просто применяют уже созданную разделяемую память.

На первом этапе мы объявим глобальные объекты, которые понадобятся нам на разных участках кода. Позже они будут инициализированы. Обратите внимание: глобальные переменные, объявленные в листинге 18.10 (такие как `mutex`), не разделяются между процессами. Они существуют в адресном пространстве каждого отдельного процесса, но привязываются к разным областям разделяемой памяти.

Листинг 18.10. Глобальные объявления в примере 18.3 (ExtremeC_examples_chapter18_3.c)

```
#include <stdio.h>
...
#include <pthread.h> // Для использования функций pthread_mutex_*

typedef uint16_t bool_t;

#define TRUE 1
#define FALSE 0

#define MUTEX_SHM_NAME "/mutex0"
#define SHM_NAME "/shm0"

// Разделяемый файловый дескриптор для обращения к объекту разделяемой
// памяти, содержащему флаг отмены
int cancel_flag_shm_fd = -1;

// Флаг, определяющий, владеет ли текущий процесс
// объектом разделяемой памяти
bool_t cancel_flag_shm_owner = FALSE;

// Разделяемый файловый дескриптор для обращения к объекту разделяемой
// памяти мьютекса
int mutex_shm_fd = -1;

// Разделяемый мьютекс
pthread_mutex_t* mutex = NULL;

// Флаг, определяющий, владеет ли текущий процесс
// объектом разделяемой памяти
bool_t mutex_owner = FALSE;

// Указатель на флаг отмены, хранящийся в разделяемой памяти
bool_t* cancel_flag = NULL;
```

В этом коде мы можем видеть глобальные объявления. Разделяемый флаг позволит уведомлять процессы о сигнале отмены. Отмечу, что в данном примере для ожидания момента, когда флаг отмены станет равен `true`, будет использоваться холостой цикл.

Здесь, как и в примере 18.2, у нас есть два отдельных объекта разделяемой памяти: один для флага отмены, а другой — для мьютекса, который защищает данный флаг.

Я мог бы создать единую структуру и определить в ней оба объекта в виде полей, что позволило бы хранить их в одной области разделяемой памяти. Но я решил использовать для этого разные области.

Одним из важных нюансов при работе с объектами разделяемой памяти в этом примере является то, что очистку должен производить процесс, который их изначально создал и инициализировал. Поскольку все процессы используют один и тот же код, нам нужно каким-то образом определить владельца объекта. Таким образом, когда мы дойдем до очистки, удалить объект сможет только его владелец. В связи с этим нам пришлось объявить две булевы переменные: `mutex_owner` и `cancel_flag_shm_owner`.

Этап 2: разделяемая память флага отмены

В листинге 18.11 показана процедура инициализации области разделяемой памяти, отведенной флагу отмены.

Листинг 18.11. Инициализация разделяемой памяти флага отмены (ExtremeC_examples_chapter18_3.c)

```
void init_shared_resource() {
  // Открываем объект разделяемой памяти
  cancel_flag_shm_fd = shm_open(SHM_NAME, O_RDWR, 0600);
  if (cancel_flag_shm_fd >= 0) {
    cancel_flag_shm_owner = FALSE;
    fprintf(stdout, "The shared memory object is opened.\n");
  } else if (errno == ENOENT) {
    fprintf(stderr,
            "WARN: The shared memory object doesn't exist.\n");
    fprintf(stdout, "Creating the shared memory object ...\n");
    cancel_flag_shm_fd = shm_open(SHM_NAME,
           O_CREAT | O_EXCL | O_RDWR, 0600);
    if (cancel_flag_shm_fd >= 0) {
      cancel_flag_shm_owner = TRUE;
      fprintf(stdout, "The shared memory object is created.\n");
    } else {
      fprintf(stderr,
          "ERROR: Failed to create shared memory: %s\n",
          strerror(errno));
      exit(1);
    }
  } else {
      fprintf(stderr,
          "ERROR: Failed to create shared memory: %s\n",
          strerror(errno));
    exit(1);
  }
  if (cancel_flag_shm_owner) {
    // Выделяем и усекаем разделяемую область памяти
```

```
    if (ftruncate(cancel_flag_shm_fd, sizeof(bool_t)) < 0) {
      fprintf(stderr, "ERROR: Truncation failed: %s\n",
              strerror(errno));
      exit(1);
    }
    fprintf(stdout, "The memory region is truncated.\n");
  }
  // Отображаем разделяемую память и инициализируем флаг отмены
  void* map = mmap(0, sizeof(bool_t), PROT_WRITE, MAP_SHARED,
      cancel_flag_shm_fd, 0);
  if (map == MAP_FAILED) {
    fprintf(stderr, "ERROR: Mapping failed: %s\n",
            strerror(errno));
    exit(1);
  }
  cancel_flag = (bool_t*)map;
  if (cancel_flag_shm_owner) {
    *cancel_flag = FALSE;
  }
}
```

Выбранный нами подход отличается от того, который использовался в примере 18.2. Дело в том, что каждый новый процесс должен проверять, был ли объект разделяемой памяти создан другим процессом. Обратите внимание: в этом примере создание новых процессов обходится без функции `fork`, и пользователь может самостоятельно создавать новые процессы в своей командной оболочке.

В связи с этим новый процесс сначала пытается открыть область разделяемой памяти, предоставляя один лишь флаг `O_RDWR`. Успех будет признаком того, что текущий процесс не владеет данной областью, поэтому на следующем шаге привязывает его. Неудача означает отсутствие области разделяемой памяти, и это говорит о том, что текущий процесс должен ее создать и стать ее владельцем. Таким образом, он продолжает работу и пытается открыть область с помощью других флагов: `O_CREAT` и `O_EXCL`. Эти флаги создают объект разделяемой памяти, если он еще не существует.

Если создание пройдет успешно, то текущий процесс становится владельцем области разделяемой памяти и переходит к ее усечению и привязке.

В описанном выше сценарии существует небольшой шанс, что другой процесс создаст ту же область разделяемой памяти прямо между двумя вызовами `shm_open`, в результате чего второй из них завершится неудачей. Флаг `O_EXCL` не дает текущему процессу создать объект, если тот уже существует; в таком случае процесс завершает работу, показывая подходящее сообщение об ошибке. Если это произойдет (что маловероятно), то мы всегда можем запустить процесс повторно, и во второй раз подобной проблемы уже не будет.

Код, показанный в листинге 18.12, выполняет обратную процедуру: уничтожает флаг отмены и его область разделяемой памяти.

Листинг 18.12. Освобождение ресурсов, выделенных для разделяемой памяти флага отмены (ExtremeC_examples_chapter18_3.c)

```
void shutdown_shared_resource() {
  if (munmap(cancel_flag, sizeof(bool_t)) < 0) {
    fprintf(stderr, "ERROR: Unmapping failed: %s\n",
            strerror(errno));
    exit(1);
  }
  if (close(cancel_flag_shm_fd) < 0) {
    fprintf(stderr,
        "ERROR: Closing the shared memory fd filed: %s\n",
        strerror(errno));
    exit(1);
  }
  if (cancel_flag_shm_owner) {
    sleep(1);
    if (shm_unlink(SHM_NAME) < 0) {
      fprintf(stderr,
          "ERROR: Unlinking the shared memory failed: %s\n",
          strerror(errno));
      exit(1);
    }
  }
}
```

Логика освобождения объекта разделяемой памяти, представленная в данном листинге, очень похожа на ту, которую мы видели в предыдущих примерах. Но здесь есть одно отличие: удалить объект разделяемой памяти может только владеющий им процесс. Обратите внимание: прежде чем удалять объект, владелец ждет 1 секунду, чтобы другие процессы успели освободить свои ресурсы. Эта задержка обычно не требуется, поскольку в большинстве POSIX-совместимых систем объект разделяемой памяти остается на месте, пока не завершатся все процессы, которые зависят от него.

Этап 3: разделяемая память именованного мьютекса

В листинге 18.13 показано, как инициализировать разделяемый мьютекс и связанный с ним объект разделяемой памяти.

Листинг 18.13. Инициализация разделяемого мьютекса и соответствующей области разделяемой памяти (ExtremeC_examples_chapter18_3.c)

```
void init_control_mechanism() {
  // Открываем разделяемую память мьютекса
  mutex_shm_fd = shm_open(MUTEX_SHM_NAME, O_RDWR, 0600);
  if (mutex_shm_fd >= 0) {
    // Разделяемый объект мьютекса существует, и я теперь его владелец
    mutex_owner = FALSE;
```

```
  fprintf(stdout,
          "The mutex's shared memory object is opened.\n");
} else if (errno == ENOENT) {
  fprintf(stderr,
          "WARN: Mutex's shared memory doesn't exist.\n");
  fprintf(stdout,
          "Creating the mutex's shared memory object ...\n");
  mutex_shm_fd = shm_open(MUTEX_SHM_NAME,
          O_CREAT | O_EXCL | O_RDWR, 0600);
  if (mutex_shm_fd >= 0) {
    mutex_owner = TRUE;
    fprintf(stdout,
            "The mutex's shared memory object is created.\n");
  } else {
    fprintf(stderr,
        "ERROR: Failed to create mutex's shared memory: %s\n",
        strerror(errno));
    exit(1);
  }
} else {
  fprintf(stderr,
      "ERROR: Failed to create mutex's shared memory: %s\n",
      strerror(errno));
  exit(1);
}
if (mutex_owner) {
  // Выделяем и усекаем область разделяемой памяти мьютекса
  if (ftruncate(mutex_shm_fd, sizeof(pthread_mutex_t)) < 0) {
    fprintf(stderr,
        "ERROR: Truncation of the mutex failed: %s\n",
        strerror(errno));
    exit(1);
  }
}
// Отображаем разделяемую память мьютекса
void* map = mmap(0, sizeof(pthread_mutex_t),
        PROT_READ | PROT_WRITE, MAP_SHARED, mutex_shm_fd, 0);
if (map == MAP_FAILED) {
  fprintf(stderr, "ERROR: Mapping failed: %s\n",
          strerror(errno));
  exit(1);
}
mutex = (pthread_mutex_t*)map;
if (mutex_owner) {
  int ret = -1;
  pthread_mutexattr_t attr;
  if ((ret = pthread_mutexattr_init(&attr))) {
    fprintf(stderr,
        "ERROR: Initializing mutex attributes failed: %s\n",
        strerror(ret));
```

```
      exit(1);
    }
    if ((ret = pthread_mutexattr_setpshared(&attr,
                    PTHREAD_PROCESS_SHARED))) {
      fprintf(stderr,
          "ERROR: Setting the mutex attribute failed: %s\n",
          strerror(ret));
      exit(1);
    }
    if ((ret = pthread_mutex_init(mutex, &attr))) {
      fprintf(stderr,
          "ERROR: Initializing the mutex failed: %s\n",
          strerror(ret));
      exit(1);
    }
    if ((ret = pthread_mutexattr_destroy(&attr))) {
      fprintf(stderr,
          "ERROR: Destruction of mutex attributes failed: %s\n",
          strerror(ret));
      exit(1);
    }
  }
}
```

Здесь мы создаем и инициализируем область разделяемой памяти, лежащей в основе разделяемого мьютекса. Это похоже на то, как мы пытались создать область разделяемой памяти для флага отмены. Стоит отметить: как и в примере 18.2, мьютекс имеет атрибут `PTHREAD_PROCESS_SHARED`, благодаря чему его могут использовать разные процессы.

Листинг 18.14 демонстрирует процедуру освобождения разделяемого мьютекса.

Листинг 18.14. Освобождение разделяемого мьютекса и связанной с ним области разделяемой памяти (ExtremeC_examples_chapter18_3.c)

```
void shutdown_control_mechanism() {
  sleep(1);
  if (mutex_owner) {
    int ret = -1;
    if ((ret = pthread_mutex_destroy(mutex))) {
      fprintf(stderr,
          "WARN: Destruction of the mutex failed: %s\n",
          strerror(ret));
    }
  }
  if (munmap(mutex, sizeof(pthread_mutex_t)) < 0) {
    fprintf(stderr, "ERROR: Unmapping the mutex failed: %s\n",
        strerror(errno));
    exit(1);
  }
```

```
  if (close(mutex_shm_fd) < 0) {
    fprintf(stderr, "ERROR: Closing the mutex failed: %s\n",
        strerror(errno));
    exit(1);
  }
  if (mutex_owner) {
    if (shm_unlink(MUTEX_SHM_NAME) < 0) {
      fprintf(stderr, "ERROR: Unlinking the mutex failed: %s\n",
          strerror(errno));
      exit(1);
    }
  }
}
```

И снова отмечаем, что разделяемую память мьютекса может освободить только процесс, который ею владеет.

Этап 4: установка флага отмены

В листинге 18.15 показаны функции, которые позволяют процессу считывать и устанавливать флаг отмены.

Листинг 18.15. Синхронизированные функции, которые считывают и устанавливают флаг отмены, защищенный разделяемым мьютексом (ExtremeC_examples_chapter18_3.c)

```
bool_t is_canceled() {
  pthread_mutex_lock(mutex); // Нужно проверить возвращаемое значение
  bool_t temp = *cancel_flag;
  pthread_mutex_unlock(mutex); // Нужно проверить возвращаемое значение
  return temp;
}

void cancel() {
  pthread_mutex_lock(mutex); // Нужно проверить возвращаемое значение
  *cancel_flag = TRUE;
  pthread_mutex_unlock(mutex); // Нужно проверить возвращаемое значение
}
```

Эти две функции обеспечивают синхронизированный доступ к разделяемому флагу отмены. Функция `is_canceled` используется для проверки значения флага, а функция `cancel` — для его установки. Как видите, обе они защищены одним и тем же разделяемым мьютексом.

Этап 5: главная функция

И, наконец, в листинге 18.16 представлены функция `main` и *обработчик сигналов*, о котором мы поговорим чуть ниже.

Листинг 18.16. Функция main и обработчик сигналов в примере 18.3 (ExtremeC_examples_chapter18_3.c)

```
void sigint_handler(int signo) {
  fprintf(stdout, "\nHandling INT signal: %d ...\n", signo);
  cancel();
}

int main(int argc, char** argv) {

  signal(SIGINT, sigint_handler);

  // Родительский процесс должен инициализировать разделяемый ресурс
  init_shared_resource();

  // Родительский процесс должен инициализировать механизм управления
  init_control_mechanism();

  while(!is_canceled()) {
    fprintf(stdout, "Working ...\n");
    sleep(1);
  }

  fprintf(stdout, "Cancel signal is received.\n");

  shutdown_shared_resource();
  shutdown_control_mechanism();

  return 0;
}
```

Логика внутри функции `main` выглядит вполне понятно. Она инициализирует разделяемые флаг и мьютекс, после чего входит в холостой цикл и ждет, пока флаг отмены будет равен `true`. В конце она освобождает все разделяемые ресурсы и завершает работу.

В этом коде есть кое-что новое: использование функции `signal`, которая назначает обработчик для определенного набора *сигналов*. Сигналы — один из механизмов, доступный во всех POSIX-совместимых операционных системах, с помощью которого процессы могут слать друг другу информацию в рамках одной ОС. *Терминал* — обычный процесс, с которым взаимодействуют пользователи и который позволяет отправлять сигналы другим процессам. Нажать Ctrl+C — удобный способ послать сигнал `SIGINT` активному процессу, запущенному в терминале.

Сигнал `SIGINT` представляет собой *сигнал прерывания*, который может получить процесс. В предыдущем листинге мы назначили функцию `sigint_handler` в качестве обработчика сигнала `SIGINT`. Иными словами, эта функция вызывается каждый раз, когда процесс получает этот сигнал. Если `SIGINT` не обработать, то будет выполнена процедура по умолчанию, которая завершит процесс, но ее можно переопределить с помощью обработчиков сигналов, как показано выше.

Существует множество способов, как отправить процессу сигнал `SIGINT`, но один из самых простых — нажать сочетание клавиш Ctrl+C на клавиатуре, в результате которого сигнал будет сразу же получен. Внутри обработчика сигнала мы присваиваем разделяемому флагу отмены значение `true`, после чего все процессы начинают выходить из своих холостых циклов.

В терминале 18.2 продемонстрированы компиляция и выполнение предыдущего листинга. Соберем наш код и запустим первый процесс.

Терминал 18.2. Компиляция примера 18.3 и запуск первого процесса

```
$ gcc ExtremeC_examples_chapter18_3.c -lpthread -lrt -o ex18_3.out
$ ./ex18_3.out
WARN: The shared memory object doesn't exist.
Creating a shared memory object ...
The shared memory object is created.
The memory region is truncated.
WARN: Mutex's shared memory object doesn't exist.
Creating the mutex's shared memory object ...
The mutex's shared memory object is created.
Working ...
Working ...
Working ...
```

Как видите, данный процесс запускается первым, поэтому становится владельцем мьютекса и флага отмены. В терминале 18.3 показан запуск второго процесса.

Терминал 18.3. Запуск второго процесса

```
$ ./ex18_3.out
The shared memory object is opened.
The mutex's shared memory object is opened.
Working ...
Working ...
Working ...
```

Второй процесс не является владельцем объектов разделяемой памяти, поэтому только открывает их. В следующем выводе демонстрируется нажатие Ctrl+C в первом процессе (терминал 18.4).

Терминал 18.4. Вывод первого процесса при нажатии Ctrl+C

```
...
Working ...
Working ...
^C
Handling INT signal: 2 ...
Cancel signal is received.
$
```

Первый процесс сообщает о том, что обрабатывает сигнал с номером 2 (это стандартный номер SIGINT). Он устанавливает флаг отмены и сразу же завершается. Вслед за этим прекращает работу и второй процесс. В терминале 18.5 показан вывод второго процесса.

Терминал 18.5. Вывод второго процесса в момент, когда устанавливается флаг отмены

```
...
Working ...
Working ...
Working ...
Cancel signal is received.
$
```

Тот же результат можно получить, если послать сигнал SIGINT второму процессу; оба процесса получат сигнал и завершат работу. Кроме того, вы можете создать несколько процессов, и все они синхронно завершатся, применяя одни и те же разделяемую память и мьютекс.

В следующем разделе мы рассмотрим использование условных переменных. Если поместить условную переменную в область разделяемой памяти, то к ней, как и к именованному мьютексу, можно будет обращаться из разных процессов, задействуя имя соответствующей области.

Именованные условные переменные

Ранее я уже объяснял: чтобы использовать условные переменные POSIX в многопроцессной системе, их, как и именованные POSIX-мьютексы, необходимо выделять в области разделяемой памяти. В примере 18.4 показано, как это делается; мы создадим ряд процессов, которые будут выполнять отсчет в определенном порядке. В главе 16 мы обсудили, что каждая условная переменная должна использоваться в связке с объектом мьютекса, защищающим ее. Поэтому в примере 18.4 будет три области разделяемой памяти: для разделяемого счетчика, для разделяемой *именованной условной переменной* и для разделяемого *именованного мьютекса*, который будет ее защищать.

Обратите внимание: вместо трех разных областей разделяемой памяти мы могли бы использовать одну общую. Для этого можно было бы определить структуру, которая заключает в себе все необходимые объекты. Но в данном примере мы не станем применять этот подход и определим для каждого объекта отдельную область разделяемой памяти.

Пример 18.4 состоит из ряда процессов, которые должны выполнять отсчет в порядке возрастания. Каждому процессу назначается номер, начиная с 1 и заканчивая

количеством процессов. Данный номер обозначает приоритет. Первоочередное право отсчета отдается процессу с наименьшим номером; только когда он закончит, следующий процесс сможет выполнить свой отсчет и завершить работу. Это значит, первым отсчет выполняет процесс с номером 1, даже если был создан последним.

Поскольку у нас будет три разные области разделяемой памяти, каждую из которых нужно отдельно инициализировать и освобождать, нам пришлось бы дублировать много кода, реши мы использовать тот же подход, что и в предыдущих примерах. Сделаем код более лаконичным и организованным, а также оформим повторяющиеся участки в виде функций с помощью объектно-ориентированного стиля. Для этого задействуем концепции и процедуры, описанные в главах 6, 7 и 8. Пример 18.4 будет написан в объектно-ориентированной манере с применением наследования, чтобы уменьшить объем дублируемого кода.

Мы определим общий родительский класс для всех областей разделяемой памяти, а также по одному дочернему классу для разделяемого счетчика, разделяемого именованного мьютекса и разделяемой именованной условной переменной. У каждого из этих классов будет своя пара заголовочных и исходных файлов, и все они будут использоваться в главной функции нашего примера.

В следующих разделах мы последовательно пройдемся по каждому из вышеупомянутых классов. Начнем с родительского класса: разделяемой памяти.

Этап 1: класс разделяемой памяти

В листинге 18.17 показаны определения класса разделяемой памяти.

Листинг 18.17. Публичный интерфейс класса разделяемой памяти (ExtremeC_examples_chapter18_4_shared_mem.h)

```
struct shared_mem_t;

typedef int32_t bool_t;


struct shared_mem_t* shared_mem_new();
void shared_mem_delete(struct shared_mem_t* obj);

void shared_mem_ctor(struct shared_mem_t* obj,
                     const char* name,
                     size_t size);
void shared_mem_dtor(struct shared_mem_t* obj);

char* shared_mem_getptr(struct shared_mem_t* obj);
bool_t shared_mem_isowner(struct shared_mem_t* obj);
void shared_mem_setowner(struct shared_mem_t* obj,
                         bool_t is_owner);
```

В этом листинге содержатся объявления (публичный API), необходимые для работы с объектом разделяемой памяти. Функции `shared_mem_getptr`, `shared_mem_isowner` и `shared_mem_setowner` представляют собой поведение данного класса.

Если вам незнаком этот синтаксис, то, пожалуйста, прочитайте главы 6, 7 и 8.

В листинге 18.18 показаны определения функций, объявленных в рамках публичного интерфейса класса в листинге 18.17.

Листинг 18.18. Определения всех функций, принадлежащих классу разделяемой памяти (ExtremeC_examples_chapter18_4_shared_mem.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/mman.h>

#define TRUE 1
#define FALSE 0

typedef int32_t bool_t;

bool_t owner_process_set = FALSE;
bool_t owner_process = FALSE;

typedef struct {
  char* name;
  int shm_fd;
  void* map_ptr;
  char* ptr;
  size_t size;
} shared_mem_t;

shared_mem_t* shared_mem_new() {
  return (shared_mem_t*)malloc(sizeof(shared_mem_t));
}

void shared_mem_delete(shared_mem_t* obj) {
  free(obj->name);
  free(obj);
}

void shared_mem_ctor(shared_mem_t* obj, const char* name,
        size_t size) {
  obj->size = size;
  obj->name = (char*)malloc(strlen(name) + 1);
  strcpy(obj->name, name);
```

```
  obj->shm_fd = shm_open(obj->name, O_RDWR, 0600);
  if (obj->shm_fd >= 0) {
    if (!owner_process_set) {
      owner_process = FALSE;
      owner_process_set = TRUE;
    }
    printf("The shared memory %s is opened.\n", obj->name);
  } else if (errno == ENOENT) {
    printf("WARN: The shared memory %s does not exist.\n",
           obj->name);
    obj->shm_fd = shm_open(obj->name,
           O_CREAT | O_RDWR, 0600);
    if (obj->shm_fd >= 0) {
      if (!owner_process_set) {
        owner_process = TRUE;
        owner_process_set = TRUE;
      }
      printf("The shared memory %s is created and opened.\n",
             obj->name);
      if (ftruncate(obj->shm_fd, obj->size) < 0) {
        fprintf(stderr, "ERROR(%s): Truncation failed: %s\n",
            obj->name, strerror(errno));
        exit(1);
      }
    } else {
      fprintf(stderr,
          "ERROR(%s): Failed to create shared memory: %s\n",
          obj->name, strerror(errno));
      exit(1);
    }
  } else {
      fprintf(stderr,
          "ERROR(%s): Failed to create shared memory: %s\n",
          obj->name, strerror(errno));
    exit(1);
  }
  obj->map_ptr = mmap(0, obj->size, PROT_READ | PROT_WRITE,
      MAP_SHARED, obj->shm_fd, 0);
  if (obj->map_ptr == MAP_FAILED) {
    fprintf(stderr, "ERROR(%s): Mapping failed: %s\n",
        name, strerror(errno));
    exit(1);
  }
  obj->ptr = (char*)obj->map_ptr;
}

void shared_mem_dtor(shared_mem_t* obj) {
  if (munmap(obj->map_ptr, obj->size) < 0) {
    fprintf(stderr, "ERROR(%s): Unmapping failed: %s\n",
        obj->name, strerror(errno));
    exit(1);
```

```
  }
  printf("The shared memory %s is unmapped.\n", obj->name);
  if (close(obj->shm_fd) < 0) {
    fprintf(stderr,
        "ERROR(%s): Closing the shared memory fd failed: %s\n",
        obj->name, strerror(errno));
    exit(1);
  }
  printf("The shared memory %s is closed.\n", obj->name);
  if (owner_process) {
    if (shm_unlink(obj->name) < 0) {
      fprintf(stderr,
          "ERROR(%s): Unlinking the shared memory failed: %s\n",
          obj->name, strerror(errno));
      exit(1);
    }
    printf("The shared memory %s is deleted.\n", obj->name);
  }
}

char* shared_mem_getptr(shared_mem_t* obj) {
  return obj->ptr;
}

bool_t shared_mem_isowner(shared_mem_t* obj) {
  return owner_process;
}

void shared_mem_setowner(shared_mem_t* obj, bool_t is_owner) {
    owner_process = is_owner;
}
```

Как видите, мы просто скопировали код, написанный для работы с разделяемой памятью в предыдущих примерах. Структура `shared_mem_t` инкапсулирует все, что нам нужно для обращения к объекту разделяемой памяти POSIX. Обратите внимание на глобальную булеву переменную `process_owner`. Она определяет, является ли текущий процесс владельцем всех областей разделяемой памяти. Она устанавливается только однажды.

Этап 2: класс разделяемого 32-битного целочисленного счетчика

В листинге 18.19 содержится определение класса разделяемого счетчика, который представляет собой 32-битное целое число. Этот класс наследуется от класса разделяемой памяти. Как вы могли заметить, для реализации наследования мы используем только второй подход, описанный в главе 8.

Листинг 18.19. Публичный интерфейс класса разделяемого счетчика (ExtremeC_examples_chapter18_4_shared_int32.h)

```
struct shared_int32_t;

struct shared_int32_t* shared_int32_new();
void shared_int32_delete(struct shared_int32_t* obj);

void shared_int32_ctor(struct shared_int32_t* obj,
                       const char* name);
void shared_int32_dtor(struct shared_int32_t* obj);

void shared_int32_setvalue(struct shared_int32_t* obj,
                           int32_t value);
void shared_int32_setvalue_ifowner(struct shared_int32_t* obj,
                                   int32_t value);
int32_t shared_int32_getvalue(struct shared_int32_t* obj);
```

В листинге 18.20 представлены реализации функций, объявленных выше.

Листинг 18.20. Определения всех функций в классе разделяемого счетчика (ExtremeC_examples_chapter18_4_shared_int32.c)

```
#include "ExtremeC_examples_chapter18_4_shared_mem.h"

typedef struct {
  struct shared_mem_t* shm;
  int32_t* ptr;
} shared_int32_t;

shared_int32_t* shared_int32_new(const char* name) {
  shared_int32_t* obj =
      (shared_int32_t*)malloc(sizeof(shared_int32_t));
  obj->shm = shared_mem_new();
  return obj;
}

void shared_int32_delete(shared_int32_t* obj) {
  shared_mem_delete(obj->shm);
  free(obj);
}

void shared_int32_ctor(shared_int32_t* obj, const char* name) {
  shared_mem_ctor(obj->shm, name, sizeof(int32_t));
  obj->ptr = (int32_t*)shared_mem_getptr(obj->shm);
}

void shared_int32_dtor(shared_int32_t* obj) {
  shared_mem_dtor(obj->shm);
}
```

```
void shared_int32_setvalue(shared_int32_t* obj, int32_t value) {
  *(obj->ptr) = value;
}

void shared_int32_setvalue_ifowner(shared_int32_t* obj,
                                   int32_t value) {
  if (shared_mem_isowner(obj->shm)) {
    *(obj->ptr) = value;
  }
}

int32_t shared_int32_getvalue(shared_int32_t* obj) {
  return *(obj->ptr);
}
```

Как видите, благодаря наследованию наш код получился намного более компактным. Все операции по управлению соответствующим объектом разделяемой памяти доступны в поле `shm` структуры `shared_int32_t`.

Этап 3: класс разделяемого мьютекса

Листинг 18.21 содержит объявления класса разделяемого мьютекса.

Листинг 18.21. Публичный интерфейс класса разделяемого мьютекса (ExtremeC_examples_chapter18_4_shared_mutex.h)

```
#include <pthread.h>

// Прямое объявление
struct shared_mutex_t;

struct shared_mutex_t* shared_mutex_new();
void shared_mutex_delete(struct shared_mutex_t* obj);

void shared_mutex_ctor(struct shared_mutex_t* obj,
                       const char* name);
void shared_mutex_dtor(struct shared_mutex_t* obj);

pthread_mutex_t* shared_mutex_getptr(struct shared_mutex_t* obj);

void shared_mutex_lock(struct shared_mutex_t* obj);
void shared_mutex_unlock(struct shared_mutex_t* obj);

#if !defined(__APPLE__)
void shared_mutex_make_consistent(struct shared_mutex_t* obj);
#endif
```

Представленный выше класс, как и ожидалось, содержит три операции: `shared_mutex_lock`, `shared_mutex_unlock` и `shared_mutex_make_consistent`. Но есть один

нюанс: операция `shared_mutex_make_consistent` доступна только в системах POSIX, которые не принадлежат к семейству macOS (Apple). Дело в том, что системы Apple не поддерживают *устойчивые мьютексы*. Мы еще вернемся к этому в следующих абзацах. Обратите внимание: здесь используется макрос `__APPLE__`, позволяющий определить, компилируется ли код в системе Apple.

Реализация этого класса представлена в листинге 18.22.

Листинг 18.22. Определения всех функций, которые находятся в классе разделяемого именованного мьютекса (ExtremeC_examples_chapter18_4_shared_mutex.c)

```
#include "ExtremeC_examples_chapter18_4_shared_mem.h"

typedef struct {
  struct shared_mem_t* shm;
  pthread_mutex_t* ptr;
} shared_mutex_t;

shared_mutex_t* shared_mutex_new() {
  shared_mutex_t* obj =
      (shared_mutex_t*)malloc(sizeof(shared_mutex_t));
  obj->shm = shared_mem_new();
  return obj;
}

void shared_mutex_delete(shared_mutex_t* obj) {
  shared_mem_delete(obj->shm);
  free(obj);
}

void shared_mutex_ctor(shared_mutex_t* obj, const char* name) {
  shared_mem_ctor(obj->shm, name, sizeof(pthread_mutex_t));
  obj->ptr = (pthread_mutex_t*)shared_mem_getptr(obj->shm);
  if (shared_mem_isowner(obj->shm)) {
    pthread_mutexattr_t mutex_attr;
    int ret = -1;
    if ((ret = pthread_mutexattr_init(&mutex_attr))) {
      fprintf(stderr,
          "ERROR(%s): Initializing mutex attrs failed: %s\n",
          name, strerror(ret));
      exit(1);
    }
#if !defined(__APPLE__)
    if ((ret = pthread_mutexattr_setrobust(&mutex_attr,
                    PTHREAD_MUTEX_ROBUST))) {
      fprintf(stderr,
          "ERROR(%s): Setting the mutex as robust failed: %s\n",
          name, strerror(ret));
      exit(1);
    }
#endif
```

```
    if ((ret = pthread_mutexattr_setpshared(&mutex_attr,
                    PTHREAD_PROCESS_SHARED))) {
      fprintf(stderr,
          "ERROR(%s): Failed to set ass process-shared: %s\n",
          name, strerror(ret));
      exit(1);
    }
    if ((ret = pthread_mutex_init(obj->ptr, &mutex_attr))) {
      fprintf(stderr,
          "ERROR(%s): Initializing the mutex failed: %s\n",
          name, strerror(ret));
      exit(1);
    }
    if ((ret = pthread_mutexattr_destroy(&mutex_attr))) {
      fprintf(stderr,
          "ERROR(%s): Destruction of mutex attrs failed: %s\n",
          name, strerror(ret));
      exit(1);
    }
  }
}

void shared_mutex_dtor(shared_mutex_t* obj) {
  if (shared_mem_isowner(obj->shm)) {
    int ret = -1;
    if ((ret = pthread_mutex_destroy(obj->ptr))) {
      fprintf(stderr,
          "WARN: Destruction of the mutex failed: %s\n",
          strerror(ret));
    }
  }
  shared_mem_dtor(obj->shm);
}

pthread_mutex_t* shared_mutex_getptr(shared_mutex_t* obj) {
  return obj->ptr;
}

#if !defined(__APPLE__)
void shared_mutex_make_consistent(shared_mutex_t* obj) {
  int ret = -1;
  if ((ret = pthread_mutex_consistent(obj->ptr))) {
    fprintf(stderr,
        "ERROR: Making the mutex consistent failed: %s\n",
        strerror(ret));
    exit(1);
  }
}
#endif
```

```
void shared_mutex_lock(shared_mutex_t* obj) {
  int ret = -1;
  if ((ret = pthread_mutex_lock(obj->ptr))) {
#if !defined(__APPLE__)
    if (ret == EOWNERDEAD) {
        fprintf(stderr,
                "WARN: The owner of the mutex is dead ...\n");
        shared_mutex_make_consistent(obj);
        fprintf(stdout, "INFO: I'm the new owner!\n");
        shared_mem_setowner(obj->shm, TRUE);
        return;
    }
#endif
    fprintf(stderr, "ERROR: Locking the mutex failed: %s\n",
        strerror(ret));
    exit(1);
  }
}

void shared_mutex_unlock(shared_mutex_t* obj) {
  int ret = -1;
  if ((ret = pthread_mutex_unlock(obj->ptr))) {
    fprintf(stderr, "ERROR: Unlocking the mutex failed: %s\n",
        strerror(ret));
    exit(1);
  }
}
```

В данном коде выполняются инициализация и освобождение POSIX-мьютекса, а также предоставляется доступ к таким тривиальным операциям, как блокировка и разблокировка. Все остальное, касающееся объекта разделяемой памяти, находится в родительском классе. Это одно из преимуществ использования наследования.

Обратите внимание: в функции-конструкторе `shared_mutex_ctor` мы делаем мьютекс разделяемым (`PTHREAD_PROCESS_SHARED`), чтобы он был доступен для всех процессов. В многопроцессном ПО это совершенно необходимо. Кроме того, в системах, не принадлежащих к семейству Apple, мы идем еще дальше и помечаем мьютекс как *устойчивый* (`PTHREAD_MUTEX_ROBUST`).

Когда процесс завершается неожиданно, обычный мьютекс, заблокированный им, переходит в несогласованное состояние. Однако устойчивый мьютекс в этой ситуации можно опять сделать согласованным. Следующий процесс, который обычно ждет разблокировка мьютекса, может заблокировать его только после его согласования. В функции `shared_mutex_lock` показано, как это делается. Отмечу, что в системах Apple данной возможности нет.

Этап 4: класс разделяемой условной переменной

В листинге 18.23 показано объявление класса разделяемой условной переменной.

Листинг 18.23. Публичный интерфейс класса разделяемой условной переменной (ExtremeC_examples_chapter18_4_shared_cond.h)

```
struct shared_cond_t;
struct shared_mutex_t;

struct shared_cond_t* shared_cond_new();
void shared_cond_delete(struct shared_cond_t* obj);

void shared_cond_ctor(struct shared_cond_t* obj,
                      const char* name);
void shared_cond_dtor(struct shared_cond_t* obj);

void shared_cond_wait(struct shared_cond_t* obj,
                      struct shared_mutex_t* mutex);
void shared_cond_timedwait(struct shared_cond_t* obj,
                           struct shared_mutex_t* mutex,
                           long int time_nanosec);
void shared_cond_broadcast(struct shared_cond_t* obj);
```

Здесь предоставляется доступ к трем операциям: `shared_cond_wait`, `shared_cond_timedwait` и `shared_cond_broadcast`. Как вы помните из главы 16, операция `shared_cond_wait` ждет сигнала, относящегося к условной переменной.

Выше мы добавили новую версию операции ожидания, `shared_cond_timedwait`. Она ждет получения сигнала на протяжении заданного времени, и если по его прошествии сигнал так и не получен, то завершается. А вот `shared_cond_wait` заканчивает работу только при получении сигнала. В примере 18.4 будет использоваться версия с временем ожидания. Стоит отметить, что обе операции принимают указатель на сопутствующий разделяемый мьютекс, как это происходило в многопоточных средах.

Листинг 18.24 содержит реализацию класса разделяемой условной переменной.

Листинг 18.24. Определения всех функций в классе разделяемой условной переменной (ExtremeC_examples_chapter18_4_shared_cond.c)

```
#include "ExtremeC_examples_chapter18_4_shared_mem.h"
#include "ExtremeC_examples_chapter18_4_shared_mutex.h"

typedef struct {
  struct shared_mem_t* shm;
  pthread_cond_t* ptr;
} shared_cond_t;
```

```
shared_cond_t* shared_cond_new() {
  shared_cond_t* obj =
      (shared_cond_t*)malloc(sizeof(shared_cond_t));
  obj->shm = shared_mem_new();
  return obj;
}

void shared_cond_delete(shared_cond_t* obj) {
  shared_mem_delete(obj->shm);
  free(obj);
}

void shared_cond_ctor(shared_cond_t* obj, const char* name) {
  shared_mem_ctor(obj->shm, name, sizeof(pthread_cond_t));
  obj->ptr = (pthread_cond_t*)shared_mem_getptr(obj->shm);
  if (shared_mem_isowner(obj->shm)) {
    pthread_condattr_t cond_attr;
    int ret = -1;
    if ((ret = pthread_condattr_init(&cond_attr))) {
      fprintf(stderr,
          "ERROR(%s): Initializing cv attrs failed: %s\n",
          name, strerror(ret));
      exit(1);
    }
    if ((ret = pthread_condattr_setpshared(&cond_attr,
                    PTHREAD_PROCESS_SHARED))) {
      fprintf(stderr,
          "ERROR(%s): Setting as process shared failed: %s\n",
          name, strerror(ret));
      exit(1);
    }
    if ((ret = pthread_cond_init(obj->ptr, &cond_attr))) {
      fprintf(stderr,
          "ERROR(%s): Initializing the cv failed: %s\n",
          name, strerror(ret));
      exit(1);
    }
    if ((ret = pthread_condattr_destroy(&cond_attr))) {
      fprintf(stderr,
          "ERROR(%s): Destruction of cond attrs failed: %s\n",
          name, strerror(ret));
      exit(1);
    }
  }
}

void shared_cond_dtor(shared_cond_t* obj) {
  if (shared_mem_isowner(obj->shm)) {
    int ret = -1;
    if ((ret = pthread_cond_destroy(obj->ptr))) {
```

```
      fprintf(stderr, "WARN: Destruction of the cv failed: %s\n",
          strerror(ret));
    }
  }
  shared_mem_dtor(obj->shm);
}

void shared_cond_wait(shared_cond_t* obj,
                      struct shared_mutex_t* mutex) {
  int ret = -1;
  if ((ret = pthread_cond_wait(obj->ptr,
                shared_mutex_getptr(mutex)))) {
    fprintf(stderr, "ERROR: Waiting on the cv failed: %s\n",
            strerror(ret));
    exit(1);
  }
}

void shared_cond_timedwait(shared_cond_t* obj,
                           struct shared_mutex_t* mutex,
                           long int time_nanosec) {
  int ret = -1;

  struct timespec ts;
  ts.tv_sec = ts.tv_nsec = 0;
  if ((ret = clock_gettime(CLOCK_REALTIME, &ts))) {
    fprintf(stderr,
            "ERROR: Failed at reading current time: %s\n",
            strerror(errno));
    exit(1);
  }
  ts.tv_sec += (int)(time_nanosec / (1000L * 1000 * 1000));
  ts.tv_nsec += time_nanosec % (1000L * 1000 * 1000);

  if ((ret = pthread_cond_timedwait(obj->ptr,
                shared_mutex_getptr(mutex), &ts))) {
#if !defined(__APPLE__)
    if (ret == EOWNERDEAD) {
      fprintf(stderr,
              "WARN: The owner of the cv's mutex is dead ...\n");
      shared_mutex_make_consistent(mutex);
      fprintf(stdout, "INFO: I'm the new owner!\n");
      shared_mem_setowner(obj->shm, TRUE);
      return;
    } else if (ret == ETIMEDOUT) {
#else
    if (ret == ETIMEDOUT) {
#endif
      return;
    }
    fprintf(stderr, "ERROR: Waiting on the cv failed: %s\n",
```

```
            strerror(ret));
    exit(1);
  }
}

void shared_cond_broadcast(shared_cond_t* obj) {
  int ret = -1;
  if ((ret = pthread_cond_broadcast(obj->ptr))) {
    fprintf(stderr, "ERROR: Broadcasting on the cv failed: %s\n",
        strerror(ret));
    exit(1);
  }
}
```

В нашем классе разделяемой условной переменной доступна только операция `shared_cond_broadcast`. Мы бы также могли предоставить доступ к операции *отправки сигнала*. Как вы, наверное, помните из главы 16, отправка сигнала, касающегося условной переменной, пробуждает только один из ожидающих процессов; притом мы не можем указать или предсказать, какой именно. Для сравнения, операция `shared_cond_broadcast` пробуждает все ожидающие процессы. В примере 18.4 мы будем использовать лишь ее, поэтому предоставляем доступ исключительно к данной функции.

Поскольку у каждой условной переменной есть сопутствующий мьютекс, ее класс должен уметь использовать экземпляр класса разделяемого мьютекса. Вот почему мы выполнили предварительное объявление `shared_mutex_t`.

Этап 5: основная логика

Листинг 18.25 содержит основную логику, реализованную в нашем примере.

Листинг 18.25. Главная функция в примере 18.4 (ExtremeC_examples_chapter18_4_main.c)

```
#include "ExtremeC_examples_chapter18_4_shared_int32.h"
#include "ExtremeC_examples_chapter18_4_shared_mutex.h"
#include "ExtremeC_examples_chapter18_4_shared_cond.h"

int int_received = 0;
struct shared_cond_t* cond = NULL;
struct shared_mutex_t* mutex = NULL;

void sigint_handler(int signo) {
  fprintf(stdout, "\nHandling INT signal: %d ...\n", signo);
  int_received = 1;
}

int main(int argc, char** argv) {
```

```
  signal(SIGINT, sigint_handler);

  if (argc < 2) {
    fprintf(stderr,
            "ERROR: You have to provide the process number.\n");
    exit(1);
  }

  int my_number = atol(argv[1]);
  printf("My number is %d!\n", my_number);

  struct shared_int32_t* counter = shared_int32_new();
  shared_int32_ctor(counter, "/counter0");
  shared_int32_setvalue_ifowner(counter, 1);

  mutex = shared_mutex_new();
  shared_mutex_ctor(mutex, "/mutex0");

  cond = shared_cond_new();
  shared_cond_ctor(cond, "/cond0");

  shared_mutex_lock(mutex);
  while (shared_int32_getvalue(counter) < my_number) {
    if (int_received) {
        break;
    }
    printf("Waiting for the signal, just for 5 seconds ...\n");
    shared_cond_timedwait(cond, mutex, 5L * 1000 * 1000 * 1000);
    if (int_received) {
        break;
    }
    printf("Checking condition ...\n");
  }
  if (int_received) {
    printf("Exiting ...\n");
    shared_mutex_unlock(mutex);
    goto destroy;
  }
  shared_int32_setvalue(counter, my_number + 1);
  printf("My turn! %d ...\n", my_number);
  shared_mutex_unlock(mutex);
  sleep(1);
  // ПРИМЕЧАНИЕ: вещание может начаться после открытия мьютекса
  shared_cond_broadcast(cond);

destroy:
  shared_cond_dtor(cond);
  shared_cond_delete(cond);

  shared_mutex_dtor(mutex);
  shared_mutex_delete(mutex);
```

```
  shared_int32_dtor(counter);
  shared_int32_delete(counter);

  return 0;
}
```

Как видите, программа принимает аргумент, обозначающий ее номер. Узнав его, процесс начинает инициализацию разделяемого счетчика, разделяемого мьютекса и разделяемой условной переменной. Затем заходит на критический участок, защищенный разделяемым мьютексом.

Внутри цикла процесс ждет, когда счетчик станет равным его номеру. Поскольку ожидание длится 5 секунд, время теоретически может истечь, в результате чего функция `shared_cond_timedwait` завершится. Это фактически означает, что условная переменная не была уведомлена на протяжении этих 5 секунд. Затем процесс проверяет условие и использует еще один пятисекундный период ожидания. Так продолжается до тех пор, пока не придет его очередь.

Когда это случится, процесс выведет свой номер, инкрементирует разделяемый счетчик и уведомит о данном изменении остальные ожидающие процессы, разослав им сигналы об объекте условной переменной. И только после этого он приступит к завершению работы.

Тем временем, если пользователь нажмет Ctrl+C, обработчик сигнала, определенный в рамках основной логики, установит локальный флаг `int_received` и, как только процесс, находящийся внутри главного цикла, покинет функцию `shared_mutex_timedwait`, заметит сигнал прерывания, и цикл будет завершен.

В терминале 18.6 показано, как скомпилировать пример 18.4. Мы сделаем это в Linux.

Терминал 18.6. Компиляция исходников примера 18.4 и создание итогового исполняемого файла

```
$ gcc -c ExtremeC_examples_chapter18_4_shared_mem.c -o shared_mem.o
$ gcc -c ExtremeC_examples_chapter18_4_shared_int32.c -o shared_int32.o
$ gcc -c ExtremeC_examples_chapter18_4_shared_mutex.c -o shared_mutex.o
$ gcc -c ExtremeC_examples_chapter18_4_shared_cond.c -o shared_cond.o
$ gcc -c ExtremeC_examples_chapter18_4_main.c -o main.o
$ gcc shared_mem.o shared_int32.o shared_mutex.o shared_cond.o \
  main.o -lpthread -lrt -o ex18_4.out
$
```

Итак, получив итоговый исполняемый файл, `ex18_4.out`, мы можем запустить три процесса и посмотреть, как один за другим они выполняют отсчет, независимо от того, каким образом им были назначены номера, и от порядка их запуска. Запустим первый процесс и назначим ему номер 3, передав соответствующий аргумент исполняемому файлу (терминал 18.7).

Терминал 18.7. Запуск первого процесса, который принимает число 3

```
$ ./ex18_4.out 3
My number is 3!
WARN: The shared memory /counter0 does not exist.
The shared memory /counter0 is created and opened.
WARN: The shared memory /mutex0 does not exist.
The shared memory /mutex0 is created and opened.
WARN: The shared memory /cond0 does not exist.
The shared memory /cond0 is created and opened.
Waiting for the signal, just for 5 seconds ...
Checking condition ...
Waiting for the signal, just for 5 seconds ...
Checking condition ...
Waiting for the signal, just for 5 seconds ...
```

Как видите, первый процесс создает все необходимые разделяемые объекты и становится их владельцем. Теперь запустим в отдельном терминале второй процесс. Он принимает число 2 (терминал 18.8).

Терминал 18.8. Запуск второго процесса, который принимает число 2

```
$ ./ex18_4.out 2
My number is 2!
The shared memory /counter0 is opened.
The shared memory /mutex0 is opened.
The shared memory /cond0 is opened.
Waiting for the signal, just for 5 seconds ...
Checking condition ...
Waiting for the signal, just for 5 seconds ...
```

Наконец, последний процесс принимает число 1. Поскольку ему был назначен наименьший номер, он немедленно его выводит, инкрементирует разделяемый счетчик и уведомляет об этом остальные процессы (листинг 18.9).

Терминал 18.9. Запуск третьего процесса, который принимает 1. Этот процесс немедленно завершается, поскольку ему назначен наименьший номер

```
$ ./ex18_4.out 1
My number is 1!
The shared memory /counter0 is opened.
The shared memory /mutex0 is opened.
The shared memory /cond0 is opened.
My turn! 1 ...
The shared memory /cond0 is unmapped.
The shared memory /cond0 is closed.
The shared memory /mutex0 is unmapped.
The shared memory /mutex0 is closed.
The shared memory /counter0 is unmapped.
The shared memory /counter0 is closed.
$
```

Теперь, вернувшись ко второму процессу, можно заметить, что он тоже вывел свой номер, инкрементировал разделяемый счетчик и уведомил об этом третий процесс (терминал 18.10).

Терминал 18.10. Второй процесс выводит свой номер и завершается

```
...
Waiting for the signal, just for 5 seconds ...
Checking condition ...
My turn! 2 ...
The shared memory /cond0 is unmapped.
The shared memory /cond0 is closed.
The shared memory /mutex0 is unmapped.
The shared memory /mutex0 is closed.
The shared memory /counter0 is unmapped.
The shared memory /counter0 is closed.
$
```

Вернемся к первому процессу. Получив уведомление от второго процесса, он выводит свой номер и завершается.

Терминал 18.11. Первый процесс выводит свой номер и завершается. Он также удаляет все объекты разделяемой памяти

```
...
Waiting for the signal, just for 5 seconds ...
Checking condition ...
My turn! 3 ...
The shared memory /cond0 is unmapped.
The shared memory /cond0 is closed.
The shared memory /cond0 is deleted.
The shared memory /mutex0 is unmapped.
The shared memory /mutex0 is closed.
The shared memory /mutex0 is deleted.
The shared memory /counter0 is unmapped.
The shared memory /counter0 is closed.
The shared memory /counter0 is deleted.
$
```

Поскольку первый процесс владеет всеми объектами разделяемой памяти, при выходе он должен их удалить. Освобождение выделенных ресурсов в многопроцессной среде может вызвать определенные сложности, так как всего одной простой ошибки достаточно для того, чтобы вывести из строя все процессы. Когда разделяемый ресурс должен быть удален из системы, требуется дополнительная синхронизация.

Представьте, что в этом примере сначала запустился процесс под номером 2, а за ним процесс под номером 3. Следовательно, первый процесс должен вывести свой номер перед вторым. Но перед выходом первый процесс удаляет все разделяемые

объекты, поскольку является их владельцем; в результате, когда второй процесс пытается к ним обратиться, происходит сбой.

Это лишь простая иллюстрация того, какие проблемы могут возникать при освобождении памяти в многопроцессных системах. Чтобы уменьшить вероятность подобных сбоев, нужно обеспечить дополнительную синхронизацию процессов.

В предыдущих разделах мы обсудили механизмы, с помощью которых можно синхронизировать разные процессы, выполняемые на одном компьютере. Далее проведем краткий обзор распределенных механизмов управления конкурентностью и их характеристик.

Распределенное управление конкурентностью

До сих пор в данной главе предполагалось, что все процессы существуют в рамках одной операционной системы и, следовательно, одного компьютера. Иными словами, речь всегда шла о локальном ПО.

Однако настоящие программные продукты обычно этим не ограничиваются. Помимо локальных систем существуют еще и распределенные. В них процессы разбросаны по разным устройствам и взаимодействуют по сети.

Некоторые аспекты систем с распределенными процессами создают трудности, отсутствующие (или не проявляющиеся в такой степени) в централизованных или локальных системах. Кратко их обсудим.

- *В распределенной программной системе вы, скорее всего, имеете дело с параллелизмом, а не с конкурентностью.* Поскольку каждый процесс выполняется на отдельном компьютере со своим собственным процессором, мы будем наблюдать не конкурентность, а параллелизм. Конкурентность обычно ограничена рамками одного компьютера. Обратите внимание: чередования по-прежнему существуют, равно как и недетерминированность, которую мы встречали в конкурентных системах.
- *Не все процессы внутри распределенной программной системы написаны на одном и том же языке.* В распределенных системах довольно часто встречаются разные языки программирования. Такое же разнообразие нередко присутствует и в локальных системах. Здесь мы исходим из того, что все наши процессы написаны на C, но в действительности они могут быть разработаны на любом другом языке. Разные языки предоставляют различные способы организации конкурентности и управляющие механизмы. Поэтому в некоторых языках, к примеру, вам будет непросто задействовать именованные мьютексы. Использование разнообразных технологий и языков в программных системах, будь они локальными или распределенными, вынуждает нас применять достаточно

абстрактные механизмы управления конкурентностью, доступные во всех средах. Это может ограничить выбор методов синхронизации теми, которые совместимы с определенной технологией или языком программирования.

- *В распределенной системе в качестве канала между процессами, находящимися на разных компьютерах, всегда используется сеть.* Это совершенно противоречит нашему предположению о локальной системе, где все процессы работают в рамках одного компьютера и общаются между собой с помощью имеющейся инфраструктуры для обмена сообщениями.
- *Наличие сети между процессами вызывает латентность.* В локальных системах тоже есть небольшая латентность, однако она предсказуема и с ней можно совладать. Она намного меньше той латентности, которую вы можете встретить в сети. Латентность попросту означает, что по многим причинам, связанным с сетевой инфраструктурой, процесс может не получить сообщение сразу. В таких системах нужно быть готовыми к тому, что ничего не происходит мгновенно.
- *Наличие сети между процессами также приводит к проблемам с безопасностью.* Когда все процессы находятся в одной системе, они взаимодействуют в общих границах, используя механизмы с чрезвычайно низкой латентностью, поэтому проблемы с безопасностью у них совершенно другие. Для взлома процессов необходимо сперва получить доступ к самой системе. В распределенной же среде весь обмен сообщениями происходит по сети, в которую может кто-то *вклиниться* и прочитать или, что еще хуже, изменить передаваемые данные. То же самое касается сообщений, предназначенных для синхронизации процессов внутри распределенной системы.
- Помимо проблем с латентностью и безопасностью, у вас могут возникнуть трудности с доставкой, которые в локальных системах случаются намного реже. Чтобы обработать сообщения, их нужно сначала доставить. Когда один процесс передает сообщение другому в рамках одной системы, отправитель должен как-то убедиться в том, что оно было получено. Для этого можно реализовать механизмы гарантии доставки, но они приводят к накладным расходам, а иногда их и вовсе нельзя использовать. В таких случаях при обмене сообщениями можно наблюдать особого рода проблемы, которые обычно моделируются с помощью задачи двух генералов.

Перечисленные выше отличия и потенциальные проблемы оказались достаточным поводом для изобретения новых способов синхронизации процессов и различных компонентов в огромных распределенных системах. В целом распределенную систему можно сделать транзакционной и синхронизированной двумя путями.

- *Через централизованную синхронизацию процессов.* Этот подход подразумевает наличие центрального процесса (или узла), который управляет остальными компонентами. Все другие процессы в системе должны постоянно с ним взаимодействовать и запрашивать его разрешения для захода на свои критические участки.

- *Через распределенную (пиринговую) синхронизацию процессов.* Построение инфраструктуры синхронизации процессов, не имеющей центрального узла, — непростая задача. На самом деле в этой области ведутся активные исследования. Кроме того, существуют некоторые узкоспециализированные алгоритмы.

В данном разделе я пытался пролить свет на сложности, присущие управлению конкурентностью в распределенных многопроцессных системах. Дальнейшее обсуждение этой темы выходит за рамки данной книги.

Резюме

В этой главе мы завершили наше обсуждение многопроцессных сред. На ее страницах мы:

- рассмотрели, что такое именованный семафор, как его можно создать и использовать в разных процессах;
- познакомились с именованным мьютексом и тем, как его применять с помощью области разделяемой памяти;
- рассмотрели пример организованного завершения работы, в котором несколько процессов ждали сигнала о завершении; этот сигнал принимался и обрабатывался одним процессом и затем распространялся между остальными. Мы реализовали этот пример, задействовав разделяемые мьютексы;
- изучили, что такое условная переменная, как ее можно сделать разделяемой и именованной, используя область разделяемой памяти;
- рассмотрели еще один пример с процессами, которые выполняют отсчет. При его реализации применялось наследование, чтобы уменьшить объем повторяющегося кода для работы с объектами мьютекса и условной переменной, привязанными к области разделяемой памяти;
- привели краткий обзор особенностей и сложностей, присущих распределенным системам;
- вдобавок кратко обсудили методы, с помощью которых в распределенном программном обеспечении можно организовать *управление* конкурентностью.

В следующей главе мы приступим к обсуждению методов *межпроцессного взаимодействия* (inter-process communication, IPC). Дискуссия займет две главы и будет включать множество тем, таких как компьютерные сети, транспортные протоколы, программирование сокетов и др.

19 Локальные сокеты и IPC

В предыдущей главе мы обсудили методики, с помощью которых два процесса могут конкурентно и синхронно работать с одним и тем же разделяемым ресурсом. Здесь я представлю новую категорию методов, позволяющих двум процессам передавать данные. Новые методы и уже знакомые нам по предыдущей главе относятся к *межпроцессному взаимодействию* (Inter-Process Communication, IPC).

В данной и следующей главах мы поговорим о методах IPC, которые предусматривают некий *обмен сообщениями* или *сигналами* между разными процессами. Передающиеся сообщения не хранятся ни в каком общем месте наподобие файла или разделяемой памяти; вместо этого процессы их генерируют и принимают.

В этой главе мы рассмотрим две основные темы. Во-первых, обсудим методы IPC, локальное межпроцессное взаимодействие и API POSIX. Во-вторых, начнем знакомство с программированием сокетов и сопутствующими темами. В ходе нашего разговора мы затронем компьютерные сети, модель «слушатель — соединитель» и способы установления соединения между двумя процессами.

В рамках данной главы будут рассмотрены:

- различные методы IPC. Мы познакомимся с пассивными и активными методами IPC и заодно отметим, какие из методов, рассмотренных в предыдущей главе, относятся к активным;
- коммуникационные протоколы и присущие им характеристики. Вы узнаете, что такое сериализация и десериализация и какую роль эти механизмы играют в полноценном межпроцессном взаимодействии;
- файловые дескрипторы и их ключевая роль в создании каналов IPC;
- API для работы с POSIX-сигналами, POSIX-каналами и очередями сообщений. Будут продемонстрированы общие примеры использования каждой из этих методик;
- компьютерные сети и то, как два процесса могут взаимодействовать по существующей сети;

- модель «слушатель — соединитель» и то, как два процесса могут установить транспортное соединение по разным сетям. Это послужит основой для будущих обсуждений, касающихся программирования сокетов;
- программирование сокетов и что собой представляет объект сокета;
- последовательность действий, которые выполняет каждый процесс, устанавливающий соединение вида «слушатель — соединитель», и API библиотеки POSIX-сокетов, который используется при этом.

В первом разделе мы еще раз пройдемся по методам IPC.

Методы межпроцессного взаимодействия

К методам IPC обычно относят любые средства взаимодействия и передачи данных между процессами. В предыдущей главе в качестве начального подхода к разделению данных между процессами были предложены файловая система и разделяемая память. На тот момент мы не использовали термин IPC, но здесь он вполне уместен! В этой главе мы прибавим еще несколько методов IPC к уже знакомым нам, но следует помнить: они имеют ряд отличий. Однако прежде, чем рассматривать эти отличия и пытаться поделить их на категории, перечислим их:

- разделяемая память;
- файловая система (как на диске, так и в памяти);
- POSIX-сигналы;
- POSIX-каналы;
- очереди сообщений POSIX;
- сокеты домена Unix;
- интернет-сокеты (или сетевые сокеты).

С точки зрения программирования, методики разделяемой памяти и файловой системы определенно чем-то схожи, поэтому их можно отнести к одной категории, известной как *активные* методы IPC. Остальные подходы выделяются на их фоне, и мы будем называть их *пассивными*. Эта и следующая главы посвящены пассивному межпроцессному взаимодействию и описывают различные методики.

Обратите внимание: все методы IPC сводятся к передаче сообщений между двумя процессами. Поскольку термин *«сообщение»* активно используется в следующих абзацах, его сначала стоит определить.

Каждое сообщение содержит последовательность байтов, которые собираются вместе в соответствии с четко определенным интерфейсом, протоколом или стандар-

том. Структура сообщения должна быть известна обоим процессам, работающим с ним, и обычно описывается в рамках коммуникационного протокола.

Ниже перечислены различия между активными и пассивными методиками.

- Активные методики подразумевают наличие разделяемого ресурса, или *носителя*, который доступен в пользовательском пространстве и является внешним по отношению к обоим процессам. Роль разделяемого ресурса может играть файл, разделяемая память или даже сетевой сервис, такой как NFS-сервер. Эти носители служат основным средством хранения сообщений, создаваемых и потребляемых процессами. В пассивных методах вместо таких разделяемых ресурсов, или носителей, используется *канал*, через который процессы отправляют и принимают сообщения. Сами сообщения при этом нигде не хранятся.
- При активном подходе все процессы должны сами *извлекать* из носителя доступные сообщения. При пассивном — входящие сообщения *передаются (доставляются)* принимающей стороне.
- Активные методики подразумевают наличие разделяемого ресурса или носителя, конкурентный доступ к которому необходимо синхронизировать. Вот почему в предыдущей главе мы исследовали различные способы синхронизации для этой категории IPC. Обратите внимание: в случае с пассивными методиками синхронизация не требуется.
- При использовании активных методик процессы могут работать независимо друг от друга. Это объясняется тем, что сообщения могут храниться в разделяемом ресурсе, и их извлечение можно отложить. Иными словами, процессам доступно выполнение в *асинхронной* манере. Для сравнения, в случае применения пассивных методов IPC оба процесса должны находиться в активном состоянии. Кроме того, ввиду мгновенного поступления сообщений принимающий процесс может потерять какую-то их часть, если не запущен в это время. То есть процессы выполняются в *синхронной* манере.

При использовании пассивных методик каждый процесс имеет временный буфер, который хранит поступающие сообщения. Он находится в ядре и существует, пока процесс не завершится. Доступ к нему может быть конкурентным, но в этом случае синхронизация должна гарантироваться самим ядром.

Сообщения, которые либо передаются по IPC-каналу (пассивный подход), либо хранятся на IPC-накопителе (активный подход), должны иметь содержимое, понятное принимающему процессу. Это значит, что оба процесса: отправитель и получатель — должны знать, как их создавать и разбирать. Сообщения состоят из байтов; отсюда следует, что обе стороны должны уметь превращать объект (текст или видео) в байтовую последовательность и восстанавливать тот же объект из полученных

байтов. Чуть позже вы увидите, что совместимость процессов обеспечивается общим *коммуникационным протоколом*, который они используют.

В следующем разделе мы подробно обсудим коммуникационные протоколы.

Коммуникационные протоколы

Одного лишь коммуникационного канала или носителя недостаточно. Две стороны, желающие взаимодействовать по общему каналу, должны понимать друг друга! Представьте, что два человека хотят пообщаться на одном языке — например, на английском или японском. В этом случае язык общения можно считать коммуникационным протоколом, который используют две стороны взаимодействия.

То же самое в контексте IPC можно сказать и о процессах; им нужен общий язык, на котором они могут общаться. Формально он называется *протоколом*. В рамках этого раздела мы обсудим коммуникационные протоколы и их различные характеристики, такие как *длина* и *содержимое* сообщения. Но сначала коммуникационный протокол необходимо описать в более глубоком смысле. Отмечу, что в данной главе основное внимание уделяется методам межпроцессного взаимодействия, поэтому речь будет идти только о протоколах, связывающих два процесса. Любые виды взаимодействия сторон, не являющиеся процессами, выходят за рамки текущей главы.

Процессы могут передавать только байты. То есть речь фактически о том, что каждый фрагмент информации, передаваемый с помощью любого метода IPC, должен быть предварительно переведен в байтовую последовательность. Это называется *сериализацией* или *маршалингом*. Текстовый абзац, отрезок аудиозаписи, музыкальная композиция или любой другой объект перед попаданием в канал IPC должен быть сериализован или сохранен на носителе IPC. В контексте межпроцессного взаимодействия это означает, что сообщения, передаваемые между двумя процессами, представляют собой байты, упорядоченные строго определенным образом.

С другой стороны, когда процесс получает последовательность байтов по каналу IPC, он должен уметь воссоздать из них исходный объект. Это называется *десериализацией* или *демаршалингом*.

Поговорим о том, как происходит сериализация и десериализация. Когда процесс хочет отправить объект по какому-либо имеющемуся каналу IPC, он сначала сериализует его в байтовый массив и передает результат другой стороне. Принимающий процесс десериализует поступившие байты и воссоздает отправленный объект. Как видите, эти операции являются обратными по отношению друг к другу и используются обеими сторонами для обмена информацией по каналу IPC, ориентированному на передачу байтов. Без этого невозможно обойтись. Любая тех-

нология межпроцессного взаимодействия (RPC, RMI и т. д.) активно задействует сериализацию и десериализацию различных объектов. Пока под сериализацией мы будем понимать совокупность этих двух операций.

Обратите внимание: сериализация применяется не только в пассивных методах IPC, которые рассматривались до сих пор. Активные методы IPC, такие как файловая система или разделяемая память, тоже нуждаются в сериализации. Дело в том, что носители в этих методах могут хранить последовательности байтов, и если процесс хочет записать некий объект, скажем, в разделяемый файл, то должен сначала сериализовать его. Таким образом, сериализация — неотъемлемая часть всех методов IPC; независимо от выбранного вами метода, при работе с каналом или носителем постоянно приходится иметь дело с сериализацией и десериализацией.

Выбор коммуникационного протокола сам по себе определяет способ сериализации, так как в рамках протокола детально описывается порядок, в котором размещаются байты. Это очень важно, поскольку сериализованный объект должен быть десериализован обратно принимающей стороной. Следовательно, механизмы сериализации и десериализации должны подчиняться правилам, которые диктует протокол. Если эти механизмы несовместимы между собой, то стороны фактически теряют всякую возможность взаимодействовать, просто ввиду того, что получатель не может воссоздать переданный объект.

Иногда в качестве синонима десериализации используется термин «разбор» (parsing), однако на самом деле это совершенно разные понятия.

Чтобы перевести наш разговор в более практическую плоскость, рассмотрим реальные примеры. Веб-сервер и веб-клиент взаимодействуют по *протоколу передачи гипертекста* (Hyper Text Transfer Protocol, HTTP). Следовательно, обе стороны, желающие общаться друг с другом, должны применять совместимые средства сериализации и десериализации HTTP-сообщений. В качестве еще одного примера можно привести протокол *системы доменных имен* (Domain Name Service, DNS). DNS-клиент и DNS-сервер должны использовать для взаимодействия совместимые средства сериализации и десериализации. Отмечу, что в отличие от протокола HTTP, предназначенного для передачи текста, DNS является двоичным протоколом. Мы поговорим об этом в следующих разделах.

Сериализация может проводиться в разных компонентах программного проекта, потому ее обычно реализуют в виде библиотек, которые можно подключать там, где они нужны. Для таких распространенных протоколов, как HTTP, DNS и FTP существуют широко известные и легко доступные сторонние библиотеки. Но если у вас есть собственные протоколы, то библиотеки сериализации для них придется писать самостоятельно.

Такие распространенные протоколы, как HTTP, FTP и DNS, — это стандарты, описанные в публичных документах, которые называются рабочими предложениями (request for comments, RFC). Например, протокол HTTP/1.1 описан в RFC-2616. В Google можно легко найти страницу соответствующего документа.

Вдобавок следует отметить, что *библиотеки сериализации* могут быть доступны в разных языках программирования. Обратите внимание: сам алгоритм сериализации не зависит ни от какого языка, поскольку всего лишь описывает порядок размещения байтов и способ их интерпретации. Поэтому алгоритмы сериализации и десериализации можно разрабатывать на любых языках. Это чрезвычайно важный аспект. В крупных программных проектах некоторые компоненты могут быть написаны на разных языках, и бывают случаи, когда этим компонентам нужно обмениваться информацией. В результате нам необходим один и тот же алгоритм сериализации, реализованный с помощью разных технологий. Например, существуют HTTP-сериализаторы для C, C++, Java, Python и т. д.

Если подытожить, то главная идея данного раздела в том, что двум сторонам, которые хотят общаться друг с другом, необходим четко определенный протокол. Протокол IPC — это стандарт, который диктует, как в целом должно осуществляться взаимодействие и какие правила относительно порядка следования байтов в сообщениях должны соблюдаться. Для передачи объектов по байтовым каналам IPC требуются алгоритмы сериализации.

В следующем подразделе будут перечислены характеристики протоколов межпроцессного взаимодействия.

Характеристики протоколов

Протоколы IPC имеют различные характеристики. Если коротко, то любой протокол должен определять разные виды содержимого для сообщений, передающихся по IPC-каналу. В одних протоколах сообщения могут иметь фиксированную длину, а в других — переменную. Некоторые протоколы вынуждают использовать предоставляемые ими операции в синхронной манере; есть и те, что поддерживают асинхронную работу. Далее мы рассмотрим эти отличительные особенности. Существующие протоколы можно разделить на разные категории в зависимости от характеристик.

Тип содержимого

Сообщения, отправляемые по IPC-каналам, могут иметь *текстовое*, *двоичное* или гибридное содержимое. Двоичные данные состоят из байтов со значениями в диапазоне от 0 до 255. Текстовые представляют собой символы, используемые в тексте.

Иными словами, в текстовом содержимом допускаются только алфавитно-цифровые и некоторые дополнительные символы.

Текстовые данные можно считать частным случаем двоичных, но мы стараемся их разделять и работать с ними по-разному. Например, текстовые сообщения имеет смысл сжимать перед отправкой, а вот у двоичных сообщений плохой *коэффициент сжатия* (реальный размер, поделенный на размер сжатых данных). Следует понимать: одни протоколы — сугубо текстовые (например, JSON), а другие — полностью двоичные (такие как DNS). Но протоколы наподобие BSON и HTTP позволяют хранить в сообщениях сочетание из текстовых и двоичных данных; то есть итоговое сообщение может быть смесью обычных байтов и текста.

Отмечу, что двоичное содержимое может передаваться в текстовом виде. Существуют разные кодировки, которые позволяют представлять двоичные данные с помощью текстовых символов. Один из самых известных алгоритмов *кодирования двоичных данных в текст*, который выполняет такое преобразование, — *Base64*. Подобные алгоритмы широко используются в сугубо текстовых протоколах, таких как JSON, для отправки двоичной информации.

Длина сообщений

Сообщения, сгенерированные в соответствии с протоколом IPC, могут иметь либо *фиксированную*, либо *переменную* длину. В первом случае длина всех сообщений совпадает, а во втором может быть разной. Прием сообщений фиксированной или переменной длины оказывает непосредственное влияние на то, как их содержимое будет десериализовано принимающей стороной. Использование протоколов, которые всегда генерируют сообщения одинаковой длины, может упростить их разбор, поскольку получатель заранее знает, сколько байтов ему нужно прочитать из канала; к тому же сообщения одинаковой длины обычно (но не всегда) имеют идентичную структуру. Чтение из IPC-канала сообщений фиксированной длины, которые устроены внутри по одному и тому же принципу, дает нам отличную возможность с помощью структур языка C обратиться к их содержимому, используя заранее подготовленные поля. Это похоже на то, как мы работали с объектами разделяемой памяти в предыдущей главе.

Если протокол может генерировать сообщения разной длины, то определить, где заканчивается отдельно взятое сообщение, может быть непросто. Принимающая сторона должна неким образом (об этом чуть позже) знать, прочитано ли сообщение до конца, или в канале еще что-то осталось. Стоит отметить: для получения всего сообщения иногда необходимо прочитать несколько блоков и один из них может содержать данные, принадлежащие двум смежным сообщениям. Пример этого будет показан в главе 20.

Большинство протоколов используют переменную длину, и работа с сообщениями фиксированной длины обычно является роскошью, поэтому имеет смысл обсудить

методы, с помощью которых разные протоколы позволяют различать или разделять сообщения с разной длиной. Иными словами, эти протоколы задействуют механизмы, позволяющие им пометить конец сообщения, благодаря чему получатель может свериться с этими метками и понять, прочитано ли сообщение целиком. Некоторые из таких методов перечислены ниже.

- *Использование разделителя*. Разделитель — это последовательность байтов (в двоичных протоколах) или символов (в текстовых протоколах), которые обозначают конец сообщения. Разделитель следует выбирать с учетом содержимого сообщений, чтобы его было легко отличить от самих данных.
- *Обрамление фиксированной длины*. В таких протоколах у каждого сообщения есть префикс фиксированной длины (обычно 4 байта или даже больше), в котором указано, сколько байтов необходимо прочитать, чтобы получить все сообщение целиком. Примеры использования этой методики — протоколы *TLV* (Tag-Value-Length) и *ASN* (Abstract Syntax Notation).
- *Использование конечного автомата*. Такие протоколы имеют *формальную грамматику*, которую можно смоделировать с помощью конечного автомата. Чтобы прочитать полное сообщение из IPC-канала, принимающая сторона должна знать грамматику протокола и применить подходящий десериализатор, основанный на конечном автомате.

Последовательность

В большинстве протоколов *общение* между процессами проходит по принципу *«запрос — ответ»*. Одна сторона шлет запрос, а другая на него отвечает. Такая схема обычно используется в клиент-серверных системах. Процесс-слушатель (которым зачастую выступает серверный процесс) ожидает сообщения и, когда оно приходит, отвечает соответствующим образом.

Если протокол синхронный или последовательный, то отправитель (клиент) подождет, пока слушатель (сервер) не завершит обработку запроса и не вернет ответ. То есть, пока слушатель не ответит, отправитель находится в *заблокированном* состоянии. В асинхронном протоколе процесс-отправитель не блокируется и может заниматься другими делами, в то время как запрос обрабатывается слушателем. Это значит, что во время подготовки ответа отправитель не будет заблокирован.

В асинхронном протоколе должен быть предусмотрен *активный* или *пассивный* механизм, позволяющий отправителю проверять наличие ответа. Активный подход означает, что отправитель будет регулярно запрашивать результат у слушателя. В случае с пассивным подходом слушатель сам передаст ответ отправителю по тому же или другому каналу взаимодействия.

Последовательность протокола не ограничена сценариями вида «запрос — ответ». Приложения для обмена сообщениями обычно используют эту методику для обеспечения максимальной отзывчивости как на серверной, так и на клиентской стороне.

Взаимодействие в рамках одного компьютера

В этом разделе мы поговорим о локальном межпроцессном взаимодействии. IPC между разными компьютерами будет темой следующей главы. Ниже перечислены четыре формальных метода, с помощью которых могут общаться процессы, находящиеся в одной и той же системе:

- POSIX-сигналы;
- POSIX-каналы;
- очереди сообщений POSIX;
- сокеты домена Unix.

POSIX-сигналы, в отличие от методик, рассмотренных ранее, не создают канал связи между процессами, но могут служить для уведомления процессов о событиях. В определенных сценариях процессы могут использовать такие сигналы, чтобы сообщать друг другу о тех или иных событиях, происходящих в системе.

Прежде чем переходить к первому методу IPC — POSIX-сигналам, — обсудим файловые дескрипторы. Их так или иначе применяет любой механизм межпроцессного взаимодействия (если не считать POSIX-сигналов). Поэтому их подробному рассмотрению посвящен весь следующий подраздел.

Файловые дескрипторы

Два взаимодействующих процесса могут выполняться как на одном, так и на двух разных компьютерах, соединенных по сети. В данном подразделе и в большей части этой главы основное внимание уделяется первому варианту, в котором процессы находятся в одной системе. И здесь очень важную роль играют файловые дескрипторы. Отмечу, что они применяются и в распределенном IPC, но там называются *сокетами*. Мы подробно расскажем о них в следующей главе.

Файловый дескриптор — абстрактная ссылка на локальный объект, который можно использовать для чтения и записи данных. Несмотря на свое название, файловые дескрипторы могут обозначать широкий спектр различных механизмов, предназначенных для чтения и изменения байтовых потоков.

Естественно, в число объектов, на которые могут ссылаться файловые дескрипторы, входят обычные файлы, размещенные в файловой системе (либо на жестком диске, либо в памяти).

С помощью файловых дескрипторов можно обращаться и к устройствам. Как вы уже видели в главе 10, у каждого устройства есть свой файл, который обычно находится в каталоге /dev.

Что касается пассивных методов IPC, то файловый дескриптор может представлять IPC-канал с возможностью чтения и записи. Вот почему первый этап подготовки IPC-канала заключается в определении ряда файловых дескрипторов.

Теперь, когда вы лучше ориентируетесь в файловых дескрипторах и понимаете, что они собой представляют, мы переходим к первому методу межпроцессного взаимодействия, который можно использовать в системах, состоящих из одного компьютера, — POSIX-сигналах. Однако стоит отметить: файловые дескрипторы в этом подходе не применяются; мы еще вернемся к ним в разделах, посвященных POSIX-каналам и очередям сообщений POSIX. Но сначала поговорим о POSIX-сигналах.

POSIX-сигналы

В POSIX-системах процессы и потоки могут отправлять и принимать ряд заранее определенных сигналов, каждый из которых может быть отправлен процессом, потоком или даже самим ядром. На самом деле сигналы предназначены для уведомления процессов или потоков о событиях или ошибках. Например, когда система должна перезагрузиться, она рассылает всем процессам сигнал `SIGTERM`, уведомляя их о том, что им следует немедленно завершить работу. Процесс, получивший сигнал, должен действовать соответствующим образом. В некоторых случаях ему ничего не нужно делать, но иногда он должен сохранить текущее состояние.

В табл. 19.1 перечислены сигналы, доступные в системе Linux. Она взята со справочной страницы по сигналам, с которой можно ознакомиться, перейдя по ссылке http://www.man7.org/linux/man-pages/man7/signal.7.html.

Таблица 19.1. Список всех сигналов, доступных в системе Linux

Сигнал	Стандарт	Действие	Комментарий
SIGABRT	P1990	Core	Сигнал о прекращении, посланный abort(3)
SIGALRM	P1990	Term	Сигнал таймера от alarm(2)
SIGBUS	P2001	Core	Ошибка шины (некорректный доступ к памяти)
SIGCHLD	P1990	Ign	Дочерний процесс остановлен или прерван
SIGCLD	—	Ign	Синоним SIGCHLD
SIGCONT	P1990	Cont	Продолжить в случае остановки
SIGEMT	—	Term	Ловушка эмулятора
SIGFPE	P1990	Core	Неправильная операция с плавающей запятой
SIGHUP	P1990	Term	Обнаружено закрытие управляющего терминала
SIGILL	P1990	Core	Некорректная инструкция от процессора

Сигнал	Стандарт	Действие	Комментарий
SIGINFO	—	—	Синоним SIGPWR
SIGINT	P1990	Term	Прерывание с клавиатуры
SIGIO	—	Term	Теперь возможен ввод/вывод (4.2 BSD)
SIGIOT	—	Core	Ловушка IOT. Синоним SIGABRT
SIGKILL	P1990	Term	Сигнал принудительного завершения
SIGLOST	—	Term	Не действует блокировка файла
SIGPIPE	P1990	Term	Запись в канале, не имеющем считывающих процессов
SIGPOLL	P2001	Term	Событие, которое можно отложить (Sys V). Синоним SIGIO
SIGPROF	P2001	Term	Закончилось время профилирующего таймера
SIGPWR	—	Term	Отказ системы питания (System V)
SIGQUIT	P1990	Core	Прекратить работу с клавиатурой
SIGSEGV	P1990	Core	Некорректное обращение к памяти
SIGSTKFLT	—	Term	Ошибка в стеке сопроцессора (не используется)
SIGSTOP	P1990	Stop	Процесс остановлен
SIGTSTP	P1990	Stop	Остановка с помощью клавиатуры
SIGSYS	P2001	Core	Некорректный системный вызов (SVr4); см. также seccomp(2)
SIGTERM	P1990	Term	Сигнал снятия
SIGTRAP	P2001	Core	Ловушка отладки
SIGTTIN	P1990	Stop	Запрос на ввод с терминала для фонового процесса
SIGTTOU	P1990	Stop	Запрос на вывод с терминала для фонового процесса
SIGUNUSED	—	Core	Синоним SIGSYS
SIGURG	P2001	Ign	Приоритетные данные в сокете (4.2 BSD)
SIGUSR1	P1990	Term	Определяемый пользователем сигнал № 1
SIGUSR2	P1990	Term	Определяемый пользователем сигнал № 2
SIGVTALRM	P2001	Term	Виртуальный таймер (4.2 BSD)
SIGXCPU	P2001	Core	Превышено время работы процессора (4.2 BSD); см setrlimit(2)
SIGXFSZ	P2001	Core	Превышен размер файла (4.2 BSD); см. setrlimit(2)
SIGWINCH	—	Ign	Сигнал изменения размера окна (4.3 BSD, Sun)

В этой таблице видно, что у Linux есть собственные сигналы, которые не входят в стандарт POSIX. Большинство сигналов относятся к общеизвестным событиям, но два из них может определить пользователь. Обычно это нужно в ситуациях, когда во время выполнения программы должны быть совершены определенные действия. В примере 19.1 показано, как применять и обрабатывать сигналы в программе на языке C (листинг 19.1).

Листинг 19.1. Обработка POSIX-сигналов (ExtremeC_examples_chapter19_1.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

void handle_user_signals(int signal) {
  switch (signal) {
    case SIGUSR1:
      printf("SIGUSR1 received!\n");
      break;
    case SIGUSR2:
      printf("SIGUSR2 received!\n");
      break;
    default:
      printf("Unsupported signal is received!\n");
  }
}

void handle_sigint(int signal) {
  printf("Interrupt signal is received!\n");
}

void handle_sigkill(int signal) {
  printf("Kill signal is received! Bye.\n");
  exit(0);
}

int main(int argc, char** argv) {
  signal(SIGUSR1, handle_user_signals);
  signal(SIGUSR2, handle_user_signals);
  signal(SIGINT, handle_sigint);
  signal(SIGKILL, handle_sigkill);
  while (1);
  return 0;
}
```

В этом примере мы использовали функцию `signal`, чтобы назначить различные обработчики для определенных сигналов. Один обработчик предназначен для сигналов, определенных пользователем, второй — для сигнала `SIGINT`, третий — для сигнала `SIGKILL`.

Программа представляет собой обычный бесконечный цикл, и наша единственная цель — обработка некоторых сигналов. В терминале 19.1 показаны команды для компиляции этого примера и его запуска в фоновом режиме.

Терминал 19.1. Компиляция и запуск примера 19.1

```
$ gcc ExtremeC_examples_chapter19_1.c -o ex19_1.out
$ ./ex19_1.out &
[1] 4598
$
```

Узнав PID программы, мы можем послать ей несколько сигналов. В нашем случае программа имеет PID 4598 (у вас он будет отличаться) и выполняется в фоне. Чтобы послать процессу сигнал, можно воспользоваться командой `kill`. Проверим наш пример с помощью следующих команд (терминал 19.2).

Терминал 19.2. Передача разных сигналов фоновому процессу

```
$ kill -SIGUSR2 4598
SIGUSR2 received!
$ kill -SIGUSR1 4598
SIGUSR2 received!
$ kill -SIGINT 4598
Interrupt signal is received!
$ kill -SIGKILL 4598
$
[1]+  Stopped                 ./ex19_1.out
$
```

Как видите, программа обрабатывает все сигналы, кроме `SIGKILL`. Это объясняется тем, что процессы не могут обрабатывать `SIGKILL`; уведомление о принудительном завершении процесса обычно поступает его родителю.

Обратите внимание: `SIGINT` (сигнал прерывания) можно послать программе, находящейся на переднем плане, нажав Ctrl+C. Следовательно, нажимая сочетание этих клавиш, вы отправляете активной программе сигнал прерывания. По умолчанию сигнал `SIGINT` просто останавливает выполнение программы, но, как показано в предыдущем примере, мы можем написать собственный обработчик, в котором `SIGINT` будет игнорироваться.

Сигналы можно отправлять не только из командной оболочки, но и из самих процессов, при условии, что нам известен PID получателя. Мы можем использовать функцию `kill` (объявленную в `signal.h`), которая делает то же самое, что и ее консольный аналог. Она принимает два аргумента: PID адресата и номер сигнала. Процессы и потоки также могут применять функции `kill` и `raise` для отправки сигналов самим себе. Обратите внимание: функция `raise` посылает сигнал текущему

потоку. Это может быть довольно полезно в ситуациях, когда вам нужно уведомить о событии другую часть своей программы.

В завершение следует сказать: сигналы поступают асинхронно, и потому факт занятости главного потока бесконечным циклом не играет никакой роли. Это можно наблюдать в терминале 19.2. Таким образом, вы можете быть уверены в том, что ваш процесс всегда сможет получить входящие сигналы.

Теперь пришло время поговорить еще об одном локальном методе межпроцессного взаимодействия, который может пригодиться в определенных обстоятельствах, — о POSIX-каналах.

POSIX-каналы

Unix поддерживает однонаправленные POSIX-каналы (pipes), которые можно использовать для обмена сообщениями между двумя процессами. При создании POSIX-канала вы получаете два файловых дескриптора: один для записи в канал, а второй для чтения из него. Простой пример использования POSIX-каналов показан в листинге 19.2.

Листинг 19.2. Пример 19.2, демонстрирующий использование POSIX-каналов (ExtremeC_examples_chapter19_2.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>

int main(int argc, char** argv) {
  int fds[2];
  pipe(fds);

  int childpid = fork();
  if (childpid == -1) {
    fprintf(stderr, "fork error!\n");
    exit(1);
  }
  if (childpid == 0) {
    // Потомок закрывает файловый дескриптор для чтения
    close(fds[0]);
    char str[] = "Hello Daddy!";
    // Потомок записывает в файловый дескриптор, открытый для записи
    fprintf(stdout, "CHILD: Waiting for 2 seconds ...\n");
    sleep(2);
    fprintf(stdout, "CHILD: Writing to daddy ...\n");
    write(fds[1], str, strlen(str) + 1);
  } else {
```

```
    // Родитель закрывает файловый дескриптор для записи
    close(fds[1]);
    char buff[32];
    // Родитель читает из файлового дескриптора, открытого для чтения
    fprintf(stdout, "PARENT: Reading from child ...\n");
    int num_of_read_bytes = read(fds[0], buff, 32);
    fprintf(stdout, "PARENT: Received from child: %s\n", buff);
  }
  return 0;
}
```

Во второй строчке функции `main` используется вызов `pipe`. Как уже говорилось ранее, он принимает массив из двух файловых дескрипторов (один для чтения, а другой для записи) и открывает каждый из них. Первый дескриптор имеет индекс 0, а второй — 1. Первый должен использоваться для чтения из канала, а второй — для записи в канал.

Чтобы получить второй процесс, мы воспользовались вызовом `fork`. Как уже объяснялось в главе 17, этот вызов создает дочерний процесс, который является копией родительского. Следовательно, после выполнения функции `fork` дочернему процессу будут доступны открытые файловые дескрипторы.

После вызова `fork` родительский процесс входит в блок `else`, а дочерний — в блок `if`. Первым делом каждый процесс закрывает файловый дескриптор, который не будет использоваться в дальнейшем. В этом примере родитель собирается читать из канала, а потомок хочет туда записывать. Вот почему родительский процесс закрывает второй файловый дескриптор (предназначенный для записи), а дочерний — первый (предназначенный для чтения). Обратите внимание: каналы являются однонаправленными и не поддерживают обратную связь.

В терминале 19.3 показан вывод примера 19.2.

Терминал 19.3. Результат выполнения примера 19.2

```
$ gcc ExtremeC_examples_chapter19_2.c -o ex19_2.out
$ ./ex19_2.out
PARENT: Reading from child ...
CHILD: Waiting for 2 seconds ...
CHILD: Writing to daddy ...
PARENT: Received from child: Hello Daddy!
$
```

Как можно видеть в листинге 19.2, для операций чтения и записи используются функции `read` и `write`. Ранее мы уже упоминали о том, что в пассивном межпроцессном взаимодействии файловый дескриптор указывает на байтовый канал, что позволяет использовать функции для работы с файловыми дескрипторами. Функции `read` и `write` принимают файловый дескриптор и производят с ним действия независимо от того, какой IPC-канал он представляет.

В данном примере новый процесс порождался с помощью функции `fork`. Но представьте ситуацию с двумя разными процессами, созданными независимо друг от друга. Возникает вопрос: каким образом они могут взаимодействовать через разделяемый канал? Процесс, который хочет иметь доступ к объекту канала, должен иметь соответствующий файловый дескриптор. Этого можно добиться двумя путями:

- один из процессов должен подготовить канал и передать соответствующие файловые дескрипторы другому процессу;
- процесс должен использовать именованный канал.

В первом случае процессы должны применять для обмена файловыми дескрипторами канал на основе сокета домена Unix. Но проблема в том, что если между процессами уже установлен такой канал, то они могут использовать его для дальнейшего взаимодействия. В результате отпадает необходимость в POSIX-канале, у которого к тому же менее дружественный API, по сравнению с сокетами домена Unix.

Второй вариант выглядит более разумным. Один из процессов может использовать функцию `mkfifo` и создать файл очереди по определенному пути. Затем второй процесс может взять путь к уже созданному файлу и открыть его для последующего общения. Стоит отметить: канал по-прежнему однонаправленный и в некоторых ситуациях один из процессов должен открывать файл только для чтения, а другой — только для записи.

Предыдущий пример имеет еще одну особенность, на которую стоит обратить внимание. Прежде чем записывать в канал, дочерний процесс ждет 2 секунды. А родительский процесс тем временем заблокирован на функции `read`. Поэтому, когда в канал ничего не записывается, читающий процесс блокируется.

В заключение напомню, что POSIX-каналы относятся к пассивным методам IPC. Как уже объяснялось ранее, пассивные методы подразумевают наличие в ядре буфера для хранения входящих сообщений, и POSIX-каналы не исключение. Записанные сообщения хранятся в ядре, пока их не прочитают. Если же процесс-владелец завершает работу, то объект канала и его буфер в ядре уничтожаются.

В следующем подразделе мы обсудим очереди сообщений POSIX.

Очереди сообщений POSIX

Очереди сообщений, размещенные в ядре, — часть стандарта POSIX. Они во многом отличаются от POSIX-каналов. Ниже перечислены некоторые фундаментальные отличия.

- В канале хранятся байты, а в очереди — сообщения. Каналу ничего не известно о структуре записываемых данных, тогда как очередь хранит отдельные со-

общения, которые добавляются при каждом вызове функции `write`. Очередь соблюдает границы между записанными сообщениями. Чтобы это проиллюстрировать, представьте: у нас есть три сообщения размером 10, 20 и 30 байт и каждое из них записывается как в POSIX-канал, так и в очередь сообщений POSIX. Канал знает лишь то, что внутри у него 60 байт, и позволяет программе считывать 15-байтные фрагменты. А вот очередь сообщений знает только то, что у нее есть три сообщения, и, поскольку ни одно из них не занимает 15 байт, программа не может прочитать 15-байтный фрагмент.

- Каналы и очереди сообщений имеют ограниченный размер, который исчисляется в байтах и сообщениях соответственно. Кроме того, каждое сообщение имеет максимальный размер, измеряемый в байтах.
- Каждая очередь сообщений, такая как именованная разделяемая память или именованный семафор, открывает файл. Это не обычные файлы, но другие процессы могут использовать их для доступа к одному и тому же экземпляру очереди сообщений.
- Сообщения в очереди могут иметь разный приоритет, тогда как в канале все байты равны.

Эти два механизма имеют и некоторые сходства:

- оба однонаправленные; для двунаправленного взаимодействия необходимо создать два экземпляра очереди или канала;
- оба имеют ограниченную емкость; вы можете записать только определенное количество байтов или сообщений;
- в большинстве POSIX-систем оба механизма представлены файловыми дескрипторами, поэтому для работы с ними можно использовать функции ввода/вывода, такие как `read` и `write`;
- обе методики работают *без соединения*. Иными словами, если два разных процесса запишут два разных сообщения, то один из них может прочитать сообщение другого. У сообщений нет владельца, и их могут читать любые процессы. Это чревато проблемами, особенно если с одним и тем же каналом или очередью сообщений работают несколько конкурентных процессов.

Очереди сообщений POSIX, о которых идет речь в этой главе, не следует путать с брокерами сообщений, применяемыми в архитектуре MQM (Message Queue Middleware — связующее ПО для работы с очередями сообщений).

В Интернете есть разные ресурсы, в которых объясняется, как работают очереди сообщений POSIX. По ссылке, представленной ниже, эта тема подается в контексте операционной системы QNX, но большая часть материала применима и к другим POSIX-системам: https://users.pja.edu.pl/~jms/qnx/help/watcom/clibref/mq_overview.html.

Пришло время посмотреть, как это работает на практике. В примере 19.3 реализован тот же сценарий, что и в предыдущем разделе, но вместо POSIX-канала используется очередь сообщений POSIX. Все функции, относящиеся к очередям сообщений, объявлены в заголовочном файле `mqueue.h`. Некоторые из них мы вскоре обсудим.

Обратите внимание: код, представленный в листинге 19.3, не скомпилируется в macOS, поскольку эта система не поддерживает очереди сообщений POSIX.

Листинг 19.3. Пример 19.3 с использованием очереди сообщений POSIX (ExtremeC_examples_chapter19_3.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <mqueue.h>

int main(int argc, char** argv) {
  // Обработчик очереди сообщений
  mqd_t mq;

  struct mq_attr attr;
  attr.mq_flags = 0;
  attr.mq_maxmsg = 10;
  attr.mq_msgsize = 32;
  attr.mq_curmsgs = 0;

  int childpid = fork();
  if (childpid == -1) {
    fprintf(stderr, "fork error!\n");
    exit(1);
  }
  if (childpid == 0) {
    // Потомок ждет, пока родитель создает очередь
    sleep(1);
    mqd_t mq = mq_open("/mq0", O_WRONLY);
    char str[] = "Hello Daddy!";
    // Потомок записывает в файловый дескриптор, открытый для записи
    fprintf(stdout, "CHILD: Waiting for 2 seconds ...\n");
    sleep(2);
    fprintf(stdout, "CHILD: Writing to daddy ...\n");
    mq_send(mq, str, strlen(str) + 1, 0);
    mq_close(mq);
  } else {
    mqd_t mq = mq_open("/mq0", O_RDONLY | O_CREAT, 0644, &attr);
    char buff[32];
    fprintf(stdout, "PARENT: Reading from child ...\n");
    int num_of_read_bytes = mq_receive(mq, buff, 32, NULL);
    fprintf(stdout, "PARENT: Received from child: %s\n", buff);
    mq_close(mq);
```

```
    mq_unlink("/mq0");
  }
  return 0;
}
```

Чтобы скомпилировать этот код, выполните команды, приведенные в терминале 19.4. Отмечу, что в Linux код следует скомпоновать с библиотекой `rt`.

Терминал 19.4. Сборка примера 19.3 в Linux

```
$ gcc ExtremeC_examples_chapter19_3.c -lrt -o ex19_3.out
$
```

Вывод примера 19.3 показан в терминале 19.5. Как видите, он ничем не отличается от того, который мы имели в примере 19.2, только теперь для выполнения той же логики используются очереди сообщений POSIX.

Терминал 19.5. Выполнение примера 19.3. в Linux

```
$ ./ex19_3.out
PARENT: Reading from child ...
CHILD: Waiting for 2 seconds ...
CHILD: Writing to daddy ...
PARENT: Received from child: Hello Daddy!
$
```

Напомню, что POSIX-каналы и очереди сообщений имеют ограниченный размер буфера в ядре. Следовательно, запись в канал или очередь, из которой никто не читает, может привести к блокировке любых операций записи. Иными словами, любой вызов функции `write` будет оставаться заблокированным, пока потребитель не прочитает сообщение из очереди или байты из канала.

В следующем подразделе я кратко опишу сокеты домена Unix. Этот механизм является самым очевидным выбором в ситуации, когда необходимо соединить два локальных процесса в одной системе.

Сокеты домена Unix

Еще один подход, с помощью которого разные процессы могут взаимодействовать на одном компьютере, состоит в использовании сокетов домена Unix. Это особая разновидность сокетов, работающих только в рамках одной системы. Этим они отличаются от сетевых сокетов, позволяющих двум процессам, находящимся на разных компьютерах, общаться друг с другом по сети. Сокеты домена Unix имеют различные характеристики, которые делают их важными и нетривиальными в использовании по сравнению с POSIX-каналами и очередями сообщений POSIX. Самая главная характеристика — тот факт, что они двунаправленные. Благодаря

этому одного сокета достаточно для чтения и записи в связанный с ним канал. То есть каналы, основанные на сокетах домена Unix, являются полнодуплексными. Кроме того, такие сокеты могут работать как с *сеансами*, так и с отдельными *сообщениями*. Это делает их еще более гибкими. Мы вернемся к поддержке сеансов и сообщений в следующих разделах.

Поскольку сокеты домена Unix нельзя обсуждать, не имея понимания основных принципов программирования сокетов, данная тема в этой главе больше рассматриваться не будет. В следующих разделах вы познакомитесь с областью программирования сокетов и сопутствующими концепциями. А полноценное обсуждение сокетов домена Unix будет продолжено в главе 20.

Введение в программирование сокетов

Прежде чем переходить к примерам реального кода на языке C, которые будут представлены в следующей главе, мы сначала поговорим о программировании сокетов. Дело вот в чем: чтобы понимать код, вам необходимо знать некоторые фундаментальные концепции.

Программированием сокетов можно заниматься и локально, и на разных компьютерах. Как вы уже могли догадаться, в первом случае используются сокеты домена Unix. А вот взаимодействие разных систем требует создания и применения сетевых сокетов. Оба вида сокетов имеют примерно одинаковые API и принцип работы, поэтому вполне логично, что в следующей главе они рассматриваются бок о бок.

Прежде чем начинать задействовать сетевые сокеты, необходимо знать, как работают компьютерные сети. Мы поговорим об этом в следующем разделе. Существует множество понятий и концепций, с которыми нужно познакомиться, чтобы быть готовыми к написанию первого примера с использованием сокетов.

Компьютерные сети

Подход к объяснению сетевых технологий, выбранный мной в данном подразделе, отличается от того, который обычно можно встретить в других текстах. Моя задача — дать вам базовое понимание того, что происходит внутри компьютерной сети, особенно между двумя процессами. Мы попытаемся взглянуть на это с точки зрения программиста. И главными «действующими лицами» нашего обсуждения будут процессы, а не компьютеры. Из-за этого порядок следования материалов может сначала показаться необычным, но все вместе поможет вам получить представление о том, как происходит межпроцессное взаимодействие по сети.

Обратите внимание: данный подраздел не следует считать полноценным описанием компьютерных сетей. Очевидно, что его просто невозможно уместить в это количество страниц и всего один подраздел.

Физический уровень

Для начала забудем о процессах и сосредоточимся на компьютерах. Прежде чем продолжать, отмечу, что компьютер в сети может называться по-разному: устройство, хост, узел или даже система. Конечно, чтобы понять настоящее значение того или иного термина, нужен контекст.

Первый шаг на пути к созданию распределенной системы — наличие нескольких компьютеров, соединенных по сети, или, если быть более точным, компьютерной сети. Пока ограничимся двумя компьютерами, которые нужно соединить, — двумя физическими устройствами. Соединить их между собой, очевидно, следует с помощью некой физической промежуточной среды наподобие кабеля или беспроводного соединения.

Без этой среды (которая не обязательно должна быть видимой, как в случае с беспроводной сетью) соединение было бы невозможным. Такие физические соединения похожи на дороги между городами. Я буду придерживаться данной аналогии, поскольку она поможет мне очень подробно объяснить происходящее внутри компьютерной сети.

Любое аппаратное оборудование, необходимое для физического подключения двух устройств, относится к *физическому уровню*. Это первый и самый низкий уровень, который мы исследуем. Без него невозможно было бы передавать данные между двумя компьютерами и считать их соединенными. Все остальные уровни, находящиеся выше, являются программными и представляют собой набор различных стандартов, определяющих то, как должны передаваться данные.

Перейдем к следующему, канальному уровню.

Канальный уровень

Организация дорожного движения между городами требует не только дорог. То же самое относится и к физическому соединению между компьютерами. Чтобы пользоваться дорогами, нам нужны законы и правила, регулирующие транспортные средства, дорожные знаки, строительные материалы, бровки, скорость, разметку, направление движения и т. д. Похожие правила требуются и для прямого физического соединения между двумя компьютерами.

Если все аппаратные компоненты и устройства, необходимые для соединения разных компьютеров, принадлежат к физическому уровню, то обязательные нормы и протоколы, определяющие способ передачи данных, относятся к более высокому, *канальному уровню*.

В рамках правил, соблюдение которых обеспечивается канальными протоколами, сообщения разбиваются на фрагменты, называемые *кадрами*. Ситуация аналогична тому, как в нормах, регулирующих дорожное движение, описывается максимальная

длина автомобилей, которым позволено перемещаться по определенной дороге. Вы не можете вести по трассе автомобиль с прицепом длиной 1 км (если предположить, что это в принципе возможно). Вам нужно разделить его на более короткие сегменты или автомобили меньшей длины. Точно так же длинный фрагмент данных следует разбить на несколько кадров, которые будут перемещаться по сети независимо друг от друга.

Стоит упомянуть, что сеть может быть установлена между любыми двумя вычислительными устройствами, не обязательно компьютерами. Подобных устройств существует очень много. Промышленные сети имеют собственные стандарты касательно кабелей, разъемов, терминаторов и т. д., а также отдельный набор канальных протоколов и спецификаций.

Существует множество стандартов, описывающих такие канальные соединения, — например, как настольный компьютер можно подключить к промышленному устройству. Один из самых известных канальных протоколов, предназначенный для проводного соединения компьютеров, — *Ethernet*. Он описывает все правила и нормы относительно передачи данных по компьютерным сетям. Еще один широко распространенный протокол, определяющий работу беспроводных сетей, называется IEEE 802.11.

Сеть, состоящая из компьютеров (или любых других вычислительных машин/устройств одного типа), соединенных физически с помощью определенного канального протокола, называется *локальной вычислительной сетью* (local area network, LAN). Обратите внимание: любое устройство, которое подключается к LAN, должно обладать физическим компонентом под названием *«сетевой адаптер»* или *«контроллер сетевого интерфейса»* (Network Interface Controller, NIC). Например, у компьютера, который мы подключаем к сети Ethernet, должен иметься *Ethernet NIC*.

У компьютера может быть несколько сетевых адаптеров, каждый из которых подключен к отдельной локальной сети. Таким образом, компьютер с тремя NIC способен работать с тремя локальными сетями одновременно.

Кроме того, возможно, что все три сетевых адаптера используются для подключения к одной сети. Способ конфигурации NIC и то, как вы соединяетесь с различными сетями, следует продумать заранее и разработать подробный план.

Каждый NIC имеет уникальный адрес, который задается управляющим канальным протоколом. Этот адрес используется для передачи данных между узлами внутри LAN. Протоколы Ethernet и IEEE 802.11 назначают каждому совместимому сетевому адаптеру MAC-адрес (media access control — управление доступом к сети). Таким образом, адаптеру Ethernet или IEEE 802.11 Wi-Fi следует иметь уникальный MAC-адрес, иначе он не сможет подключиться к LAN. MAC-адреса не должны повторяться внутри одной локальной сети. В идеале им надлежит быть совершенно

уникальными и неизменяемыми. Однако в реальности это не так; вы даже можете установить MAC-адрес сетевого адаптера вручную.

Если подытожить все вышесказанное, то мы имеем стек из двух уровней: физического (снизу) и канального (сверху). Этого достаточно, чтобы соединить ряд компьютеров в одной локальной сети. Но это еще не все. Нам нужен еще один уровень поверх указанных двух, чтобы иметь возможность соединять компьютеры из разных локальных сетей с промежуточными LAN или без них.

Сетевой уровень

Мы уже знаем, что для соединения разных узлов в локальных сетях Ethernet используются MAC-адреса. Но что, если соединить нужно компьютеры из двух разных LAN, которые, к слову, могут быть несовместимы между собой?

Например, одна из них может быть проводной сетью Ethernet, а другая — *FDDI-сетью* (fiber distributed data interface — волоконно-оптический распределенный интерфейс передачи данных), которая на физическом уровне в основном использует оптическое волокно. Еще одним примером могут быть промышленные устройства, подключенные к сети *IE* (Industrial Ethernet — промышленный Ethernet) и взаимодействующие с компьютерами операторов, которые обычно находятся в локальной сети Ethernet. Эти и многие другие примеры показывают: поверх упомянутых выше протоколов нужен еще один уровень для соединения узлов из разных LAN. Стоит отметить: данный уровень нужен, даже если локальные сети совместимы между собой. Это особенно важно для передачи данных из одной сети (совместимой или неоднородной) в другую через ряд промежуточных LAN. Чуть ниже будут даны более подробные объяснения.

Сетевой уровень работает с *пакетами* точно так же, как канальный с кадрами. Длинные сообщения разбиваются на более мелкие части, пакеты. Кадры и пакеты представляют собой две разные концепции на двух разных уровнях, но для простоты мы будем считать, что это одно и то же, и до конца данной главы будем называть и то и другое *пакетами*.

Ключевое различие между этими понятиями, о котором необходимо знать, состоит в том, что пакеты инкапсулируются (содержатся) в кадрах. Мы не станем углубляться в данную тему, но в Интернете можно найти много материала, посвященного разным аспектам обеих концепций.

Сетевой протокол заполняет пробел между разными локальными сетями, соединяя их между собой. У каждой локальной сети могут быть свои отдельные стандарты и протоколы физического и канального уровней, но все они имеют общий управляющий сетевой протокол. В противном случае неоднородные (несовместимые) LAN не могли бы соединяться друг с другом. Самый известный сетевой протокол на текущий момент — *IP* (Internet Protocol — интернет-протокол). Он активно

используется в крупных компьютерных сетях, которые обычно состоят из более мелких LAN на основе Ethernet или Wi-Fi. У IP есть две версии с разной длиной адресов: IPv4 и IPv6.

Но как соединить два компьютера из двух разных LAN? Ответ кроется в механизме *маршрутизации*. Получение данных из внешней локальной сети требует наличия узла-*маршрутизатора*. Представьте, что мы хотим соединить две разные сети: LAN1 и LAN2. Маршрутизатор — обычный узел, который находится в обеих сетях благодаря наличию двух сетевых адаптеров. Один принадлежит LAN1, а другой — LAN2. Затем специальный алгоритм маршрутизации определяет, какие пакеты следует передавать между сетями и как это делать.

Благодаря механизмам маршрутизации через узел-маршрутизатор могут проходить данные из разных сетей и в разных направлениях. Для этого в каждой локальной сети должен иметься маршрутизатор. Таким образом, данные, отправленные компьютеру, который находится в другой географической зоне, на пути к адресату могут пройти через десятки маршрутизаторов. Я не стану углубляться в данную тему, но в Интернете можно найти огромное количество информации об этом механизме.

Существует утилита под названием traceroute, которая позволяет просматривать маршруты между вашим компьютером и адресатом.

Итак, два компьютера из двух разных LAN могут соединяться друг с другом напрямую или через промежуточные локальные сети. Любые более специализированные соединения должны устанавливаться поверх этого уровня. Следовательно, в основе любого взаимодействия двух программ, находящихся на разных узлах, лежит трехуровневый стек из трех протоколов: физического, канального и сетевого. Но что именно мы имеем в виду, когда говорим о соединении между двумя компьютерами?

Утверждение о том, что два узла соединены между собой, звучит слегка расплывчато, по крайней мере для программистов. Если быть более точным, то соединяются и обмениваются данными операционные системы, установленные на этих узлах. Способность подключаться к LAN и взаимодействовать с другими узлами в той же или другой локальной сети — неотъемлемая часть большинства современных ОС. Все операционные системы на основе Unix, которым в данной книге уделяется основное внимание, поддерживают работу с сетью и могут быть установлены на узлы, участвующие в сетевом взаимодействии.

Сетевые возможности присутствуют в Linux, Microsoft Windows и почти любой другой ОС. И в самом деле, вряд ли операционная система сможет выжить без доступа к сети. Обратите внимание: управлением сетевыми соединениями занимается ядро (или, точнее, один из его компонентов), поэтому более правильно говорить, что сетевые возможности предоставляются ядром.

Поскольку ядро обеспечивает работу с сетью, эта возможность доступна любому процессу в пользовательском пространстве, который благодаря этому может соединяться с другими процессами, размещенными на разных сетевых узлах. Вам как программисту не нужно беспокоиться о различных уровнях (физическом, канальном и сетевом), с которыми имеет дело ядро; уровни, имеющие отношение к вашему коду, находятся выше, и вы можете сосредоточиться именно на них.

У каждого узла в IP-сети есть IP-адрес. Как уже было сказано ранее, существует два вида IP-адресов: *IP версии 4 (IPv4)* и *IP версии 6 (IPv6)*. В IPv4 адреса состоят из четырех сегментов, каждый из которых может содержать числовое значение от 0 до 255. То есть IPv4-адреса находятся в диапазоне от `0.0.0.0` до `255.255.255.255`. Как видите, для хранения адресов этого формата достаточно 4 байт (или 32 бит). Если взять IPv6-адреса, то они могут достигать 16 байт (128 бит). Кроме того, IP-адреса бывают приватными и публичными, но подробности этого выходят далеко за рамки темы, обсуждаемой в данной главе. Нам достаточно знать о том, что каждый узел в IP-сети имеет уникальный IP-адрес.

Если вернуться к предыдущему разделу, то в отдельно взятой локальной сети каждый узел имеет как адрес канального уровня, так и IP-адрес, но для соединения с узлами мы будем использовать только последний. Например, в сети Ethernet у узла есть MAC-адрес, который протоколы канального уровня задействуют для передачи данных внутри LAN, и IP-адрес, с помощью которого программы, размещенные на разных узлах, устанавливают сетевые соединения как в рамках одной локальной сети, так и с рядом других сетей.

Главная обязанность сетевого уровня состоит в соединении двух или больше локальных сетей. В итоге множество отдельных LAN можно объединить в одну громадную сеть. На самом деле такая сеть уже существует; мы называем ее Интернетом.

Как и в любой другой сети, у узла, подключенного к Интернету, должен быть IP-адрес. Но главное отличие между узлом с выходом в Интернет и без него состоит в том, что первый должен иметь публичный IP-адрес, а второй может обойтись приватным.

Возьмем, к примеру, вашу домашнюю сеть, подключенную к Интернету. Внешний узел, размещенный в Интернете, не может соединиться с вашим ноутбуком, поскольку у того есть только приватный IP-адрес. То есть ноутбук доступен в вашей домашней сети, но не в Интернете. Следовательно, если вы хотите открыть доступ к своему программному обеспечению снаружи, то его следует разместить на компьютере с публичным IP-адресом.

Сетевым IP-технологиям посвящено огромное количество информации, и мы не станем рассматривать ее в полном объеме, но вы, как программист, должны знать, чем приватные адреса отличаются от публичных.

Программист не отвечает за управление сетевыми соединениями между узлами, но должен быть способен обнаруживать неисправности в сети. Это очень важно, поскольку благодаря данным навыкам вы будете знать, что является виной ошибки или некорректного поведения — ваш код или инфраструктура (сеть). Вот почему мы должны затронуть некоторые дополнительные концепции и инструменты.

Простейший инструмент, который позволяет убедиться в том, что два хоста (узла) в одной или разных локальных сетях могут обмениваться данными или «видеть» друг друга, — утилита *ping*. Вы уже, наверное, с ней знакомы. Она отправляет ICMP-пакеты (Internet Control Message Protocol — протокол межсетевых управляющих сообщений); если они возвращаются обратно, то это значит, другой компьютер работает, подключен к сети и отвечает на запросы.

ICMP — еще один протокол сетевого уровня, который в основном используется для мониторинга и администрирования сетей на основе IP в ситуациях, когда возникают проблемы с соединением или с качеством обслуживания.

Представьте, что хотите узнать, видит ли ваш компьютер публичный IP-адрес `8.8.8.8` (если он подключен к Интернету, то должен видеть). Следующие команды помогут вам проверить ваше соединение (терминал 19.6).

Терминал 19.6. Использование утилиты ping для проверки подключения к Интернету

```
$ ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8): 56 data bytes
64 bytes from 8.8.8.8: icmp_seq=0 ttl=123 time=12.190 ms
64 bytes from 8.8.8.8: icmp_seq=1 ttl=123 time=25.254 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=123 time=15.478 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=123 time=22.287 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=123 time=21.029 ms
64 bytes from 8.8.8.8: icmp_seq=5 ttl=123 time=28.806 ms
64 bytes from 8.8.8.8: icmp_seq=6 ttl=123 time=20.324 ms
^C
--- 8.8.8.8 ping statistics ---
7 packets transmitted, 7 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 12.190/20.767/28.806/5.194 ms
$
```

Вывод показывает: было отправлено семь ICMP-пакетов, и ни один из них не был потерян во время передачи. Это значит, операционная система, находящаяся за IP-адресом `8.8.8.8`, работает и отвечает на запросы.

Публичный IP-адрес 8.8.8.8 принадлежит публичному DNS-сервису Google. Более подробно об этом можно почитать на странице https://ru.wikipedia.org/wiki/Google_Public_DNS.

Мы увидели, как два компьютера могут соединиться по сети. Теперь вплотную подошли к моменту, когда один процесс может подключиться к другому и передать данные по ряду локальных сетей. Для этого поверх сетевого уровня нужен еще один. Именно с этого начинается сетевое программирование.

Транспортный уровень

Итак, мы уже знаем, что два компьютера можно соединить с помощью стека из трех уровней: физического, канального и сетевого. Межпроцессное взаимодействие требует, чтобы соединение и общение происходило на уровне процессов. Однако на каждом из соединенных компьютеров подобных процессов может быть много, и мы должны иметь возможность установить соединение между любыми двумя процессами, размещенными на разных компьютерах. Поэтому соединение, действующее лишь на сетевом уровне, будет слишком общим для поддержки взаимодействия нескольких отдельных процессов.

Вот почему поверх сетевого уровня необходим еще один. *Транспортный уровень* закрывает эту потребность. Если компьютеры общаются на сетевом уровне, то запущенные на них процессы могут соединяться на транспортном уровне, который функционирует поверх сетевого. Транспортный уровень, как и любой другой, имеет собственные уникальные идентификаторы, роль которых в данном случае играют *порты*. Ниже мы обсудим это более подробно, но сначала необходимо объяснить модель *«слушатель — соединитель»*, которая позволяет двум сторонам взаимодействовать по каналу. Далее для описания этой модели будет использоваться сравнение между компьютерными и телефонными сетями.

Модель «слушатель — соединитель» в контексте телефонных сетей

Лучший пример, с которого только можно начать, — *телефонные сети общего пользования* (ТСОП). На первый взгляд они могут показаться не очень похожими на компьютерные, но у них есть ряд сходств, которые позволят нам должным образом объяснить принцип работы транспортного протокола.

В нашей аналогии люди, пользующиеся телефоном, играют ту же роль, что и процессы в компьютерной сети. Следовательно, телефонный звонок — эквивалент *транспортного соединения*. Абоненты могут звонить, только если предусмотрена вся необходимая инфраструктура. Это подобно сетевой инфраструктуре, без которой процессы не могли бы взаимодействовать.

Предположим, вся необходимая инфраструктура уже готова и работает без каких-либо проблем. Мы хотим, чтобы два субъекта, находящиеся в данных системах, создали канал и передали данные. Это аналогично двум абонентам ТСОП или двум процессам, которые размещены на разных узлах компьютерной сети.

Использование ТСОП требует наличия телефонного аппарата. Точно так же компьютерный узел должен иметь сетевой адаптер. Помимо этого, существуют разные уровни, состоящие из различных протоколов. Эти уровни, на которые опирается инфраструктура, делают возможным создание транспортного канала.

Вернемся к ТСОП. Один из телефонных аппаратов ждет звонка. Пусть это будет *слушающая* сторона. Отмечу, что телефон, подключенный к ТСОП, всегда ожидает сигнала вызова и звонит при его получении.

Теперь поговорим о другой стороне, которая инициирует звонок. Это эквивалент создания транспортного канала. Здесь тоже используется телефонный аппарат. Слушатель доступен по телефонному номеру, который можно считать его адресом. Чтобы сделать звонок, сторона *соединителя* должна знать данный номер. Таким образом, соединитель набирает телефонный номер слушателя, и инфраструктура информирует последнего о том, что ему поступил входящий звонок.

Когда сторона слушателя берет трубку, она принимает входящее соединение, в результате чего между слушателем и соединителем создается канал. С этого момента люди, находящиеся по разные стороны соединения, могут общаться друг с другом по каналу ТСОП. Обратите внимание: если стороны разговаривают на разных языках, то общение не сможет продолжаться и один из абонентов повесит трубку. В результате канал будет уничтожен.

Транспортное взаимодействие с соединениями и без

С помощью аналогии, представленной выше, я попытался объяснить, как происходит взаимодействие в компьютерной сети. Однако на самом деле это лишь один вид взаимодействия: *с установлением соединения*. Мы рассмотрим другой подход, в котором соединения не используются. Но сначала более подробно поговорим о первом способе.

В рамках взаимодействия, ориентированного на соединения, для соединителя создается отдельный канал. Следовательно, если один слушатель общается с тремя соединителями, то для этого требуется три разных канала. Неважно, насколько велико передаваемое сообщение, оно дойдет до получателя в корректном виде и без какой-либо потери данных внутри канала. Если одному адресату послать несколько сообщений, то порядок их отправки будет сохранен и принимающий процесс не заметит никаких неполадок в работе инфраструктуры.

Как уже объяснялось в предыдущих разделах, любое сообщение при передаче по компьютерной сети всегда разбивается на мелкие фрагменты, которые называют пакетами. В случае ориентации взаимодействия на соединения ни одна из сторон (ни слушатель, ни соединитель) не будет иметь никакого представления о *комму-*

тации отдельных пакетов. Даже если пакеты будут получены в другом порядке, принимающая операционная система их отсортирует и воссоздаст сообщение в его исходном виде и процесс-получатель ничего не заметит.

Более того, если один из пакетов потеряется во время передачи, то операционная система получателя запросит его снова, чтобы восстановить все сообщение целиком. Например, *TCP* (Transport Control Protocol — протокол управления передачей) — это протокол транспортного уровня, который ведет себя в точности так, как описано выше. Поэтому TCP-каналы ориентированы на соединения.

Взаимодействие можно проводить и без соединений. Наличие соединения гарантирует два фактора: *доставку* отдельных пакетов и их *упорядоченность*. Такие протоколы, как TCP, обеспечивают одновременное выполнение этих двух условий. А вот транспортные протоколы, которые не устанавливают соединение, их не гарантируют.

Иными словами, вы не можете гарантировать, что будет доставлен каждый отдельный пакет в сообщении или что все пакеты дойдут в правильном порядке. И то и другое возможно по отдельности и вместе! Например, *UDP* (User Datagram Protocol — протокол пользовательских датаграмм) не гарантирует доставку пакетов и их упорядоченность. Стоит отметить: корректность содержимого отдельно взятого пакета обеспечивается протоколом на сетевом и канальном уровнях.

Теперь пришло время обсудить термины, которые часто используются в сетевом программировании. *Поток данных*[1] — последовательность байтов, передающаяся по каналу, ориентированному на соединения. Это значит, что взаимодействие, в котором отсутствуют соединения, фактически не поддерживает потоки данных. Для единицы данных, передающихся по каналу без соединения, предусмотрен специальный термин — *датаграмма*. Это фрагмент данных, который можно доставить целиком в условиях отсутствия соединения. Если фрагмент данных превышает максимальный размер датаграммы, то нам не под силу гарантировать его доставку и итоговая последовательность может оказаться неверной. Концепция датаграммы существует на транспортном уровне и аналогична концепции пакета, принадлежащей сетевому уровню.

Например, протокол UDP гарантирует корректную доставку отдельно взятой датаграммы (пакета), но о соотношении двух смежных датаграмм (пакетов) ничего сказать нельзя. Вы должны принять тот факт, что целостность данных в UDP существует только на уровне датаграммы. В TCP все не так. Данный протокол гарантирует доставку и упорядоченность отправленных пакетов, поэтому мы можем рассматривать передачу потока байтов между двумя процессами.

---

[1] Не путать с потоком выполнения. — *Примеч. ред.*

Последовательность инициализации транспортного протокола

Здесь мы поговорим о шагах, которые предпринимает каждый процесс в целях установления транспортного взаимодействия. Последовательность этих шагов зависит от наличия или отсутствия соединения, и потому оба варианта будут рассмотрены в отдельных фрагментах текста. Отмечу, что разница видна только при инициализации канала и после этого обе стороны используют примерно одинаковые API для чтения и записи в созданный канал.

Процесс-слушатель всегда *привязывается* к конечной точке (обычно это сочетание IP-адреса и порта), а процесс-соединитель к ней *подключается*. Это происходит как при наличии соединения, так и при его отсутствии.

Обратите внимание: в описанных ниже последовательностях предполагается, что между компьютерами, на которых размещены процесс-слушатель и процесс-соединитель, установлена IP-сеть.

Последовательности инициализации при отсутствии соединения

Чтобы создать канал связи без соединения, процесс-слушатель делает следующее.

1. Привязывается к порту на одном из имеющихся сетевых интерфейсов (или даже на каждом из них). Это значит, что процесс-слушатель просит свою операционную систему перенаправлять ему входящий трафик через данный порт. Порт представляет собой обычное число между 0 и 65 535 (2 байта); необходимо выбрать номер порта, который еще не привязан к другому процессу-слушателю, иначе получится ошибка. Если мы привязываемся к порту в определенном сетевом интерфейсе, то операционная система будет перенаправлять процессу все пакеты, которые проходят по данному интерфейсу и предназначены для этого порта.
2. Процесс ждет и считывает сообщения, которые становятся доступными в созданном канале, и отвечает на них, выполняя запись в тот же канал.

А процесс-соединитель выполняет следующие действия.

1. Он должен знать IP-адрес и номер порта, принадлежащие процессу-слушателю. Поэтому для соединения с этим процессом он предоставляет своей системе соответствующую информацию. Если нужный процесс не прослушивает указанный порт или если IP-адрес указывает не на тот компьютер, то соединение не удается установить.
2. Если соединение установлено успешно, то процесс-соединитель может записывать в канал и читать из него почти таким же способом (то есть используя тот же API), что и процесс-слушатель.

Помимо выполнения описанных выше шагов, процесс-слушатель и процесс-соединитель должны использовать один и тот же транспортный протокол, иначе их операционные системы не смогут прочитать и понять передаваемые сообщения.

Последовательности инициализации при использовании соединения

При использовании подхода, ориентированного на соединения, процесс-слушатель инициализируется, выполняя такую последовательность.

1. Он привязывается к порту, как и в предыдущем сценарии, в котором не было соединений. Порт ничем не отличается от описанного в прошлом разделе и подвержен тем же ограничениям.
2. Далее процесс слушателя задает размер очереди *отставания*. Она содержит ожидающие соединения, еще не принятые процессом. При взаимодействии, ориентированном на соединения, сторона слушателя получает возможность отправлять данные только после приема входящих соединений. Настроив очередь отставания, процесс-слушатель переходит в *режим прослушивания*.
3. Теперь процесс-слушатель начинает *принимать* входящие соединения. Это обязательный этап создания транспортного канала. Только приняв входящее соединение, слушатель может передать данные. Если процесс-соединитель инициирует соединение со слушателем, но тот не может его принять, то данное соединение будет оставаться в очереди отставания, пока не будет принято или пока не истечет *время ожидания*. Это может произойти, если процесс-слушатель слишком занят обработкой других соединений. В таком случае входящие соединения будут накапливаться в очереди отставания, и когда та заполнится, соединения начнут немедленно отклоняться операционной системой.

Процесс-соединитель проходит через последовательность шагов, очень похожую на ту, которую мы видели выше, при рассмотрении взаимодействия без использования соединений. Соединитель подключается к определенной конечной точке, предоставляя IP-адрес и порт, и, после того как его примет слушатель, может задействовать тот же API для чтения и записи в канал.

Поскольку созданный канал ориентирован на соединения, между слушателем и соединителем устанавливается отдельное соединение; это позволяет им обмениваться потоком байтов неограниченной длины. Следовательно, оба процесса могут передавать огромные объемы данных, корректность которых гарантируется транспортным и сетевым протоколами.

В заключение напомню: процесс-слушатель (независимо от того, ориентирован канал на соединения или нет) должен привязаться к конечной точке. Если говорить конкретно о UDP и TCP, то эта конечная точка состоит из IP-адреса и номера порта.

Прикладной уровень

Транспортный канал соединяет два процесса, находящихся на разных его концах, и позволяет им общаться друг с другом. Под общением мы понимаем передачу последовательности байтов, которую могут понять обе стороны. Как уже объяснялось

в начальных разделах данной главы, для этого требуется коммуникационный протокол. Поскольку он находится на *прикладном уровне* и используется процессами (или приложениями, которые выполняются в виде процессов), этот протокол называется *прикладным*.

На канальном, сетевом и транспортном уровнях существует не так уж много протоколов, и большинство из них хорошо известны. Протоколов прикладного уровня намного больше. Это опять же аналогично ситуации в телекоммуникационных сетях. Для телефонных сетей существует всего несколько стандартов, однако люди общаются между собой на множестве языков, которые могут сильно разниться. В компьютерных сетях каждому приложению, запущенному в виде процесса, нужен прикладной протокол, позволяющий взаимодействовать с другими процессами.

Следовательно, программист использует либо общеизвестный прикладной протокол, такой как HTTP или FTP, либо спроектированный и реализованный внутри своей команды.

Итак, мы обсудили пять уровней: физический, канальный, сетевой, транспортный и прикладной. Теперь можно объединить их и использовать в качестве отправной точки для разработки и развертывания компьютерных сетей. В следующем пункте текста речь пойдет о наборе интернет-протоколов.

Набор интернет-протоколов

Ежедневно мы имеем дело с широко распространенной сетевой моделью *IPS* (Internet Protocol Suite — набор интернет-протоколов). Она в основном используется в Интернете, и, поскольку доступ к нему поддерживают практически все компьютеры, мы можем наблюдать ее повсеместное присутствие. Тем не менее IPS не является официальным стандартом ISO. Стандартной моделью для компьютерных сетей считается *OSI* (Open System Interconnections — модель взаимодействия открытых систем), которая носит скорее теоретический характер и почти никогда не развертывается и не используется в публичных сетях. Ниже перечислены уровни, из которых состоит IPS, а также известные протоколы, которые применяются на каждом из них:

- физический уровень;
- канальный уровень — Ethernet, IEEE 802.11 Wi-Fi;
- сетевой уровень — IPv4, IPv6 и ICMP;
- транспортный уровень — TCP, UDP;
- прикладной уровень — многочисленные протоколы наподобие HTTP, FTP, DNS, DHCP и многие другие.

Как видите, данная модель напрямую связана с теми уровнями, которые мы обсудили в текущей главе, только сетевой уровень иногда называют интернет-уровнем.

Причина в том, что в IPS распространенными сетевыми протоколами являются только IPv4 и IPv6. В остальном к IPS применимо все представленное нами ранее. Это основная модель, с которой мы будем иметь дело в данной книге и в реальных рабочих условиях.

Мы уже знаем, как работают компьютерные сети, и готовы к тому, чтобы познакомиться с *программированием сокетов*. На оставшихся страницах этой главы и в следующей вы увидите, что между уже знакомыми нам понятиями транспортного уровня и концепциями программирования сокетов существует глубокая связь.

Что такое программирование сокетов

Итак, мы рассмотрели модель IPS и различные уровни сети. Теперь вам будет намного легче понять, что такое программирование сокетов. Прежде чем объяснять технические аспекты, следует отметить: программирование сокетов — метод межпроцессного взаимодействия, который позволяет соединить два процесса, размещенных на одном или разных узлах с установленным между ними сетевым соединением. Если мы говорим о сценарии с двумя узлами, то они должны быть подключены к рабочей сети. Уже сам этот факт делает программирование сокетов неотрывным от компьютерных сетей и всего того, что объяснялось ранее.

Строго говоря, программирование сокетов происходит в основном на транспортном уровне. Как уже отмечалось, транспортный уровень отвечает за соединение двух процессов поверх имеющегося сетевого уровня. Следовательно, транспортный уровень — ключевой элемент установления контекста для программирования сокетов. Это, в сущности, причина, по которой программист должен иметь хорошее представление о транспортном уровне и его различных протоколах. Некоторые ошибки, связанные с программированием сокетов, возникают в транспортном канале, лежащем в основе взаимодействия.

В этом виде взаимодействия сокеты выступают главным средством прокладывания транспортного канала. Несмотря на то, о чем мы говорили выше, программирование распространяется не только на транспортный *(процесс — процесс)*, но и на сетевой уровень *(хост — хост)*. Это значит, сокеты могут оперировать как на сетевом, так и на транспортном уровне. Учитывая сказанное, следует сказать, что в этой и следующей главах мы в основном будем рассматривать и использовать транспортные сокеты.

Что такое сокет

Как уже было сказано, программирование сокетов на самом деле происходит на транспортном уровне. Все, что находится выше, всего лишь делает взаимодействие более узкоспециализированным; однако сам канал, лежащий в основе программирования сокетов, создается на транспортном уровне.

Я также упоминал, что интернет-соединение (сетевое соединение), в рамках которого создается транспортный канал, на самом деле соединяет операционные системы или, точнее, их ядра. Следовательно, в ядре должна существовать абстракция, напоминающая соединение. Более того, одно ядро может инициировать и принимать много соединений, поскольку в его операционной системе может выполняться несколько процессов, которым нужен доступ к сети.

Роль этой абстракции играет *сокет*. Для любого соединения в системе, уже существующего или устанавливаемого, выделяется сокет, который его идентифицирует. Для отдельно взятого соединения между двумя процессами нужно по одному сокету на каждом конце. Ранее уже объяснялось, что один из этих сокетов принадлежит стороне соединителя, а другой — стороне слушателя. API, который позволяет нам определять сокеты и работать с ними, описывается *библиотекой сокетов*, предоставляемой операционной системой.

Поскольку нас в основном интересуют POSIX-системы, мы можем ожидать, что в составе API POSIX такая библиотека есть, и это действительно так. В оставшейся части этой главы мы будем обсуждать *библиотеку сокетов POSIX*. Вы увидите, как с ее помощью можно устанавливать соединения между двумя процессами.

Библиотека сокетов POSIX

У каждого объекта сокета есть три атрибута: *домен*, *тип* и *протокол*. Они подробно описаны в справочных страницах операционной системы, и потому здесь мы поговорим о значениях, обычно присваиваемых этим атрибутам. Начнем с домена, иначе называемого *семейством адресов* (address family, AF) или *семейством протоколов* (protocol family, PF). Ниже перечислено несколько значений, которые можно часто встретить. Отмечу, что эти семейства адресов поддерживаются транспортными каналами независимо от наличия соединения.

- Сокеты `AF_LOCAL` или `AF_UNIX` — локальные; работают, только если процессы соединителя и слушателя находятся на одном компьютере.
- Сокеты `AF_INET` позволяют двум процессам соединяться друг с другом по IPv4.
- Сокеты `AF_INET6` дают возможность двум процессам соединяться друг с другом по IPv6.

В некоторых POSIX-системах константы, которые используются в качестве домена, могут иметь префикс PF_ вместо AF_. Обычно эти константы имеют одни и те же значения, что делает их взаимозаменяемыми.

В следующей главе будет продемонстрировано использование доменов `AF_UNIX` и `AF_INET`, но вы также можете легко найти примеры с доменом `AF_INET6`. Кроме

того, в ряде операционных систем есть уникальные семейства адресов, которые не встречаются в других ОС.

Ниже перечислены самые распространенные значения, которые используются в качестве типа сокета.

- Тип сокетов `SOCK_STREAM` представляет транспортный канал, ориентированный на соединения, с гарантией доставки, корректности и упорядоченности отправляемых данных. На это также указывает термин `STREAM` (поток), который мы рассматривали выше. Обратите внимание: на данном этапе мы не можем сказать, используется ли в качестве транспортного протокола TCP, поскольку здесь также могут применяться локальные сокеты, принадлежащие к семейству адресов `AF_UNIX`.
- Тип сокетов `SOCK_DGRAM` представляет транспортный канал без поддержки соединений. Как уже объяснялось выше, термин «датаграмма» обозначает последовательность байтов, которые нельзя считать потоком. Вместо этого их следует рассматривать в качестве отдельных фрагментов данных, именуемых датаграммами. На более техническом уровне датаграмма представляет собой пакет данных, передаваемый по сети.
- Сокет типа `SOCK_RAW` может представлять каналы с соединениями и без. Основное отличие между `SOCK_RAW` и `SOCK_DGRAM` или `SOCK_STREAM` в том, что ядро на самом деле знает о том, какой транспортный протокол используется внутри (UDP или TCP). Это позволяет ему проанализировать пакет и извлечь из него заголовок и содержимое. Однако с сокетами типа `SOCK_RAW` ядро этого не делает, и потому программа, которая открыла сокет, сама должна его прочитать и извлечь из него нужные элементы.

  Иными словами, при использовании `SOCK_RAW` пакеты доставляются непосредственно программе, которая должна понимать их структуру и уметь извлекать их содержимое. Обратите внимание: если канал является потоковым (ориентированным на соединения), то восстановлением потерянных пакетов и их упорядочением должно заниматься не ядро, а сама программа. Из этого следует, что при выборе протокола TCP ядро выполняет восстановление и упорядочение пакетов за вас.

Третий атрибут определяет протокол, который должен использоваться для объекта сокета. Этот атрибут может быть выбран операционной системой в момент создания сокета, поскольку в большинстве случаев его можно определить по сочетанию семейства адресов и типа. Если же существует несколько потенциальных протоколов, то данный атрибут необходимо установить вручную.

Программирование сокетов предлагает решения для межпроцессного взаимодействия и внутри одной системы, и между разными компьютерами. То есть два процесса, которые мы соединяем, могут вполне находиться как на разных хостах

и даже в разных локальных сетях, так и в одной и той же системе; в первом случае используются сетевые сокеты, а во втором — сокеты домена Unix.

В заключение стоит отметить, что соединения на основе сокетов являются двунаправленными и полнодуплексными. Это значит, обе стороны могут читать из канала и записывать в него, не мешая друг другу. Это полезное свойство, которое требуется в большинстве ситуаций, связанных с межпроцессным взаимодействием.

Итак, мы познакомились с концепцией сокетов. Теперь заново рассмотрим последовательности действий, которые выполняют процессы слушателя и соединителя. Однако на сей раз я уделю больше внимания подробностям и покажу, как эти действия можно выполнить с помощью сокетов.

Повторное рассмотрение последовательности инициализации слушателя и соединителя

Как уже упоминалось ранее, в почти любом соединении в компьютерной сети одна из сторон всегда ожидает входящих соединений, а другая пытается к ней подключиться. Мы также обсудили пример с телефонной сетью и увидели, как телефон используется для ожидания входящих звонков и как с его помощью можно соединяться с другими ожидающими устройствами. Аналогичная ситуация существует и в программировании сокетов. Здесь мы исследуем действия, которые процессы должны выполнить на двух противоположных концах канала, чтобы успешно установить транспортное соединение.

В следующих подразделах мы подробно поговорим о создании сокета и различных операциях, которые должны выполнить оба процесса, желающие участвовать в соединении. Последовательность шагов, описанная ниже, не зависит от конкретной инфраструктуры. Это возможно благодаря тому, что программирование сокетов инкапсулирует в себе разного рода транспортные соединения.

Как вы помните, ранее мы обсуждали последовательность действий слушателя и соединителя отдельно для взаимодействия с соединениями и без. Здесь будет использоваться тот же подход. Начнем с потоковой последовательности (ориентированной на соединения) на стороне слушателя.

Потоковая последовательность на стороне слушателя. Процесс, который хочет принимать новые потоковые соединения (слушатель), должен выполнить следующие шаги. Вы уже знакомы с этапами привязки, прослушивания и приема по предыдущим разделам, здесь мы обсудим их с точки зрения программирования сокетов. Обратите внимание: соответствующие возможности реализованы в ядре, а процесс, который хочет перейти в режим прослушивания, должен всего лишь вызвать подходящие функции из библиотеки сокетов.

1. Процесс должен создать объект сокета, используя функцию `socket`. Этот объект обычно называют *слушающим сокетом*. Он представляет весь процесс слушателя и служит для приема входящих соединений. Аргументы, передаваемые функции `socket`, могут разниться в зависимости от вида канала. В качестве семейства адресов можно указать либо `AF_UNIX`, либо `AF_INET`, но тип сокета может быть только `SOCK_STREAM`, поскольку мы имеем дело с потоковым каналом. Протокол сокета может определить операционная система. Например, если вы укажете для своего объекта атрибуты `AF_INET` и `SOCK_STREAM`, то в качестве протокола по умолчанию будет выбран TCP.
2. Теперь сокет нужно привязать к *конечной точке*, доступной процессу соединителя. Для этого предусмотрена функция `bind`. Свойства выбранной конечной точки во многом зависят от заданного семейства адресов. Например, в случае с интернет-каналом она представляет собой сочетание IP-адреса и порта. Если мы имеем дело с сокетом домена Unix, то конечная точка будет иметь вид пути к *файлу сокета*, размещенному в файловой системе.
3. Сокет должен быть сконфигурирован для прослушивания. Здесь используется функция `listen`. Как уже объяснялось ранее, она просто создает очередь отставания для слушающего сокета. Эта очередь содержит список ожидающих соединений, которые еще не были приняты процессом слушателя. Ядро хранит новые входящие соединения в соответствующей очереди отставания, пока слушатель не освободится и не начнет их принимать. Как только очередь заполнится, ядро будет отклонять любые последующие соединения. Если сделать очередь слишком короткой, то в периоды загруженности слушателя многие входящие соединения могут быть отклонены; если же очередь слишком длинная, то вы получите кучу сообщений, время ожидания которых в конечном счете истечет, что сделает их недействительными. Размер очереди отставания следует выбирать с учетом поведения слушающей программы.
4. После настройки очереди отставания можно приступать к приему входящих соединений. Для каждого входящего соединения должна быть вызвана функция `accept`. В связи с этим вызов данной функции зачастую помещают в бесконечный цикл. Каждый раз, когда процесс слушателя перестает принимать новые соединения, запросы соединителей направляются в очередь отставания; как только очередь заполняется, новые запросы начинают отклоняться. Отмечу, что каждый вызов функции `accept` берет следующее соединение, ожидающее в очереди сокета. Если очередь пустая, а слушающий сокет сделан блокирующим, то любой вызов `accept` будет блокироваться до тех пор, пока не появится новое соединение.

Обратите внимание: функция `accept` возвращает новый объект сокета. Это значит, что ядро выделяет новый уникальный сокет для каждого принятого соединения. Иными словами, процесс слушателя, принявший 100 клиентов, использует как минимум 101 сокет: 1 для прослушивания и 100 для входящих соединения. Сокет,

возвращаемый функцией `accept`, следует применять для дальнейшего взаимодействия с клиентом, находящимся на противоположном конце канала.

Как видите, данная последовательность вызовов характерна для всех типов потокового (ориентированного на соединения) межпроцессного взаимодействия на основе сокетов. В следующей главе мы покажем реальные примеры того, как эти шаги реализуются на языке C. Ниже описана потоковая последовательность на стороне соединителя.

Потоковая последовательность на стороне соединителя. Когда процесс соединителя хочет подключиться к процессу слушателя, который уже находится в режиме прослушивания, он должен выполнить ту же последовательность действий. Обратите внимание: если слушатель не находится в режиме прослушивания, то его ядро отклонит соединение.

1. Процесс соединителя должен создать сокет, вызвав функцию `socket`. Этот сокет будет использоваться для подключения к процессу на противоположной стороне. По своим характеристикам он должен быть похож на слушающий сокет или по крайней мере быть с ним совместимым, иначе установить соединение не получится. Следовательно, обоим сокетам следует иметь одно и то же семейство адресов, а тип, как и прежде, должен быть `SOCK_STREAM`.
2. Затем нужно вызвать функцию `connect` и передать ей аргументы, которые однозначно идентифицируют конечную точку слушателя. Соединитель должен иметь возможность обратиться к этой конечной точке, а слушателю следует сделать ее доступной. Успешное выполнение `connect` говорит о том, что соединение было принято нужным нам процессом. Но перед этим соединение может какое-то время находиться в очереди отставания слушателя. Если по какой-то причине конечная точка недоступна, то соединение не будет установлено и процесс соединителя получит ошибку.

Функция `connect`, как и функция `accept` на стороне соединителя, возвращает объект сокета. Этот сокет служит идентификатором соединения, и его нужно использовать для дальнейшего взаимодействия с процессом слушателя. В следующей главе я покажу рассмотренные нами последовательности шагов на примере программы-калькулятора.

Датаграммная последовательность на стороне слушателя. При инициализации датаграммного процесса слушателя выполняются следующие шаги.

1. Датаграммный слушатель, как и потоковый, создает объект сокета, вызывая функцию `socket`. Однако на этот раз в качестве типа сокета следует указать `SOCK_DGRAM`.
2. После создания слушающего сокета процесс слушателя должен привязать его к конечной точке. Конечная точка и присущие ей ограничения очень похожи на те, которые используются на стороне потокового соединителя. Отмечу, что

у датаграммного слушающего сокета нет режима прослушивания и этапа приема, поскольку лежащий в его основе канал не поддерживает соединения, и потому мы не можем выделить отдельный сеанс для каждого входящего запроса.

Как уже упоминалось ранее, у датаграммного серверного сокета нет режима прослушивания и этапа приема. Для чтения из процесса соединителя и записи в него датаграммные слушатели должны использовать функции `recvfrom` и `sendto`. Чтение по-прежнему можно выполнять с помощью функции `read`, однако для записи одного вызова `write` будет недостаточно. Причину этого вы увидите при рассмотрении примера с датаграммным слушателем в следующей главе.

Датаграммная последовательность на стороне соединителя. Датаграммный соединитель выполняет почти ту же последовательность действий, что и потоковый. Единственное отличие состоит в типе сокета: в случае с датаграммным соединителем это должен быть `SOCK_DGRAM`. Особый случай представляют датаграммные соединители на основе сокетов домена Unix: в целях получения ответов от сервера они должны быть привязаны к файлу соответствующего сокета. Более подробно об этом поговорим в следующей главе, в которой будет представлен пример калькулятора с использованием датаграмм и сокетов домена Unix.

Итак, мы прошлись по всем возможным последовательностям. Теперь можно поговорить о том, как связаны между собой сокеты и *дескрипторы сокетов*. Это будет заключительный подраздел в данной главе, а уже в главе 20 мы рассмотрим реальные примеры на языке C, которые иллюстрируют все описанные последовательности.

У сокетов есть собственные дескрипторы!

В отличие от других пассивных методов межпроцессного взаимодействия, которые работают с файловыми дескрипторами, методики на основе сокетов имеют дело с объектами сокетов. Каждый объект имеет целочисленный идентификатор, играющий роль дескриптора сокета внутри ядра. С помощью этого дескриптора можно ссылаться на соответствующий канал.

Заметьте, что файловые дескрипторы и дескрипторы сокетов — это разные вещи. Первые указывают на обычные файлы или файлы устройств, а вторые ссылаются на объекты сокетов, созданные путем вызова функций `socket`, `accept` и `connect`.

Несмотря на разницу файловых дескрипторов и дескрипторов сокетов, для чтения и записи в них используется один и тот же API (или набор функций). Поэтому с сокетами, как и с файлами, можно работать с помощью функций `read` и `write`.

Эти дескрипторы имеют нечто общее: их API позволяет сделать их неблокирующими. Такие дескрипторы можно использовать для работы с файлом или сокетом в неблокирующей манере.

Резюме

В этой главе мы начали обсуждать методы межпроцессного взаимодействия, которые позволяют двум процессам общаться и обмениваться данными. Мы завершим изучение этой темы уже в следующей главе, где поговорим именно о программировании сокетов и рассмотрим различные реальные примеры на языке C.

В ходе данной главы мы:

- рассмотрели пассивные и активные методы межпроцессного взаимодействия, их различия и сходства;
- сравнили методы IPC на одном и нескольких компьютерах;
- познакомились с коммуникационными протоколами и их различными характеристиками;
- рассмотрели концепции сериализации и десериализации, поговорив о том, как они работают в рамках определенного коммуникационного протокола;
- выяснили, каким образом такие характеристики протокола, как содержимое, длина и упорядоченность, могут влиять на процесс-получатель;
- познакомились с POSIX-каналами и увидели пример их использования;
- выяснили, что такое очередь сообщений POSIX и как с ее помощью можно организовать взаимодействие двух процессов;
- кратко затронули тему сокетов домена Unix и их основные свойства;
- разобрались, что такое компьютерные сети и каким образом совокупность различных уровней сети позволяет устанавливать транспортные соединения;
- обсудили, что такое программирование сокетов;
- рассмотрели последовательность инициализации процессов слушателя и соединителя, а также шаги, которые они выполняют при подготовке к взаимодействию;
- сравнили файловые дескрипторы и дескрипторы сокетов.

В следующей главе мы продолжим обсуждать программирование сокетов и рассмотрим реальные примеры на языке C. Мы напишем клиентскую и серверную программы, которые будут выполнять функции калькулятора. Затем воспользуемся сокетами домена Unix и сетевыми сокетами, чтобы организовать полноценное клиент-серверное взаимодействие клиентского процесса калькулятора и его сервера.

20 Программирование сокетов

В предыдущей главе мы обсудили локальное межпроцессное взаимодействие и познакомились с концепцией программирования сокетов. Здесь же завершим наше введение и подробно рассмотрим тему программирования сокетов на примере реального клиент-серверного приложения: проекта «Калькулятор».

Порядок, в котором представлены темы, может показаться немного необычным, но это сделано для того, чтобы вы лучше ориентировались в различных типах сокетов и понимали, как они себя ведут в реальном проекте. В рамках данной главы мы:

- для начала подытожим материал, представленный в предыдущей главе. Это будет всего лишь краткое повторение пройденного материала. Вы обязательно должны прочитать вторую часть главы 19, посвященную программированию сокетов;
- в процессе повторения обсудим разные типы сокетов, датаграммные и потоковые последовательности, а также некоторые другие темы, которые понадобятся нам при рассмотрении примера с калькулятором;
- опишем и полностью проанализируем пример клиент-серверного проекта «Калькулятор». Это позволит нам перейти к обсуждению различных его компонентов и представить код на языке C;
- разработаем один из важнейших компонентов нашего примера — библиотеку сериализации/десериализации. Она реализует главный протокол взаимодействия клиентской и серверной частей калькулятора;
- необходимо понимать, что клиентская и серверная части калькулятора должны уметь взаимодействовать с помощью сокетов любых типов. Поэтому в нашем примере рассмотрим интеграцию с разного рода сокетами, начиная с *сокетов домена Unix* (Unix domain sockets, UDS);
- увидим в нашем примере, как с их помощью установить клиент-серверное соединение в рамках одного компьютера;
- вслед за этим рассмотрим сетевые сокеты; узнаем, как TCP- и UDP-сокеты можно интегрировать в проект «Калькулятор».

Для начала повторим все, что уже знаем о сокетах и программировании сокетов в целом. Прежде чем продолжать, крайне желательно ознакомиться со второй частью предыдущей главы. Я буду исходить из того, что вы уже обладаете необходимыми знаниями.

Краткий обзор программирования сокетов

В данном разделе мы еще раз поговорим о том, что такое сокеты, каких типов они бывают и что мы в целом понимаем под программированием сокетов. Это будет краткий обзор, но он станет основой для дальнейшего углубленного обсуждения в последующих разделах.

Если помните, в предыдущей главе мы говорили о наличии двух категорий межпроцессного взаимодействия, которые позволяют двум и более процессам общаться и обмениваться данными. К первой категории относятся *активные* (pull-based) методики, требующие наличия доступного *носителя* (такого как разделяемая память или обычный файл) для хранения и извлечения данных. Вторая категория охватывает *пассивные* (push-based) методики и подразумевает создание *канала*, доступного всем процессам, участвующим во взаимодействии. Главное различие между этими категориями связано с тем, как именно данные извлекаются из носителя (при активном подходе) или канала (при пассивном подходе).

Говоря простым языком, при использовании активных методик данные должны извлекаться или считываться с носителя, а в пассивном подходе данные доставляются читающему процессу автоматически. В первом случае процессы извлекают данные из разделяемого носителя, и если несколько из них могут туда записывать, то это чревато состояниями гонки.

Если быть более точным, то в активных методиках данные всегда доставляются в буфер внутри ядра, который доступен принимающему процессу с помощью дескриптора (файла или сокета).

Затем принимающий процесс может либо заблокироваться, пока не станут доступными какие-то новые данные, либо *запросить* дескриптор и проверить, появилось ли в буфере ядра нечто новое, и в случае отрицательного ответа заняться чем-то другим. Первый подход называется *блокирующим вводом/выводом*, а второй — *неблокирующим*, или *асинхронным*, *вводом/выводом*. В этой главе все активные методики используют блокирующий подход.

Мы уже знаем, что программирование сокетов — особая разновидность межпроцессного взаимодействия, относящаяся ко второй категории. Поэтому все методы IPC, основанные на сокетах, являются активными. Однако основная характеристика, которая отличает программирование сокетов от других активных методов IPC, заключается в использовании *сокетов*. Сокеты — это специальные объекты в Unix-подобных и других операционных системах, включая даже Microsoft Windows, представляющие *двунаправленные каналы*.

Иными словами, один объект сокета можно использовать для чтения и записи в один и тот же канал. Таким образом, два процесса, находящиеся на разных концах одного канала, могут участвовать в *двунаправленном взаимодействии*.

В предыдущей главе мы видели, что сокеты представлены дескрипторами по аналогии с тем, как файловые дескрипторы представляют файлы. Оба вида дескрипторов имеют некоторые общие аспекты, такие как операции ввода/вывода и возможность их *запрашивать*, однако на самом деле это разные вещи. Дескриптор сокета всегда представляет канал, а вот файловый дескриптор может представлять такие носители, как обычный файл или POSIX-канал. В связи с этим дескрипторы сокетов не поддерживают определенные операции с файлами, например `seek`; то же самое можно сказать даже о файловых дескрипторах, которые представляют канал.

Взаимодействие, основанное на сокетах, может работать как с соединениями, так и без. В первом случае канал представляет *поток* байтов, передающийся между двумя определенными процессами, а во втором по каналу могут передаваться *датаграммы*, и при этом никакого соединения между процессами нет. Несколько процессов могут использовать один и тот же канал для разделения состояния или обмена данными.

Таким образом, мы имеем два типа каналов: *потоковые* и *датаграммные*. Каждый потоковый канал в программе представлен *потоковым сокетом*, а каждый датаграммный — *датаграммным*. При подготовке канала мы должны сделать его либо потоковым, либо датаграммным. Чуть позже вы увидите, что наш пример с калькулятором поддерживает оба вида каналов.

Сокеты бывают разных типов. Каждый тип предназначен для определенных задач и ситуаций. В целом сокеты можно разделить на две категории: UDS и сетевые. Как вы уже, наверное, знаете из предыдущей главы, UDS можно использовать в случаях, когда все процессы, желающие участвовать в межпроцессном взаимодействии, находятся на одном компьютере. Иными словами, UDS подходит только для проектов, развернутых в рамках одной системы.

Для сравнения, сетевые сокеты можно применять в почти любой конфигурации, независимо от того, как именно развернуты процессы и где они находятся. Они могут размещаться на одном компьютере или быть распределены по сети. В случае с локальным развертыванием более предпочтительны сокеты UDS, поскольку они более быстрые и имеют меньше накладных расходов по сравнению с сетевыми сокетами. В рамках нашего примера с калькулятором мы реализуем поддержку как UDS, так и сетевых сокетов.

UDS и сетевые сокеты могут представлять как потоковые, так и датаграммные каналы. Следовательно, мы имеем четыре разные комбинации: UDS поверх потокового канала, UDS поверх датаграммного канала, сетевой сокет поверх потокового канала и, наконец, сетевой сокет поверх датаграммного канала. Все эти четыре варианта будут реализованы в нашем примере.

Сетевой сокет, предоставляющий потоковый канал, обычно работает по TCP. Дело в том, что TCP — самый распространенный транспортный протокол для этого вида сокетов. С другой стороны, сетевой сокет, предоставляющий датаграммный канал,

обычно работает по UDP. Это объясняется тем, что в большинстве случаев для такого рода сокетов используется транспортный протокол UDP. Обратите внимание: сокеты UDS, предоставляющие потоковые или датаграммные каналы, не имеют каких-то специальных названий, поскольку не применяют никаких транспортных протоколов.

Все указанные разновидности сокетов и каналов лучше всего показать на реальном примере. Вот, собственно, почему мы пошли таким необычным путем. Это позволит вам обратить внимание на общие аспекты разных типов сокетов и каналов и оформить их в виде блоков кода, пригодных к повторному использованию. В следующем разделе мы обсудим проект «Калькулятор» и его внутреннюю структуру.

Проект «Калькулятор»

Мы выделили целый раздел на то, чтобы объяснить назначение проекта «Калькулятор». Это масштабный пример, поэтому, прежде чем углубляться в него, будет полезно получить четкое понимание того, что он собой представляет. Данный проект должен помочь вам достичь следующих целей:

- рассмотреть полнофункциональный пример с рядом простых и четко определенных возможностей;
- извлечь общие аспекты различных типов сокетов и каналов и оформить их в виде библиотек, пригодных к повторному использованию. Это существенно уменьшит количество кода, который нужно будет написать, и продемонстрирует вам границы, пролегающие между разными видами сокетов и каналов;
- организовать взаимодействие с помощью четко определенного прикладного протокола. Обычным примерам программирования сокетов недостает этого очень важного свойства. Они, как правило, демонстрируют очень простые и зачастую одноразовые сценарии взаимодействия клиента и сервера;
- поработать над примером, имеющим все характеристики полнофункциональной клиент-серверной программы, такой как прикладной протокол с поддержкой разных видов каналов, возможностью сериализации/десериализации и т. д. Это позволит вам по-новому взглянуть на программирование сокетов.

С учетом всего вышесказанного этот проект станет нашим главным примером в данной главе. Мы будем разрабатывать его шаг за шагом, и я проведу вас через различные этапы, кульминацией которых станет завершенное и рабочее приложение.

Для начала следует разработать относительно простой и полноценный прикладной протокол. Он будет использоваться для взаимодействия клиента и сервера. Как уже объяснялось ранее, две стороны не могут общаться между собой без четко определенного протокола прикладного уровня. Они могут быть соединены и передавать данные, поскольку эту возможность предоставляет программирование сокетов, но не будут понимать друг друга.

Вот почему мы должны уделить немного времени обсуждению прикладного протокола, который будет использоваться в проекте «Калькулятор». Но прежде, чем переходить к самому протоколу, рассмотрим иерархию исходного кода, которую можно увидеть в кодовой базе проекта. Это существенно облегчит поиск прикладного протокола и библиотеки сериализации/десериализации.

Иерархия исходного кода

С точки зрения программиста, API для программирования POSIX-сокетов обращается со всеми потоковыми каналами одинаково, независимо от того, что лежит в их основе: UDS или сетевой сокет. Как вы можете помнить из материалов предыдущей главы, для потоковых каналов предусмотрены определенные последовательности шагов, которые выполняются на стороне слушателя и соединителя, и эти последовательности не зависят от типа сокетов.

Таким образом, если вы собираетесь поддерживать разные типы сокетов в сочетании с разными типами каналов, то лучше определить их общие аспекты и реализовать их отдельно, чтобы не повторяться. Именно такой подход мы станем применять в проекте «Калькулятор», и он отражен в исходном коде. Поэтому вам будут встречаться различные библиотеки, и некоторые из них будут содержать общий код, повторно использующийся разными компонентами.

Теперь пришло время взяться за кодовую базу. Прежде всего, исходный код проекта находится по адресу https://github.com/PacktPublishing/Extreme-C/tree/master/ch20-socket-programming. Если пройти по данной ссылке и взглянуть на код, то можно увидеть ряд каталогов с исходными файлами. Очевидно, что анализ каждого файла занял бы слишком много времени, поэтому мы сосредоточимся на важных участках кода. Вы можете сами пройтись по кодовой базе, собрать ее и запустить полученную программу; благодаря этому вы получите общее представление о том, как разрабатывался наш пример.

Отмечу, что весь код, относящийся к сокетам UDS, UDP и TCP, находится в одном каталоге. Далее мы рассмотрим иерархию кодовой базы.

Если перейти в корень проекта и выполнить команду `tree`, то можно увидеть дерево файлов и каталогов, представленное в терминале 20.1. В нем же показано, как клонировать репозиторий этой книги на GitHub и перейти в корневой каталог данного примера.

Терминал 20.1. Клонирование кодовой базы проекта «Калькулятор» и вывод его файлов и каталогов

```
$ git clone https://github.com/PacktPublishing/Extreme-C
Cloning into 'Extreme-C'...
...
```

```
Resolving deltas: 100% (458/458), done.
$ cd Extreme-C/ch20-socket-programming
$ tree
.
├── CMakeLists.txt
├── calcser
...
├── calcsvc
...
├── client
│   ├── CMakeLists.txt
│   ├── clicore
...
│   ├── tcp
│   │   ├── CMakeLists.txt
│   │   └── main.c
│   ├── udp
│   │   ├── CMakeLists.txt
│   │   └── main.c
│   └── Unix
│       ├── CMakeLists.txt
│       ├── datagram
│       │   ├── CMakeLists.txt
│       │   └── main.c
│       └── stream
│           ├── CMakeLists.txt
│           └── main.c
├── server
│   ├── CMakeLists.txt
│   ├── srvcore
...
│   ├── tcp
│   │   ├── CMakeLists.txt
│   │   └── main.c
│   ├── udp
│   │   ├── CMakeLists.txt
│   │   └── main.c
│   └── Unix
│       ├── CMakeLists.txt
│       ├── datagram
│       │   ├── CMakeLists.txt
│       │   └── main.c
│       └── stream
│           ├── CMakeLists.txt
│           └── main.c
└── types.h

18 directories, 49 files
$
```

Как можно видеть в этом списке файлов и каталогов, наш проект состоит из ряда компонентов, в том числе и библиотек. Все компоненты находятся в отдельных каталогах, каждый из которых описан ниже.

- Библиотека сериализации/десериализации в каталоге /calcser содержит соответствующие исходные файлы. Она описывает прикладной протокол, по которому общаются клиентская и серверная часть калькулятора. В итоге собирается в статическую библиотеку `libcalcser.a`.
- Библиотека в каталоге /calcsvc содержит исходники *вычислительного сервиса*. Это не то же самое, что серверный процесс. Данный сервис предоставляет основные функции калькулятора и не привязан к серверному процессу, и потому его можно использовать отдельно в качестве самостоятельной библиотеки С. В результате сборки этого каталога получается статическая библиотека `libcalcsvc.a`.
- Библиотека в каталоге /server/srvcore содержит исходники, общие для потоковых и датаграммных процессов, независимо от типа сокета. Поэтому ее могут использовать все серверные процессы калькулятора, включая те, которые основаны на UDS и сетевых сокетах и работают с потоковыми и датаграммными каналами. Итоговым результатом сборки этого каталога будет статическая библиотека `libsrvcore.a`.
- Каталог /server/unix/stream содержит исходники серверной программы, которая использует потоковые каналы внутри сокетов UDS. Она будет собрана в исполняемый файл `unix_stream_calc_server`. Это одна из нескольких программ, которые можно применять в качестве серверной части калькулятора. Этот конкретный сервер прослушивает сокет UDS для установления потоковых соединений.
- Каталог /server/unix/datagram содержит исходники серверной программы, которая использует датаграммные каналы внутри сокетов UDS. Она будет собрана в исполняемый файл `unix_datagram_calc_server`. Это одна из нескольких программ, которые можно задействовать в качестве серверной части калькулятора. Этот конкретный сервер прослушивает сокет UDS для приема датаграммных сообщений.
- Каталог /server/tcp содержит исходники серверной программы, которая использует потоковые каналы внутри сетевых сокетов TCP. Она будет собрана в исполняемый файл `tcp_calc_server`. Это одна из нескольких программ, которые можно применять в качестве серверной части калькулятора. Этот конкретный сервер прослушивает сокет TCP для установления потоковых соединений.
- Каталог /server/udp содержит исходники серверной программы, которая использует датаграммные каналы внутри сетевых сокетов UDP. Она будет собрана в исполняемый файл `udp_calc_server`. Это одна из нескольких программ,

которые можно задействовать в качестве серверной части калькулятора. Этот конкретный сервер прослушивает сокет UDP для приема датаграммных сообщений.

- Библиотека в каталоге /client/clicore содержит исходники, общие для потоковых и датаграммных клиентских процессов, независимо от типа сокета. Поэтому ее могут использовать все клиентские процессы калькулятора, включая те, которые основаны на UDS и сетевых сокетах и работают с потоковыми и датаграммными каналами. Итоговым результатом сборки этого каталога будет статическая библиотека `libclicore.a`.
- Каталог /client/unix/stream содержит исходники клиентской программы, которая использует потоковые каналы внутри сокетов UDS. Она будет собрана в исполняемый файл `unix_stream_calc_client`. Это одна из нескольких программ, которые можно применять для запуска клиентской части калькулятора. Этот конкретный клиент подключается к конечной точке UDS и устанавливает потоковое соединение.
- Каталог /client/unix/datagram содержит исходники клиентской программы, которая использует датаграммные каналы внутри сокетов UDS. Она будет собрана в исполняемый файл `unix_datagram_calc_client`. Это одна из нескольких программ, которые можно задействовать в целях запуска клиентской части калькулятора. Этот конкретный клиент подключается к конечной точке UDS и отправляет датаграммные сообщения.
- Каталог /client/tcp содержит исходники клиентской программы, которая использует потоковые каналы внутри сокетов TCP. Она будет собрана в исполняемый файл `tcp_calc_client`. Это одна из нескольких программ, которые можно применять для запуска клиентской части калькулятора. Этот конкретный клиент подключается к конечной точке TCP-сокета и устанавливает потоковое соединение.
- Каталог /client/udp содержит исходники клиентской программы, которая использует датаграммные каналы внутри сокетов UDP. Она будет собрана в исполняемый файл `udp_calc_client`. Это одна из нескольких программ, которые можно задействовать для запуска клиентской части калькулятора. Этот конкретный клиент подключается к конечной точке UDP-сокета и отправляет датаграммные сообщения.

Сборка проекта

Итак, мы прошлись по всем каталогам проекта. Теперь нам нужно показать, как он собирается. Проект использует систему CMake, поэтому, прежде чем переходить к сборке, убедитесь в том, что она у вас установлена.

Чтобы собрать проект, выполните следующие команды в корневом каталоге главы (терминал 20.2).

Терминал 20.2. Программы для сборки проекта «Калькулятор»

```
$ mkdir -p build
$ cd build
$ cmake ..
...
$ make
...
$
```

Запуск проекта

Ничто так не помогает убедиться в работоспособности проекта, как его самостоятельный запуск. Поэтому, прежде чем переходить к техническим подробностям, я хочу, чтобы вы по очереди запустили серверную и клиентскую части калькулятора и понаблюдали за тем, как они общаются друг с другом.

Перед запуском процессов необходимо открыть два отдельных терминала (или командные оболочки), чтобы ввести два разных набора команд. В первом терминале мы запустим потоковый сервер, прослушивающий сокет UDS. Соответствующая команда приводится ниже (терминал 20.3).

Обратите внимание: перед вводом этой команды вы должны перейти в каталог build, который был создан в рамках предыдущего раздела.

Терминал 20.3. Запуск потокового сервера, который прослушивает сокет UDS

```
$ ./server/unix/stream/unix_stream_calc_server
```

Убедитесь в том, что сервер работает. Запустите во втором терминале потоковый клиент, собранный для использования UDS (терминал 20.4).

Терминал 20.4. Запуск клиентской части калькулятора и отправка нескольких запросов

```
$ ./client/unix/stream/unix_stream_calc_client
? (type quit to exit) 3++4
The req(0) is sent.
req(0) > status: OK, result: 7.000000
? (type quit to exit) mem
The req(1) is sent.
req(1) > status: OK, result: 7.000000
? (type quit to exit) 5++4
The req(2) is sent.
req(2) > status: OK, result: 16.000000
? (type quit to exit) quit
Bye.
$
```

Как видите, у клиентского процесса есть собственная командная строка. Она принимает команды от пользователя, превращает их в запросы, соответствующие прикладному протоколу, и отправляет их серверу для дальнейшей обработки. Затем клиент ждет ответа и сразу после того, как тот будет получен, выводит результат. Отмечу, что данная командная строка является частью общего кода, написанного для всех клиентов, поэтому вы всегда сможете работать с ней, независимо от типа канала или сокета.

Теперь пришло время углубиться в детали прикладного протокола и посмотреть на то, как выглядят запросы и ответы.

Прикладной протокол

Любые два процесса, которые хотят общаться друг с другом, должны соблюдать общий прикладной протокол. Он может быть написан специально для проекта «Калькулятор», но мы можем воспользоваться и общеизвестным стандартом, таким как HTTP. Наш протокол будет называться *протоколом калькулятора*.

Сообщения в протоколе калькулятора имеют переменную длину. То есть длина сообщений может отличаться, и между ними должны находиться разделители. У нас будет один тип запросов и один тип ответов. Вдобавок следует сказать, что протокол будет текстовым. То есть запросы и ответы могут состоять только из алфавитно-цифровых и нескольких других символов. Это позволит сделать сообщения калькулятора понятными человеку.

Запрос состоит из четырех полей: *идентификатора*, *метода*, *первого* и *второго операнда*. Каждый запрос обладает уникальным идентификатором, благодаря которому сервер знает, кому отправлять соответствующий ответ.

Метод — операция, выполняемая сервисом калькулятора. В листинге 20.1 показан заголовочный файл `calcser/calc_proto_req.h`, который описывает структуру запроса в нашем протоколе.

Листинг 20.1. Определение объекта запроса (calcser/calc_proto_req.h)

```
#ifndef CALC_PROTO_REQ_H
#define CALC_PROTO_REQ_H

#include <stdint.h>

typedef enum {
  NONE,
  GETMEM, RESMEM,
  ADD, ADDM,
  SUB, SUBM,
```

```
  MUL, MULM,
  DIV
} method_t;

struct calc_proto_req_t {
  int32_t id;
  method_t method;
  double operand1;
  double operand2;
};

method_t str_to_method(const char*);
const char* method_to_str(method_t);

#endif
```

Как видите, в рамках нашего протокола определено девять методов. Любой приличный калькулятор должен иметь внутреннюю память, и потому у нас есть операции с памятью, относящиеся к сложению, вычитанию и умножению.

Например, метод `ADD` просто складывает два числа с плавающей запятой, а `ADDM` — его разновидность, которая прибавляет к этим двум числам значение, хранящееся во внутренней памяти, и заносит результат в ту же память для дальнейшего использования. Это аналогично применению кнопки памяти в настольном калькуляторе; вы можете найти ее по надписи +M.

У нас также есть специальный метод для чтения и сброса внутренней памяти калькулятора. Операцию деления с внутренней памятью выполнять нельзя, поэтому других вариаций не предусмотрено.

Представьте, что клиент хочет создать запрос с ID `1000`, методом `ADD` и двумя операндами: `1.5` и `5.6`. В языке C для этого нужно создать объект `calc_proto_req_t` (данная структура объявлена в предыдущем заголовке в листинге 20.1) и заполнить его нужными значениями. В листинге 20.2 показано, как это делается.

Листинг 20.2. Создание объекта запроса на C

```
struct calc_proto_req_t req;
req.id = 1000;
req.method = ADD;
req.operand1 = 1.5;
req.operand2 = 5.6;
```

Как уже объяснялось в предыдущей главе, объект `req` в этом листинге можно отправить серверу только после сериализации и превращения его в запрос. Иными словами, нам нужно сериализовать данный *объект запроса* в соответствующее *сообщение запроса*. В соответствии с нашим прикладным протоколом результат сериализации объекта `req` будет выглядеть следующим образом (листинг 20.3).

Листинг 20.3. Сериализованное сообщение, эквивалентное объекту req, который был определен в листинге 20.2

```
1000#ADD#1.5#5.6$
```

Символ # служит для *разделения полей*, а символ $ играет роль *разделителя сообщений*. Кроме того, у каждого сообщения есть ровно четыре поля. *Десериализатор* на другом конце канала использует эти факты для разбора входящих байтов и воссоздания оригинального объекта.

С другой стороны, серверный процесс, который отвечает на запрос, должен сериализовать объект ответа. Ответ состоит из трех полей: *идентификатора запроса*, *статуса* и *результата*. Идентификатор уникален и указывает на запрос, на который хочет ответить сервер.

Заголовочный файл `calcser/calc_proto_resp.h` описывает то, как должен выглядеть ответ. Вы можете видеть его в листинге 20.4.

Листинг 20.4. Определение объекта ответа (calcser/calc_proto_resp.h)

```
#ifndef CALC_PROTO_RESP_H
#define CALC_PROTO_RESP_H

#include <stdint.h>

#define STATUS_OK              0
#define STATUS_INVALID_REQUEST 1
#define STATUS_INVALID_METHOD  2
#define STATUS_INVALID_OPERAND 3
#define STATUS_DIV_BY_ZERO     4
#define STATUS_INTERNAL_ERROR  20

typedef int status_t;

struct calc_proto_resp_t {
  int32_t req_id;
  status_t status;
  double result;
};

#endif
```

По аналогии с объектом запроса из листинга 20.2, `req`, серверный процесс создает *объект ответа*, выполняя следующие инструкции (листинг 20.5).

Листинг 20.5. Создание объекта ответа для объекта req, который был определен в листинге 20.2

```
struct calc_proto_resp_t resp;
resp.req_id = 1000;
resp.status = STATUS_OK;
resp.result = 7.1;
```

Результат сериализации этого объекта выглядит так (листинг 20.6).

Листинг 20.6. Сериализованное ответное сообщение, эквивалентное объекту resp, который был создан в листинге 20.5

```
1000#0#7.1$
```

И снова мы используем символы # для разделения полей и $ для разделения сообщений. Обратите внимание: поле `status` является числовым и сигнализирует об успешном или неуспешном запросе. В случае неудачи оно имеет ненулевое значение, описанное в заголовочном файле ответа (или, точнее, в протоколе калькулятора).

Теперь более подробно поговорим о библиотеке сериализации/десериализации и посмотрим, как она выглядит внутри.

Библиотека сериализации/десериализации

В предыдущем подразделе было показано, как выглядят сообщения запроса и ответа. Здесь же поговорим об алгоритмах сериализации и десериализации, которые используются в проекте «Калькулятор». Предоставление соответствующих операций будет выполняться с помощью класса `serializer` с `calc_proto_ser_t` в качестве структуры атрибутов.

Я уже упоминал о том, что эти возможности предоставляются другим частям проекта в виде статической библиотеки `libcalcser.a`. В листинге 20.7 вы можете видеть публичный API класса `serializer`, который находится в файле `calcser/calc_proto_ser.h`.

Листинг 20.7. Публичный интерфейс класса serializer (calcser/calc_proto_ser.h)

```
#ifndef CALC_PROTO_SER_H
#define CALC_PROTO_SER_H

#include <types.h>

#include "calc_proto_req.h"
#include "calc_proto_resp.h"

#define ERROR_INVALID_REQUEST          101
#define ERROR_INVALID_REQUEST_ID       102
#define ERROR_INVALID_REQUEST_METHOD   103
#define ERROR_INVALID_REQUEST_OPERAND1 104
#define ERROR_INVALID_REQUEST_OPERAND2 105

#define ERROR_INVALID_RESPONSE         201
#define ERROR_INVALID_RESPONSE_REQ_ID  202
```

```
#define ERROR_INVALID_RESPONSE_STATUS  203
#define ERROR_INVALID_RESPONSE_RESULT  204

#define ERROR_UNKNOWN 220

struct buffer_t {
  char* data;
  int len;
};

struct calc_proto_ser_t;

typedef void (*req_cb_t)(
        void* owner_obj,
        struct calc_proto_req_t);

typedef void (*resp_cb_t)(
        void* owner_obj,
        struct calc_proto_resp_t);

typedef void (*error_cb_t)(
        void* owner_obj,
        const int req_id,
        const int error_code);

struct calc_proto_ser_t* calc_proto_ser_new();
void calc_proto_ser_delete(
        struct calc_proto_ser_t* ser);

void calc_proto_ser_ctor(
        struct calc_proto_ser_t* ser,
        void* owner_obj,
        int ring_buffer_size);

void calc_proto_ser_dtor(
        struct calc_proto_ser_t* ser);

void* calc_proto_ser_get_context(
        struct calc_proto_ser_t* ser);

void calc_proto_ser_set_req_callback(
        struct calc_proto_ser_t* ser,
        req_cb_t cb);

void calc_proto_ser_set_resp_callback(
        struct calc_proto_ser_t* ser,
        resp_cb_t cb);
void calc_proto_ser_set_error_callback(
        struct calc_proto_ser_t* ser,
        error_cb_t cb);

void calc_proto_ser_server_deserialize(
        struct calc_proto_ser_t* ser,
```

```
        struct buffer_t buffer,
        bool_t* req_found);

struct buffer_t calc_proto_ser_server_serialize(
        struct calc_proto_ser_t* ser,
        const struct calc_proto_resp_t* resp);

void calc_proto_ser_client_deserialize(
        struct calc_proto_ser_t* ser,
        struct buffer_t buffer,
        bool_t* resp_found);

struct buffer_t calc_proto_ser_client_serialize(
        struct calc_proto_ser_t* ser,
        const struct calc_proto_req_t* req);

#endif
```

Помимо конструктора и деструктора, которые нужны для создания и уничтожения объекта `serializer`, у нас есть две пары функций: первая пара для серверного процесса, а вторая — для клиентского.

На клиентской стороне мы сериализуем объект запроса и десериализуем сообщение с ответом. А на серверной стороне десериализуем сообщение с запросом и сериализуем объект ответа.

Помимо операций сериализации и десериализации, у нас также есть три *функции обратного вызова*:

- обратный вызов для получения объекта запроса, десериализованного из соответствующего канала;
- обратный вызов для получения объекта ответа, который был десериализован из соответствующего канала;
- обратный вызов для получения ошибки в случае провала сериализации или десериализации.

Эти обратные вызовы используются клиентскими и серверными процессами для получения входящих запросов и ответов, а также ошибок, которые могут возникнуть при сериализации или десериализации сообщения.

Теперь поближе рассмотрим функции сериализации/десериализации для серверной стороны.

Функции сериализации/десериализации для серверной стороны

Для серверного процесса предусмотрено две функции: одна сериализует объект ответа, а другая десериализует сообщение с запросом. Начнем с функции сериализации.

В листинге 20.8 представлен код функции `calc_proto_ser_server_serialize`, которая сериализует ответ.

Листинг 20.8. Функция сериализации ответа для серверной стороны (calcser/calc_proto_ser.c)

```
struct buffer_t calc_proto_ser_server_serialize(
    struct calc_proto_ser_t* ser,
    const struct calc_proto_resp_t* resp) {
  struct buffer_t buff;
  char resp_result_str[64];
  _serialize_double(resp_result_str, resp->result);
  buff.data = (char*)malloc(64 * sizeof(char));
  sprintf(buff.data, "%d%c%d%c%s%c", resp->req_id,
          FIELD_DELIMITER, (int)resp->status, FIELD_DELIMITER,
      resp_result_str, MESSAGE_DELIMITER);
  buff.len = strlen(buff.data);
  return buff;
}
```

Элемент `resp` — это указатель на объект ответа, который нужно сериализовать. Функция возвращает объект `buffer_t`, который объявлен в заголовочном файле `calc_proto_ser.h` и выглядит так (листинг 20.9).

Листинг 20.9. Определение объекта buffer_t (calcser/calc_proto_ser.h)

```
struct buffer_t {
  char* data;
  int len;
};
```

Сериализатор имеет простой код, основная часть которого — инструкция `sprintf`, предназначенная для создания строки с ответным сообщением. Теперь взглянем на функцию десериализации запроса. Десериализатор обычно сложнее реализовать, и если найти в кодовой базе вызовы следующих функций, то можно увидеть, насколько сложными они бывают.

В листинге 20.10 показана функция для десериализации запроса.

Листинг 20.10. Функция десериализации запроса для серверной стороны (calcser/calc_proto_ser.c)

```
void calc_proto_ser_server_deserialize(
    struct calc_proto_ser_t* ser,
    struct buffer_t buff,
    bool_t* req_found) {
  if (req_found) {
    *req_found = FALSE;
  }
  _deserialize(ser, buff, _parse_req_and_notify,
          ERROR_INVALID_REQUEST, req_found);
}
```

Приведенная выше функция выглядит просто, однако на самом деле использует приватные методы `_deserialize` и `_parse_req_and_notify`, которые определены в файле `calc_proto_ser.c`, где содержится сама реализация класса `Serializer`.

Мы не станем углубляться в код упомянутых приватных методов, поскольку это уже выходит за рамки данной книги. Но чтобы вы имели общее представление, особенно если вам хочется почитать исходный код, отмечу: десериализатор использует *кольцевой буфер* фиксированной длины и пытается найти символ `$`, который служит разделителем сообщений.

При нахождении `$` десериализатор применяет указатель, который в нашем случае ссылается на функцию `_parse_req_and_notify` (третий аргумент, переданный функции `_deserialize`). Она пытается извлечь поля и воссоздать объект запроса. Затем использует функции обратного вызова, отправляя уведомления зарегистрированному *наблюдателю*, роль которого в данном случае исполняет объект сервера, а тот уже приступает к обработке запроса.

Теперь рассмотрим функции, применяемые на стороне клиента.

Функции сериализации/десериализации для клиентской стороны

Как и в случае с серверной стороной, для стороны клиента предусмотрено две функции: одна для сериализации объекта запроса, а другая для сериализации входящего ответа.

Начнем с сериализатора запроса. Его определение можно видеть в листинге 20.11.

Листинг 20.11. Функция сериализации запроса для клиентской стороны (calcser/calc_proto_ser.c)

```
struct buffer_t calc_proto_ser_client_serialize(
    struct calc_proto_ser_t* ser,
    const struct calc_proto_req_t* req) {
  struct buffer_t buff;
  char req_op1_str[64];
  char req_op2_str[64];
  _serialize_double(req_op1_str, req->operand1);
  _serialize_double(req_op2_str, req->operand2);
  buff.data = (char*)malloc(64 * sizeof(char));
  sprintf(buff.data, "%d%c%s%c%s%c%s%c", req->id, FIELD_DELIMITER,
          method_to_str(req->method), FIELD_DELIMITER,
          req_op1_str, FIELD_DELIMITER, req_op2_str,
          MESSAGE_DELIMITER);
  buff.len = strlen(buff.data);
  return buff;
}
```

Данная функция принимает объект запроса и возвращает объект `buffer`; это очень похоже на сериализацию ответа на серверной стороне. Здесь даже применяется тот же подход: использование инструкции `sprintf` для создания сообщения с запросом.

В листинге 20.12 показана функция десериализации ответа.

Листинг 20.12. Функция десериализации ответа для клиентской стороны (calcser/calc_proto_ser.c)

```
void calc_proto_ser_client_deserialize(
    struct calc_proto_ser_t* ser,
    struct buffer_t buff, bool_t* resp_found) {
  if (resp_found) {
    *resp_found = FALSE;
  }
  _deserialize(ser, buff, _parse_resp_and_notify,
          ERROR_INVALID_RESPONSE, resp_found);
}
```

Здесь применяется тот же механизм и задействуются похожие приватные методы. Я настоятельно рекомендую должным образом прочесть эти исходники, чтобы лучше понять, как разные части кода были собраны вместе и максимально хорошо подготовлены для повторного использования существующих компонентов.

Мы не станет углубляться в класс `Serializer`; можете сами пройтись по коду и посмотреть, как он работает.

Итак, у нас есть библиотека сериализации. Теперь мы можем написать клиентскую и серверную программы. Разработка библиотеки, которая сериализует и десериализует сообщения в соответствии с согласованным прикладным протоколом, — обязательный шаг при написании многопроцессного программного обеспечения. Заметьте, что это не зависит от того, как развернута ваша система: на одном или нескольких компьютерах. Процессы должны понимать друг друга, и для этого должны быть определены подходящие протоколы прикладного уровня.

Прежде чем переходить к коду, относящемуся к программированию сокетов, необходимо объяснить еще кое-что: сервис калькулятора. Он лежит в основе серверного процесса и выполняет сами вычисления.

Сервис калькулятора

Сервис калькулятора — основная логика нашего примера. Стоит отметить, что он должен работать независимо от того или иного механизма межпроцессного взаимодействия. В листинге 20.13 показано объявление его класса.

Как видите, этот сервис спроектирован таким образом, что его можно использовать даже в простейшей программе, состоящей из одной лишь функции `main` и не имеющей никакого отношения к IPC.

Листинг 20.13. Публичный интерфейс класса, принадлежащего сервису калькулятора (calcsvc/calc_service.h)

```
#ifndef CALC_SERVICE_H
#define CALC_SERVICE_H

#include <types.h>

static const int CALC_SVC_OK = 0;
static const int CALC_SVC_ERROR_DIV_BY_ZERO = -1;

struct calc_service_t;

struct calc_service_t* calc_service_new();
void calc_service_delete(struct calc_service_t*);

void calc_service_ctor(struct calc_service_t*);
void calc_service_dtor(struct calc_service_t*);

void calc_service_reset_mem(struct calc_service_t*);
double calc_service_get_mem(struct calc_service_t*);
double calc_service_add(struct calc_service_t*, double, double b, bool_t mem);
double calc_service_sub(struct calc_service_t*, double, double b, bool_t mem);
double calc_service_mul(struct calc_service_t*, double, double b, bool_t mem);
int calc_service_div(struct calc_service_t*, double, double, double*);
#endif
```

Как видите, в этом классе даже предусмотрены отдельные типы ошибок. Входные аргументы имеют стандартные типы языка C, которые никоим образом не зависят от классов или структур, связанных с сериализацией. Поскольку это изолированная и самостоятельная логика, мы компилируем ее в виде отдельной статической библиотеки `libcalcsvc.a`.

Каждый серверный процесс должен использовать объекты данного сервиса для выполнения вычислений. Эти объекты обычно называются *сервисными* или *служебными*. И потому итоговая серверная программа должна быть скомпонована с данной библиотекой.

Прежде чем двигаться дальше, необходимо сделать следующее замечание: если клиенту не требуется определенный контекст, то мы можем ограничиться одним служебным объектом. То есть если сервис на клиентской стороне не требует от нас сохранения какого-либо состояния из предыдущих запросов данного клиента, то служебный объект можно сделать *синглтоном*. Это значит, что у объекта *нет состояния*.

И наоборот, если для обработки текущего запроса нужна какая-либо информация о предыдущих запросах, то в каждом клиенте необходимо использовать отдельный служебный объект. Именно это происходит в нашем проекте. Как вы уже знаете, у калькулятора есть внутренняя память, уникальная для каждого клиента. Поэтому мы не можем задействовать один и тот же объект в разных клиентах. Это значит, у объекта *есть состояние*.

Если подытожить вышесказанное, то для каждого клиента нам нужно создавать новый объект сервиса. Благодаря этому у каждого клиента будет свой калькулятор с отдельной внутренней памятью. Служебные объекты калькулятора имеют состояние (значение во внутренней памяти), которое им нужно загружать.

Теперь мы готовы перейти к обсуждению различных типов сокетов с примерами в контексте проекта «Калькулятор».

Сокеты домена Unix

Мы уже знаем по предыдущей главе, при установлении соединения между двумя процессами, размещенными на одном компьютере, одним из лучших решений являются сокеты домена Unix (Unix domain sockets, UDS). В данной главе мы продолжили наше обсуждение и поговорили чуть более подробно о пассивных методах межпроцессного взаимодействия, а также о потоковых и датаграммных каналах. Теперь пришло время объединить эти знания и посмотреть на UDS в действии.

Текущий раздел состоит из четырех частей, посвященных разным видам процессов, находящимся на стороне слушателя или соединителя и использующим потоковые или датаграммные каналы. Все эти процессы работают с сокетами UDS. Мы рассмотрим все шаги, которые они должны выполнить в целях создания канала с учетом последовательностей, представленных нами в предыдущей главе. Для начала поговорим о слушающем процессе, работающем с потоковым каналом. Это будет *потоковый сервер*.

Потоковый сервер на основе UDS

Как вы помните, в предыдущей главе мы рассматривали разные последовательности действий, которые выполняют слушатель и соединитель, взаимодействуя с помощью транспортного канала. Сервер играет роль слушателя, поэтому должен выполнять его последовательность. В частности, поскольку в данном подразделе речь идет о потоковом канале, ему следует использовать последовательность потокового слушателя.

В рамках этой последовательности сервер должен сначала создать объект сокета. В нашем проекте потоковый сервер, желающий принимать соединения по UDS, должен выполнять те же шаги.

Следующий фрагмент кода находится в главной функции серверной программы. Как видно в листинге 20.14, процесс сначала создает объект `socket`.

Листинг 20.14. Создание потокового объекта UDS (server/unix/stream/main.c)

```
int server_sd = socket(AF_UNIX, SOCK_STREAM, 0);
if (server_sd == -1) {
  fprintf(stderr, "Could not create socket: %s\n",
strerror(errno));
  exit(1);
}
```

Как видите, объект сокета создается с помощью функции `socket`. Она подключается из заголовка `<sys/socket.h>`, который входит в стандарт POSIX. Обратите внимание: мы пока не знаем, каким будет данный объект: клиентским или серверным. Это смогут определить только последующие функции.

В предыдущей главе мы уже объясняли, что у каждого объекта сокета есть три атрибута, которые определяются тремя аргументами, переданными функции `socket`. Эти аргументы указывают семейство адресов, тип и протокол, которые будет использовать данный объект.

Согласно последовательности инициализации потокового слушателя, особенно той ее части, которая относится к UDS после создания объекта сокета, серверная программа должна привязаться к *файлу сокета*. Это будет наш следующий шаг. Листинг 20.15 используется в проекте «Калькулятор» в целях привязки объекта сокета к файлу, расположенному по заранее известному пути. Этот путь указан в виде массива символов `sock_file`.

Листинг 20.15. Привязка потокового объекта UDS к файлу сокета, заданному с помощью массива символов sock_file (server/unix/stream/main.c)

```
struct sockaddr_un addr;
memset(&addr, 0, sizeof(addr));
addr.sun_family = AF_UNIX;
strncpy(addr.sun_path, sock_file, sizeof(addr.sun_path) - 1);

int result = bind(server_sd, (struct sockaddr*)&addr,
sizeof(addr));
if (result == -1) {
  close(server_sd);
  fprintf(stderr, "Could not bind the address: %s\n",
strerror(errno));
  exit(1);
}
```

Данный код состоит из двух этапов. На первом создается экземпляр типа `struct sockaddr_un` с именем `addr`, который после инициализации указывает на файл

сокета. На втором этапе объект `addr` передается функции `bind`, чтобы она знала, какой файл следует *привязать* к объекту сокета. Вызов данной функции завершается успешно, только если к заданному файлу не привязан никакой другой объект. Следовательно, в случае с UDS два объекта сокетов, которые, вероятно, принадлежат разным процессам, не могут быть привязаны к одному и тому же файлу.

В Linux UDS можно привязать к абстрактному адресу сокета. Это в основном полезно в ситуациях, когда для размещения файла сокета нельзя задействовать файловую систему. В приведенном выше листинге в целях инициализации структуры адреса, addr, можно применять строку, начинающуюся с нулевого символа, \0, в результате чего предоставленное имя будет привязано к объекту сокета внутри ядра. Данное имя должно быть уникальным в рамках всей системы, и к нему не должен быть привязан никакой другой объект.

Продолжая говорить о пути к файлу сокета, следует отметить, что в большинстве систем Unix он не может превышать 104 байтов. Однако в системах Linux его длина составляет 108 байт. Обратите внимание: строковая переменная, которая хранит этот путь в виде массива типа `char`, всегда содержит в конце дополнительный нулевой символ. Поэтому путь к файлу сокета фактически имеет длину 103 или 107 байт в зависимости от операционной системы.

Если функция `bind` возвращает `0`, то это значит, что привязка прошла успешно и вы можете приступать к следующему шагу в последовательности потокового слушателя: настройке размера *очереди отставания*.

В коде, представленном в листинге 20.16, показана процедура настройки очереди отставания для потокового сервера, прослушивающего сокет UDS.

Листинг 20.16. Настройка размера очереди отставания для привязанного потокового сокета (server/unix/stream/main.c)

```
result = listen(server_sd, 10);
if (result == -1) {
  close(server_sd);
  fprintf(stderr, "Could not set the backlog: %s\n",
strerror(errno));
  exit(1);
}
```

Функция `listen` задает размер очереди отставания для уже привязанного сокета. Как объяснялось в предыдущей главе, когда занятый серверный процесс не в состоянии принимать от клиентов дальнейшие входящие запросы, некоторое количество этих запросов может подождать в очереди отставания, пока у программы не появится возможность их обработать. Это важный этап подготовки потокового сокета перед приемом клиентских запросов.

Согласно последовательности действий потокового слушателя, вслед за привязкой потокового сокета и настройки размера его очереди отставания мы можем приступить к приему новых клиентских запросов. В листинге 20.17 показано, как это делается.

Листинг 20.17. Принятие новых клиентских запросов с помощью сокета потокового слушателя (server/unix/stream/main.c)

```
while (1) {
  int client_sd = accept(server_sd, NULL, NULL);
  if (client_sd == -1) {
    close(server_sd);
    fprintf(stderr, "Could not accept the client: %s\n", strerror(errno));
    exit(1);
  }
  ...
}
```

Все волшебство происходит в функции `accept`, которая возвращает новый сокет при получении нового запроса со стороны клиента. Возвращаемый объект сокета указывает на соответствующий потоковый канал, созданный между сервером и клиентом, запрос которого был принят. Обратите внимание: у клиента есть свой потоковый канал и, следовательно, собственный дескриптор сокета.

Отмечу: если потоковый слушающий сокет является блокирующим (что происходит по умолчанию), то функция `accept` блокирует выполнение, пока не поступит новый клиентский запрос. То есть при отсутствии новых запросов поток выполнения, вызывающий функцию `accept`, блокируется на ней.

Теперь пришло время собрать все перечисленные выше шаги в единое целое. В листинге 20.18 показан потоковый сервер из проекта «Калькулятор», который прослушивает сокет UDS.

Листинг 20.18. Главная функция потокового сервиса калькулятора, прослушивающая конечную точку UDS (server/unix/stream/main.c)

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>

#include <sys/socket.h>
#include <sys/un.h>

#include <stream_server_core.h>

int main(int argc, char** argv) {
  char sock_file[] = "/tmp/calc_svc.sock";
```

```
  // ----------- 1. Создаем объект сокета ------------------
  int server_sd = socket(AF_UNIX, SOCK_STREAM, 0);
  if (server_sd == -1) {
    fprintf(stderr, "Could not create socket: %s\n", strerror(errno));
    exit(1);
  }

  // ----------- 2. Привязываем файл сокета ------------------

  // Удаляем ранее созданный файл сокета, если таковой имеется
  unlink(sock_file);

  // Подготавливаем адрес
  struct sockaddr_un addr;
  memset(&addr, 0, sizeof(addr));
  addr.sun_family = AF_UNIX;
  strncpy(addr.sun_path, sock_file, sizeof(addr.sun_path) - 1);

  int result = bind(server_sd,
          (struct sockaddr*)&addr, sizeof(addr));
  if (result == -1) {
    close(server_sd);
    fprintf(stderr, "Could not bind the address: %s\n",
            strerror(errno));
    exit(1);
  }

  // ----------- 3. Подготавливаем резерв ------------------
  result = listen(server_sd, 10);
  if (result == -1) {
    close(server_sd);
    fprintf(stderr, "Could not set the backlog: %s\n",
            strerror(errno));
    exit(1);
  }

  // ----------- 4. Начинаем принимать клиентов ---------
  accept_forever(server_sd);

  return 0;
}
```

Найти блоки кода, ответственные за выполнение вышеупомянутых действий по инициализации серверного сокета, должно быть довольно легко. Здесь не хватает лишь кода, который принимает запросы от клиентов. Он вынесен в отдельную функцию с именем `accept_forever`. Обратите внимание: данная функция является блокирующей, поэтому главный поток будет заблокирован до завершения работы сервера.

В листинге 20.19 показано определение функции `accept_forever`. Она является частью общей серверной библиотеки, которая находится в каталоге srvcore.

Подобное размещение объясняется тем, что ее определение используется другими потоковыми сокетами, включая работающие по TCP. Таким образом, мы можем повторно задействовать имеющуюся логику, вместо того чтобы создавать ее заново.

Листинг 20.19. Функция, принимающая новые клиентские запросы на потоковом сокете, который прослушивает конечную точку UDS (server/srvcore/stream_server_core.c)

```
void accept_forever(int server_sd) {
  while (1) {
    int client_sd = accept(server_sd, NULL, NULL);
    if (client_sd == -1) {
      close(server_sd);
      fprintf(stderr, "Could not accept the client: %s\n",
              strerror(errno));
      exit(1);
    }
    pthread_t client_handler_thread;
    int* arg = (int *)malloc(sizeof(int));
    *arg = client_sd;
    int result = pthread_create(&client_handler_thread, NULL,
            &client_handler, arg);
    if (result) {
      close(client_sd);
      close(server_sd);
      free(arg);
      fprintf(stderr, "Could not start the client handler thread.\n");
      exit(1);
    }
  }
}
```

В данном листинге видно, что в момент приема нового клиентского запроса мы создаем новый поток выполнения, который отвечает за работу с соединением. Это фактически сводится к чтению байтов из клиентского канала, передаче прочитанных байт десериализатору и генерации подходящих ответов при обнаружении запроса.

Создание нового потока выполнения для каждого клиента — стандартный подход, который обычно используется серверными процессами, работающими с блокирующим потоковым каналом, независимо от типа сокета. Поэтому в таких ситуациях важнейшую роль играет многопоточность и все связанные с ней темы.

Что касается неблокирующих потоковых каналов, то для них обычно используется другой подход, известный как цикл событий.

Полученный объект сокета можно использовать для чтения и записи на клиентской стороне. Если следовать принципу, по которому разрабатывалась библиотека

srvcore, то следующим шагом будет анализ функции-компаньона в клиентском потоке, client_handler. В исходных текстах эту функцию можно найти сразу за accept_forever. В листинге 20.20 показано ее определение.

Листинг 20.20. Функция-компаньон потока выполнения, который работает с клиентом (server/srvcore/stream_server_core.c)

```
void* client_handler(void *arg) {
  struct client_context_t context;

  context.addr = (struct client_addr_t*)
      malloc(sizeof(struct client_addr_t));
  context.addr->sd = *((int*)arg);
  free((int*)arg);

  context.ser = calc_proto_ser_new();
  calc_proto_ser_ctor(context.ser, &context, 256);
  calc_proto_ser_set_req_callback(context.ser, request_callback);
  calc_proto_ser_set_error_callback(context.ser, error_callback);

  context.svc = calc_service_new();
  calc_service_ctor(context.svc);

  context.write_resp = &stream_write_resp;

  int ret;
  char buffer[128];
  while (1) {
    int ret = read(context.addr->sd, buffer, 128);
    if (ret == 0 || ret == -1) {
      break;
    }
    struct buffer_t buf;
    buf.data = buffer; buf.len = ret;
    calc_proto_ser_server_deserialize(context.ser, buf, NULL);
  }

  calc_service_dtor(context.svc);
  calc_service_delete(context.svc);

  calc_proto_ser_dtor(context.ser);
  calc_proto_ser_delete(context.ser);

  free(context.addr);

  return NULL;
}
```

У этого кода есть много разных аспектов, но среди них я хочу выделить несколько особенно важных. Как вы сами можете видеть, для чтения блоков информации, передаваемой клиентом, используется функция read. Если помните, данная функция

принимает файловый дескриптор, однако здесь ей передается дескриптор сокета. Это демонстрация того, что, несмотря на разницу этих двух видов дескрипторов, для работы с ними можно использовать одни и те же функции ввода/вывода.

В этом коде мы читаем блоки байтов из ввода и передаем их десериализатору, вызывая функцию `calc_proto_ser_server_deserialize`. Прежде чем ответ полностью десериализуется, данную функцию, возможно, придется вызвать три или четыре раза. Это во многом зависит от размера блоков, которые вы читаете из ввода, и длины сообщений, передаваемых по каналу.

Вдобавок следует отметить: у каждого клиента есть свой объект-сериализатор. То же самое касается объекта сервиса. Эти объекты создаются и уничтожаются вместе со своим потоком выполнения.

Заключительным аспектом, на который стоит обратить внимание, является то, что ответы клиенту возвращаются с помощью функции `stream_write_response`, предназначенной для работы с потоковым сокетом. Эту функцию можно найти в том же файле, что и код из предыдущих листингов. В листинге 20.21 представлено ее определение.

Листинг 20.21. Функция, которая используется для возвращения ответов клиенту (server/srvcore/stream_server_core.c)

```
void stream_write_resp(
        struct client_context_t* context,
        struct calc_proto_resp_t* resp) {
  struct buffer_t buf =
      calc_proto_ser_server_serialize(context->ser, resp);
  if (buf.len == 0) {
    close(context->addr->sd);
    fprintf(stderr, "Internal error while serializing response\n");
    exit(1);
  }
  int ret = write(context->addr->sd, buf.data, buf.len);
  free(buf.data);
  if (ret == -1) {
    fprintf(stderr, "Could not write to client: %s\n",
            strerror(errno));
    close(context->addr->sd);
    exit(1);
  } else if (ret < buf.len) {
    fprintf(stderr, "WARN: Less bytes were written!\n");
    exit(1);
  }
}
```

В данном листинге видно, что для записи сообщения, которое возвращается клиенту, мы используем функцию `write`. Как мы уже знаем, она может принимать файловые дескрипторы, но, по всей видимости, ей можно передавать и дескрипторы

сокетов. Это наглядная демонстрация того, что API ввода/вывода POSIX совместим с обоими видами дескрипторов.

То же самое относится и к функции `close`. Мы использовали ее для разрыва соединения. Как известно, она умеет работать с файловыми дескрипторами, поэтому ей смело можно передавать дескрипторы сокетов.

Итак, мы прошлись по некоторым важнейшим аспектам потокового сервера UDS и получили общее представление о том, как он работает. Теперь пришло время перейти к обсуждению потокового клиента UDS. Конечно, многие участки кода остались без внимания, но вы можете проанализировать их самостоятельно.

Потоковый клиент на основе UDS

Как и серверная программа, описанная в предыдущем подразделе, клиент должен первым делом создать объект сокета. Вы помните, что мы должны выполнить последовательность шагов потокового соединителя. Здесь используется такой же фрагмент кода, что и на серверной стороне, включая те же аргументы, которые сигнализируют о работе с UDS. Дальше нам нужно подключиться к процессу сервера, указав конечную точку UDS, аналогично тому, как это сделал сервер. После создания потокового канала клиентский процесс может читать и записывать в него с помощью открытого дескриптора сокета.

В листинге 20.22 показана функция `main` потокового клиента, который соединяется с конечной точкой UDS.

Листинг 20.22. Главная функция потокового клиента, соединяющегося с конечной точкой UDS (client/unix/stream/main.c)

```
int main(int argc, char** argv) {
  char sock_file[] = "/tmp/calc_svc.sock";

  // ----------- 1. Создаем объект сокета ------------------

  int conn_sd = socket(AF_UNIX, SOCK_STREAM, 0);
  if (conn_sd == -1) {
    fprintf(stderr, "Could not create socket: %s\n",
            strerror(errno));
    exit(1);
  }

  // ----------- 2. Подключаемся к серверу ----------------------

  // Подготавливаем адрес
  struct sockaddr_un addr;
  memset(&addr, 0, sizeof(addr));
  addr.sun_family = AF_UNIX;
  strncpy(addr.sun_path, sock_file, sizeof(addr.sun_path) - 1);
```

```
  int result = connect(conn_sd,
          (struct sockaddr*)&addr, sizeof(addr));
  if (result == -1) {
    close(conn_sd);
    fprintf(stderr, "Could no connect: %s\n", strerror(errno));
    exit(1);
  }

  stream_client_loop(conn_sd);

  return 0;
}
```

Первая часть этого листинга очень похожа на код сервера, но дальше вместо `bind` клиент вызывает `connect`. Обратите внимание: код подготовки адреса ничем не отличается от того, который используется сервером.

Успешное возвращение вызова `connect` означает, что дескриптор сокета `conn_sd` был привязан к открытому каналу. С этого момента `conn_sd` можно использовать для взаимодействия с сервером. Мы передаем даный дескриптор функции `stream_client_loop`, которая выводит командную строку клиента и выполняет все остальные действия, которые будут инициированы клиентской стороной. Это блокирующая функция, выполняющаяся, пока клиент не завершит работу.

Клиент также использует функции `read` и `write` для передачи и получения сообщений от сервера. Листинг 20.23 содержит определение функции `stream_client_loop`, которая входит в состав общей клиентской библиотеки и используется всеми потоковыми клиентами, независимо от типа сокета — UDS или TCP. Как видите, она вызывает функцию `write` для отправки серверу сообщения с сериализованным запросом.

Листинг 20.23. Функция, выполняющая потоковый клиент (client/clicore/stream_client_core.c)

```
void stream_client_loop(int conn_sd) {
  struct context_t context;

  context.sd = conn_sd;
  context.ser = calc_proto_ser_new();
  calc_proto_ser_ctor(context.ser, &context, 128);
  calc_proto_ser_set_resp_callback(context.ser, on_response);
  calc_proto_ser_set_error_callback(context.ser, on_error);

  pthread_t reader_thread;
  pthread_create(&reader_thread, NULL,
          stream_response_reader, &context);

  char buf[128];
  printf("? (type quit to exit) ");
  while (1) {
```

```
    scanf("%s", buf);
    int brk = 0, cnt = 0;
    struct calc_proto_req_t req;
    parse_client_input(buf, &req, &brk, &cnt);
    if (brk) {
      break;
    }
    if (cnt) {
      continue;
    }
    struct buffer_t ser_req =
        calc_proto_ser_client_serialize(context.ser, &req);
    int ret = write(context.sd, ser_req.data, ser_req.len);
    if (ret == -1) {
      fprintf(stderr, "Error while writing! %s\n",
              strerror(errno));
      break;
    }
    if (ret < ser_req.len) {
      fprintf(stderr, "Wrote less than anticipated!\n");
      break;
    }
    printf("The req(%d) is sent.\n", req.id);
  }
  shutdown(conn_sd, SHUT_RD);
  calc_proto_ser_dtor(context.ser);
  calc_proto_ser_delete(context.ser);
  pthread_join(reader_thread, NULL);
  printf("Bye.\n");
}
```

Данный код демонстрирует, что все клиентские процессы имеют один общий объект-сериализатор, и это логично. Для сравнения, на серверной стороне у каждого клиента был свой сериализатор.

Более того, клиентский процесс создает новый поток выполнения для чтения ответов, которые присылает сервер. Это вызвано тем, что чтение из серверного процесса — блокирующая операция, поэтому ее следует выполнять в отдельном потоке.

В рамках главного потока мы выводим командную строку клиента, которая принимает пользовательский ввод через терминал. Во время завершения работы главный поток присоединяет поток чтения и ждет, когда тот завершится.

В данном листинге также следует обратить внимание на то, что клиентский процесс использует тот же API ввода/вывода для чтения и записи в потоковый канал. Как уже говорилось ранее, для этого предусмотрены функции `read` и `write`, и пример их использования показан в листинге 20.23.

В следующем подразделе мы поговорим о датаграммных каналах, но с применением все тех же сокетов UDS. Начнем с датаграммного сервера.

Датаграммный сервер на основе UDS

Возможно, из предыдущей главы вы помните, что датаграммные процессы, слушатель и соединитель, имеют собственные последовательности действий по передаче данных. Пришло время показать, как может выглядеть датаграммный сервер на основе UDS.

Согласно последовательности датаграммного слушателя, процесс должен сначала создать объект сокета. Это показано в листинге 20.24.

Листинг 20.24. Создание объекта UDS, предназначенного для работы с датаграммным каналом (server/unix/datagram/main.c)

```
int server_sd = socket(AF_UNIX, SOCK_DGRAM, 0);
if (server_sd == -1) {
  fprintf(stderr, "Could not create socket: %s\n",
          strerror(errno));
  exit(1);
}
```

Вместо `SOCK_STREAM` мы используем `SOCK_DGRAM`. Это значит, объект сокета будет работать с датаграммным каналом. Остальные два аргумента остаются без изменений.

Второй шаг в последовательности датаграммного слушателя состоит в привязке сокета к конечной точке UDS. Как уже говорилось ранее, это файл сокета. Данный шаг ничем не отличается от того, который мы выполнили в потоковом сервере, и потому я не стану его здесь приводить. Можете взглянуть на него в листинге 20.15.

Это все действия, которые выполняет слушающий датаграммный процесс; датаграммному сокету не назначается очередь отставания. Более того, здесь нет этапа приема клиентских запросов, поскольку у нас не может быть клиентских соединений с выделенным каналом между двумя процессами.

В листинге 20.25 показана функция `main` датаграммного сервера, который прослушивает конечную точку UDS в рамках проекта «Калькулятор».

Листинг 20.25. Главная функция датаграммного сервера, который прослушивает конечную точку UDS (server/unix/datagram/main.c)

```
int main(int argc, char** argv) {
  char sock_file[] = "/tmp/calc_svc.sock";

  // ----------- 1. Создаем объект сокета ------------------
  int server_sd = socket(AF_UNIX, SOCK_DGRAM, 0);
  if (server_sd == -1) {
    fprintf(stderr, "Could not create socket: %s\n",
```

```
            strerror(errno));
    exit(1);
  }

  // ----------- 2. Привязываем файл сокета ------------------

  // Удаляем ранее созданный файл сокета, если таковой существует
  unlink(sock_file);

  // Подготавливаем адрес
  struct sockaddr_un addr;
  memset(&addr, 0, sizeof(addr));
  addr.sun_family = AF_UNIX;
  strncpy(addr.sun_path, sock_file, sizeof(addr.sun_path) - 1);

  int result = bind(server_sd,
          (struct sockaddr*)&addr, sizeof(addr));
  if (result == -1) {
    close(server_sd);
    fprintf(stderr, "Could not bind the address: %s\n",
            strerror(errno));
    exit(1);
  }

  // ----------- 3. Начинаем обслуживать запросы ---------
  serve_forever(server_sd);

  return 0;
}
```

Вы уже знаете, что датаграммные каналы не поддерживают соединения и работают не так, как потоковые каналы. Иными словами, мы не можем установить выделенное соединение между двумя процессами. Поэтому процессы передают по каналу только отдельные фрагменты данных. Клиент отправляет отдельные и независимые друг от друга датаграммы, а сервер их принимает и в свою очередь возвращает другие датаграммы в качестве ответа.

Таким образом, важнейшим аспектом датаграммного канала является то, что сообщение с запросом или ответом должно влезать в одну датаграмму. В противном случае его нельзя разделить между двумя датаграммами, и ни сервер, ни клиент не смогут его обработать. К счастью, сообщения в нашем проекте в основном достаточно короткие.

Размер датаграммы во многом зависит от канала, по которому она проходит. Например, датаграммы UDS являются довольно гибкими, поскольку проходят через ядро. А вот при использовании UDP-сокетов все зависит от конфигурации сети. Что касается UDS, то информация, доступная по следующей ссылке, более подробно объясняет, как выбрать корректный размер: https://stackoverflow.com/questions/21856517/whats-the-practical-limit-on-the-size-of-single-packet-transmitted-over-domain.

Еще одно различие между датаграммными и потоковыми сокетами, заслуживающее внимания, состоит в том, что для передачи данных по ним используются разные API ввода/вывода. Для работы с датаграммным сокетом, как и с потоковым, можно применять операции `read` и `write`, однако для чтения и отправки данных в датаграммный канал мы обычно используем другие функции: `recvfrom` и `sendto`.

Это вызвано тем, что потоковые сокеты имеют выделенный канал и при записи в него мы знаем, что находится на обоих его концах. Касательно датаграммных сокетов, один и тот же канал используется множеством разных сторон. Упомянутые выше функции способны запомнить нужный процесс и отправить ему датаграмму.

Ниже вы можете видеть определение функции `serve_forever`, которая использовалась в конце функции `main` в листинге 20.25. Она входит в состав общей серверной библиотеки и предназначена для датаграммных серверов, независимо от типа сокета. В листинге 20.26 наглядно показано, как работает операция `recvfrom`.

Листинг 20.26. Функция для обработки датаграмм, принадлежащая общей серверной библиотеке и предназначенная для датаграммных серверов (server/srvcore/datagram_server_core.c)

```
void serve_forever(int server_sd) {
  char buffer[64];
  while (1) {
    struct sockaddr* sockaddr = sockaddr_new();
    socklen_t socklen = sockaddr_sizeof();
    int read_nr_bytes = recvfrom(server_sd, buffer,
            sizeof(buffer), 0, sockaddr, &socklen);
    if (read_nr_bytes == -1) {
      close(server_sd);
      fprintf(stderr, "Could not read from datagram socket: %s\n",
              strerror(errno));
      exit(1);
    }
    struct client_context_t context;
    context.addr = (struct client_addr_t*)
        malloc(sizeof(struct client_addr_t));
    context.addr->server_sd = server_sd;
    context.addr->sockaddr = sockaddr;
    context.addr->socklen = socklen;

    context.ser = calc_proto_ser_new();
    calc_proto_ser_ctor(context.ser, &context, 256);
    calc_proto_ser_set_req_callback(context.ser, request_callback);
    calc_proto_ser_set_error_callback(context.ser, error_callback);

    context.svc = calc_service_new();
    calc_service_ctor(context.svc);

    context.write_resp = &datagram_write_resp;
```

```
    bool_t req_found = FALSE;
    struct buffer_t buf;
    buf.data = buffer;
    buf.len = read_nr_bytes;
    calc_proto_ser_server_deserialize(context.ser, buf, &req_found);

    if (!req_found) {
      struct calc_proto_resp_t resp;
      resp.req_id = -1;
      resp.status = ERROR_INVALID_RESPONSE;
      resp.result = 0.0;
      context.write_resp(&context, &resp);
    }

    calc_service_dtor(context.svc);
    calc_service_delete(context.svc);

    calc_proto_ser_dtor(context.ser);
    calc_proto_ser_delete(context.ser);

    free(context.addr->sockaddr);
    free(context.addr);
  }
}
```

Как показано в этом листинге, датаграммный сервер представляет собой программу с одним потоком выполнения и без какой-либо многопоточности. Более того, данная программа работает с каждой датаграммой отдельно и независимо. Она получает датаграмму, десериализует ее содержимое, создает объект запроса, обрабатывает запрос с помощью объекта сервиса, сериализует объект ответа, помещает его в новую датаграмму и отправляет процессу, который послал исходный запрос. Эта процедура повторяется по кругу снова и снова для каждой входящей датаграммы.

Отмечу: у каждой датаграммы есть собственные объект-сериализатор и объект сервиса. Мы могли бы спроектировать приложение так, чтобы для всех датаграмм использовались одни и те же сериализатор и сервис. Подумайте, благодаря чему это возможно и почему такой подход может оказаться несовместимым с проектом «Калькулятор». Это спорное решение, и у разных людей могут быть разные мнения на сей счет.

Обратите внимание: в листинге 20.26 при получении датаграммы мы сохраняем ее клиентский адрес. Позже данный адрес можно использовать для записи напрямую в клиентский процесс. Вам стоит ознакомиться с тем, как датаграмма возвращается исходному клиенту. Как и в случае с потоковым сервером, мы используем для этого функцию. В листинге 20.27 показано определение функции `datagram_write_resp`, которая находится в общей библиотеке датаграммного сервера сразу за функцией `serve_forever`.

Листинг 20.27. Функция, возвращающая датаграммы клиентам (server/srvcore/datagram_server_core.c)

```
void datagram_write_resp(struct client_context_t* context,
        struct calc_proto_resp_t* resp) {
  struct buffer_t buf =
      calc_proto_ser_server_serialize(context->ser, resp);
  if (buf.len == 0) {
    close(context->addr->server_sd);
    fprintf(stderr, "Internal error while serializing object.\n");
    exit(1);
  }
  int ret = sendto(context->addr->server_sd, buf.data, buf.len,
      0, context->addr->sockaddr, context->addr->socklen);
  free(buf.data);
  if (ret == -1) {
    fprintf(stderr, "Could not write to client: %s\n",
            strerror(errno));
    close(context->addr->server_sd);
    exit(1);
  } else if (ret < buf.len) {
    fprintf(stderr, "WARN: Less bytes were written!\n");
    close(context->addr->server_sd);
    exit(1);
  }
}
```

Как вы можете видеть, мы берем адрес клиента и передаем его функции `sendto` вместе с сериализованным ответом. Все остальное делает за нас операционная система, в результате чего датаграмма возвращается клиенту, который послал запрос.

Мы уже имеем достаточно хорошее представление о датаграммном сервере и о том, как следует использовать сокеты. Теперь обратим внимание на датаграммный клиент, который основан на сокетах того же типа.

Датаграммный клиент на основе UDS

С технической точки зрения потоковые и датаграммные клиенты очень похожи. Это значит, что их общая структура должна быть почти идентичной, а различия будут связаны с тем, что мы передаем датаграммы, вместо того чтобы работать с потоковым каналом.

Однако существует одна важная особенность, довольно уникальная и присущая датаграммным клиентам, подключающимся к конечным точкам UDS.

Дело в том, что датаграммный клиент, как и серверная программа, обязан привязаться к файлу сокета, чтобы получать направляемые ему датаграммы. Но, как вы вскоре увидите, это не относится к датаграммным клиентам, которые используют

сетевые сокеты. Отмечу, что клиент и сервер должны привязываться к разным файлам сокетов.

Главная причина этого различия состоит в том, что серверной программе нужен адрес, по которому можно вернуть ответ, и если датаграммный клиент не привязан к файлу сокета, то данный файл не будет иметь никакого отношения к конечной точке. Если же говорить о сетевых сокетах, то у клиента всегда есть соответствующий дескриптор, привязанный к IP-адресу и порту, и потому подобной проблемы не возникает.

Если не считать этих различий, то код в целом выглядит довольно похоже. В листинге 20.28 вы можете видеть функцию `main` датаграммного клиента.

Листинг 20.28. Функция, возвращающая датаграммы клиентам (server/srvcore/datagram_server_core.)

```
int main(int argc, char** argv) {
  char server_sock_file[] = "/tmp/calc_svc.sock";
  char client_sock_file[] = "/tmp/calc_cli.sock";

  // ----------- 1. Создаем объект сокета ------------------

  int conn_sd = socket(AF_UNIX, SOCK_DGRAM, 0);
  if (conn_sd == -1) {
    fprintf(stderr, "Could not create socket: %s\n",
            strerror(errno));
    exit(1);
  }

  // ----------- 2. Привязываем файл клиентского сокета ------------
  // Удаляем ранее созданный файл сокета, если таковой существует
  unlink(client_sock_file);

  // Подготавливаем клиентский адрес
  struct sockaddr_un addr;
  memset(&addr, 0, sizeof(addr));
  addr.sun_family = AF_UNIX;
  strncpy(addr.sun_path, client_sock_file,
          sizeof(addr.sun_path) - 1);

  int result = bind(conn_sd,
          (struct sockaddr*)&addr, sizeof(addr));
  if (result == -1) {
    close(conn_sd);
    fprintf(stderr, "Could not bind the client address: %s\n",
            strerror(errno));
    exit(1);
  }
```

```
  // ----------- 3. Подключаемся к серверу --------------------

  // Подготавливаем серверный адрес
  memset(&addr, 0, sizeof(addr));
  addr.sun_family = AF_UNIX;
  strncpy(addr.sun_path, server_sock_file,
          sizeof(addr.sun_path) - 1);

  result = connect(conn_sd,
          (struct sockaddr*)&addr, sizeof(addr));
  if (result == -1) {
    close(conn_sd);
    fprintf(stderr, "Could no connect: %s\n", strerror(errno));
    exit(1);
  }

  datagram_client_loop(conn_sd);

  return 0;
}
```

Как уже объяснялось ранее и как мы можем видеть в данном листинге, клиент обязан привязаться к файлу сокета. И конечно, чтобы начать клиентский цикл, в конце `main` следует вызвать другую функцию. В данном случае это `datagram_client_loop`. В ней по-прежнему много общего между потоковым и датаграммным клиентами. Основное различие заключается в использовании функций `recvfrom` и `sendto` вместо `read` и `write`. Объяснение, которое приводилось в предыдущем подразделе, актуально и для датаграммного клиента.

Теперь пришло время поговорить о сетевых сокетах. Как вы увидите, все различия при переходе с UDS на сетевые сокеты будут находиться в функциях `main` клиентской и серверной программ.

Сетевые сокеты

Еще одно широко используемое семейство адресов сокетов — `AF_INET`. К нему относятся любые каналы, создаваемые поверх сетевого соединения. В отличие от потоковых и датаграммных сокетов UDS, не имеющих никаких протоколов, для сетевых сокетов существует два общеизвестных протокола. TCP-сокеты создают потоковый канал между двумя процессами, а UDP-сокеты — датаграммный канал, который может применять любое количество процессов.

В следующих разделах мы увидим, как разрабатываются программы на основе TCP- и UDP-сокетов, и рассмотрим реальные примеры в рамках проекта «Калькулятор».

TCP-сервер

Программа, использующая TCP-сокет для прослушивания и приема разных запросов (то есть TCP-сервер), отличается от потокового сервера, который прослушивает конечную точку UDS. Этих отличий два: во-первых, при вызове функции `socket` указывается другое семейство адресов, `AF_INET` вместо `AF_UNIX`, и, во-вторых, адрес, который она использует для привязки, имеет другую структуру.

В остальном, если говорить об операциях ввода/вывода, то TCP-сокет ведет себя так же, как и UDS. Следует отметить, что TCP-сокет является потоковым, поэтому для него должен подойти код, рассчитанный на потоковые сокеты домена Unix.

Возвращаясь к проекту «Калькулятор», все отличия следует искать в функциях `main`, где мы создаем объект сокета и привязываем его к конечной точке. Остальной код должен остаться без изменений. В листинге 20.29 вы можете видеть функцию `main`, принадлежащую TCP-серверу.

Листинг 20.29. Главная функция TCP-сервера (server/tcp/main.c)

```
int main(int argc, char** argv) {

  // ----------- 1. Создаем объект сокета ------------------
  int server_sd = socket(AF_INET, SOCK_STREAM, 0);
  if (server_sd == -1) {
    fprintf(stderr, "Could not create socket: %s\n",
            strerror(errno));
    exit(1);
  }

  // ----------- 2. Привязываем файл сокета ------------------

  // Подготавливаем адрес
  struct sockaddr_in addr;
  memset(&addr, 0, sizeof(addr));
  addr.sin_family = AF_INET;
  addr.sin_addr.s_addr = INADDR_ANY;
  addr.sin_port = htons(6666);

  ...

  // ----------- 3. Подготавливаем резерв ------------------
  ...

  // ----------- 4. Начинаем принимать клиентов ---------
  accept_forever(server_sd);

  return 0;
}
```

Если сравнить этот код с функцией `main`, которую мы видели в листинге 20.17, то можно заметить упомянутые ранее отличия. Для привязанного адреса конеч-

ной точки теперь применяется структура `sockaddr_in`, а не `sockaddr_un`. Функция `listen` используется тем же образом, а для обработки входящих соединений вызывается та же функция `accept_forever`.

В завершение отмечу кое-что относительно операций ввода/вывода: будучи потоковым, TCP-сокет обладает теми же свойствами; следовательно, его можно использовать так же, как и любой другой потоковый сокет. Иными словами, мы можем вызывать все те же функции `read`, `write` и `close`.

Теперь обсудим TCP-клиент.

TCP-клиент

Здесь тоже все должно быть похоже на потоковый клиент, работающий с UDS. Отличия, упомянутые выше, актуальны и для TCP-сокета на стороне соединителя и ограничены функцией `main`.

В листинге 20.30 вы можете видеть главную функцию TCP-клиента.

Листинг 20.30. Главная функция TCP-клиента (client/tcp/main.c)

```
int main(int argc, char** argv) {

  // ----------- 1. Создаем объект сокета ------------------

  int conn_sd = socket(AF_INET, SOCK_STREAM, 0);
  if (conn_sd == -1) {
    fprintf(stderr, "Could not create socket: %s\n",
            strerror(errno));
    exit(1);
  }

  // ------------ 2. Подключаемся к серверу --------------------

  // Находим IP-адрес, которому принадлежит это сетевое имя
  ...

  // Подготавливаем адрес
  struct sockaddr_in addr;
  memset(&addr, 0, sizeof(addr));
  addr.sin_family = AF_INET;
  addr.sin_addr = *((struct in_addr*)host_entry->h_addr);
  addr.sin_port = htons(6666);

  ...

  stream_client_loop(conn_sd);

  return 0;
}
```

Изменения очень похожи на те, которые мы видели в TCP-сервере. Здесь используется другое семейство адресов и другая структура для хранения адреса сокета. Остальной код не претерпел изменений, поэтому подробно обсуждать TCP-клиент нет нужды.

Поскольку TCP-сокеты потоковые, тот же общий код позволяет обрабатывать новые клиентские запросы. Это можно видеть на примере функции `stream_client_loop`, которая является частью общей клиентской библиотеки в проекте «Калькулятор». Теперь вы знаете, зачем мы создали две общие библиотеки: одну для клиентской программы, а вторую для серверной. Это было сделано для того, чтобы сократить объем кода. Если один и тот же код подходит для двух разных ситуаций, то его всегда лучше вынести в библиотеку, которую можно будет использовать повторно.

Рассмотрим серверную и клиентскую UDP-программы; как вы сами сможете убедиться, они более или менее похожи на то, что мы уже видели в TCP-программах.

UDP-сервер

UPD-сокеты являются сетевыми и датаграммными. Поэтому мы можем ожидать высокой степени сходства с кодом, написанным для TCP-сервера и для датаграммного сервера, который использовал UDS.

Кроме того, главное различие между UDP- и TCP-сокетами, независимо от того, в какой программе они применяются: клиентской или серверной, состоит в том, что UDP-сокет имеет тип `SOCK_DGRAM`. Семейство адресов остается тем же, поскольку оба вида сокетов являются сетевыми. Листинг 20.31 содержит главную функцию UDP-сервера.

Листинг 20.31. Главная функция UDP-сервера (server/udp/main.c)

```
int main(int argc, char** argv) {

  // ----------- 1. Создаем объект сокета ------------------
  int server_sd = socket(AF_INET, SOCK_DGRAM, 0);
  if (server_sd == -1) {
    fprintf(stderr, "Could not create socket: %s\n",
            strerror(errno));
    exit(1);
  }

  // ----------- 2. Привязываем файл сокета ------------------

  // Подготавливаем адрес
  struct sockaddr_in addr;
  memset(&addr, 0, sizeof(addr));
  addr.sin_family = AF_INET;
  addr.sin_addr.s_addr = INADDR_ANY;
  addr.sin_port = htons(9999);
```

```
  ...

  // ----------- 3. Начинаем обслуживать запросы ---------
  serve_forever(server_sd);

  return 0;
}
```

Обратите внимание: UDP-сокеты являются датаграммными. Поэтому весь код, написанный для датаграммных сокетов домена Unix, актуален и для них. Например, работа с UDP-сокетами требует применения функций `recvfrom` и `sendto`. В связи с этим для обслуживания входящих датаграмм была использована та же функция `serve_forever`. Она входит в состав общей серверной библиотеки, в которой собран код, относящийся к датаграммам.

О коде UDP-сервера было сказано достаточно. Посмотрим, как выглядит UDP-клиент.

UDP-клиент

Код UDP- и TCP-клиентов очень похож, однако они используют сокеты разных типов и различные функции для обработки входящих сообщений; в UDP-клиенте задействована функция из датаграммного клиента, основанного на UDS. В листинге 20.32 вы можете видеть функцию `main`.

Листинг 20.32. Главная функция UDP-клиента (client/udp/main.c)

```
int main(int argc, char** argv) {

  // ----------- 1. Создаем объект сокета ------------------

  int conn_sd = socket(AF_INET, SOCK_DGRAM, 0);
  if (conn_sd == -1) {
    fprintf(stderr, "Could not create socket: %s\n",
            strerror(errno));
    exit(1);
  }

  // ------------ 2. Подключаемся к серверу --------------------
  ...

  // Подготавливаем адрес
  ...

  datagram_client_loop(conn_sd);

  return 0;
}
```

Это была последняя концепция данной главы. Мы рассмотрели разные общеизвестные типы сокетов и показали, как на языке C реализовать последовательности действий слушателя и соединителя в контексте потоковых и датаграммных каналов.

Проект «Калькулятор» имеет много аспектов, о которых я даже не упомянул. Поэтому настоятельно рекомендую вам пройтись по коду, найти эти места и попытаться их понять. Наличие полностью рабочего примера поможет вам исследовать концепции, которые можно встретить в реальных приложениях.

Резюме

В этой главе мы:

- в рамках обзора методов IPC познакомились с различными типами взаимодействия, каналов, носителей и сокетов;
- исследовали проект «Калькулятор», рассмотрев его прикладной протокол и алгоритм сериализации, который он использует;
- увидели, как с помощью сокетов UDS установить клиент-серверное соединение и как их задействовать в проекте «Калькулятор»;
- по очереди обсудили потоковые и датаграммные каналы, создаваемые с применением сокетов домена Unix;
- рассмотрели, как с помощью TCP- и UDP-сокетов, которые использовались в примере с калькулятором, можно создать клиент-серверный канал межпроцессного взаимодействия.

Следующая глава посвящена интеграции C с другими языками программирования. Такая интеграция позволяет загрузить библиотеку, написанную на C, в среду выполнения другого языка, такого как Java. В рамках следующей главы мы также поговорим об интеграции с C++, Java, Python и Golang.

21 Интеграция с другими языками

Навыки написания программ и библиотек на языке C могут оказаться еще полезнее, чем можно было бы ожидать. Благодаря важной роли в разработке операционных систем язык C проникает в другие области. Написанные на нем библиотеки потенциально могут загружаться и применяться в других языках программирования. Пользуясь преимуществами высокоуровневых языков, вы можете задействовать в их средах эффективный и быстрый код, написанный на C.

В данной главе подробно это обсудим и покажем, как интегрировать разделяемые библиотеки C с некоторыми популярными языками программирования.

Мы рассмотрим следующие ключевые темы:

- выясним, благодаря чему возможна интеграция как таковая. Это даст вам общее представление о том, как интегрируются разные языки;
- создадим библиотеку для работы со стеком на C и соберем ее в виде разделяемого объектного файла. Он будет использоваться рядом других языков программирования;
- пройдемся по C++, Java, Python и Golang и рассмотрим загрузку и использование библиотеки для работы со стеком.

В целом следует отметить, что в данной главе мы будем работать с пятью разными подпроектами, написанными на разных языках, поэтому в целях предотвращения любых проблем с их сборкой и выполнением все действия будут выполняться в Linux. Я предоставлю достаточно информации и для macOS, но основное внимание будет уделяться сборке и запуску исходников в Linux. В репозитории GitHub для этой книги доступны дополнительные скрипты, которые помогут вам собрать исходники в macOS.

В первом разделе речь пойдет об интеграции с другими языками как таковой. Вы увидите, что делает ее возможной. Это заложит фундамент для дальнейшего обсуждения в контексте других сред выполнения за пределами C.

Что делает интеграцию возможной

Как уже объяснялось в главе 10, язык C кардинально изменил разработку операционных систем. Но, помимо этого, он также сделал возможным создание поверх него других языков программирования общего назначения. В наши дни мы называем их высокоуровневыми. В большинстве своем компиляторы этих языков написаны на C, а остальные разработаны с помощью других инструментов и компиляторов, которые тоже написаны на C.

Язык программирования общего назначения, неспособный применять или предоставлять возможности операционной системы, совершенно бесполезен. Вы можете писать на нем, однако написанное нельзя будет выполнить ни в одной ОС. Такой язык можно задействовать в теоретических целях, но в промышленном смысле он ни на что не годится. Поэтому языки, и особенно их компиляторы, должны уметь генерировать рабочие программы. Как вы уже знаете, возможности компьютера предоставляются с помощью операционной системы. И, какой бы та ни была, они должны быть доступны в языке, чтобы написанная на нем программа, которая выполняется в этой системе, могла использовать их.

И здесь на помощь приходит C. Возможности Unix-подобных операционных систем доступны в виде API, который предоставляет стандартная библиотека C. Если компилятор хочет создать рабочую программу, то должен дать возможность скомпилированной программе работать со стандартной библиотекой C опосредованным образом. Независимо от языка и наличия в нем скомпилированной стандартной библиотеки, когда написанная на нем программа обращается к той или иной функции (такой как открытие файла), этот запрос должен быть передан стандартной библиотеке C, откуда он может быть направлен на выполнение в ядро. Примером этого служит язык Java, предоставляющий пакет *Java Standard Edition (Java SE)*.

Раз уж мы заговорили о Java, программы, написанные на этом языке, компилируются в промежуточный код, который называется *байт-кодом*. Его выполнение требует установки *среды выполнения Java* (Java Runtime Environment, JRE). В основе JRE лежит виртуальная машина (ВМ), которая загружает байт-код и выполняет его внутри себя. Эта ВМ должна уметь имитировать возможности и сервисы, реализованные в стандартной библиотеке C, и предоставлять их программам, которые выполняются внутри нее. Поскольку каждая платформа имеет собственную реализацию стандартной библиотеки C и по-своему поддерживает стандарты POSIX и SUS, для каждой операционной системы должна быть собрана своя виртуальная машина.

В заключение отмечу: в качестве библиотек, которые загружаются в другие языки, могут выступать только разделяемые объектные файлы. Статическую библиотеку загрузить не получится, поскольку ее можно скомпоновать только с исполняемым или разделяемым объектным файлом. В Unix-подобных системах разделяемые объектные файлы имеют расширение `.so`, а в macOS — `.dylib`.

В этом коротком разделе я попытался дать вам базовое понимание того, зачем нужна возможность загружать библиотеки C (в частности, разделяемые библиотеки), и показать, как они используются в большинстве языков программирования, поскольку многие из них поддерживают загрузку и применение разделяемых объектных файлов.

Дальше мы напишем библиотеку C и загрузим ее в разные языки программирования для дальнейшего использования. Именно этим мы вскоре займемся, но сначала следует объяснить, как получить материалы, предусмотренные для данной главы, и выполнить команды, показанные в терминалах.

Получение необходимых материалов

В данной главе полно ресурсов из разных языков программирования, и я хочу, чтобы вы могли собрать и запустить все примеры. Поэтому я выделил данную главу для перечисления некоторых основных моментов, которые следует учитывать при сборке исходного кода.

Прежде всего вам нужно получить материалы, подготовленные для этой главы. Как вам уже должно быть известно, у нее есть репозиторий, поделенный на главы. Каталог этой главы называется ch21-integration-with-other-languages. В терминале 21.1 показаны команды, которые позволят вам клонировать данный репозиторий и перейти в корень текущей главы.

Терминал 21.1. Клонирование репозитория GitHub этой книги и переход в корневой каталог главы

```
$ git clone https://github.com/PacktPublishing/Extreme-C.git
...
$ cd Extreme-C/ch21-integration-with-other-languages
$
```

Что касается терминалов, представленных в этой главе, то мы исходим из того, что команды выполняются в корневом каталоге ch21-integration-with-other-languages. Если нам нужно перейти в другое место, то мы предоставим соответствующие команды, но все происходит внутри каталога главы.

Кроме того, в целях сборки исходного кода на вашем компьютере должны быть установлены *Java Development Kit (JDK)*, Python и Golang. В зависимости от вашей операционной системы (Linux или macOS) и дистрибутива Linux, команды установки могут отличаться.

В заключение отмечу: у исходного кода, написанного на других языках, должна быть возможность использовать библиотеку для работы со стеком на C, которая будет рассмотрена в следующем разделе. Она уже должна быть готова на момент

сборки исходников. Поэтому, прежде чем идти дальше, обязательно прочитайте следующий раздел и соберите описанный в нем разделяемый объектный файл. Итак, вы узнали, как получить материалы данной главы. Теперь можно переходить к обсуждению нашей библиотеки на языке C.

Библиотека для работы со стеком

В этом разделе мы разработаем небольшую библиотеку, которая будет загружаться и использоваться программами, написанными на других языках помимо C. В основе библиотеки лежит класс `Stack`, который предоставляет базовые операции для работы с объектами в стеке, такие как *push* и *pop*. Данные объекты создаются и уничтожаются самой библиотекой; для этого предусмотрены конструктор и деструктор.

В листинге 21.1 вы можете видеть публичный интерфейс библиотеки, который находится в заголовочном файле `cstack.h`.

Листинг 21.1. Публичный интерфейс библиотеки Stack (cstack.h)

```
#ifndef _CSTACK_H_
#define _CSTACK_H_

#include <unistd.h>

#ifdef __cplusplus
extern "C" {
#endif

#define TRUE 1
#define FALSE 0

typedef int bool_t;

typedef struct {
  char* data;
  size_t len;
} value_t;

typedef struct cstack_type cstack_t;

typedef void (*deleter_t)(value_t* value);

value_t make_value(char* data, size_t len);
value_t copy_value(char* data, size_t len);
void free_value(value_t* value);

cstack_t* cstack_new();
void cstack_delete(cstack_t*);
```

```
// Функции поведения
void cstack_ctor(cstack_t*, size_t);
void cstack_dtor(cstack_t*, deleter_t);

size_t cstack_size(const cstack_t*);

bool_t cstack_push(cstack_t*, value_t value);
bool_t cstack_pop(cstack_t*, value_t* value);

void cstack_clear(cstack_t*, deleter_t);

#ifdef __cplusplus
}
#endif

#endif
```

Как уже объяснялось в главе 6, эти объявления составляют публичный интерфейс класса `Stack`. Роль структуры атрибутов данного класса играет `cstack_t`. Я выбрал это название, поскольку `stack_t` уже используется в стандартной библиотеке C, и я предпочитаю избегать неясности в данном коде. Для структуры атрибутов применяется предварительное объявление, поэтому она не содержит никаких полей. Все детали будут предоставлены в исходном файле, который послужит ее реализацией. У этого класса также есть конструктор, деструктор и ряд операций, таких как push и pop. Все они принимают в качестве первого аргумента указатель типа `cstack_t`, обозначающий объект, с которым работают. Этот подход к написанию класса `Stack` был рассмотрен в главе 6, в момент обсуждения *неявной инкапсуляции*.

В листинге 21.2 показана реализация класса `Stack`. Она также содержит определение структуры атрибутов `cstack_t`.

Листинг 21.2. Определение класса Stack (cstack.c)

```
#include <stdlib.h>
#include <assert.h>

#include "cstack.h"

struct cstack_type {
  size_t top;
  size_t max_size;
  value_t* values;
};

value_t copy_value(char* data, size_t len) {
  char* buf = (char*)malloc(len * sizeof(char));
  for (size_t i = 0; i < len; i++) {
    buf[i] = data[i];
```

```
  }
  return make_value(buf, len);
}

value_t make_value(char* data, size_t len) {
  value_t value;
  value.data = data;
  value.len = len;
  return value;
}

void free_value(value_t* value) {
  if (value) {
    if (value->data) {
      free(value->data);
      value->data = NULL;
    }
  }
}

cstack_t* cstack_new() {
  return (cstack_t*)malloc(sizeof(cstack_t));
}

void cstack_delete(cstack_t* stack) {
  free(stack);
}

void cstack_ctor(cstack_t* cstack, size_t max_size) {
  cstack->top = 0;
  cstack->max_size = max_size;
  cstack->values = (value_t*)malloc(max_size * sizeof(value_t));
}

void cstack_dtor(cstack_t* cstack, deleter_t deleter) {
  cstack_clear(cstack, deleter);
  free(cstack->values);
}

size_t cstack_size(const cstack_t* cstack) {
  return cstack->top;
}

bool_t cstack_push(cstack_t* cstack, value_t value) {
  if (cstack->top < cstack->max_size) {
    cstack->values[cstack->top++] = value;
    return TRUE;
  }
  return FALSE;
}
```

```
bool_t cstack_pop(cstack_t* cstack, value_t* value) {
  if (cstack->top > 0) {
    *value = cstack->values[--cstack->top];
    return TRUE;
  }
  return FALSE;
}

void cstack_clear(cstack_t* cstack, deleter_t deleter) {
  value_t value;
  while (cstack_size(cstack) > 0) {
    bool_t popped = cstack_pop(cstack, &value);
    assert(popped);
    if (deleter) {
      deleter(&value);
    }
  }
}
```

В этом определении видно, что каждый объект стека основан на массиве; кроме того, в стеке можно хранить любое значение. Соберем библиотеку и сгенерируем из нее разделяемый объектный файл. В следующих разделах данный файл будет загружаться другими языками программирования.

В терминале 21.2 показано, как из имеющихся исходных файлов создать разделяемую объектную библиотеку. Эти команды рассчитаны на Linux, и чтобы они работали в macOS, их необходимо немного изменить. Обратите внимание: они должны выполняться в корневом каталоге главы, как уже объяснялось ранее.

Терминал 21.2. Сборка библиотеки для работы со стеком и создание разделяемого объектного файла в Linux

```
$ gcc -c -g -fPIC cstack.c -o cstack.o
$ gcc -shared cstack.o -o libcstack.so
$
```

Стоит отметить: те же команды можно выполнить и в macOS, но для этого в системе должна быть доступна команда `gcc`, которая ссылается на компилятор `clang`. В противном случае данную библиотеку в macOS можно собрать, используя команды, показанные в терминале 21.3. Обратите внимание: при таком раскладе разделяемые объектные файлы будут иметь расширение `.dylib`.

Терминал 21.3. Сборка библиотеки для работы со стеком и создание разделяемого объектного файла в macOS

```
$ clang -c -g -fPIC cstack.c -o cstack.o
$ clang -dynamiclib cstack.o -o libcstack.dylib
$
```

Итак, мы получили объектный файл разделяемой библиотеки. Теперь можно написать программы на других языках, которые будут его загружать. Прежде чем демонстрировать загрузку и использование нашей библиотеки в других средах выполнения, необходимо написать несколько тестов, чтобы проверить ее функциональность. Код, показанный в листинге 21.3, создает стек, выполняет некоторые доступные операции и проверяет, соответствуют ли полученные результаты нашим ожиданиям.

Листинг 21.3. Код, проверяющий функциональность класса Stack (cstack_tests.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#include "cstack.h"

value_t make_int(int int_value) {
  value_t value;
  int* int_ptr = (int*)malloc(sizeof(int));
  *int_ptr = int_value;
  value.data = (char*)int_ptr;
  value.len = sizeof(int);
  return value;
}

int extract_int(value_t* value) {
  return *((int*)value->data);
}

void deleter(value_t* value) {
  if (value->data) {
    free(value->data);
  }
  value->data = NULL;
}

int main(int argc, char** argv) {
  cstack_t* cstack = cstack_new();
  cstack_ctor(cstack, 100);
  assert(cstack_size(cstack) == 0);

  int int_values[] = {5, 10, 20, 30};

  for (size_t i = 0; i < 4; i++) {
    cstack_push(cstack, make_int(int_values[i]));
  }
  assert(cstack_size(cstack) == 4);
```

```
  int counter = 3;
  value_t value;
  while (cstack_size(cstack) > 0) {
    bool_t popped = cstack_pop(cstack, &value);
    assert(popped);
    assert(extract_int(&value) == int_values[counter--]);
    deleter(&value);
  }
  assert(counter == -1);
  assert(cstack_size(cstack) == 0);

  cstack_push(cstack, make_int(10));
  cstack_push(cstack, make_int(20));
  assert(cstack_size(cstack) == 2);

  cstack_clear(cstack, deleter);
  assert(cstack_size(cstack) == 0);

  // Чтобы при вызове деструктора в стеке что-то было.
  cstack_push(cstack, make_int(20));

  cstack_dtor(cstack, deleter);
  cstack_delete(cstack);
  printf("All tests were OK.\n");
  return 0;
}
```

Мы использовали утверждения для проверки возвращаемых значений. В терминале 21.4 показан вывод этого кода после его сборки и выполнения в Linux. Не забывайте, что мы находимся в корневом каталоге главы.

Терминал 21.4. Сборка и запуск тестов библиотеки

```
$ gcc -c -g cstack_tests.c -o tests.o
$ gcc tests.o -L$PWD -lcstack -o cstack_tests.out
$ LD_LIBRARY_PATH=$PWD ./cstack_tests.out
All tests were OK.
$
```

При запуске итогового исполняемого файла, `cstack_tests.out`, мы устанавливаем переменную среды `LD_LIBRARY_PATH`, которая указывает на каталог с `libcstack.so`, поскольку программа должна найти и загрузить необходимые разделяемые библиотеки.

Как видно в этом терминале, все тесты были успешно пройдены. Это значит, с точки зрения функциональности наша библиотека работает должным образом. Было бы неплохо проверить нашу библиотеку еще и на предмет нефункциональных требований, таких как расход ресурсов или отсутствие утечек памяти.

В терминале 21.5 показано, как с помощью утилиты `valgrind` проверить выполняемые тесты на любые потенциальные утечки памяти.

Терминал 21.5. Выполнение тестов с использованием утилиты valgrind

```
$ LD_LIBRARY_PATH=$PWD valgrind --leak-check=full ./cstack_tests.
out
==31291== Memcheck, a memory error detector
==31291== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==31291== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==31291== Command: ./cstack_tests.out
==31291==
All tests were OK.
==31291==
==31291== HEAP SUMMARY:
==31291==     in use at exit: 0 bytes in 0 blocks
==31291==   total heap usage: 10 allocs, 10 frees, 2,676 bytes allocated
==31291==
==31291== All heap blocks were freed -- no leaks are possible
==31291==
==31291== For counts of detected and suppressed errors, rerun with: -v
==31291== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
$
```

Как видите, у нас нет никаких утечек памяти. Это позволяет нам чувствовать определенную уверенность по отношению к написанной нами библиотеке. Таким образом, поиск первопричин любых проблем с памятью следует начинать со среды выполнения, в которой загружена библиотека.

Тестирование на языке C будет рассмотрено в следующей главе. Вместо инструкции `assert`, которую вы видели в терминале 21.3, лучше использовать модульные тесты и выполнять их с помощью фреймворка модульного тестирования наподобие CMocka.

В следующих разделах мы интегрируем библиотеку для работы со стеком в программы, написанные на четырех языках. Начнем с C++.

Интеграция с C++

Интеграцию с C++ можно считать самой простой. C++ — своего рода объектно-ориентированное расширение C. Компиляторы этих языков генерируют похожие объектные файлы. Поэтому программе на C++ проще загружать разделяемые библиотеки C по сравнению с другими языками. Иными словами, программа на C++ способна в равной степени загружать объектные файлы, написанные на C и C++. Единственным проблемным моментом в некоторых случаях может быть *декорирование имен* в C++, описанное в главе 2. В следующем подразделе я напомню, что это такое.

Декорирование имен в C++

Если более подробно, то в языке C++ имена символов, относящихся к функциям (как к глобальным, так и к методам классов), декорируются. В основном так поддерживаются возможности наподобие *пространства имен* и *перегрузка функций*, которых нет в C. При сборке кода на языке C с использованием компилятора C++ данная функция включена по умолчанию, поэтому мы ожидаем увидеть декорированные имена символов. Взгляните на код в листинге 21.4.

Листинг 21.4. Простая функция на C (test.c)

```
int add(int a, int b) {
  return a + b;
}
```

Если собрать этот файл с помощью компилятора C (в данном случае `clang`), то в объектном файле можно будет увидеть символы, показанные в терминале 21.6. Обратите внимание: файла `test.c` нет в репозитории GitHub данной книги.

Терминал 21.6. Сборка файла test.c с использованием компилятора C

```
$ clang -c test.c -o test.o
$ nm test.o
0000000000000000 T _add
$
```

Как видите, мы имеем символ с именем `_add`, который ссылается на функцию `add`, определенную выше. Теперь соберем этот же файл с помощью компилятора C++ (в данном случае `clang++`) (терминал 21.7).

Терминал 21.7. Сборка файла test.c с помощью компилятора C++

```
$ clang++ -c test.c -o test.o
clang: warning: treating 'c' input as 'c++' when in C++ mode, this
behavior is deprecated [-Wdeprecated]
$ nm test.o
0000000000000000 T __Z3addii
$
```

Компилятор `clang++` сгенерировал предупреждение о том, что в ближайшем будущем поддержка компиляции кода на C так, будто он написан на C++, исчезнет. Но, поскольку данная возможность все еще есть (просто помечена как устаревшая), мы видим, что имя символа, сгенерированного для представленной выше функции, было декорировано и отличается от того, которое было получено при использовании `clang`. Это определенно может привести к проблемам на стадии компоновки, когда будет проводиться поиск определенного символа.

Чтобы устранить эту проблему, код на языке C необходимо завернуть в специальный блок, который не даст компилятору C++ декорировать имена символов. В результате при компиляции с помощью `clang` и `clang++` названия символов будут совпадать. В листинге 21.5 показан код — модифицированная версия листинга 21.4.

Листинг 21.5. Определение функции в специальном блоке языка C (test.c)

```
#ifdef __cplusplus
extern "C" {
#endif

int add(int a, int b) {
  return a + b;
}

#ifdef __cplusplus
}
#endif
```

Эта функция помещается в блок `extern "C" { ... }`, только если у нас уже определен макрос `__cplusplus`. Наличие последнего — признак того, что код собирается компилятором C++. Снова скомпилируем этот код с помощью `clang++` (терминал 21.8).

Терминал 21.8. Компиляция новой версии файла test.c с использованием `clang++`

```
$ clang++ -c test.c -o test.o
clang: warning: treating 'c' input as 'c++' when in C++ mode, this
behavior is deprecated [-Wdeprecated]
$ nm test.o
0000000000000000 T _add
$
```

В этот раз сгенерированный символ не декорирован. Что касается нашей библиотеки для работы со стеком, то, учитывая все вышесказанное, все объявления необходимо поместить в блок `extern "C" { ... }`. Вот почему вы можете видеть данный блок в листинге 21.1. Таким образом, при компоновке программы на C++ с нашей библиотекой символы можно будет найти внутри `libcstack.so` (или `libcstack.dylib`).

extern "C" — это спецификация компоновки. Подробности можно найти по следующим ссылкам:

- https://isocpp.org/wiki/faq/mixing-c-and-cpp;
- https://stackoverflow.com/questions/1041866/what-is-the-effect-of-extern-c-in-c.

Теперь пришло время написать код на C++, который будет использовать нашу библиотеку для работы со стеком. Как вы вскоре увидите, это простая интеграция.

Код на C++

Мы узнали, как отключить декорирование имен при использовании кода на языке C в проекте на C++. Теперь можем написать программу на C++, которая будет использовать библиотеку для работы со стеком. Для начала обернем нашу библиотеку в класс, который является главным составным компонентом объектно-ориентированной программы на C++. Доступ к возможностям библиотеки лучше предоставлять в объектно-ориентированной манере, вместо того чтобы вызывать функции C напрямую.

В листинге 21.6 показан класс, в который заворачивается код библиотеки для работы со стеком.

Листинг 21.6. Класс языка C++, в который завернута функциональность, предоставляемая библиотекой для работы со стеком (c++/Stack.cpp)

```
#include <string.h>

#include <iostream>
#include <string>
#include "cstack.h"

template<typename T>
value_t CreateValue(const T& pValue);

template<typename T>
T ExtractValue(const value_t& value);

template<typename T>
class Stack {
public:
  // Конструктор
  Stack(int pMaxSize) {
    mStack = cstack_new();
    cstack_ctor(mStack, pMaxSize);
  }

  // Деструктор
  ~Stack() {
    cstack_dtor(mStack, free_value);
    cstack_delete(mStack);
  }
```

```
  size_t Size() {
    return cstack_size(mStack);
  }

  void Push(const T& pItem) {
    value_t value = CreateValue(pItem);
    if (!cstack_push(mStack, value)) {
      throw "Stack is full!";
    }
  }

  const T Pop() {
    value_t value;
    if (!cstack_pop(mStack, &value)) {
      throw "Stack is empty!";
    }
    return ExtractValue<T>(value);
  }

  void Clear() {
    cstack_clear(mStack, free_value);
  }

private:
  cstack_t* mStack;
};
```

Данный класс имеет несколько важных аспектов, которые необходимо отметить.

- Он содержит приватную переменную `cstack_t` — указатель на объект, создаваемый функцией `cstack_new`, принадлежащей статической библиотеке. Это можно считать *ссылкой* на объект, который существует на уровне языка C и создается/управляется отдельной библиотекой C. Указатель `mStack` аналогичен файловому дескриптору (или ссылке), ссылающемуся на файл.
- В класс завернуты все функции, предоставляемые библиотекой для работы со стеком. Так делает не всякая обертка вокруг библиотеки C; обычно доступ предоставляется к ограниченному набору функций.
- Данный класс является шаблонным. То есть способен работать с различными типами данных. Как видите, мы объявили две шаблонные функции для сериализации и десериализации объектов разных типов: `CreateValue` и `ExtractValue`. С помощью этих функций класс превращает объект C++ в байтовый массив (сериализация) и наоборот (десериализация).
- Мы определяем специализированную шаблонную функцию для типа `std::string`. Таким образом, мы можем с помощью нашего класса хранить значения данного типа. Стоит отметить, что `std::string` — это стандартный тип языка C++, предназначенный для строковых значений.

- Наша библиотека позволяет размещать множество значений разных типов в одном и том же экземпляре стека. Значение можно преобразовать в массив символов и обратно. Взгляните на структуру `value_t` в листинге 21.1. Ей нужен всего лишь указатель `car`. В отличие от библиотеки C этот класс, написанный на C++, является *типобезопасным*, и каждый его экземпляр может работать только с определенным типом данных.
- В C++ каждый класс имеет как минимум один конструктор и один деструктор. Таким образом, соответствующий объект стека будет удобно инициализировать в конструкторе и финализировать в деструкторе.

Мы хотим, чтобы наш класс на C++ умел работать со строковыми значениями. Поэтому нам нужно написать подходящие функции сериализации и десериализации, которые можно будет использовать в рамках данного класса. В листинге 21.7 содержатся определения функций, которые преобразуют массив символов языка C в объект `std::string` и наоборот.

Листинг 21.7. Специализированные шаблонные функции, предназначенные для сериализации/десериализации типа std::string. Они используются внутри класса на языке C++ (c++/Stack.cpp)

```
template<>
value_t CreateValue(const std::string& pValue) {
  value_t value;
  value.len = pValue.size() + 1;
  value.data = new char[value.len];
  strcpy(value.data, pValue.c_str());
  return value;
}

template<>
std::string ExtractValue(const value_t& value) {
  return std::string(value.data, value.len);
}
```

Это *специализация* типа `std::string` для объявленной шаблонной функции, которая используется в классе. Она определяет, как объект `std::string` должен быть преобразован в массив символов C и, наоборот, как массив символов C можно превратить в объект `std::string`.

В листинге 21.8 показан метод `main`, который использует класс C++.

Листинг 21.8. Главная функция, использующая класс Stack из C++ (c++/Stack.cpp)

```
int main(int argc, char** argv) {
  Stack<std::string> stringStack(100);
  stringStack.Push("Hello");
  stringStack.Push("World");
```

```
  stringStack.Push("!");
  std::cout << "Stack size: " << stringStack.Size() << std::endl;
  while (stringStack.Size() > 0) {
    std::cout << "Popped > " << stringStack.Pop() << std::endl;
  }
  std::cout << "Stack size after pops: " <<
      stringStack.Size() << std::endl;
  stringStack.Push("Bye");
  stringStack.Push("Bye");
  std::cout << "Stack size before clear: " <<
      stringStack.Size() << std::endl;
  stringStack.Clear();
  std::cout << "Stack size after clear: " <<
      stringStack.Size() << std::endl;
  return 0;
}
```

Описанный выше сценарий охватывает все функции, предоставляемые библиотекой для работы со стеком. Мы выполняем ряд операций и проверяем их результаты. Обратите внимание: в данном коде коде функциональность проверяется с помощью объекта `Stack<std::string>`. Поэтому вы можете помещать в стек и удалять из него только значения `std::string`.

В терминале 21.9 показано, как собрать и запустить представленный выше код. Отмечу, что весь код, который вы видели в данном разделе, написан на C++11 и, следовательно, должен собираться с использованием совместимого компилятора. Как уже говорилось ранее, эти команды выполняются в корневом каталоге главы.

Терминал 21.9. Сборка и запуск кода на C++

```
$ cd c++
$ g++ -c -g -std=c++11 -I$PWD/.. Stack.cpp -o Stack.o
$ g++ -L$PWD/.. Stack.o -lcstack -o cstack_cpp.out
$ LD_LIBRARY_PATH=$PWD/.. ./cstack_cpp.out
Stack size: 3
Popped > !
Popped > World
Popped > Hello
Stack size after pops: 0
Stack size before clear: 2
Stack size after clear: 0
$
```

С помощью параметра `-std=c++11` мы указали, что нам нужен компилятор C++11. Обратите внимание на параметры `-I` и `-L`, которые используются для задания пользовательских каталогов с подключаемыми заголовками и, соответственно, библиотеками. Параметр `-lcstack` позволяет скомпоновать код на языке C++

с библиотекой `libcstack.so`. Следует отметить, что в системах macOS разделяемые объектные библиотеки имеют расширение `.dylib`, поэтому вместо `libcstack.so` у вас может получиться `libcstack.dylib`.

Чтобы запустить исполняемый файл `cstack_cpp.out`, компилятор должен найти `libcstack.so`. Описанное отличается от процедуры сборки. Нам нужно запустить программу и перед этим следует найти файл библиотеки. Таким образом, изменив переменную среды `LD_LIBRARY_PATH`, мы сообщили загрузчику, где искать разделяемые объекты. Более подробно об этом было написано в главе 2.

Код на языке C++ тоже должен проверяться на предмет утечек памяти. В этом нам может помочь утилита `valgrind`, которая позволяет проанализировать полученный исполняемый файл. В терминале 21.10 показан вывод `valgrind` при выполнении исполняемого файла `cstack_cpp.out`.

Терминал 21.10. Сборка и запуск кода на C++ с использованием утилиты valgrind

```
$ cd c++
$ LD_LIBRARY_PATH=$PWD/.. valgrind --leak-check=full ./cstack_cpp.
out
==15061== Memcheck, a memory error detector
==15061== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==15061== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==15061== Command: ./cstack_cpp.out
==15061==
Stack size: 3
Popped > !
Popped > World
Popped > Hello
Stack size after pops: 0
Stack size before clear: 2
Stack size after clear: 0
==15061==
==15061== HEAP SUMMARY:
==15061==     in use at exit: 0 bytes in 0 blocks
==15061==   total heap usage: 9 allocs, 9 frees, 75,374 bytes allocated
==15061==
==15061== All heap blocks were freed -- no leaks are possible
==15061==
==15061== For counts of detected and suppressed errors, rerun with: -v
==15061== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0
from 0)
$
```

Данный вывод наглядно демонстрирует, что в нашем коде нет никаких утечек памяти. Обратите внимание: наличие 1081 байта в секции `still reachable` не является признаком утечек. Более подробно об этом можно почитать в практическом руководстве по утилите `valgrind`.

Мы выяснили, как написать на языке C++ обертку вокруг нашей библиотеки для работы со стеком. Смешивание кода на C и C++ может выглядеть тривиально, однако в C++ существуют правила декорирования имен, которые требуют дополнительного внимания. В следующем разделе мы кратко обсудим Java и увидим, как написанная на этом языке программа может загрузить библиотеку C.

Интеграция с Java

Java-программы собираются компилятором Java в специальный байт-код, аналогичный формату объектных файлов *ABI* (Application Binary Interface — двоичный интерфейс приложений). Файлы с байт-кодом Java нельзя выполнять как обычные программы, для работы им нужна специальная среда.

Байт-код Java может выполняться только в *виртуальной машине Java* (Java Virtual Machine, JVM). Она представляет собой процесс, который имитирует рабочую среду для байт-кода Java. Обычно написана на C или C++ и способна загружать и использовать стандартную библиотеку C вместе со всеми функциями, которые та предоставляет.

В байт-код Java можно компилировать не только сам язык Java, но и, к примеру, Scala, Kotlin и Groovy. Таким образом, все эти языки программирования могут работать внутри JVM. Их обычно называют языками JVM.

В данном разделе мы загрузим уже собранную нами библиотеку для работы со стеком в программу, написанную на Java. Если вы незнакомы с этим языком, то приведенные далее инструкции могут показаться вам запутанными и сложными для восприятия. Поэтому я настоятельно рекомендую всем читателям текущего раздела обзавестись базовыми навыками программирования на Java.

Написание кода на Java

Представьте, что у нас есть проект на C, собранный в разделяемую объектную библиотеку. Мы хотим подключить его к Java и воспользоваться его функциями. К счастью, для компиляции кода на языке Java вовсе не требуется наличие кода на C (исходного или скомпилированного). В Java эти две части проекта полностью отделены друг от друга с помощью *машинно-зависимых методов* (native methods). Конечно, программу не получится запустить с одним лишь кодом на Java, поскольку мы вызываем функции C, что требует предварительной загрузки файла с разделяемой объектной библиотекой. Мы предоставим все инструкции и исходники, необходимые для выполнения Java-программы, которая успешно загружает разделяемую библиотеку и вызывает ее функции.

Для загрузки разделяемых объектных файлов в Java используется *JNI* (Java Native Interface). Отмечу, что JNI не входит в состав данного языка программирования; скорее, это часть спецификации JVM, вследствие чего импортированную библиотеку можно будет задействовать во всех языках JVM, таких как Scala.

Ниже я покажу, как с помощью JNI загрузить нашу разделяемую объектную библиотеку.

Мы уже упомянули, что JNI использует машинно-зависимые методы, не имеющие никаких определений на языке Java; их реальные определения написаны на C или C++ и находятся во внешних разделяемых библиотеках. Можно сказать, что машинно-зависимые методы — это своеобразные порты, через которые программы, написанные на Java, могут общаться с внешним миром и выходить за пределы JVM. В листинге 21.9 показан класс с рядом статических машинно-зависимых методов, который должен предоставлять доступ к функциям, содержащимся в нашей библиотеке для работы со стеком.

Листинг 21.9. Класс NativeStack (java/src/com/packt/extreme_c/ch21/ex1/Main.java)

```
package com.packt.extreme_c.ch21.ex1;

class NativeStack {

  static {
    System.loadLibrary("NativeStack");
  }

  public static native long newStack();
  public static native void deleteStack(long stackHandler);

  public static native void ctor(long stackHandler, int maxSize);
  public static native void dtor(long stackHandler);

  public static native int size(long stackHandler);

  public static native void push(long stackHandler, byte[] item);
  public static native byte[] pop(long stackHandler);

  public static native void clear(long stackHandler);
}
```

Эти методы, как видно по их сигнатурам, соответствуют функциям нашей библиотеки на языке C. Обратите внимание: роль первого операнда играет переменная `long`. Она содержит адрес памяти, прочитанный из машинно-зависимой библиотеки, и выступает указателем, который нужно передавать другим методам в целях обозначения экземпляра стека. Для написания приведенного выше класса нам не нужен уже готовый и полностью рабочий разделяемый объектный файл. Все, что нам требуется, — это список объявлений, необходимых для определения API стека.

У нашего класса также есть *статический конструктор*. Он загружает файл разделяемой объектной библиотеки, находящийся в файловой системе, и пытается сопоставить машинно-зависимые методы с символами, найденными в загруженном файле. Стоит отметить, что выше мы работаем не с файлом `libcstack.so`, который был сгенерирован при сборке нашей библиотеки для работы со стеком. JNI имеет очень строгие правила поиска символов, которые соответствуют машинно-зависимым методам. Как следствие, мы не можем использовать символы, определенные в `libcstack.so`. Вместо этого мы должны подготовить символы, которые ищет JNI, и уже с их помощью обращаться к нашей библиотеке для работы со стеком.

Сейчас это может звучать не совсем понятно, но в следующем разделе я все проясню и покажу, как все реализовать. Продолжим, ведь нам еще нужно добавить код на Java.

В листинге 21.10 показан обобщенный класс `Stack<T>`, в который завернуты машинно-зависимые методы, доступные через JNI. Обобщенные классы в Java можно считать аналогом шаблонных классов в C++. Они используются для описания обобщенных типов, которые могут работать с другими типами.

Как можно видеть, в классе `Stack<T>` есть объект *маршаллер* (marshaller), реализующий интерфейс `Marshaller<T>`. Он используется для сериализации и десериализации входных аргументов каждого метода (от типа `T`), чтобы их можно было передавать в стек C и доставать из него обратно.

Листинг 21.10. Класс Stack<T> и интерфейс Marshaller<T> (java/src/com/packt/extreme_c/ch21/ex1/Main.java)

```
interface Marshaller<T> {

  byte[] marshal(T obj);

  T unmarshal(byte[] data);
}

class Stack<T> implements AutoCloseable {
  private Marshaller<T> marshaller;
  private long stackHandler;

  public Stack(Marshaller<T> marshaller) {
    this.marshaller = marshaller;
    this.stackHandler = NativeStack.newStack();
    NativeStack.ctor(stackHandler, 100);
  }

  @Override
  public void close() {
    NativeStack.dtor(stackHandler);
    NativeStack.deleteStack(stackHandler);
  }
```

```
  public int size() {
    return NativeStack.size(stackHandler);
  }

  public void push(T item) {
    NativeStack.push(stackHandler, marshaller.marshal(item));
  }

  public T pop() {
    return marshaller.unmarshal(NativeStack.pop(stackHandler));
  }
  public void clear() {
    NativeStack.clear(stackHandler);
  }
}
```

Ниже перечислены разные аспекты данного кода, заслуживающие внимания.

- Класс `Stack<T>` — обобщенный. Разные его экземпляры могут работать с различными типами, такими как `String`, `Integer`, `Point`. При этом каждый экземпляр будет совместим только с одним типом, который указывается в момент его создания.
- Чтобы во внутреннем стеке можно было хранить данные разных типов, этот стек должен использовать внешний маршаллер, позволяющий сериализовать и десериализовать объекты. Библиотека C способна хранить в своей структуре данных массивы байтов, а высокоуровневые языки, которым нужно использовать ее функции, должны предоставлять эти массивы путем сериализации входных объектов. Чуть ниже будет показана реализация интерфейса `Marshaller` для типа `String`.
- Экземпляр `Marshaller` будет внедрен с помощью конструктора. Это значит, у нас должен быть уже готовый экземпляр маршаллера, совместимый с *обобщенным* типом класса `T`.
- Класс `Stack<T>` реализует интерфейс `AutoCloseable`. Это просто говорит о наличии у него ряда машинно-зависимых ресурсов, которые необходимо освобождать во время уничтожения. Обратите внимание на то, что сам стек создается в коде на C, а не в Java. Следовательно, когда стек перестанет быть нужным, *сборщик мусора* JVM не может его освободить. Объекты `AutoCloseable` можно применять в качестве ресурсов с определенной областью видимости; когда они больше не нужны, у них автоматически вызывается метод `close`. Вскоре вы увидите пример использования приведенного выше класса.
- У нас есть метод-конструктор, и для инициализации соответствующего стека используются машинно-зависимые методы. Мы храним в классе ссылку на стек в виде поля `long`. В отличие от кода на C++ здесь нет никаких деструкторов. Поэтому внутренний стек может не освободиться, что в конечном счете приведет к утечке памяти. Вот почему мы пометили класс как `AutoCloseable`.

Когда объект `AutoCloseable` больше не нужен, из него вызывается метод `close`, в котором, как видно в представленном выше коде, применяется функция-деструктор из библиотеки C; она освобождает ресурсы, выделенные стеком.

В целом мы не можем рассчитывать на то, что механизм сборки мусора вызовет методы финализации (завершения) Java-объектов. Поэтому для корректного управления машинно-зависимыми ресурсами следует использовать интерфейс AutoCloseable.

В листинге 21.11 показана реализация класса `StringMarshaller`. Она очень простая благодаря тому, что класс `String` поддерживает работу с байтовыми массивами.

Листинг 21.11. Класс StringMarshaller (java/src/com/packt/extreme_c/ch21/ex1/Main.java)

```
class StringMarshaller implements Marshaller<String> {

  @Override
  public byte[] marshal(String obj) {
    return obj.getBytes();
  }

  @Override
  public String unmarshal(byte[] data) {
    return new String(data);
  }
}
```

Листинг 21.12 содержит наш класс `Main`, в котором показан пробный пример использования функций для работы со стеком на C в коде на языке Java.

Листинг 21.12. Класс Main, который содержит пример с демонстрацией возможностей библиотеки для работы со стеком на C (java/src/com/packt/extreme_c/ch21/ex1/Main.java)

```
public class Main {
  public static void main(String[] args) {
    try (Stack<String> stack = new Stack<>(new StringMarshaller())) {
      stack.push("Hello");
      stack.push("World");
      stack.push("!");
      System.out.println("Size after pushes: " + stack.size());
      while (stack.size() > 0) {
        System.out.println(stack.pop());
      }
      System.out.println("Size after pops: " + stack.size());
      stack.push("Ba");
      stack.push("Bye!");
```

```
      System.out.println("Size after before clear: " + stack.size());
      stack.clear();
      System.out.println("Size after clear: " + stack.size());
    }
  }
}
```

Как видите, переменная `stack` создается и используется внутри блока `try`. Эта конструкция, которая обычно называется *try-with-resources*, появилась в Java 7. По завершении блока `try` из объекта ресурса вызывается метод `close`, и соответствующий стек освобождается. Это тот же пример, который мы написали на C++ выше, но здесь он выполнен на Java.

В этом подразделе мы рассмотрели часть проекта, написанную на Java, и код, необходимый для импорта машинно-зависимой части. Все исходники, приведенные выше, можно скомпилировать, но запускать их нельзя, поскольку нам еще нужна часть, написанная на C. Только сочетание этих двух частей позволяет получить исполняемую программу. В следующем разделе мы обсудим все этапы написания машинно-зависимой части.

Написание машинно-зависимой части

Самым важным аспектом предыдущего раздела была концепция машинно-зависимых методов. Они объявляются в Java, но их определения находятся в разделяемой объектной библиотеке за пределами JVM. Но как виртуальная машина находит эти определения? Ответ прост: по именам символов в загруженных объектных файлах. JVM извлекает имя символа для каждого машинно-зависимого метода с учетом различных свойств, таких как пакет, содержащий и его имя. Затем в загруженных разделяемых библиотеках ищется соответствующий символ, и в случае неудачи возвращается ошибка.

Согласно тому, что мы узнали в предыдущем подразделе, JVM вынуждает нас использовать для функций, которые входят в состав загружаемого объектного файла, символы с определенными именами. Однако при создании нашей библиотеки для работы со стеком мы не следовали никаким соглашениям об именовании. Таким образом, JVM не сможет найти функции, предоставляемые нашей библиотекой. Поэтому мы должны найти другой путь. Библиотеки C обычно пишут без расчета на то, что они будут применяться в среде JVM.

На рис. 21.1 показано, как связать Java с машинно-зависимой частью, используя промежуточную библиотеку на C или C++. Мы предоставляем машине JVM те символы, которые она ожидает, и делегируем вызовы, связанные с этими символами, к подходящим функциям внутри библиотеки C. Вот так, собственно, работает JNI.

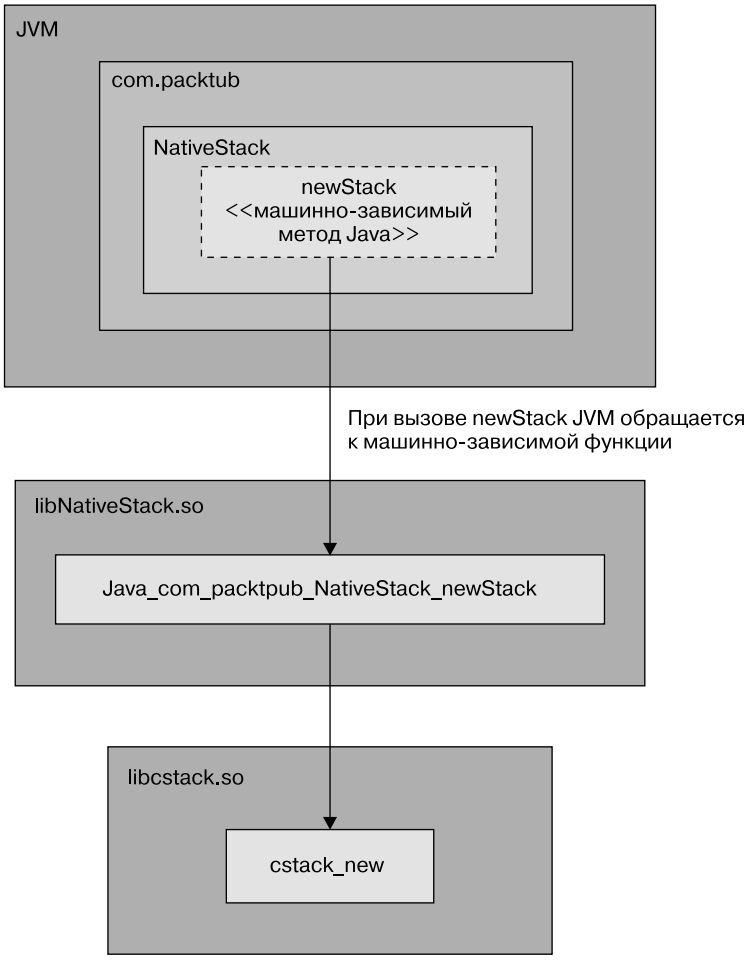

Рис. 21.1. Промежуточный разделяемый объект libNativeStack.so, который направляет вызовы из Java непосредственно библиотеке для работы со стеком, libcstack.so

Объясню на гипотетическом примере. Представьте, что мы хотим вызвать функцию C, `func`, из Java, и определение данной функции находится в разделяемом объектном файле `libfunc.so`. В коде на языке Java также имеется класс `Clazz` с методом `doFunc`. Мы знаем, что при поиске определения машинно-зависимой функции `doFunc` JVM будет искать символ `Java_Clazz_doFunc`. Поэтому создадим промежуточную разделяемую библиотеку `libNativeLibrary.so`, содержащую именно такой символ, который ищет JVM. Затем внутри данной функции мы сделаем вызов `func`. Можно сказать, что `Java_Clazz_doFunc` играет роль реле, делегируя вызов библиотеке C и в конечном счете функции `func`.

Чтобы у нас совпадали имена символов Java, можно воспользоваться заголовочным файлом C, который компилятор Java обычно генерирует из машинно-зависимых методов, найденных в коде на Java. Таким образом, нам остается только написать определения для функций, содержащихся в данном заголовочном файле. Это позволяет не ошибиться в именах символов, которые в конечном счете будет искать JVM.

Команды, представленные в терминале 21.11, показывают, как скомпилировать исходный файл Java и сделать так, чтобы компилятор сгенерировал заголовочный файл для найденных там машинно-зависимых методов. Здесь мы скомпилируем наш единственный Java-файл, `Main.java`, который содержит весь код Java, представленный в предыдущих листингах. Обратите внимание: эти команды нужно выполнять в корневом каталоге главы.

Терминал 21.11. Компиляция файла Main.java и генерация заголовка из найденных в нем машинно-зависимых методов

```
$ cd java
$ mkdir -p build/headers
$ mkdir -p build/classes
$ javac -cp src -h build/headers -d build/classes \
src/com/packt/extreme_c/ch21/ex1/Main.java
$ tree build
build
├── classes
│   └── com
│       └── packt
│           └── extreme_c
│               └── ch21
│                   └── ex1
│                       ├── Main.class
│                       ├── Marshaller.class
│                       ├── NativeStack.class
│                       ├── Stack.class
│                       └── StringMarshaller.class
└── headers
    └── com_packt_extreme_c_ch21_ex1_NativeStack.h

7 directories, 6 files
$
```

Как видите, мы передали компилятору Java, `javac`, параметр `-h`. Вдобавок указали каталог, в который должны попасть все заголовки. Утилита `tree` отображает содержимое каталога build в виде дерева. Обратите внимание на файлы `.class`. Они содержат байт-код Java, который будет использоваться при загрузке этих классов в экземпляр JVM.

Помимо файлов с классами, мы видим заголовочный файл, `com_packt_extreme_c_ch21_ex1_NativeStack.h`, который содержит соответствующие определения функций языка C для машинно-зависимых методов из класса `NativeStack`.

Если открыть заголовочный файл, то можно увидеть примерно то же, что и в листинге 21.13. В нем объявлен ряд функций с длинными, необычными именами, состоящими из имени пакета, имени класса и имени соответствующего машинно-зависимого метода.

Листинг 21.13. (Неполное) содержимое заголовочного файла, сгенерированного в соответствии с JNI

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class com_packt_extreme_c_ch21_ex1_NativeStack */

#ifndef _Included_com_packt_extreme_c_ch21_ex1_NativeStack
#define _Included_com_packt_extreme_c_ch21_ex1_NativeStack
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     com_packt_extreme_c_ch21_ex1_NativeStack
 * Method:    newStack
 * Signature: ()J
 */
JNIEXPORT jlong JNICALL Java_com_packt_extreme_1c_ch21_ex1_
NativeStack_newStack
  (JNIEnv *, jclass);

/*
 * Class:     com_packt_extreme_c_ch21_ex1_NativeStack
 * Method:    deleteStack
 * Signature: (J)V
 */
JNIEXPORT void JNICALL Java_com_packt_extreme_1c_ch21_ex1_
NativeStack_deleteStack
  (JNIEnv *, jclass, jlong);

...
...
...

#ifdef __cplusplus
}
#endif
#endif
```

Функции, объявленные в этом заголовочном файле, содержат имена символов, которые будет искать JVM при загрузке соответствующих функций для машинно-зависимых методов. Мы модифицировали данный заголовок и сделали его более

компактным с помощью макроса, чтобы объявления функций занимали меньше места. Это видно в листинге 21.14.

Листинг 21.14. Модифицированная версия сгенерированного заголовочного файла JNI (java/native/NativeStack.h)

```
// Имя файла: NativeStack.h
// Описание: измененный заголовочный файл, сгенерированный с помощью JNI

#include <jni.h>

#ifndef _Included_com_packt_extreme_c_ch21_ex1_NativeStack
#define _Included_com_packt_extreme_c_ch21_ex1_NativeStack

#define JNI_FUNC(n) Java_com_packt_extreme_1c_ch21_ex1_
NativeStack_##

#ifdef __cplusplus
extern "C" {
#endif

JNIEXPORT jlong JNICALL JNI_FUNC(newStack)(JNIEnv* , jclass);
JNIEXPORT void JNICALL JNI_FUNC(deleteStack)(JNIEnv* , jclass, jlong);

JNIEXPORT void JNICALL JNI_FUNC(ctor)(JNIEnv* , jclass, jlong, jint);
JNIEXPORT void JNICALL JNI_FUNC(dtor)(JNIEnv* , jclass, jlong);

JNIEXPORT jint JNICALL JNI_FUNC(size)(JNIEnv* , jclass, jlong);

JNIEXPORT void JNICALL JNI_FUNC(push)(JNIEnv* , jclass, jlong, jbyteArray);
JNIEXPORT jbyteArray JNICALL JNI_FUNC(pop)(JNIEnv* , jclass, jlong);

JNIEXPORT void JNICALL JNI_FUNC(clear)(JNIEnv* , jclass, jlong);

#ifdef __cplusplus
}
#endif
#endif
```

Итак, мы создали новый макрос `JNI_FUNC`, который позволяет избавиться от существенной части имени функции, общего для всех объявлений. Вдобавок мы убрали комментарии, чтобы сделать заголовочный файл еще более компактным.

Мы будем использовать макрос `JNI_FUNC` как в заголовке, так и в следующем исходном файле, который показан в листинге 21.15.

Обычно модифицировать сгенерированный заголовочный файл не принято. Мы сделали это в познавательных целях. В настоящих средах для сборки сгенерированные файлы желательно использовать, не подвергая каким-либо изменениям.

В листинге 21.15 находятся определения представленных выше функций. Как видите, они всего лишь перенаправляют вызовы к соответствующим функциям на языке C, подключенным из библиотеки для работы со стеком.

Листинг 21.15. Определения функций, объявленных в заголовочном файле JNI (java/native/NativeStack.cpp)

```
#include <stdlib.h>

#include "NativeStack.h"
#include "cstack.h"

void defaultDeleter(value_t* value) {
  free_value(value);
}

void extractFromJByteArray(JNIEnv* env,
                           jbyteArray byteArray,
                           value_t* value) {
  jboolean isCopy = false;
  jbyte* buffer = env->GetByteArrayElements(byteArray, &isCopy);
  value->len = env->GetArrayLength(byteArray);
  value->data = (char*)malloc(value->len * sizeof(char));
  for (size_t i = 0; i < value->len; i++) {
    value->data[i] = buffer[i];
  }
  env->ReleaseByteArrayElements(byteArray, buffer, 0);
}

JNIEXPORT jlong JNICALL JNI_FUNC(newStack)(JNIEnv* env,
                                           jclass clazz) {
  return (long)cstack_new();
}

JNIEXPORT void JNICALL JNI_FUNC(deleteStack)(JNIEnv* env,
                                             jclass clazz,
                                             jlong stackPtr) {
  cstack_t* cstack = (cstack_t*)stackPtr;
  cstack_delete(cstack);
}

JNIEXPORT void JNICALL JNI_FUNC(ctor)(JNIEnv *env,
                                      jclass clazz,
                                      jlong stackPtr,
                                      jint maxSize) {
  cstack_t* cstack = (cstack_t*)stackPtr;
  cstack_ctor(cstack, maxSize);
}

JNIEXPORT void JNICALL JNI_FUNC(dtor)(JNIEnv* env,
                                      jclass clazz,
                                      jlong stackPtr) {
```

```
  cstack_t* cstack = (cstack_t*)stackPtr;
  cstack_dtor(cstack, defaultDeleter);
}

JNIEXPORT jint JNICALL JNI_FUNC(size)(JNIEnv* env,
                                      jclass clazz,
                                      jlong stackPtr) {
  cstack_t* cstack = (cstack_t*)stackPtr;
  return cstack_size(cstack);
}

JNIEXPORT void JNICALL JNI_FUNC(push)(JNIEnv* env,
                                      jclass clazz,
                                      jlong stackPtr,
                                      jbyteArray item) {
  value_t value;
  extractFromJByteArray(env, item, &value);

  cstack_t* cstack = (cstack_t*)stackPtr;
  bool_t pushed = cstack_push(cstack, value);
  if (!pushed) {
    jclass Exception = env->FindClass("java/lang/Exception");
    env->ThrowNew(Exception, "Stack is full!");
  }
}

JNIEXPORT jbyteArray JNICALL JNI_FUNC(pop)(JNIEnv* env,
                                           jclass clazz,
                                           jlong stackPtr) {
  value_t value;
  cstack_t* cstack = (cstack_t*)stackPtr;
  bool_t popped = cstack_pop(cstack, &value);
  if (!popped) {
    jclass Exception = env->FindClass("java/lang/Exception");
    env->ThrowNew(Exception, "Stack is empty!");
  }

  jbyteArray result = env->NewByteArray(value.len);
  env->SetByteArrayRegion(result, 0,
          value.len, (jbyte*)value.data);
  defaultDeleter(&value);
  return result;
}


JNIEXPORT void JNICALL JNI_FUNC(clear)(JNIEnv* env,
                                       jclass clazz,
                                       jlong stackPtr) {
  cstack_t* cstack = (cstack_t*)stackPtr;
  cstack_clear(cstack, defaultDeleter);
}
```

Данный код написан на C++. Определения можно написать и на C. Единственное, на что следует обратить внимание, — это на преобразование байтовых массивов C в байтовые массивы Java, которое происходит в функциях `push` и `pop`. Мы добавили функцию `extractFromJByteArray`, которая будет создавать массив байтов C на основе аналогичного массива, полученного из кода на Java.

Команды, представленные в терминале 21.12, создают в Linux промежуточный разделяемый объект `libNativeStack.so`, который будет загружаться и использоваться Java-машиной. Стоит отметить, что перед выполнением этих команд нужно установить переменную среды `JAVA_HOME`.

Терминал 21.12. Сборка промежуточной разделяемой библиотеки libNativeStack.so

```
$ cd java/native
$ g++ -c -fPIC -I$PWD/../.. -I$JAVA_HOME/include \
  -I$JAVA_HOME/include/linux NativeStack.cpp -o NativeStack.o
$ g++ -shared -L$PWD/../.. NativeStack.o -lcstack -o
libNativeStack.so
$
```

Итоговый разделяемый объектный файл компонуется с библиотекой для работы со стеком, `libcstack.so`. Это значит, что чтобы библиотека `libNativeStack.so` могла работать, нужно загрузить `libcstack.so`. Следовательно, JVM загружает по очереди эти два файла, и в итоге код на Java сможет взаимодействовать с машинно-зависимой частью проекта, что позволит нам выполнить нашу Java-программу.

Команды, представленные в терминале 21.13, выполняют демонстрационный пример, показанный в листинге 21.12.

Терминал 21.13. Выполнение демонстрационного примера на Java

```
$ cd java
$ LD_LIBRARY_PATH=$PWD/.. java -Djava.library.path=$PWD/native \
  -cp build/classes com.packt.extreme_c.ch21.ex1.Main
Size after pushes: 3
!
World
Hello
Size after pops: 0
Size after before clear: 2
Size after clear: 0
$
```

Как видите, мы передали JVM параметр `-Djava.library.path=...`, определяющий, где нужно искать разделяемые объектные библиотеки. Мы указали каталог, в котором должна находиться библиотека `libNativeStack.so`.

В этом разделе мы показали, как загрузить скомпилированную библиотеку C в JVM и использовать ее совместно с исходным кодом на языке Java. Данный механизм можно применять и для загрузки более крупных машинно-зависимых библиотек, состоящих из нескольких частей.

Теперь пришло время заняться интеграцией нашей библиотеки с кодом на Python.

Интеграция с Python

Python — интерпретируемый язык программирования. Это значит, написанный на нем код анализируется и выполняется промежуточной программой, известной как *интерпретатор*. Если мы будем использовать внешнюю скомпилированную разделяемую библиотеку, то интерпретатор ее загрузит и сделает доступной в исходном коде. У Python есть специальный фреймворк для загрузки внешних разделяемых библиотек под названием *ctypes*. Мы будем применять его в данном разделе.

Процесс загрузки разделяемой библиотеки с помощью `ctypes` очень прост. Нам понадобится лишь сама библиотека и определения ввода и вывода функций, которые будут использоваться. Класс, представленный в листинге 21.16, служит оберткой для логики, относящейся к `ctypes`, и делает ее доступной нашему главному классу `Stack`, который будет показан в последующих листингах.

Листинг 21.16. Код, который относится к ctypes и делает библиотеку для работы со стеком доступной остальному коду на Python (python/stack.py)

```
from ctypes import *

class value_t(Structure):
  _fields_ = [("data", c_char_p), ("len", c_int)]

class _NativeStack:
  def __init__(self):
    self.stackLib = cdll.LoadLibrary(
            "libcstack.dylib" if platform.system() == 'Darwin'
            else "libcstack.so")

    # value_t make_value(char*, size_t)
    self._makevalue_ = self.stackLib.make_value
    self._makevalue_.argtypes = [c_char_p, c_int]
    self._makevalue_.restype = value_t

    # value_t copy_value(char*, size_t)
    self._copyvalue_ = self.stackLib.copy_value
```

```
        self._copyvalue_.argtypes = [c_char_p, c_int]
        self._copyvalue_.restype = value_t

        # void free_value(value_t*)
        self._freevalue_ = self.stackLib.free_value
        self._freevalue_.argtypes = [POINTER(value_t)]

        # cstack_t* cstack_new()
        self._new_ = self.stackLib.cstack_new
        self._new_.argtypes = []
        self._new_.restype = c_void_p

        # void cstack_delete(cstack_t*)
        self._delete_ = self.stackLib.cstack_delete
        self._delete_.argtypes = [c_void_p]

        # void cstack_ctor(cstack_t*, int)
        self._ctor_ = self.stackLib.cstack_ctor
        self._ctor_.argtypes = [c_void_p, c_int]

        # void cstack_dtor(cstack_t*, deleter_t)
        self._dtor_ = self.stackLib.cstack_dtor
        self._dtor_.argtypes = [c_void_p, c_void_p]

        # size_t cstack_size(cstack_t*)
        self._size_ = self.stackLib.cstack_size
        self._size_.argtypes = [c_void_p]
        self._size_.restype = c_int

        # bool_t cstack_push(cstack_t*, value_t)
        self._push_ = self.stackLib.cstack_push
        self._push_.argtypes = [c_void_p, value_t]
        self._push_.restype = c_int

        # bool_t cstack_pop(cstack_t*, value_t*)
        self._pop_ = self.stackLib.cstack_pop
        self._pop_.argtypes = [c_void_p, POINTER(value_t)]
        self._pop_.restype = c_int

        # void cstack_clear(cstack_t*, deleter_t)
        self._clear_ = self.stackLib.cstack_clear
        self._clear_.argtypes = [c_void_p, c_void_p]
```

В определении этого класса находятся все функции, которые будут использоваться в нашем коде на Python. Ссылки на функции C хранятся в виде приватных полей экземпляра класса (у такого поля в начале и в конце есть символ _), и их можно применять для вызова соответствующих функций. Обратите внимание: в представленном выше коде загружается `libcstack.dylib`, поскольку мы задействуем систему macOS. В Linux нужно загружать `libcstack.so`.

Класс, представленный в листинге 21.17, — главный компонент Python, который использует представленную выше обертку. Весь остальной код на Python будет обращаться с его помощью к функциям для работы со стеком.

Листинг 21.17. Класс Stack на Python, который использует функции C, загруженные из библиотеки для работы со стеком (python/stack.py)

```
class Stack:
  def __enter__(self):
    self._nativeApi_ = _NativeStack()
    self._handler_ = self._nativeApi_._new_()
    self._nativeApi_._ctor_(self._handler_, 100)
    return self

  def __exit__(self, type, value, traceback):
    self._nativeApi_._dtor_(self._handler_, self._nativeApi_._freevalue_)
    self._nativeApi_._delete_(self._handler_)

  def size(self):
    return self._nativeApi_._size_(self._handler_)

  def push(self, item):
    result = self._nativeApi_._push_(self._handler_,
            self._nativeApi_._copyvalue_(item.encode('utf-8'), len(item)));
    if result != 1:
      raise Exception("Stack is full!")

  def pop(self):
    value = value_t()
    result = self._nativeApi_._pop_(self._handler_, byref(value))
    if result != 1:
      raise Exception("Stack is empty!")
    item = string_at(value.data, value.len)
    self._nativeApi_._freevalue_(value)
    return item

  def clear(self):
    self._nativeApi_._clear_(self._handler_, self._nativeApi_._freevalue_)
```

Как видите, `Stack` хранит ссылку на класс `_NativeStack`, чтобы иметь возможность вызывать соответствующие функции C. Обратите внимание: мы переопределяем функции `__enter__` и `__exit__`. Это позволяет использовать данный код в качестве класса ресурсов и делает его совместимым с инструкцией `with` языка Python. Синтаксис данной инструкции будет продемонстрирован чуть позже. Пожалуйста, имейте в виду, что класс `Stack` работает только со строковыми элементами.

В листинге 21.18 показан демонстрационный пример, очень похожий на те, которые использовались для Java и C++.

Листинг 21.18. Пример на языке Python, который использует класс Stack (python/stack.py)

```
if __name__ == "__main__":
  with Stack() as stack:
    stack.push("Hello")
    stack.push("World")
    stack.push("!")
    print("Size after pushes:" + str(stack.size()))
    while stack.size() > 0:
      print(stack.pop())
    print("Size after pops:" + str(stack.size()))
    stack.push("Ba");
    stack.push("Bye!");
    print("Size before clear:" + str(stack.size()))
    stack.clear()
    print("Size after clear:" + str(stack.size()))
```

В этом коде вы можете видеть инструкцию `with` языка Python.

При входе в блок `with` вызывается функция `__enter__`, а обращение к экземпляру класса `Stack` происходит с помощью переменной `stack`. При выходе из блока `with` вызывается функция `__exit__`. Это дает нам возможность освободить соответствующие машинно-зависимые ресурсы (в данном случае объект стека C), когда они больше не нужны.

В терминале 21.14 показано, как запускать этот код. Обратите внимание: все листинги на языке Python находятся в одном файле `stack.py`. Прежде чем выполнять следующие команды, нужно перейти в корневой каталог данной главы.

Терминал 21.14. Запуск демонстрационного примера на Python

```
$ cd python
$ LD_LIBRARY_PATH=$PWD/.. python stack.py
Size after pushes:3
!
World
Hello
Size after pops:0
Size before clear:2
Size after clear:0
$
```

Интерпретатор должен как-то найти и загрузить разделяемую библиотеку C, поэтому мы присваиваем переменной среды `LD_LIBRARY_PATH` путь к каталогу, содержащему разделяемый библиотечный файл.

В следующем разделе будет показано, как библиотеку для работы со стеком загрузить и использовать в языке Go.

Интеграция с Go

Язык программирования Golang (или просто Go) поддерживает простую интеграцию с машинно-зависимыми разделяемыми библиотеками. Он преподносится как системный язык, и его можно считать следующей итерацией C и C++. Поэтому мы ожидаем, что с загрузкой и использованием машинно-зависимых библиотек в Golang не будет никаких проблем.

В Golang для вызова любого кода на C и загрузки разделяемых объектных файлов предусмотрен встроенный пакет *cgo*. В коде на Go, показанном в листинге 21.19, видно, как использовать пакет `cgo` и вызывать функции C, загруженные из библиотеки для работы со стеком. Там же определен новый класс, `Stack`, с помощью которого другой на Go может пользоваться этими функциями.

Листинг 21.19. Класс Stack, который использует загруженный объектный файл libcstack.so (go/stack.go)

```
package main

/*
#cgo CFLAGS: -I..
#cgo LDFLAGS: -L.. -lcstack
#include "cstack.h"
*/
import "C"
import (
  "fmt"
)

type Stack struct {
  handler *C.cstack_t
}

func NewStack() *Stack {
  s := new(Stack)
  s.handler = C.cstack_new()
  C.cstack_ctor(s.handler, 100)
  return s
}

func (s *Stack) Destroy() {
  C.cstack_dtor(s.handler, C.deleter_t(C.free_value))
  C.cstack_delete(s.handler)
}

func (s *Stack) Size() int {
  return int(C.cstack_size(s.handler))
}
```

```
func (s *Stack) Push(item string) bool {
  value := C.make_value(C.CString(item), C.ulong(len(item) + 1))
  pushed := C.cstack_push(s.handler, value)
  return pushed == 1
}

func (s *Stack) Pop() (bool, string) {
  value := C.make_value(nil, 0)
  popped := C.cstack_pop(s.handler, &value)
  str := C.GoString(value.data)
  defer C.free_value(&value)
  return popped == 1, str
}

func (s *Stack) Clear() {
  C.cstack_clear(s.handler, C.deleter_t(C.free_value))
}
```

Чтобы использовать пакет `cgo`, нужно импортировать `C`. Он загружает разделяемые библиотеки, указанные в псевдодирективе `#cgo`. С помощью директивы `#cgo LDFLAGS: -L.. -lcstack` мы указали, что нам нужно загрузить библиотеку `libcstack.so`. Обратите внимание: параметры `CFLAGS` и `LDFLAGS` определяют флаги, которые передаются непосредственно компилятору C и компоновщику соответственно.

Мы также указали путь, по которому нужно проводить поиск объектного файла. После этого можем использовать структуру `C` для вызова загруженных машинозависимых функций. Например, с помощью `C.cstack_new()` мы вызвали соответствующую функцию из библиотеки для работы со стеком. Пакет `cgo` позволяет сделать это довольно легко. Стоит отметить, что приведенный выше класс `Stack` работает только со строковыми элементами.

В листинге 21.20 показан пример, написанный на Golang. Обратите внимание: при выходе из функции `main` нам приходится вызывать функцию `Destroy` из объекта `stack`.

Листинг 21.20. Пример, написанный на Go и использующий класс Stack (go/stack.go)

```
func main() {
  var stack = NewStack()
  stack.Push("Hello")
  stack.Push("World")
  stack.Push("!")
  fmt.Println("Stack size:", stack.Size())
  for stack.Size() > 0 {
    _, str := stack.Pop()
    fmt.Println("Popped >", str)
  }
  fmt.Println("Stack size after pops:", stack.Size())
```

```
  stack.Push("Bye")
  stack.Push("Bye")
  fmt.Println("Stack size before clear:", stack.Size())
  stack.Clear()
  fmt.Println("Stack size after clear:", stack.Size())
  stack.Destroy()
}
```

В терминале 21.15 показано, как собрать и запустить наш пример.

Терминал 21.15. Запуск примера на Go

```
$ cd go
$ go build -o stack.out stack.go
$ LD_LIBRARY_PATH=$PWD/.. ./stack.out
Stack size: 3
Popped > !
Popped > World
Popped > Hello
Stack size after pops: 0
Stack size before clear: 2
Stack size after clear: 0
$
```

Как видите, в Golang, в отличие от Python, программу нужно сначала скомпилировать и только потом запускать. Кроме того, мы должны установить переменную среды `LD_LIBRARY_PATH`, чтобы исполняемый файл мог найти и загрузить `libcstack.so`.

В этом разделе было показано, как использовать пакет `cgo` в Golang для загрузки и работы с разделяемыми библиотеками. Поскольку Golang ведет себя как тонкая обертка вокруг кода на C, на то, чтобы загрузить и применить внешние объектные файлы, в данном языке требуется меньше усилий по сравнению с Python и Java.

Резюме

В этой главе мы обсудили интеграцию C с другими языками программирования. В ходе обсуждения мы:

- разработали библиотеку C, которая предоставляет функции для работы со стеком, такие как `push`, `pop` и т. д. Итоговым результатом сборки данной библиотеки стал разделяемый объектный файл, предназначенный для загрузки другими языками;
- обсудили декорирование имен в C++ и рассказали, как его избежать в коде на языке C при использовании компилятора C++;

- написали на C++ обертку вокруг нашей библиотеки для работы со стеком, способную загружать объектные файлы и выполнять их функции в рамках C++;
- написали JNI-обертку вокруг библиотеки C, использовав для этого машинно-зависимые методы;
- узнали, как в JNI написать машинно-зависимый код и соединить код на C с кодом на Java. В конце выполнили Java-программу, которая использует библиотеку для работы со стеком;
- удачно написали код на Python, который с помощью пакета `ctypes` загружал и использовал разделяемый объектный файл библиотеки;
- в заключительном разделе написали программу на Golang, способную загружать разделяемый объектный файл нашей библиотеки с помощью пакета `cgo`.

Следующая глава посвящена тестированию и отладке в C. Вы познакомитесь с рядом библиотек C для написания модульных тестов. Вдобавок мы поговорим об отладке и некоторых инструментах, позволяющих отлаживать программы и отслеживать их работу.

22 Модульное тестирование и отладка

Прежде чем передавать код клиентам, его необходимо тщательно протестировать, и неважно, какой язык программирования вы используете и какого рода приложение разрабатываете.

Написание тестов не является чем-то новым, и в любом современном программном проекте можно найти сотни и даже тысячи тестов. В наши дни писать тесты для ПО нужно обязательно, а доставка кода без надлежащей проверки крайне не приветствуется. Вот почему я посвятил данную главу тестированию ПО, написанного на C, и различным библиотекам, которые предназначены для этого.

Но тестирование будет не единственной темой текущей главы. Мы обсудим средства и методы отладки, с помощью которых можно искать проблемы в программах на C. Тестирование и отладка всегда дополняли друг друга; когда тест проваливается, обычно далее следует анализ и отладка соответствующего кода.

В данной главе мы не станем углубляться в философию тестирования, а просто будем исходить из того, что тесты — это хорошо. Мы познакомимся с базовой терминологией и рекомендациями по написанию тестируемого кода, которым должен следовать разработчик.

Эта глава состоит из двух разделов. В первом речь пойдет о тестировании и библиотеках, которые можно использовать в современной разработке на C. Во втором мы поговорим об отладке, начиная с обзора разных видов программных ошибок. Проблемы с памятью, конкурентностью, производительностью — самые распространенные случаи, в которых для проведения успешного расследования не обойтись без отладки.

Кроме того, мы рассмотрим самые распространенные средства отладки, доступные для C (и C++). Конечная цель главы — познакомить вас с инструментами тестирования и отладки, чтобы вы получили о них базовое представление.

В первом разделе вы узнаете основную терминологию тестирования ПО в целом. Она не будет иметь прямого отношения к C, а изученные здесь идеи и концепции можно применить и к другим языкам программирования и технологиям.

Тестирование программного обеспечения

Тестирование ПО — большой и важный раздел программирования, имеющий собственную терминологию и множество концепций. В этом разделе предлагается очень поверхностное введение в тестирование ПО. Я попытаюсь дать определение нескольким терминам, которые будут использоваться в первой части текущей главы. Поэтому имейте в виду, что здесь вы не найдете всеобъемлющего материала по тестированию. Настоятельно рекомендую вам самостоятельно продолжить исследование данной темы.

Когда речь заходит о тестировании программного обеспечения, сразу возникают вопросы: что мы тестируем и с какой целью? В целом мы тестируем отдельные аспекты системы. Они могут быть *функциональными* и *нефункциональными*. Иными словами, могут относиться как к определенным возможностям системы, так и к неким показателям, которые демонстрируются в момент предоставления этих возможностей. Ниже мы рассмотрим несколько примеров.

Функциональное тестирование относится к определенным возможностям, входящим в список *функциональных требований*. Эти тесты предоставляют определенный ввод какому-то *программному элементу*, такому как *функция*, *модуль*, *компонент* или *программная система*, и ожидают получить от них определенный вывод. Тест считается *пройденным* только в случае получения ожидаемого вывода.

Нефункциональное тестирование относится к *уровню качества*, на котором программный элемент, такой как функция, модуль, компонент или система в целом, выполняет определенные функции. Эти тесты обычно *измеряют* различные *показатели*, например *потребление памяти*, *время выполнения*, *конфликты при блокировках* и *уровень безопасности*, и оценивают, насколько хорошо элемент справился со своими обязанностями. Тест считается пройденным, только если измеренный показатель находится в допустимом диапазоне. *Ожидания* относительно этих показателей выводятся из *нефункциональных требований*, предъявляемых к системе.

Помимо функциональных и нефункциональных тестов, у нас могут быть разные *уровни* тестирования. Они спроектированы так, чтобы могли охватывать аспекты разного рода. Это может быть размер тестируемого элемента, его тип или уровень его функциональности.

Например, если говорить о размере элемента, то уровни тестирования начинаются наименьшим возможным функциональным блоком, которым, как известно, является функция (или метод), и заканчиваются самым крупным фрагментом функциональности, который предоставляется системой в целом.

Познакомимся с этими уровнями поближе.

Уровни тестирования

В любой программной системе можно предусмотреть следующие уровни тестирования (это неполный список, больше можно найти в других источниках):

- модульное тестирование;
- интеграционное тестирование;
- системное тестирование;
- приемочное тестирование;
- регрессионное тестирование.

В *модульном тестировании* проверяется единица функциональности. Это может быть функция, выполняющая определенную работу, или набор функций, совместно удовлетворяющих некие потребности. Это может быть и класс с определенной конечной целью, и даже компонент, на который возложена конкретная задача. *Компонент* — часть программной системы с четко определенным набором функций; компоненты, собранные воедино, составляют всю систему в целом.

Когда единицей функциональности выступает компонент, процесс тестирования называется *компонентным*. На данном уровне можно проводить как функциональное, так и нефункциональное тестирование. Каждый модуль должен тестироваться по отдельности, а для этого необходимо каким-то образом имитировать среду. Это единственный уровень, который мы рассматриваем в данной главе. Мы продемонстрируем модульное и компонентное тестирование на примере реального кода на языке C.

Модули в совокупности формируют компонент. В компонентном тестировании проверяются отдельные изолированные компоненты. Но если их сгруппировать, то возникнет необходимость в другом уровне тестирования, который относится к их функциональности или характеристикам, — в *интеграционном тестировании*. Как понятно из названия, тесты на этом уровне проверяют, хорошо ли интегрированы разные компоненты и соответствуют ли требованиям, предъявляемым к системе.

Тестирование всей системы в целом происходит на другом уровне. Здесь у нас есть набор всех компонентов, которые полностью интегрированы. Таким образом, мы проверяем, соответствуют ли возможности, предоставляемые системой, и ее характеристики заявленным требованиям.

Еще один уровень предусмотрен для проверки системы на соответствие бизнес-требованиям с точки зрения *заинтересованной стороны* или *конечного пользователя*. Это так называемое *приемочное тестирование*. Как и системное, оно относится

ко всей системе в целом, однако на самом деле оба уровня существенно отличаются. Вот лишь несколько аспектов разницы:

- за системное тестирование отвечают разработчики и тестировщики, а приемочное обычно проводят конечные пользователи или заинтересованная сторона;
- приемочное тестирование охватывает только функциональные требования, а системное — еще и нефункциональные;
- в системном тестировании в качестве ввода обычно используется небольшой, заранее подготовленный набор данных, тогда как приемочное имеет дело с настоящими данными, которые поступают в систему в режиме реального времени.

Прекрасное объяснение всех различий можно найти на странице https://www.javatpoint.com/acceptance-testing.

Когда в программную систему вносятся изменения, необходимо убедиться в том, что функциональные и нефункциональные тесты по-прежнему актуальны. Для этого предусмотрен отдельный уровень, известный как *регрессионное тестирование*. Его задача — подтвердить, что внесение изменений не привело к появлению *регрессий*. В рамках этого тестирования мы заново проводим все модульные, интеграционные и сквозные (системные) тесты, как функциональные, так и нефункциональные, и смотрим, провалились ли какие-либо из них.

Мы познакомились с разными уровнями тестирования. В оставшейся части главы обсудим модульные тесты. В следующем разделе начнем наше обсуждение с примера на языке C и попытаемся написать для него тестовые случаи.

Модульное тестирование

Как уже объяснялось в предыдущем подразделе, в рамках модульного тестирования проверяются изолированные модули любых размеров — от функции до компонента. Это применимо как к C, так и к C++, только во втором случае роль модулей могут играть еще и классы.

Самый важный аспект модульного тестирования состоит в том, что проверяемые модули должны быть изолированы друг от друга. Например, если одна функция зависит от другой, то мы должны как-то протестировать их по отдельности. Покажу это на реальном примере.

Пример 22.1 выводит факториалы четных чисел меньше 10, но делает это необычным способом. Код аккуратно разделен на один заголовочный и два исходных файла. Мы имеем две функции; одна из них генерирует четные числа меньше 10, а другая принимает указатель на нее и использует его в качестве источника целых чисел, вычисляя их факториалы.

В листинге 22.1 показан заголовочный файл с объявлениями функций.

Листинг 22.1. Заголовочный файл из примера 22.1 (ExtremeC_examples_chapter22_1.h)

```
#ifndef _EXTREME_C_EXAMPLE_22_1_
#define _EXTREME_C_EXAMPLE_22_1_

#include <stdint.h>
#include <unistd.h>

typedef int64_t (*int64_feed_t)();

int64_t next_even_number();

int64_t calc_factorial(int64_feed_t feed);

#endif
```

Как видите, `calc_factorial` принимает указатель на функцию, который возвращает целое число. С помощью этого указателя будут считываться целочисленные значения, для которых нужно вычислить факториал. В листинге 22.2 показаны определения этих функций.

Листинг 22.2. Определения функций, которые используются в примере 22.1 (ExtremeC_examples_chapter22_1.c)

```
#include "ExtremeC_examples_chapter22_1.h"

int64_t next_even_number() {
  static int feed = -2;
  feed += 2;
  if (feed >= 10) {
    feed = 0;
  }
  return feed;
}

int64_t calc_factorial(int64_feed_t feed) {
  int64_t fact = 1;
  int64_t number = feed();
  for (int64_t i = 1; i <= number; i++) {
    fact *= i;
  }
  return fact;
}
```

У `next_even_number` есть внутренняя статическая переменная, которая возвращается вызывающей функции. Обратите внимание: при достижении значения `8` она всегда возвращается к `0`. Таким образом вы можете вызывать `next_even_number`

сколько угодно раз и никогда не получите число больше 8 или меньше 0. В листинге 22.3 содержится исходный файл с функцией `main`.

Листинг 22.3. Главная функция в примере 22.1 (ExtremeC_examples_chapter22_1_main.c)

```
#include <stdio.h>

#include "ExtremeC_examples_chapter22_1.h"

int main(int argc, char** argv) {
  for (size_t i = 1; i <= 12; i++) {
    printf("%lu\n", calc_factorial(next_even_number));
  }
  return 0;
}
```

Функция `main` вызывает `calc_function` 12 раз и выводит полученные факториалы. Для выполнения этого примера нужно сначала скомпилировать оба исходника и затем скомпоновать получившиеся переносимые объектные файлы. В терминале 22.1 показаны команды, необходимые для сборки и запуска примера.

Терминал 22.1. Сборка и запуск примера 22.1

```
$ gcc -c ExtremeC_examples_chapter22_1.c -o impl.o
$ gcc -c ExtremeC_examples_chapter22_1_main.c -o main.o
$ gcc impl.o main.o -o ex22_1.out
$ ./ex22_1.out
1
2
24
720
40320
1
2
24
720
40320
1
2
$
```

Чтобы написать тесты для представленных выше функций, нужно сначала дать небольшое введение. В данном примере у нас две функции (не считая `main`), `next_even_number` и `calc_factorial`, поэтому нам нужно по отдельности протестировать два разных изолированных модуля. Но в главной функции видно: `calc_factorial` зависит от `next_even_number`. Можно было бы подумать, что из-за этой зависимости изолировать функцию `calc_factorial` будет намного сложнее, чем мы ожидали. Но это не так.

На самом деле `calc_factorial` зависит не от самой функции `next_even_number`, а лишь от ее *сигнатуры*. Поэтому вместо `next_even_number` можно подставить функцию с той же сигнатурой и сделать так, чтобы она всегда возвращала одно и то же целое число. То есть мы можем предоставить упрощенную версию `next_even_number`, которая предназначена сугубо для использования в *тестовых случаях*.

Что же такое тестовый случай? Как вы уже знаете, в отдельно взятом модуле есть разные случаи, которые можно протестировать. Простейший пример — передача модулю разных входных значений с последующей проверкой того, соответствует ли вывод *ожиданиям*. В нашем примере мы можем передать функции `calc_factorial` значение `0`, ожидая получить `1` на выходе. Того же результата можно ожидать при передаче `-1`.

Каждый из этих вариантов может служить тестовым случаем. Следовательно, в одном модуле может быть несколько таких случаев, которые охватывают предельно допустимые параметры. Группа тестовых случаев называется *тестовым набором*. Элементы одного тестового набора могут относиться к разным модулям.

Сначала создадим тестовый набор для функции `next_even_number`. Поскольку ее можно с легкостью проверить изолированно от остальных, никаких дополнительных усилий не требуется. В листинге 22.4 показаны тестовые случаи, написанные для `next_even_number`.

Листинг 22.4. Тестовые случаи, написанные для функции next_even_number (ExtremeC_examples_chapter22_1__next_even_number__tests.c)

```
#include <assert.h>

#include "ExtremeC_examples_chapter22_1.h"

void TESTCASE_next_even_number__even_numbers_should_be_returned()
{
  assert(next_even_number() == 0);
  assert(next_even_number() == 2);
  assert(next_even_number() == 4);
  assert(next_even_number() == 6);
  assert(next_even_number() == 8);
}

void TESTCASE_next_even_number__numbers_should_rotate() {
  int64_t number = next_even_number();
  next_even_number();
  next_even_number();
  next_even_number();
  next_even_number();
  int64_t number2 = next_even_number();
  assert(number == number2);
}
```

Как видите, в этом тестовом наборе определено два тестовых случая. Следует отметить, что принцип, по которому выбираются имена для тестовых случаев, я выработал самостоятельно; никакого стандарта на сей счет не существует. Имя тестового случая должно подсказывать, что делает тот или иной тест. Но есть еще более важный момент: тест должен легко находиться в коде, если тестовый случай провалится или его нужно будет подправить.

Чтобы тестовые случаи можно было легко отличить от обычных функций, я добавил к их именам префикс `TESTCASE` в верхнем регистре. Сами имена пытаются описать случаи и их назначение.

Оба тестовых случая содержат в конце вызов `assert`. Это то, с помощью чего любой случай проверяет результат на соответствие ожиданиям. Если условие в скобках `assert` не равно `true`, то *средство выполнения тестов* завершает работу и выводит сообщение об ошибке. Более того, оно возвращает ненулевой *код выхода*, свидетельствующий о том, что тестовые случаи провалились. При успешном прохождении всех тестов возвращаемое значение должно быть равно `0`.

В качестве полезного упражнения попробуйте самостоятельно пройтись по этим двум тестовым случаям и понять, как они проверяют наши ожидания при вызове функции `next_even_number`.

Теперь пришло время написать тестовые случаи для функции `calc_factorial`. Для этого нам потребуется *функция-заглушка*, которая возвращает тестовый вывод. О том, что такое заглушки, мы поговорим чуть позже.

В листинге 22.5 представлены три тестовых случая, которые проверяют только модуль `calc_factorial`.

Листинг 22.5. Тестовые случаи, написанные для функции calc_factorial (ExtremeC_examples_chapter22_1__calc_factorial__tests.c)

```
#include <assert.h>

#include "ExtremeC_examples_chapter22_1.h"

int64_t input_value = -1;

int64_t feed_stub() {
  return input_value;
}

void TESTCASE_calc_factorial__fact_of_zero_is_one() {
  input_value = 0;
  int64_t fact = calc_factorial(feed_stub);
  assert(fact == 1);
}
```

```
void TESTCASE_calc_factorial__fact_of_negative_is_one() {
  input_value = -10;
  int64_t fact = calc_factorial(feed_stub);
  assert(fact == 1);
}

void TESTCASE_calc_factorial__fact_of_5_is_120() {
  input_value = 5;
  int64_t fact = calc_factorial(feed_stub);
  assert(fact == 120);
}
```

Итак, мы определили три тестовых случая для функции `calc_factorial`. Обратите внимание на функцию `feed_stub`. Она соблюдает тот же контракт, что и `next_even_number` из листинга 22.2, но при этом имеет очень простое определение. Она всего лишь возвращает значение, хранящееся в статической переменной `input_value`. Тестовые случаи могут инициализировать эту переменную перед вызовом `calc_facorial`.

Используя упомянутую выше заглушку, мы можем изолировать функцию `calc_facorial` и протестировать ее отдельно. Тот же подход применим и в объектно-ориентированных языках программирования, таких как C++ или Java, но там в качестве заглушек нужно определять *классы* и *объекты*.

В языке C *заглушка* — это определение функции, соответствующее объявлению, которое используется в логике тестируемого модуля. Что еще важнее, заглушка должна быть простой и возвращать значение, которое будет использоваться исключительно в тестовом случае.

В C++ заглушкой может быть как определение функции, которое соответствует ее объявлению, так и класс, который реализует интерфейс. В других объектно-ориентированных языках, где нет самостоятельных функций (таких как Java), возможен только второй вариант. В этом случае заглушка представляет собой объект, созданный из класса-заглушки. Как бы то ни было, у заглушки должно быть простое определение, которое подходит лишь для тестов, но не для реального использования.

Наконец, нам нужно как-то выполнять наши тестовые случаи. Как уже утверждалось прежде, для этого требуется средство выполнения тестов. Следовательно, нам необходим определенный исходный файл с функцией `main`, который выполняет тестовые случаи один за другим. Пример представлен в листинге 22.6.

Листинг 22.6. Средство выполнения тестов для примера 22.1 (ExtremeC_examples_chapter22_1_tests.c)

```
#include <stdio.h>

void TESTCASE_next_even_number__even_numbers_should_be_returned();
void TESTCASE_next_even_number__numbers_should_rotate();
```

```
void TESTCASE_calc_factorial__fact_of_zero_is_one();
void TESTCASE_calc_factorial__fact_of_negative_is_one();
void TESTCASE_calc_factorial__fact_of_5_is_120();

int main(int argc, char** argv) {
  TESTCASE_next_even_number__even_numbers_should_be_returned();
  TESTCASE_next_even_number__numbers_should_rotate();
  TESTCASE_calc_factorial__fact_of_zero_is_one();
  TESTCASE_calc_factorial__fact_of_negative_is_one();
  TESTCASE_calc_factorial__fact_of_5_is_120();
  printf("All tests are run successfully.\n");
  return 0;
}
```

Данный код возвращает `0`, только если все тестовые случаи внутри функций `main` выполняются успешно. Для сборки этого средства выполнения тестов потребуются команды, представленные в терминале 22.2. Обратите внимание на параметр `-g`, который добавляет отладочные символы в итоговый исполняемый файл. Тесты чаще всего собираются в *отладочном режиме*, поскольку в случае провала одного из них нам сразу же нужна точная *трассировка стека* и дополнительная отладочная информация, чтобы продолжить наше расследование. Более того, инструкции `assert` обычно удаляются из *сборки выпуска*, но в исполняемом файле средства выполнения тестов должны присутствовать.

Терминал 22.2. Сборка и запуск средства выполнения тестов для примера 22.1

```
$ gcc -g -c ExtremeC_examples_chapter22_1.c -o impl.o
$ gcc -g -c ExtremeC_examples_chapter22_1__next_even_number__
tests.c -o tests1.o
$ gcc -g -c ExtremeC_examples_chapter22_1__calc_factorial__tests.c
-o tests2.o
$ gcc -g -c ExtremeC_examples_chapter22_1_tests.c -o main.o
$ gcc impl.o tests1.o tests2.o main.o -o ex22_1_tests.out
$ ./ex22_1_tests.out
All tests are run successfully.
$ echo $?
0
$
```

Как видите, все тесты успешно пройдены. Мы также можем проверить код выхода у процесса средства выполнения тестов, используя команду `echo $?`, и посмотреть, возвращает ли она `0`.

Теперь внесем в одну из функций простое изменение, чтобы наши тесты провалились. Посмотрим, что произойдет, если отредактировать `calc_factorial` следующим образом (листинг 22.7).

Листинг 22.7. Изменение функции calc_factorial таким образом, чтобы провалить тесты

```
int64_t calc_factorial(int64_feed_t feed) {
  int64_t fact = 1;
  int64_t number = feed();
  for (int64_t i = 1; i <= (number + 1); i++) {
    fact *= i;
  }
  return fact;
}
```

После внесения изменения, выделенного жирным шрифтом, тестовые случаи с `0` и отрицательным вводом, как и прежде, будут проходить успешно, а вот тесты с вычислением факториала `5` начнут проваливаться. Соберем средство выполнения тестов еще раз и получим следующий вывод на компьютере с macOS (терминал 22.3).

Терминал 22.3. Сборка и запуск средства выполнения тестов после изменения функции calc_factorial

```
$ gcc -g -c ExtremeC_examples_chapter22_1.c -o impl.o
$ gcc -g -c ExtremeC_examples_chapter22_1_tests.c -o main.o
$ ./ex22_1_tests.out
Assertion failed: (fact == 120), function TESTCASE_calc_
factorial__fact_of_5_is_120,
file .../22.1/ExtremeC_examples_chapter22_1__calc_factorial__
tests.c, line 29.
Abort trap: 6
$ echo $?
134
$
```

Как видите, здесь выводится сообщение `Assertion failed` и код выхода `134`. На самом деле этот код используется и возвращается системами для периодического выполнения тестов, такими как *Jenkins*, чтобы проверить успешность прохождения последних.

Если у вас есть модуль, который нужно протестировать в изоляции, то ему обычно необходимо подать на вход какие-то зависимости. Поэтому сам модуль следует писать так, чтобы он был *тестируемым*. Очень важно помнить: не всякий код можно протестировать, и тесты бывают не только модульными. Информацию о том, как писать тестируемый код, можно найти по ссылке https://blog.gurock.com/highly-testable-code/.

Чтобы вам было понятнее, представим, что мы написали функцию `calc_factorial`, показанную в листинге 22.8, вызывающую `next_even_number` напрямую, а не через указатель `next_even_number`.

Листинг 22.8. Функция calc_factorial с измененной сигнатурой, которая не принимает указатель на функцию

```
int64_t calc_factorial() {
  int64_t fact = 1;
  int64_t number = next_even_number();
  for (int64_t i = 1; i <= number; i++) {
    fact *= i;
  }
  return fact;
}
```

Данный код не такой тестируемый. Мы не можем протестировать функцию `calc_factorial`, не вызывая `next_even_number`, — то есть не внося «грязных» изменений в определение символа `next_even_number` в итоговом исполняемом файле, как делаем это в примере 22.2.

На самом деле обе версии `calc_factorial` делают одно и то же, но определение в листинге 22.2 лучше поддается тестированию, поскольку его можно проверять в изоляции. Написание тестируемого кода — непростая задача, которая всегда требует предусмотрительности и дополнительных усилий. Есть разные мнения о том, сколько именно накладных расходов для этого нужно, но неизбежность этих расходов не вызывает сомнений. Однако выгода очевидна. Вы не уследите за модулем, не имеющим тестов, в который со временем вносятся все новые изменения.

Тестовые дублеры

В предыдущем примере при написании тестовых случаев использовались функции-заглушки. Но существуют и другие термины, относящиеся к объектам, которые пытаются имитировать зависимости модуля. Такие объекты называются *тестовыми дублерами*. Ниже мы познакомимся с двумя разновидностями последних: *макетами* и *фиктивными заполнителями*. Но сначала вспомним, что такое функция-заглушка.

В данном подразделе следует обратить внимание на две вещи. Во-первых, определения этих тестовых дублеров — предмет бесконечных споров, и в данной главе я постараюсь дать такое определение, которое подходит для нашего контекста. Во-вторых, наше обсуждение относится только к языку C, поэтому мы будем иметь дело лишь с функциями, но не с объектами.

Зависимость модуля от другой функции просто означает, что он привязан к ее сигнатуре, и потому вместо одной функции можно подставить другую. Эта новая функция, в зависимости от присущих ей свойств, может называться макетной или

фиктивной. Она предназначена исключительно для удовлетворения тестовых требований и не может использоваться в реальных условиях.

Как уже объяснялось ранее, заглушка — очень простая функция, которая обычно возвращает фиксированное значение. В примере 22.1 она просто возвращала значение, установленное тестовым случаем. Пройдя по следующей ссылке, можно найти дополнительную информацию о тех тестовых дублерах, о которых мы уже говорили, и о нескольких других: https://en.wikipedia.org/wiki/Test_double. В ней заглушкой называют нечто предоставляющее тестовому коду *непрямой ввод*. Если согласиться с этим определением, то функция `feed_stub` из листинга 22.5 (см. выше) является заглушкой.

Макетные функции (или макетные объекты в целом, если говорить об объектно-ориентированных языках) можно модифицировать, указывая вывод для определенного ввода. Таким образом, перед запуском логики теста мы указываем то, что должно вернуться из макетной функции при определенном вводе, и в ходе выполнения эта функция будет вести себя так, как было определено заранее. Макетные объекты в целом могут иметь определенные ожидания, но могут и выполнять нужные нам утверждения. Согласно приведенной выше ссылке ожидания для макетных объектов задаются перед запуском теста. В разделе, посвященном компонентному тестированию, будет показан пример макетных функций на языке C.

Наконец, чтобы получить в тесте упрощенную версию настоящей и, возможно, сложной функциональности, используется фиктивная функция. Например, вместо реальной файловой системы можно задействовать некое упрощенное хранилище в памяти. В компонентном тестировании, к примеру, фиктивные реализации позволяют заменить другие компоненты, имеющие сложную функциональность.

В завершение раздела я хочу затронуть тему *покрытия кода*. Теоретически все модули должны иметь соответствующие тестовые наборы, и каждый набор должен содержать все тестовые случаи, которые проходятся по всем возможным ветвям кода. Но это теория. А на практике тесты предусмотрены лишь для какой-то части модулей. Наши тестовые случаи обычно не покрывают все имеющиеся ответвления.

Доля модулей, имеющих подходящие тестовые случаи, называется покрытием кода или *охватом тестирования*. Чем она больше, тем выше вероятность того, что вы получите уведомления о нежелательных изменениях. В действительности такие изменения обычно вносятся в ходе исправления программных ошибок или реализации новых возможностей.

Итак, мы обсудили тестовые дублеры. Пришло время поговорить о компонентном тестировании.

Компонентное тестирование

В предыдущем разделе уже объяснялось, что роль модуля может играть отдельная функция, группа функций или целый компонент. Поэтому компонентное тестирование — частный случай модульного тестирования. В данном разделе мы напишем гипотетический компонент в рамках примера 22.1 и поместим в него две функции, которые у нас уже есть. Обратите внимание: из компонента обычно получается исполняемый файл или библиотека. Представим, что наш компонент будет собираться в библиотеку с двумя функциями.

Как уже говорилось ранее, у нас должна быть возможность протестировать функциональность компонента. Мы по-прежнему будем писать тестовые случаи, но от тестов из предыдущего раздела они будут отличаться изолируемыми модулями. Раньше это были функции, а теперь мы собираемся изолировать компонент, состоящий из двух функций, которые работают сообща. Таким образом, данные функции следует тестировать вместе.

В листинге 22.9 представлены тестовые случаи, которые мы написали для компонента, созданного в рамках примера 22.1.

Листинг 22.9. Компонентные тесты, написанные для нашего гипотетического компонента из примера 22.1 (ExtremeC_examples_chapter22_1_component_tests.c)

```
#include <assert.h>

#include "ExtremeC_examples_chapter22_1.h"

void TESTCASE_component_test__factorials_from_0_to_8() {
  assert(calc_factorial(next_even_number) == 1);
  assert(calc_factorial(next_even_number) == 2);
  assert(calc_factorial(next_even_number) == 24);
  assert(calc_factorial(next_even_number) == 720);
  assert(calc_factorial(next_even_number) == 40320);
}

void TESTCASE_component_test__factorials_should_rotate() {
  int64_t number = calc_factorial(next_even_number);
  for (size_t i = 1; i <= 4; i++) {
    calc_factorial(next_even_number);
  }
  int64_t number2 = calc_factorial(next_even_number);
  assert(number == number2);
}

int main(int argc, char** argv) {
  TESTCASE_component_test__factorials_from_0_to_8();
  TESTCASE_component_test__factorials_should_rotate();
  return 0;
}
```

Итак, мы написали два тестовых случая. Как уже упоминалось, функции `calc_factorial` и `next_even_number` нашего гипотетического компонента должны работать совместно, поэтому вторая подается на вход первой. Эти и другие аналогичные тестовые случаи должны гарантировать корректную работу компонента.

Подготовка основы для написания тестовых случаев требует много усилий. Как следствие, очень часто для этого используют библиотеку тестирования. Такие библиотеки подготавливают среду для тестовых случаев: инициализируют, выполняют и уничтожают каждый из них. В следующем разделе речь пойдет о двух библиотеках тестирования, доступных для языка C.

Библиотеки тестирования для C

В данном разделе будут продемонстрированы две известные библиотеки, предназначенные для тестирования программ на C. Библиотеки модульного тестирования написаны на C и C++. Это связано с тем, что мы можем их легко интегрировать и затем использовать модули непосредственно из тестовой среды C или C++. В текущем разделе мы сфокусируемся на модульном и компонентном тестировании в C.

Для интеграционного тестирования можно выбрать другой язык программирования. В целом интеграционное и системное тестирование отличается высокой сложностью, поэтому нам нужно использовать какие-то фреймворки для автоматизации тестирования, чтобы упростить написание тестов и выполнять их без лишних усилий. В рамках данной автоматизации будет использоваться *предметно-ориентированный язык* (domain-specific language, DSL); он упростит написание и выполнение тестов. Для этой цели подходит много разных языков, однако скриптовые, такие как Unix shell, Python, JavaScript и Ruby, пользуются наибольшей популярностью. В автоматизации тестов активно применяются и другие языки программирования, включая Java.

Ниже перечислены одни из самых известных фреймворков модульного тестирования, с помощью которых можно писать модульные тесты для программ на C. Этот же список находится по ссылке http://check.sourceforge.net/doc/check_html/check_2.html#SEC3:

- Check (от автора приведенной выше ссылки);
- AceUnit;
- GNU Autounit;
- cUnit;
- CUnit;
- CppUnit;
- CuTest;

- embUnit;
- MinUnit;
- Google Test;
- CMocka.

В следующих подразделах мы познакомимся с двумя популярными фреймворками для тестирования: *CMocka*, написанным на C, и *Google Test*, написанным на C++. Мы не станем исследовать все их возможности, я лишь помогу вам получить общее представление о том, что такое фреймворк модульного тестирования. Дальнейшее самостоятельное изучение этой темы крайне приветствуется.

В следующем подразделе мы напишем модульные тесты для примера 22.1, используя фреймворк CMocka.

CMocka

Первое преимущество CMocka, на которое следует обратить внимание, таково: этот фреймворк написан исключительно на C и зависит только от стандартной библиотеки данного языка — никаких других библиотек он не использует. Поэтому тесты можно собирать компилятором C, что делает тестовую среду очень близкой к реальной. Фреймворк CMocka доступен для многих платформ, включая macOS, Linux и даже Microsoft Windows.

CMocka *де-факто* стандартный фреймворк для модульного тестирования в C. Он поддерживает *средства тестирования* (test fixtures), которые позволяют инициализировать и очищать тестовую среду перед выполнением тестового случая и после него. Вдобавок CMocka поддерживает *макетирование функций*, что очень полезно при написании макета какой-либо функции на C. Напомню, что макетную функцию можно сконфигурировать так, чтобы она возвращала соответствующее значение для определенного ввода. В примере 22.2 мы напишем макет для стандартной функции `rand`.

В листинге 22.10 показаны те же тестовые случаи, которые мы видели в примере 22.1, но переписанные с использованием CMocka. Мы поместили их в один файл, у которого есть своя функция `main`.

Листинг 22.10. Тестовые случаи на основе CMocka для примера 22.1 (ExtremeC_examples_chapter22_1_cmocka_tests.c)

```
// Нужно для библиотеки CMocka
#include <stdarg.h>
#include <stddef.h>
#include <setjmp.h>
#include <cmocka.h>
```

```
#include "ExtremeC_examples_chapter22_1.h"

int64_t input_value = -1;

int64_t feed_stub() {
  return input_value;
}

void calc_factorial__fact_of_zero_is_one(void** state) {
  input_value = 0;
  int64_t fact = calc_factorial(feed_stub);
  assert_int_equal(fact, 1);
}

void calc_factorial__fact_of_negative_is_one(void** state) {
  input_value = -10;
  int64_t fact = calc_factorial(feed_stub);
  assert_int_equal(fact, 1);
}

void calc_factorial__fact_of_5_is_120(void** state) {
  input_value = 5;
  int64_t fact = calc_factorial(feed_stub);
  assert_int_equal(fact, 120);
}
void next_even_number__even_numbers_should_be_returned(void** state) {
  assert_int_equal(next_even_number(), 0);
  assert_int_equal(next_even_number(), 2);
  assert_int_equal(next_even_number(), 4);
  assert_int_equal(next_even_number(), 6);
  assert_int_equal(next_even_number(), 8);
}

void next_even_number__numbers_should_rotate(void** state) {
  int64_t number = next_even_number();
  for (size_t i = 1; i <= 4; i++) {
    next_even_number();
  }
  int64_t number2 = next_even_number();
  assert_int_equal(number, number2);
}

int setup(void** state) {
  return 0;
}

int tear_down(void** state) {
  return 0;
}

int main(int argc, char** argv) {
  const struct CMUnitTest tests[] = {
```

```
    cmocka_unit_test(calc_factorial__fact_of_zero_is_one),
    cmocka_unit_test(calc_factorial__fact_of_negative_is_one),
    cmocka_unit_test(calc_factorial__fact_of_5_is_120),
    cmocka_unit_test(next_even_number__even_numbers_should_be_returned),
    cmocka_unit_test(next_even_number__numbers_should_rotate),
  };
  return cmocka_run_group_tests(tests, setup, tear_down);
}
```

В CMocka каждый тестовый случай должен возвращать `void` и принимать аргумент `void**`. Аргумент-указатель будет использоваться для получения информации, `state`, которая относится к отдельно взятому тестовому случаю. В функции `main` мы создадим список тестовых случаев и в конце вызовем `cmocka_run_group_tests`, чтобы выполнить все модульные тесты.

Помимо самих тестовых случаев, мы видим две новые функции: `setup` и `tear_down`. Как уже упоминалось, это средства тестирования. Они вызываются перед выполнением каждого тестового случая и после него и предназначены для его подготовки и уничтожения. Первое вызывается перед тестовым случаем, а второе — после него. Отмечу, что их можно было бы назвать как угодно, но для ясности были выбраны имена `setup` и `tear_down`.

Еще одно различие между тестовыми случаями, написанными с помощью CMocka, и теми, которые мы видели в предыдущих разделах, заключается в использовании других функций-утверждений. Библиотеки тестирования предоставляют широкое разнообразие утверждений, позволяющих получить дополнительные сведения о проваленном тесте, в то время как стандартная функция `assert` немедленно завершает программу, не предоставляя подробной информации. В представленном выше коде используется функция `assert_int_equal`, которая проверяет, равны ли два целых числа.

Чтобы скомпилировать данную программу, нужно сначала установить CMocka. В системах Linux на основе Debian для этого достаточно выполнить `sudo apt-get install libcmocka-dev`, а в macOS можно воспользоваться командой `brew install cmocka`. В Интернете можно найти множество справочных материалов, которые помогут вам с процессом установки.

Установив CMocka, вы можете собрать свой код, используя следующие команды (терминал 22.4).

Терминал 22.4. Сборка и запуск модульных тестов на основе CMocka для примера 22.1

```
$ gcc -g -c ExtremeC_examples_chapter22_1.c -o impl.o
$ gcc -g -c ExtremeC_examples_chapter22_1_cmocka_tests.c -o cmocka_tests.o
$ gcc impl.o cmocka_tests.o -lcmocka -o ex22_1_cmocka_tests.out
$ ./ex22_1_cmocka_tests.out
```

```
[==========] Running 5 test(s).
[ RUN      ] calc_factorial__fact_of_zero_is_one
[       OK ] calc_factorial__fact_of_zero_is_one
[ RUN      ] calc_factorial__fact_of_negative_is_one
[       OK ] calc_factorial__fact_of_negative_is_one
[ RUN      ] calc_factorial__fact_of_5_is_120
[       OK ] calc_factorial__fact_of_5_is_120
[ RUN      ] next_even_number__even_numbers_should_be_returned
[       OK ] next_even_number__even_numbers_should_be_returned
[ RUN      ] next_even_number__numbers_should_rotate
[       OK ] next_even_number__numbers_should_rotate
[==========] 5 test(s) run.
[  PASSED  ] 5 test(s).
$
```

Как видите, мы использовали флаг `-lcmocka`, чтобы скомпоновать нашу программу с установленной библиотекой CMocka. Вывод содержит имена тестовых случаев и количество пройденных тестов. Теперь отредактируем один из случаев, чтобы он провалился. Возьмем `next_even_number__even_numbers_should_be_returned` и изменим первое утверждение (листинг 22.11).

Листинг 22.11. Изменение одного из тестовых случаев на основе CMocka в примере 22.1

```
void next_even_number__even_numbers_should_be_returned(void** state) {
  assert_int_equal(next_even_number(), 1);
  ...
}
```

Теперь соберем тесты и выполним их заново (терминал 22.5).

Терминал 22.5. Сборка и запуск измененных модульных тестов на основе CMocka

```
$ gcc -g -c ExtremeC_examples_chapter22_1_cmocka_tests.c -o cmocka_tests.o
$ gcc impl.o cmocka_tests.o -lcmocka -o ex22_1_cmocka_tests.out
$ ./ex22_1_cmocka_tests.out
[==========] Running 5 test(s).
[ RUN      ] calc_factorial__fact_of_zero_is_one
[       OK ] calc_factorial__fact_of_zero_is_one
[ RUN      ] calc_factorial__fact_of_negative_is_one
[       OK ] calc_factorial__fact_of_negative_is_one
[ RUN      ] calc_factorial__fact_of_5_is_120
[       OK ] calc_factorial__fact_of_5_is_120
[ RUN      ] next_even_number__even_numbers_should_be_returned
[  ERROR   ] --- 0 != 0x1
[   LINE   ] --- .../ExtremeC_examples_chapter22_1_cmocka_tests.c:37: error:
Failure!
[  FAILED  ] next_even_number__even_numbers_should_be_returned
[ RUN      ] next_even_number__numbers_should_rotate
[       OK ] next_even_number__numbers_should_rotate
[==========] 5 test(s) run.
```

```
[  PASSED  ] 4 test(s).
[  FAILED  ] 1 test(s), listed below:
[  FAILED  ] next_even_number__even_numbers_should_be_returned

1 FAILED TEST(S)
$
```

В этом выводе видно, что один из тестовых случаев провалился, и причина указана в сообщении об ошибке среди журнальных записей. Не была пройдена проверка равенства целых чисел. Как уже объяснялось ранее, применение функции `assert_int_equal` вместо обычного вызова `assert` позволяет CMocka выводить в журнале выполнения полезную информацию, не ограничиваясь одной остановкой программы.

В следующем примере демонстрируется макетирование функций с помощью CMocka. Этот фреймворк позволяет создать макет функции, который при получении определенного ввода будет возвращать заданный результат.

В примере 22.2 мы хотим показать, как пользоваться этой возможностью. Мы возьмем стандартную функцию `rand` для генерирования случайных чисел. У нас также есть функция `random_boolean`, которая возвращает булево значение в зависимости от того, какое число мы получили из `rand`: четное или нечетное. Прежде чем переходить к макетированию, следует показать, как создается заглушка для функции `rand`. Вы увидите, что этот пример отличается от 22.1. В листинге 22.12 показано объявление функции `random_boolean`.

Листинг 22.12. Заголовочный файл в примере 22.2 (ExtremeC_examples_chapter22_2.h)

```
#ifndef _EXTREME_C_EXAMPLE_22_2_
#define _EXTREME_C_EXAMPLE_22_2_

#define TRUE 1
#define FALSE 0

typedef int bool_t;

bool_t random_boolean();

#endif
```

А в листинге 22.13 содержится определение.

Листинг 22.13. Определение функции random_boolean в примере 22.2 (ExtremeC_examples_chapter22_2.c)

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

#include "ExtremeC_examples_chapter22_2.h"
```

```
bool_t random_boolean() {
  int number = rand();
  return (number % 2);
}
```

Прежде всего нужно сказать, что мы не можем позволить `random_boolean` использовать настоящее определение функции `rand`, поскольку, как можно догадаться по ее имени, она генерирует случайные числа, а элемент случайности в наших тестах недопустим. Тесты проверяют результат на соответствие ожиданиям, поэтому предоставляемый ввод должен быть предсказуемым. Более того, определение `rand` — часть стандартной библиотеки C, такой как *glibc* в Linux, и его не получится легко заменить заглушкой, как мы сделали это в примере 22.1.

В предыдущем примере мы могли легко передать определению заглушки указатель на функцию. Но здесь функция `rand` используется напрямую. Мы не можем изменить определение `random_boolean`, поэтому должны придумать какое-то другое решение, которое позволит нам применять заглушку вместо `rand`.

Один за простейших способов подмены определения `rand` в C заключается в правке *символов* в итоговом объектном файле. В *таблице символов* объектного файла можно найти запись для `rand`, которая ссылается на соответствующее определение в стандартной библиотеке C. Если изменить эту запись так, чтобы она указывала на другое определение функции `rand` в наших тестовых двоичных файлах, то мы сможем легко подставить вместо `rand` нашу заглушку.

В листинге 22.14 показано, как мы определили функцию-заглушку вместе с тестами. Это очень похоже на то, что мы сделали в примере 22.1.

Листинг 22.14. Написание тестовых случаев CMocka с использованием функции-заглушки (ExtremeC_examples_chapter22_2_cmocka_tests_with_stub.c)

```
#include <stdlib.h>

// Нужно для библиотеки CMocka
#include <stdarg.h>
#include <stddef.h>
#include <setjmp.h>
#include <cmocka.h>

#include "ExtremeC_examples_chapter22_2.h"

int next_random_num = 0;

int __wrap_rand() {
  return next_random_num;
}

void test_even_random_number(void** state) {
  next_random_num = 10;
```

```
  assert_false(random_boolean());
}

void test_odd_random_number(void** state) {
  next_random_num = 13;
  assert_true(random_boolean());
}
int main(int argc, char** argv) {
  const struct CMUnitTest tests[] = {
    cmocka_unit_test(test_even_random_number),
    cmocka_unit_test(test_odd_random_number)
  };
  return cmocka_run_group_tests(tests, NULL, NULL);
}
```

В этом коде в основном используется тот же подход, который мы видели в тестах CMocka в листинге 22.10, написанных для примера 22.1. Соберем представленные выше файлы и выполним тесты (терминал 22.6). Мы ожидаем, что все они провалятся, поскольку, независимо от определения заглушки, `random_boolean` берет функцию `rand` из стандартной библиотеки C.

Терминал 22.6. Сборка и запуск модульных тестов на основе CMocka для примера 22.2

```
$ gcc -g -c ExtremeC_examples_chapter22_2.c -o impl.o
$ gcc -g -c ExtremeC_examples_chapter22_2_cmocka_tests_with_stub.c -o tests.o
$ gcc impl.o tests.o -lcmocka -o ex22_2_cmocka_tests_with_stub.out
$ ./ex22_2_cmocka_tests_with_stub.out
[==========] Running 2 test(s).
[ RUN      ] test_even_random_number
[  ERROR   ] --- random_boolean()
[   LINE   ] --- ExtremeC_examples_chapter22_2_cmocka_tests_with_stub.c:23:
 error: Failure!
[  FAILED  ] test_even_random_number
[ RUN      ] test_odd_random_number
[  ERROR   ] --- random_boolean()
[   LINE   ] --- ExtremeC_examples_chapter22_2_cmocka_tests_with_stub.c:28:
error: Failure!
[  FAILED  ] test_odd_random_number
[==========] 2 test(s) run.
[  PASSED  ] 0 test(s).
[  FAILED  ] 2 test(s), listed below:
[  FAILED  ] test_even_random_number
[  FAILED  ] test_odd_random_number

2 FAILED TEST(S)
$
```

Теперь пришло время прибегнуть к необычному приему и заменить определение, на которое ссылается символ `rand` в исполняемом файле `ex22_2_cmocka_tests_with_stub.out`. Обратите внимание: следующие команды рассчитаны только на системы Linux. Вот как это делается (терминал 22.7).

Терминал 22.7. Сборка и запуск модульных тестов на основе CMocka для примера 22.2 после создания обертки для символа rand

```
$ gcc impl.o tests.o -lcmocka -Wl,--wrap=rand -o
ex22_2_cmocka_tests_with_stub.out
$ ./ex22_2_cmocka_tests_with_stub.out
[==========] Running 2 test(s).
[ RUN      ] test_even_random_number
[       OK ] test_even_random_number
[ RUN      ] test_odd_random_number
[       OK ] test_odd_random_number
[==========] 2 test(s) run.
[  PASSED  ] 2 test(s).
$
```

Как видно в выводе, стандартная функция `rand` больше не вызывается, а вместо нее наша заглушка возвращает заданное нами значение. Чтобы функция `__wrap_rand` вызывалась вместо `rand`, при компоновке с помощью `gcc` использовался параметр `-Wl,--wrap=rand`.

Отмечу, что этот параметр доступен только в программе `ld` в Linux, поэтому для подмены функций в macOS или других системах, в которых нет компоновщика GNU, следует использовать другие приемы, такие как *взаимное позиционирование*.

Параметр `--wrap=rand` заставляет компоновщик обновить запись с символом `rand` в таблице символов итогового исполняемого файла, чтобы она ссылалась на определение функции `__wrap_rand`. Обратите внимание: это не произвольное имя и вы должны назвать свою заглушку именно так. Функцию `__wrap_rand` называют *оберткой*. После обновления таблицы символов любое обращение к функции `rand` будет приводить к вызову `__wrap_rand`. В этом можно убедиться, взглянув на таблицу символов в итоговом двоичном файле с тестами.

Помимо обновления символа `rand` в таблице символов, компоновщик также создает новую запись. В ней находится символ `__real_rand`, который ссылается непосредственно на определение стандартной функции `rand`. Поэтому если нам нужно вызвать `rand`, то мы можем сделать это с помощью функции `__real_rand`. Это отличный пример использования таблицы символов и ее содержимого для вызова заглушки, хотя некоторым людям он не по душе; они предпочитают совершать предварительную загрузку разделяемого объекта, который служит оберткой для настоящей функции `rand`. К какому бы способу вы ни прибегли, вам в конечном счете необходимо перенаправлять вызовы символа `rand` к другой функции.

Представленный выше механизм мог бы послужить основой для демонстрации макетирования функций в CMocka. Вместо глобальной переменной `next_random_num`, как в листинге 22.14, мы можем использовать макетную функцию, чтобы вернуть определенное значение. В листинге 22.15 можно видеть те же тесты CMocka, только с применением макетной функции, которая читает тестовый ввод.

Листинг 22.15. Написание тестовых случаев на основе CMocka с использованием макетной функции (ExtremeC_examples_chapter22_2_cmocka_tests_with_mock.c)

```
#include <stdlib.h>

// Нужно для библиотеки CMocka
#include <stdarg.h>
#include <stddef.h>
#include <setjmp.h>
#include <cmocka.h>
#include "ExtremeC_examples_chapter22_2.h"

int __wrap_rand() {
  return mock_type(int);
}

void test_even_random_number(void** state) {
  will_return(__wrap_rand, 10);
  assert_false(random_boolean());
}

void test_odd_random_number(void** state) {
  will_return(__wrap_rand, 13);
  assert_true(random_boolean());
}
int main(int argc, char** argv) {
  const struct CMUnitTest tests[] = {
    cmocka_unit_test(test_even_random_number),
    cmocka_unit_test(test_odd_random_number)
  };
  return cmocka_run_group_tests(tests, NULL, NULL);
}
```

Теперь, зная о вызове функции-обертки `__wrap_rand`, мы можем объяснить код, относящийся к макетированию. Макет создается с помощью пары функций `will_return` и `mock_type`. Сначала вызывается функция `will_return`; она определяет значение, которое должен вернуть макет. Затем, когда вызывается макет (в данном случае `__wrap_rand`), функция `mock_type` возвращает заданное значение.

В качестве примера мы указали, что вызов `__wrap_rand` должен возвращать `10`, использовав для этого выражение `will_return(__wrap_rand, 10)`. В результате, когда внутри `__wrap_rand` вызывается функция `mock_type`, возвращается значение `10`. Следует отметить: каждый вызов `will_return` должен использоваться в паре с `mock_type`, иначе тест провалится. Как следствие, если `__wrap_rand` по какой-то причине не вызывается, то это приводит к провалу теста.

В завершение отмечу, что вывод представленного выше кода будет таким же, как показанный в листингах 22.6 и 22.7. Кроме того, для сборки и запуска тестов к исходному файлу `ExtremeC_examples_chapter22_2_cmocka_tests_with_mock.c` следует применять те же команды.

В этом подразделе мы рассмотрели, как с помощью библиотеки CMocka создавать тестовые случаи, делать утверждения и писать макетные функции. Далее речь пойдет о Google Test — еще одном фреймворке, который можно использовать для модульного тестирования программ на C.

Google Test

Google Test — фреймворк, пригодный для модульного тестирования программ на C и C++. Он написан на языке C++, но подходит и для тестирования кода на C. Некоторые считают, что так делать не стоит, поскольку в этом случае тестовая и реальная среды используют разные компиляторы и компоновщики.

Прежде чем приступать к написанию тестовых случаев для примера 22.1 с помощью Google Test, необходимо сначала поправить имеющийся заголовочный файл. В листинге 22.16 показана его измененная версия.

Листинг 22.16. Измененный заголовочный файл в примере 22.1 (ExtremeC_examples_chapter22_1.h)

```
#ifndef _EXTREME_C_EXAMPLE_22_1_
#define _EXTREME_C_EXAMPLE_22_1_

#include <stdint.h>
#include <unistd.h>

#if __cplusplus
extern "C" {
#endif

typedef int64_t (*int64_feed_t)();

int64_t next_even_number();

int64_t calc_factorial(int64_feed_t feed);

#if __cplusplus
}
#endif

#endif
```

Как видите, мы разместили определение в блоке `extern C { ... }`, который остается в программе только при наличии макроса `_cplusplus`. В результате этого изменения, в случае использования компилятора C++, символы в итоговых объектных файлах не должны декорироваться. Иначе, когда компоновщик попытается найти определение *декорированных символов*, произойдет ошибка компоновки. Если вы незнакомы с *декорированием имен* в C++, то, пожалуйста, обратитесь к последнему разделу главы 2.

Теперь напишем тестовые случаи с использованием Google Test (листинг 22.17).

Листинг 22.17. Тестовые случаи для примера 22.1, написанные с помощью Google Test (ExtremeC_examples_chapter22_1_gtests.cpp)

```
// Нужно для библиотеки Google Test
#include <gtest/gtest.h>
#include "ExtremeC_examples_chapter22_1.h"

int64_t input_value = -1;

int64_t feed_stub() {
  return input_value;
}

TEST(calc_factorial, fact_of_zero_is_one) {
  input_value = 0;
  int64_t fact = calc_factorial(feed_stub);
  ASSERT_EQ(fact, 1);
}

TEST(calc_factorial, fact_of_negative_is_one) {
  input_value = -10;
  int64_t fact = calc_factorial(feed_stub);
  ASSERT_EQ(fact, 1);
}

TEST(calc_factorial, fact_of_5_is_120) {
  input_value = 5;
  int64_t fact = calc_factorial(feed_stub);
  ASSERT_EQ(fact, 120);
}

TEST(next_even_number, even_numbers_should_be_returned) {
  ASSERT_EQ(next_even_number(), 0);
  ASSERT_EQ(next_even_number(), 2);
  ASSERT_EQ(next_even_number(), 4);
  ASSERT_EQ(next_even_number(), 6);
  ASSERT_EQ(next_even_number(), 8);
}

TEST(next_even_number, numbers_should_rotate) {
  int64_t number = next_even_number();
  for (size_t i = 1; i <= 4; i++) {
    next_even_number();
  }
  int64_t number2 = next_even_number();
   ASSERT_EQ(number, number2);
}

int main(int argc, char** argv) {
  ::testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```

Тестовые случаи определены с помощью макроса `TEST(...)`. Это пример того, насколько легко макросы позволяют сформировать DSL. Существуют и другие макросы, такие как `TEST_F(...)` и `TEST_P(...)`, но они предназначены для C++. Первый аргумент макроса представляет собой имя тестового класса (фреймворк Google Test написан для объектно-ориентированного кода на C++), который можно считать набором тестов, содержащим ряд тестовых случаев. Второй аргумент — имя тестового случая.

Обратите внимание: макрос `ASSERT_EQ` позволяет делать утверждения о равенстве объектов, а не только целых чисел. Google Test предоставляет большое количество макросов для проверки ожиданий, что делает этот проект полноценным фреймворком для модульного тестирования. Вдобавок отмечу: представленный выше код нужно собирать с помощью компилятора, совместимого с C++11, такого как `g++` или `clang++`.

Для сборки кода используются команды, показанные в терминале 22.8. Обратите внимание: мы выбрали компилятор `g++` и передаем ему параметр `-std=c++11`, который требует, чтобы использовался стандарт C++11.

Терминал 22.8. Сборка и запуск модульных тестов Google Test для примера 22.1

```
$ gcc -g -c ExtremeC_examples_chapter22_1.c -o impl.o
$ g++ -std=c++11 -g -c ExtremeC_examples_chapter22_1_gtests.cpp -o gtests.o
$ g++ impl.o gtests.o -lgtest -lpthread -o ex19_1_gtests.out
$ ./ex19_1_gtests.out
[==========] Running 5 tests from 2 test suites.
[----------] Global test environment set-up.
[----------] 3 tests from calc_factorial
[ RUN      ] calc_factorial.fact_of_zero_is_one
[       OK ] calc_factorial.fact_of_zero_is_one (0 ms)
[ RUN      ] calc_factorial.fact_of_negative_is_one
[       OK ] calc_factorial.fact_of_negative_is_one (0 ms)
[ RUN      ] calc_factorial.fact_of_5_is_120
[       OK ] calc_factorial.fact_of_5_is_120 (0 ms)
[----------] 3 tests from calc_factorial (0 ms total)

[----------] 2 tests from next_even_number
[ RUN      ] next_even_number.even_numbers_should_be_returned
[       OK ] next_even_number.even_numbers_should_be_returned (0 ms)
[ RUN      ] next_even_number.numbers_should_rotate
[       OK ] next_even_number.numbers_should_rotate (0 ms)
[----------] 2 tests from next_even_number (0 ms total)

[----------] Global test environment tear-down
[==========] 5 tests from 2 test suites ran. (1 ms total)
[  PASSED  ] 5 tests.
$
```

Этот вывод похож на тот, который мы получили при использовании CMocka. Он сигнализирует о прохождении пяти тестовых случаев. Изменим тот же тестовый случай, что и в примере с CMocka, с целью нарушить работу тестового набора (листинг 22.18).

Листинг 22.18. Изменение одного из тестовых случаев, написанных с использованием Google Test

```
TEST(next_even_number, even_numbers_should_be_returned) {
  ASSERT_EQ(next_even_number(), 1);
  ...
}
```

Снова их соберем и запустим (терминал 22.9).

Терминал 22.9. Сборка и запуск модульных тестов Google Test для примера 22.1 после изменения одного из тестовых случаев

```
$ g++ -std=c++11 -g -c ExtremeC_examples_chapter22_1_gtests.cpp –o gtests.o
$ g++ impl.o gtests.o -lgtest -lpthread -o ex22_1_gtests.out
$ ./ex22_1_gtests.out
[==========] Running 5 tests from 2 test suites.
[----------] Global test environment set-up.
[----------] 3 tests from calc_factorial
[ RUN      ] calc_factorial.fact_of_zero_is_one
[       OK ] calc_factorial.fact_of_zero_is_one (0 ms)
[ RUN      ] calc_factorial.fact_of_negative_is_one
[       OK ] calc_factorial.fact_of_negative_is_one (0 ms)
[ RUN      ] calc_factorial.fact_of_5_is_120
[       OK ] calc_factorial.fact_of_5_is_120 (0 ms)
[----------] 3 tests from calc_factorial (0 ms total)

[----------] 2 tests from next_even_number
[ RUN      ] next_even_number.even_numbers_should_be_returned
.../ExtremeC_examples_chapter22_1_gtests.cpp:34: Failure
Expected equality of these values:
  next_even_number()
    Which is: 0
  1
[  FAILED  ] next_even_number.even_numbers_should_be_returned (0 ms)
[ RUN      ] next_even_number.numbers_should_rotate
[       OK ] next_even_number.numbers_should_rotate (0 ms)
[----------] 2 tests from next_even_number (0 ms total)

[----------] Global test environment tear-down
[==========] 5 tests from 2 test suites ran. (0 ms total)
[  PASSED  ] 4 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] next_even_number.even_numbers_should_be_returned

 1 FAILED TEST
$
```

Google Test, в точности как CMocka, указывает на то, где провалились тесты, и выводит информативный отчет. В завершение следует сказать, что Google Test тоже поддерживает средства тестирования, но не так, как это делает CMocka. В Google Test они имеют вид *тестовых классов*.

Для создания макетных объектов и макетирования в целом можно использовать библиотеку Google Mock (или gmock), но в этой книге она не рассматривается.

В данном разделе мы обсудили две самые известные библиотеки для модульного тестирования на C. Вторая часть главы посвящена навыкам отладки, которыми, несомненно, должен владеть любой программист.

Отладка

Бывают случаи, когда проваливается один тест или группа тестов. Вдобавок иногда можно столкнуться с программной ошибкой. В обеих ситуациях вы имеете дело с дефектом в коде, который нужно найти и исправить. Для этого следует выполнить множество сеансов отладки, пройтись по исходному коду в поисках причины неполадок и составить план необходимых исправлений. Но что такое *отладка* ПО?

Бытует мнение, будто термин debug (отладка) появился во времена, когда компьютеры были такими большими, что в системное оборудование могли забираться настоящие насекомые (bugs) типа мотыльков, вызывая неполадки. Обязанностью некоторых работников с официальной должностью debuggers (дословно «борцы с насекомыми» — то, что мы называем отладчиками) было вылавливать насекомых из оборудования в аппаратном зале. Более подробно об этом можно почитать на странице https://ru.wikipedia.org/wiki/Отладка_программы.

Отладка — это расследование, целью которого является определение причины наблюдаемого дефекта путем анализа внутренностей или внешнего поведения программы. Обычно, запуская программу, мы рассматриваем ее в качестве некоего черного ящика. Но когда с результатами что-то не так или работа программы почему-то прерывается, нам нужно заглянуть поглубже и выяснить, откуда берется данная проблема. Это значит, что программу следует анализировать по принципу белого ящика, в котором все видно.

Вот почему у нас может быть две разные сборки программы: *сборка выпуска* и *отладочная сборка*. В сборке выпуска акцент на работе и функциональности, а программа в основном рассматривается в качестве черного ящика; в отладочных сборках мы можем отслеживать любые происходящие события и анализировать программу

по принципу белого ящика. Отладочные сборки в основном предназначены для разработки и тестирования, а сборки выпуска используются для развертывания и работы в реальных условиях.

Чтобы получить отладочную сборку, все элементы программного проекта или какая-то их часть должны содержать *отладочные* символы, которые позволяют разработчику выполнять *трассировку стека* программы и отслеживать ее поток выполнения. Обычно продукты сборки выпуска (исполняемый файл или библиотеки) не подходят для отладки — они недостаточно прозрачные для того, чтобы внешний наблюдатель мог исследовать внутренности программы. В главах 4 и 5 была описана процедура сборки исходников C для последующей отладки.

Для отладки программ, как правило, используются отладчики. Отладчик — это самостоятельный инструмент, который подключается к заданному процессу, чтобы управлять им или отслеживать его работу. Отладчики — основное средство исследования проблем, но есть и другие инструменты, которые можно использовать для анализа памяти, конкурентных потоков выполнения или производительности программы. Мы поговорим о них чуть позже.

Существенная часть программных ошибок подлежит *воспроизведению*, но есть и такие ошибки, которые нельзя воспроизвести или наблюдать во время сеанса отладки. Это в основном связано с *эффектом наблюдателя*, согласно которому при попытке заглянуть внутрь программы вы влияете на то, как она работает, и можете предотвратить появление ряда дефектов. Такого рода проблемы являются катастрофическими, и их зачастую очень сложно исправить, поскольку вы не можете использовать средства отладки для поиска их причин!

К этой категории можно отнести некоторые ошибки с управлением потоками выполнения в высокопроизводительных средах.

В следующих подразделах речь пойдет о разных видах программных ошибок. Кроме того, будут представлены инструменты, которые используются для исследования дефектов в современной разработке на C/C++.

Категории программных ошибок

За годы эксплуатации программного продукта можно получить тысячи отчетов об ошибках. Но категорий, на которые можно разделить эти ошибки, не так уж много. Ниже перечислены виды программных дефектов, которые, как мне кажется, имеют большое значение и для устранения которых требуются специальные навыки. Конечно, это не полный список; есть другие виды проблем, которые не упоминаются в этом перечне.

- *Логические ошибки*. Для исследования таких ошибок вам необходимо ориентироваться в коде и его потоках выполнения. Чтобы увидеть поток выполнения

программы, к запущенному процессу нужно подключить отладчик. Только после этого данный поток можно будет *трассировать* и анализировать. При отладке программы можно также использовать *журнальные записи работы*, особенно если в итоговых двоичных файлах нет отладочных символов или если отладчик нельзя подключить к активному экземпляру программы.

- *Ошибки памяти*. Связаны с памятью. Обычно их вызывают висячие указатели, переполнения буфера, двойное освобождение и т. д. Эти ошибки необходимо исследовать с помощью *профилировщика памяти*, который играет роль средства отладки для мониторинга и наблюдения за памятью.
- *Ошибки конкурентности*. Многопроцессным и многопоточным программам всегда были свойственны одни из самых трудно выявляемых ошибок в компьютерной промышленности. Для обнаружения особенно коварных проблем, таких как состояние гонки или гонка данных, требуются специальные инструменты наподобие *средств отладки потоков* (thread sanitizers).
- *Проблемы с производительностью*. Обновление программного обеспечения может привести к *ухудшению производительности*. Такие проблемы нужно исследовать за счет дополнительного тестирования или даже отладки. Журнальные записи выполнения, содержащие хронологические сведения о предыдущих прогонах программы с примечаниями, могут помочь в поиске конкретного изменения или изменений, которые вызвали ухудшения производительности.

Далее мы поговорим о различных инструментах, упомянутых в этом списке.

Отладчики

Мы уже обсуждали отладчики, особенно `gdb`, в главе 4 и заглядывали с их помощью в память приложения. В данном разделе вернемся к этой теме и рассмотрим, какую роль отладчики играют в повседневной разработке ПО. Ниже перечислены распространенные возможности, доступные в большинстве отладчиков.

- Отладчик, как и любая другая программа, выполняется в виде процесса. Процесс отладчика может присоединиться к другому процессу с заданным идентификатором.
- После успешного присоединения отладчик может управлять выполнением инструкций в заданном процессе; таким образом, пользователь может останавливать и возобновлять поток выполнения заданного процесса с помощью интерактивного сеанса отладки.
- Отладчики могут заглядывать внутрь защищенной памяти процесса. Они также способны изменять ее содержимое, благодаря чему разработчик может выполнить ту же группу инструкций после намеренного внесения изменений в память.

- Почти все известные отладчики могут отследить исходный код, который относится к той или иной инструкции; для этого лишь требуется, чтобы переносимые объектные файлы содержали отладочные символы. Иными словами, остановившись на инструкции, вы можете перейти к соответствующей строчке исходного кода.
- Если в заданном объектном файле нет отладочных символов, то отладчик может показать дизассемблированный код соответствующей инструкции, что тоже может быть полезно.
- Некоторые отладчики предназначены только для определенных языков, но большинство из них универсальны. Языки на основе *виртуальной машины Java* (Java Virtual Machine, JVM), такие как Java, Scala и Groovy, должны использовать отладчики JVM, позволяющие просматривать и управлять внутренностями экземпляра JVM.
- У интерпретируемых языков, таких как Python, тоже есть свои отладчики, которые позволяют останавливать скрипты и управлять их выполнением. Для JVM и скриптовых языков тоже можно использовать низкоуровневые средства отладки наподобие `gdb`, однако они будут анализировать процесс виртуальной машины или интерпретатора, а не сам байт-код Java или скрипт Python.

Перечень отладчиков можно найти на странице в «Википедии»: https://en.wikipedia.org/wiki/List_of_debuggers. В этом списке можно отметить следующие проекты.

1. *Advanced Debugger (adb)* — стандартный отладчик в Unix. У него есть разные реализации для различных Unix-систем. Отладчик по умолчанию в Solaris Unix.
2. *GNU Debugger (gdb)* — версия Unix-отладчика от проекта GNU. Используется по умолчанию во многих Unix-подобных операционных системах, включая Linux.
3. *LLDB* — отладчик, который в основном предназначен для отладки объектных файлов, генерируемых компиляторами LLVM.
4. *Python Debugger* — используется для отладки Python-сценариев.
5. *Java Platform Debugger Architecture (JPDA)* — не совсем отладчик, а, скорее, API, предназначенный для отладки программ, которые выполняются внутри JVM.
6. *OllyDbg* — отладчик и дизассемблер, применяемый в Microsoft Windows для отладки приложений с графическим пользовательским интерфейсом.
7. *Microsoft Visual Studio Debugger* — основной отладчик, применяемый в Microsoft Visual Studio.

Помимо `gdb`, можно также использовать `cgdb`. Эта программа выводит консольный редактор кода рядом с интерактивной командной оболочкой `gdb`, чтобы вам было легче перемещаться по коду.

В этом подразделе мы обсудили отладчики — основное средство для исследования проблем в ПО. Дальше речь пойдет о профилировщиках памяти, незаменимых при анализе программных ошибок, которые связаны с памятью.

Средства проверки памяти

Иногда при возникновении ошибки или сбоя, связанного с памятью, одного лишь отладчика недостаточно. Вам нужен другой инструмент, способный обнаруживать повреждения памяти и некорректные обращения на чтение или запись к отдельным ячейкам. Речь идет о *средстве проверки* или *профилировщике памяти*. Профилировщик может входить в состав отладчика, но обычно выступает отдельной программой, и его подход к обнаружению некорректного поведения отличается от процедуры отладки.

От средства проверки памяти обычно можно ожидать таких возможностей, как:

- подсчет общего объема выделенной, освобожденной и задействованной статической памяти, а также количества выделений кучи, стека и т. д.;
- обнаружение утечек памяти, что можно считать важнейшей функцией профилировщиков;
- обнаружение некорректных операций чтения/записи, таких как выход за пределы буферов или массивов, запись на уже освобожденный участок памяти и т. д.;
- обнаружение проблемы *двойного освобождения*. Она возникает, когда программа пытается освободить область памяти, которая уже подверглась этой операции.

Мы уже рассматривали такие средства проверки памяти, как *Memcheck* (один из инструментов Valgrind) в предыдущих главах, в частности в главе 5. Там же обсуждались разные виды профилировщиков и средств проверки памяти. Здесь мы еще раз вернемся к данной теме и рассмотрим кое-какие подробности о каждом из этих инструментов.

Все средства проверки памяти делают одно и то же, но внутренние механизмы мониторинга операций с памятью, которые они используют, могут отличаться. Поэтому я разделю их в зависимости от тех подходов, которые в них применяются.

1. *Переопределение на этапе компиляции*. Чтобы использовать средство проверки памяти на основе этой методики, вам необходимо внести некоторые (обычно незначительные) изменения в свой исходный код — например, подключить заголовочный файл соответствующей библиотеки. Затем вам нужно заново скомпилировать свои двоичные файлы. Иногда двоичные файлы приходится компоновать с библиотекой, предоставляемой средством проверки памяти. Преимущество данного подхода в том, что он меньше всего влияет на производительность программы по сравнению с другими методиками. Но у него есть и недостаток: двоичные файлы приходится перекомпилировать. Среди профилировщиков памяти, которые используют этот подход, можно выделить LLVM AddressSanitizer (ASan), Memwatch, Dmalloc и Mtrace.

2. *Переопределение на этапе компоновки*. Данный вид средств проверки памяти похож на предыдущий, но отличается тем, что вам не нужно изменять свой исходный код. Вместо этого итоговые двоичные файлы достаточно скомпоновать с предоставленной библиотекой. Для проверки памяти на этапе компоновки можно использовать утилиту *heap checker* и *gperftools*.
3. *Перехват на этапе выполнения*. Средство проверки памяти на основе этого подхода выступает медиатором между программой и ОС, пытаясь перехватывать и отслеживать все операции работы с памятью и сообщать о любом некорректном поведении или обращении не по тому адресу. Оно также может создавать отчеты об утечках, вычисляя общий объем выделенных и освобожденных блоков памяти. Основным преимуществом данной методики является то, что для его использования вам не нужно заново компилировать или компоновать свою программу. Его большой недостаток заключается в существенных накладных расходах при выполнении программы. К тому же расход памяти будет намного больше, чем в случае запуска программы без профилировщика. Это явно не идеальная среда для отладки высокопроизводительных и встраиваемых программ. Для перехвата на этапе выполнения можно использовать утилиту Memcheck, входящую в состав Valgrind. Такие профилировщики следует применять к отладочной сборке кодовой базы.
4. *Предварительная загрузка библиотек*. Некоторые профилировщики применяют *взаимное позиционирование* для создания оберток вокруг стандартных функций для работы с памятью. Таким образом, предварительно загружая предоставленные разделяемые библиотеки с помощью переменной среды `LD_PRELOAD`, программа может обращаться к функциям-оберткам, а профилировщик может перехватывать вызовы соответствующих стандартных функций для работы с памятью.

Обычно для устранения всех проблем, связанных с памятью, одного инструмента недостаточно, поскольку у каждого из них есть сильные и слабые стороны, из-за которых его имеет смысл использовать только в определенном контексте.

В этом разделе мы прошлись по существующим профилировщикам памяти и разделили их по группам в зависимости от того, как именно они записывают операции выделения и освобождения памяти. Следующий подраздел посвящен *средствам отладки потоков*.

Средства отладки потоков

Средства отладки потоков — это программы, которые позволяют отлаживать многопоточные процессы и находить в них проблемы, связанные с конкурентностью. Вот лишь несколько проблем, которые они способны выявить:

- гонки данных и точное местонахождение операций чтения/записи, в которых они возникают. Эти операции могут находиться в разных потоках;

- неправильное использование API для работы с потоками, особенно того, который входит в стандарт POSIX и применяется в POSIX-совместимых системах;
- потенциальная взаимная блокировка;
- проблемы с порядком блокировки.

И средства отладки потоков, и профилировщики памяти могут иметь *ложные срабатывания*. Иными словами, могут находить проблемы, которые, как становится очевидно после дальнейшего анализа, таковыми не являются. Это очень зависит от подхода к отслеживанию событий, которые используют эти библиотеки, и от того, как принимается окончательное решение.

В следующем списке представлен ряд широко известных средств отладки потоков.

- *Helgrind (om Valgrind)* — еще один инструмент из состава Valgrind, в основном применяемый для отладки потоков. Кроме того, следует упомянуть о DRD — аналогичном средстве, которое тоже является частью Valgrind. Списки возможностей Helgrind и DRD находятся по ссылкам http://valgrind.org/docs/manual/hg-manual.html и http://valgrind.org/docs/manual/drd-manual.html. Helgrind, как и любой другой инструмент из проекта Valgrind, не требует изменения исходного кода. Для его запуска необходимо выполнить команду `valgrind --tool=helgrind [path-to-executable]`.
- *Intel Inspector* — данный преемник *Intel Thread Checker* проводит анализ ошибок потоков выполнения и проблем с памятью. Поэтому его можно считать как отладчиком потоков, так и профилировщиком памяти. В отличие от бесплатного Valgrind, для его использования необходимо приобрести подходящую лицензию.
- *LLVM ThreadSanitizer (TSan)* — часть набора инструментов LLVM, которая поставляется вместе с утилитой LLVM AddressSanitizer, описанной в предыдущем подразделе. Чтобы использовать данный отладчик, необходимо внести небольшие изменения на этапе компиляции и заново собрать кодовую базу.

В этом разделе мы обсудили средства отладки потоков и познакомились с некоторыми инструментами, позволяющими отлаживать проблемы с многопоточностью. Далее рассмотрим программы и наборы инструментов для оптимизации производительности программы.

Профилировщики производительности

Иногда результаты прохождения группы нефункциональных тестов сигнализируют об ухудшении производительности. Исследование таких проблем проводится специальными инструментами. В этом подразделе мы проведем краткий обзор средств, с помощью которых можно проанализировать производительность и выявить узкие места.

Такие профилировщики производительности обычно предлагают некоторые из следующих возможностей:

- сбор статистики о каждом вызове функции;
- предоставление *графа вызовов функций* для трассировки вызовов;
- сбор статистики относительно памяти для каждого вызова функции;
- сбор статистики о конфликтах при блокировке;
- сбор статистики о выделении/освобождении памяти;
- анализ кэша, предоставление статистики обращений к кэшу и выделение участков кода, которые плохо оптимизированы для работы с кэшем;
- сбор статистики о событиях, относящихся к многопоточности и синхронизации.

Ниже перечислены самые известные программы и наборы инструментов, которые можно использовать для профилирования производительности.

- *Google Performance Tools (gperftools)* — на самом деле это производительная реализация `malloc`, но, как утверждается на главной странице данного проекта, он предоставляет ряд средств анализа производительности, таких как *heap checker*, который в предыдущих разделах был представлен в качестве профилировщика памяти. Условием его использования является его компоновка с итоговым двоичным файлом.
- *Callgrind (из состава Valgrind)* — в основном собирает статистику о вызовах функций и отношениях между вызывающей и вызываемой функциями. Вам не нужно изменять исходный код или компоновать итоговые двоичные файлы. Этот инструмент можно использовать динамически, но, конечно, только в сочетании с отладочной сборкой.
- *Intel VTune* — это пакет профилирования производительности от Intel, который поддерживает все возможности, описанные в предыдущем списке. Его использование требует приобретения подходящей лицензии.

Резюме

Эта глава была посвящена модульному тестированию и отладке программ на языке C. Подытожим:

- мы поговорили о тестировании и увидели его важную роль для программистов и команд разработки;
- обсудили разные уровни тестирования: модульное, интеграционное и системное;
- рассмотрели функциональное и нефункциональное тестирование;

- выяснили, что такое регрессионное тестирование;
- исследовали две общеизвестные библиотеки тестирования для C, CMocka и Google Test, а также привели некоторые примеры;
- поговорили об отладке и разных видах программных дефектов;
- обсудили отладчики, профилировщики памяти, отладчики потоков и профилировщики производительности, которые позволяют более успешно исследовать программные проблемы.

В следующей главе речь пойдет о *системах сборки*, доступных для проектов на языке C. Вы узнаете, что такое система сборки и какие возможности она предоставляет; в конечном счете это поможет нам автоматизировать процесс сборки крупных проектов на C.

23 Системы сборки

Для нас, программистов, сборка проекта и запуск различных его компонентов — первый этап в разработке новой возможности или исправлении программной ошибки. На самом деле это касается не только C или C++; почти любой проект, написанный на компилируемом языке программирования, таком как C, C++, Java или Go, нужно сначала собрать.

Таким образом, возможность быстро и легко собирать программные проекты — фундаментальное требование со стороны практически любой стороны, вовлеченной в процесс создания ПО, будь то разработчики, тестировщики, интеграторы, инженеры DevOps или даже служба поддержки.

Более того, когда к команде присоединяется новичок, он первым делом собирает кодовую базу, над которой ему нужно будет работать. Учитывая все вышесказанное, можно с уверенностью утверждать, что вопрос сборки программного проекта заслуживает нашего внимания; это важный элемент процесса разработки ПО.

Программистам регулярно приходится собирать свою кодовую базу, чтобы видеть результаты вносимых изменений. Сборка проекта, состоящего всего из нескольких исходных файлов, выглядит простой и быстрой, но по мере увеличения количества исходников (что, поверьте, не редкость) сборка кодовой базы становится для разработчиков реальным препятствием. Поэтому надлежащий механизм сборки программных проектов крайне важен.

Когда-то люди писали скрипты командной оболочки для сборки огромного количества исходных файлов. Это работало, однако для использования данных скриптов в разных проектах нужно было сделать их достаточно универсальными, что требовало много усилий. Позже, около 1976 года, в Bell Labs создали первую (или по крайней мере одну из первых) *систему сборки*. Она называлась *Make* и применялась во внутренних проектах.

В итоге Make начали использовать повсеместно во всех проектах на C и C++ и даже там, где эти языки программирования не были основными.

В данной главе мы обсудим широко распространенные *системы сборки* и *генераторы скриптов сборки* для проектов на C и C++. Здесь мы:

- сначала рассмотрим, что такое системы сборки и для чего они нужны;
- затем познакомимся с Make и узнаем, как нужно использовать файлы Makefile;
- далее перейдем к CMake, поговорим о генераторах скриптов сборки и напишем простые файлы `CMakeLists.txt`;
- разберемся, что такое система Ninja и чем она отличается от Make;
- выясним, как задействовать CMake для генерации скриптов сборки Ninja;
- подробно обсудим проект Bazel и способы его применения, рассмотрим файлы `WORKSPACE` и `BUILD` и простой пример их использования;
- просмотрим ссылки на уже опубликованные сравнения различных систем сборки.

Следует отметить, что все средства сборки, которые используются в этой главе, должны быть установлены заранее. Соответствующие материалы и документацию можно найти в Интернете, поскольку эти инструменты применяются в огромных масштабах.

В первом разделе я попробую объяснить, что собой представляют системы сборки.

Что такое система сборки

Говоря простым языком, система сборки — набор программ и сопутствующих текстовых файлов, которые позволяют собрать кодовую базу программного обеспечения. В наши дни у любого языка программирования есть собственные системы сборки. Например, в Java есть *Ant*, *Maven*, *Gradle* и т. д. Но что означает «собрать кодовую базу»?

Сборка кодовой базы — получение конечных продуктов компиляции из исходных файлов. Например, если говорить о C, то конечными продуктами могут быть исполняемые файлы, разделяемые объектные файлы или статические библиотеки, и задача системы сборки состоит в том, чтобы сгенерировать их из исходных файлов, из которых состоит кодовая база. То, какие именно операции нужно для этого выполнить, во многом зависит от языков программирования, на которых написан исходный код.

Многие современные системы сборки, особенно в проектах, написанных на *JVM-языках*, таких как Java или Scala, имеют дополнительную функцию. Они занимаются

управлением зависимостями. Это значит, что система сборки находит все зависимости в кодовой базе, загружает их и использует загруженные артефакты в *процессе сборки*. Это очень удобно, особенно если проект имеет множество зависимостей (что характерно для крупных кодовых баз).

Например, в проектах на Java одной из известнейших систем сборки является *Maven*; она использует XML-файлы и поддерживает управление зависимостями. К сожалению, в C/C++ хороших средств для управления зависимостями не существует. О том, почему в мире C/C++ все еще нет систем наподобие Maven, есть разные мнения, но факт отсутствия этих систем может быть признаком того, что они нам не нужны.

Еще один аспект систем сборки — их способность собирать огромные проекты с множеством модулей внутри. Конечно, то же самое можно сделать с помощью скриптов командной оболочки или путем написания рекурсивных файлов Makefile, которые могут перебирать модули на любом количестве уровней, но мы говорим о встроенной поддержке такой возможности. К сожалению, Make не умеет делать этого по умолчанию. А вот другое известное средство сборки, CMake, может нам помочь. Мы еще вернемся к этому в разделе, посвященном CMake.

На сегодня многие проекты используют Make в качестве стандартной системы сборки, действуя через CMake. Это один из аспектов инструмента CMake, который делает его чрезвычайно важным, и вы должны научиться с ним работать, прежде чем присоединяться к проекту на C/C++. Отмечу, что CMake можно использовать не только для C и C++, но и для других языков программирования.

В следующем разделе речь пойдет о системе сборки Make и о том, как она собирает проекты. Мы рассмотрим пример проекта на языке C с несколькими модулями и будем использовать его на страницах этой главы для изучения разных систем сборки, которые можно в нем применять.

Make

Система сборки Make использует файлы Makefile. Makefile — текстовый файл (без какого-либо расширения), который находится в каталоге с исходниками и содержит *цели сборки* и команды, позволяющие Make знать, как собирать текущую кодовую базу.

Начнем с простого многомодульного проекта на C и добавим в него Make. В терминале 23.1 показаны файлы и каталоги, из которых состоит проект. Как видите, у него есть модуль `calc`, который используется другим модулем, `exec`.

Результатом сборки модуля `calc` будет статическая объектная библиотека, а модуль `exec` собирается в исполняемый файл:

Терминал 23.1. Файлы и каталоги проекта

```
$ tree ex23_1
ex23_1/
├── calc
│   ├── add.c
│   ├── calc.h
│   ├── multiply.c
│   └── subtract.c
└── exec
    └── main.c

2 directories, 5 files
$
```

Чтобы собрать этот проект с помощью системы сборки и получить соответствующие продукты, необходимо выполнить команды, представленные в терминале 23.2. Обратите внимание: в качестве платформы для этого проекта используется Linux.

Терминал 23.2. Сборка проекта

```
$ mkdir -p out
$ gcc -c calc/add.c -o out/add.o
$ gcc -c calc/multiply.c -o out/multiply.o
$ gcc -c calc/subtract.c -o out/subtract.o
$ ar rcs out/libcalc.a out/add.o out/multiply.o out/subtract.o
$ gcc -c -Icalc exec/main.c -o out/main.o
$ gcc -Lout out/main.o -lcalc -o out/ex23_1.out
$
```

Мы получили артефакты: статическую библиотеку `libcalc.a` и исполняемый файл `ex23_1.out`. Если вы не знаете, как скомпилировать проект на языке C, или представленные выше команды выглядят незнакомыми, то, пожалуйста, прочитайте главы 2 и 3.

Первая команда в терминале 23.2 создает каталог out. В него должны попасть все переносимые объектные файлы и конечные продукты компиляции.

Следующие три команды используют `gcc` для компиляции исходных файлов в каталоге calc и создания соответствующих переносимых объектных файлов. Затем эти файлы применяются в пятой команде для получения статической библиотеки `libcalc.a`.

Наконец, последние две команды компилируют файл `main.c` из каталога exec и компонуют его с библиотекой `libcalc.a`, чтобы сгенерировать итоговый исполняемый файл, `ex23_1.out`. Напомню, что все эти файлы помещаются в каталог out.

Количество команд может расти с увеличением количества исходных файлов. Мы можем хранить приведенные выше команды в так называемом *скрипте сборки*, но сначала необходимо ответить на ряд вопросов.

- Будем ли мы выполнять одни и те же команды на всех платформах? Некоторые детали зависят от компилятора и системной среды, поэтому команды могут варьироваться. В простейшем случае следует предусмотреть разные скрипты командной оболочки для разных платформ. Но это фактически означает, что наш скрипт не является *переносимым*.
- Что произойдет, если в проект будет добавлен новый каталог или модуль? Придется ли менять скрипт сборки?
- Как на скрипт сборки повлияет добавление новых исходных файлов?
- Что, если нам понадобится новый продукт компиляции — библиотека или исполняемый файл?

Хорошая система сборки должна уметь справляться с большинством перечисленных ситуаций. Рассмотрим наш первый Makefile. Он будет собирать представленный выше проект и генерировать продукты его компиляции. Все файлы систем сборки, представленные в данном и следующих разделах, позволяют собирать этот конкретный проект и ничего больше.

В листинге 23.1 показано содержимое простейшего файла Makefile, который только можно написать для нашего проекта.

Листинг 23.1. Очень простой файл Makefile, написанный для нашего проекта (Makefile-very-simple)

```
build:
    mkdir -p out
    gcc -c calc/add.c -o out/add.o
    gcc -c calc/multiply.c -o out/multiply.o
    gcc -c calc/subtract.c -o out/subtract.o
    ar rcs out/libcalc.a out/add.o out/multiply.o out/subtract.o
    gcc -c -Icalc exec/main.c -o out/main.o
    gcc -Lout -lcalc out/main.o -o out/ex23_1.out
clean:
    rm -rfv out
```

Этот Makefile содержит две цели: `build` и `clean`. Каждая из них имеет набор команд, которые должны быть выполнены при ее вызове. Этот набор называют *рецептом* цели.

Чтобы выполнить команды, указанные в Makefile, необходимо использовать утилиту `make`. Вы должны сообщить ей, какую цель нужно выполнить; если этого не сделать, то по умолчанию всегда выполняется первая.

Для сборки нашего проекта достаточно скопировать содержимое листинга 23.1 в файл Makefile и поместить его в корневой каталог. Каталог проекта должен быть похож на вывод командной строки, показанный в терминале 23.3.

Терминал 23.3. Файлы и каталоги нашего проекта после добавления Makefile

```
$ tree ex23_1
ex23_1/
├── Makefile
├── calc
│   ├── add.c
│   ├── calc.h
│   ├── multiply.c
│   └── subtract.c
└── exec
    └── main.c

2 directories, 6 files
$
```

После этого можно просто выполнить команду `make`. Утилита `make` автоматически ищет файл Makefile в корневом каталоге и выполняет его первую цель. Если нам нужно выполнить цель `clean`, то мы должны указать команду `make clean`. Цель `clean` позволяет удалить файлы, сгенерированные в процессе сборки, чтобы мы могли начать с чистого листа.

В терминале 23.4 показан результат выполнения `make`.

Терминал 23.4. Сборка проекта с использованием очень простого файла Makefile

```
$ cd ex23_1
$ make
mkdir -p out
gcc -c -Icalc exec/main.c -o out/main.o
gcc -c calc/add.c -o out/add.o
gcc -c calc/multiply.c -o out/multiply.o
gcc -c calc/subtract.c -o out/subtract.o
ar rcs out/libcalc.a out/add.o out/multiply.o out/subtract.o
gcc -Lout -lcalc out/main.o -o out/ex23_1.out
$
```

Вы можете спросить: чем скрипт сборки (представляющий собой shell-скрипт) отличается от приведенного выше Makefile? Это хороший вопрос! Наш файл Makefile не похож на то, как мы обычно собираем наши проекты с помощью Make.

На самом деле это упрощенный пример применения Make, в котором не используются многие возможности, предлагаемые данной системой сборки.

Иными словами, пока наш Makefile был на удивление похож на скрипт командной оболочки, хотя последний потребовал бы больше усилий с нашей стороны. Мы подошли к тому моменту, когда файлы Makefile становятся интересными и по-настоящему уникальными.

Следующий пример Makefile тоже довольно прост, однако в нем уже можно увидеть несколько аспектов системы сборки Make, которые нас интересуют (листинг 23.2).

Листинг 23.2. Новый, но по-прежнему простой файл Makefile для нашего проекта (Makefile-simple)

```
CC = gcc

build: prereq out/main.o out/libcalc.a
    ${CC} -Lout -lcalc out/main.o -o out/ex23_1.out

prereq:
    mkdir -p out

out/libcalc.a: out/add.o out/multiply.o out/subtract.o
    ar rcs out/libcalc.a out/add.o out/multiply.o out/subtract.o

out/main.o: exec/main.c calc/calc.h
    ${CC} -c -Icalc exec/main.c -o out/main.o

out/add.o: calc/add.c calc/calc.h
    ${CC} -c calc/add.c -o out/add.o

out/subtract.o: calc/subtract.c calc/calc.h
    ${CC} -c calc/subtract.c -o out/subtract.o

out/multiply.o: calc/multiply.c calc/calc.h
    ${CC} -c calc/multiply.c -o out/multiply.o

clean: out
    rm -rf out
```

Мы объявили внутри Makefile переменную и использовали ее в разных местах по аналогии с `CC`. Переменные в сочетании с условиями позволяют писать гибкие инструкции сборки, прилагая меньше усилий по сравнению с написанием аналогичных shell-сценариев.

Еще одна замечательная особенность системы Make — возможность подключать другие файлы Makefile. Это позволяет задействовать существующие файлы, написанные для других проектов.

Как видно в приведенном выше коде, каждый файл Makefile может иметь несколько целей. Цель начинается с новой строки и содержит двоеточие в конце. Все инструкции цели (рецепта) *должны* иметь отступы в виде символов табуляции, чтобы программа `make` могла их распознать. Цели обладают одним интересным свойством: могут зависеть от других целей.

Например, в приведенном выше Makefile цель `build` зависит от целей `prereq`, `out/main.o` и `out/libcalc.a`. Таким образом, при вызове цели `build` система Make сначала проверяет цели, от которых та зависит, и вызывает их, если они еще не были выполнены. Внимательно присмотревшись к целям в этом Makefile, можно увидеть их поток выполнения.

Это то, чего явно не хватает в скриптах командной оболочки; чтобы заставить их работать по такому же принципу, потребуется много механизмов управления потоком выполнения (циклы, условные выражения и т. д.). Файлы Makefile более компактные и декларативные — вот почему мы используем их. Мы хотим объявить только то, что должно быть собрано, и нам не нужно знать, по какому пути пойдет процесс сборки. Нельзя сказать, что Make полностью удовлетворяет данному требованию, но с этого начинается любая полноценная система сборки.

Еще одно свойство целей в Makefile состоит в том, что они могут ссылаться на файлы и каталоги, размещенные на диске, такие как `out/multiply.o`, и если программа `make` не обнаружит в них свежих изменений с момента последней сборки, то соответствующая цель будет пропущена. То же самое касается зависимости `out/multiply.o`, которая относится к файлу `calc/multiply.c`. Если файл `calc/multiply.c` в последнее время не менялся и если его уже скомпилировали, то повторная его компиляция будет бессмысленной. Это еще одна возможность, которую сложно реализовать при написании скриптов командной оболочки.

Благодаря этому вы можете компилировать только те исходные файлы, которые изменились с момента последней сборки, что позволяет избежать компиляции огромного количества кода. Конечно, чтобы данная возможность заработала, проект нужно полностью скомпилировать хотя бы раз. Но после этого компиляции и компоновке будут подлежать лишь измененные исходники.

Еще один важный аспект предыдущего файла Makefile — цель `calc/calc.h`. От файла `calc/calc.h` зависит несколько целей — в основном исходных файлов. Поэтому, учитывая возможности, описанные выше, простое изменение заголовочного файла может инициировать компиляцию нескольких исходных файлов, которые от него зависят.

Вот почему мы пытаемся подключать в исходных файлах только необходимые заголовки и по возможности использовать предварительное объявление вместо подключения. Предварительные объявления обычно не применяются в исходных файлах, поскольку зачастую требуют доступа к определению структур или функций, но в случае с заголовками такой проблемы нет.

Наличие множества зависимостей между заголовочными файлами обычно имеет катастрофические последствия. Даже небольшое изменение заголовка, подключенного к большому количеству других заголовков, которые, в свою очередь,

подключаются ко многим исходным файлам, может привести к сборке всего проекта или кода аналогичного масштаба. Это фактически ухудшает качество разработки и затягивает ожидание между сборками до нескольких минут.

Предыдущий файл Makefile получился слишком многословным. Мы хотим, чтобы цели менялись при добавлении каждого нового исходного файла. Файл Makefile будет при этом меняться, но это не будет выражаться в добавлении новых целей или изменении его общей структуры, поскольку это фактически исключило бы повторное использование данного Makefile в аналогичных проектах.

Кроме того, многие цели названы по одному и тому же принципу, поэтому поддержка *регулярных выражений* в Make могла бы существенно сократить количество целей и сделать код в Makefile более компактным. Это еще одна крайне полезная возможность Make, которую сложно повторить при написании скриптов командной оболочки.

Следующий пример Makefile будет последним в данном проекте, хотя это по-прежнему не самый оптимальный вариант по сравнению с тем, что мог бы написать опытный пользователь Make (листинг 23.3).

Листинг 23.3. Новый пример Makefile, написанный для нашего проекта и использующий регулярные выражения (Makefile-by-pattern)

```
BUILD_DIR = out
OBJ = ${BUILD_DIR}/calc/add.o \
                ${BUILD_DIR}/calc/subtract.o \
                ${BUILD_DIR}/calc/multiply.o \
                ${BUILD_DIR}/exec/main.o
CC = gcc
HEADER_DIRS = -Icalc
LIBCALCNAME = calc
LIBCALC = ${BUILD_DIR}/lib${LIBCALCNAME}.a
EXEC = ${BUILD_DIR}/ex23_1.out

build: prereq ${BUILD_DIR}/exec/main.o ${LIBCALC}
    ${CC} -L${BUILD_DIR} -l${LIBCALCNAME} ${BUILD_DIR}/exec/main.o -o ${EXEC}

prereq:
    mkdir -p ${BUILD_DIR}
    mkdir -p ${BUILD_DIR}/calc
    mkdir -p ${BUILD_DIR}/exec

${LIBCALC}: ${OBJ}
    ar rcs ${LIBCALC} ${OBJ}

${BUILD_DIR}/calc/%.o: calc/%.c
    ${CC} -c ${HEADER_DIRS} $< -o $@
```

```
${BUILD_DIR}/exec/%.o: exec/%.c
    ${CC} -c ${HEADER_DIRS} $< -o $@

clean: ${BUILD_DIR}
    rm -rf ${BUILD_DIR}
```

Этот файл Makefile использует регулярные выражения в своих целях. Переменная `OBJ` хранит список ожидаемых переносимых объектных файлов и применяется везде, где такой список может понадобиться.

Описание того, как работают регулярные выражения, выходит за рамки данной книги, но вы можете видеть, что в названиях целей используется много символов-заполнителей, таких как `%`, `$<` и `$@`.

Выполнение этого файла Makefile дает тот же результат, что и в предыдущих примерах, но теперь мы можем задействовать разные удобные возможности системы Make, и в конечном счете у нас получился Make-сценарий, который легко сопровождать и использовать повторно.

В терминале 23.5 показано, как запустить этот файл Makefile и каким будет его вывод.

Терминал 23.5. Сборка проекта с использованием заключительной версии Makefile

```
$ make
mkdir -p out
mkdir -p out/calc
mkdir -p out/exec
gcc -c -Icalc exec/main.c -o out/exec/main.o
gcc -c -Icalc calc/add.c -o out/calc/add.o
gcc -c -Icalc calc/subtract.c -o out/calc/subtract.o
gcc -c -Icalc calc/multiply.c -o out/calc/multiply.o
ar rcs out/libcalc.a out/calc/add.o out/calc/subtract.o out/calc/
multiply.o out/exec/main.o
gcc -Lout -lcalc out/exec/main.o -o out/ex23_1.out
$
```

В следующих разделах мы поговорим о CMake — отличном инструменте для генерации настоящих файлов Makefile. На самом деле, спустя некоторое время после того, как система Make приобрела популярность, появились новые инструменты, *генераторы скриптов сборки*, способные генерировать файлы Makefile и скрипты для других систем по заданному описанию. CMake — один из них, и это, наверное, самый популярный проект такого рода.

Вот главная ссылка на описание GNU Make — реализацию Make от проекта GNU: https://www.gnu.org/software/make/manual/html_node/index.html.

CMake — не система сборки!

CMake — генератор скриптов сборки для других систем, таких как Make и Ninja. Написание эффективных и кросс-платформенных файлов Makefile — утомительный процесс. CMake и аналогичные инструменты вроде *Autotools* созданы для того, чтобы вы могли получить хорошо оптимизированные, кросс-платформенные скрипты сборки, такие как Make или Ninja. Отмечу, что Ninja — еще одна система сборки; мы познакомимся с ней в следующем разделе.

Больше об Autotools можно узнать на странице https://www.gnu.org/software/automake/manual/html_node/Autotools-Introduction.html.

Еще один важный аспект — управление зависимостями, которого нет в файлах Makefile. Эти генераторы умеют проверять установленные зависимости и не генерировать скрипты сборки, если какая-то из них отсутствует в системе. Проверка компиляторов и их версий, определение их местоположения и возможностей — все это выполняется перед генерацией сценария сборки.

Если системе Make нужен файл Makefile, то CMake ищет `CMakeLists.txt`. Наличие этого файла в проекте означает, что для генерации Makefile используется CMake. К счастью, CMake, в отличие от Make, поддерживает вложенные модули. То есть у вас может быть несколько файлов `CMakeList.txt`, размещенных в разных каталогах проекта, и все они будут найдены; если запустить утилиту CMake в корне проекта, она сгенерирует для каждого из них подходящий файл Makefile.

В продолжение данного раздела добавим в наш демонстрационный проект поддержку CMake. Создадим для этого три файла `CMakeLists.txt`. В терминале 23.6 вы можете видеть иерархию проекта после их добавления.

Терминал 23.6. Иерархия проекта после добавления трех файлов CMakeLists.txt

```
$ tree ex23_1
ex23_1/
├── CMakeLists.txt
├── calc
│   ├── CMakeLists.txt
│   ├── add.c
│   ├── calc.h
│   ├── multiply.c
│   └── subtract.c
└── exec
    ├── CMakeLists.txt
    └── main.c

2 directories, 8 files
$
```

Итак, у нас теперь три файла `CMakeLists.txt`: один в корневом каталоге, другой в каталоге calc и третий — внутри exec. Содержимое первого из них показано в листинге 23.4. Как видите, он добавляет подкаталоги calc и exec.

Каждый из этих каталогов должен содержать файл `CMakeLists.txt`, и наша конфигурация соответствует этому требованию.

Листинг 23.4. Файл CMakeLists.txt, размещенный в корневом каталоге проекта (CMakeLists.txt)

```
cmake_minimum_required(VERSION 3.8)

include_directories(calc)

add_subdirectory(calc)
add_subdirectory(exec)
```

Этот файл CMake добавляет calc в число подключенных каталогов, которые будут использоваться компилятором C во время сборки исходных файлов. Как уже говорилось ранее, он также добавляет два подкаталога, calc и exec, каждый из которых содержит собственный файл `CMakeLists.txt`, описывающий процедуру компиляции ее содержимого. В листинге 23.5 представлен файл `CMakeLists.txt`, который находится в каталоге calc.

Листинг 23.5. Файл CMakeLists.txt, размещенный в каталоге calc (calc/CMakeLists.txt)

```
add_library(calc STATIC
  add.c
  subtract.c
  multiply.c
)
```

Как видите, это всего лишь простое *объявление цели* `calc`. Оно означает, что у нас должна быть статическая библиотека `calc` (на самом деле после сборки она будет называться `libcalc.a`) с переносимыми объектными файлами для исходных файлов `add.c`, `subtract.c` и `multiply.c`. Обратите внимание: в CMake цели обычно представляют конечные продукты компиляции кодовой базы. Следовательно, в случае с модулем `calc` у нас получится ровно один продукт — статическая библиотека.

Для цели `calc` больше ничего не указано. В частности, мы не указали ни расширение, ни имя файла статической библиотеки (хотя могли это сделать). Все остальные конфигурации, необходимые для сборки данного модуля, либо унаследованы от родительского файла `CMakeLists.txt`, либо основаны на стандартной конфигурации самой системы CMake.

Например, мы знаем, что Linux и macOS имеют разные расширения разделяемых объектных файлов. Поэтому, если целью выступает разделяемая библиотека, то в ее объявлении не нужно указывать расширение. CMake может самостоятельно

позаботиться об этом платформозависимом аспекте, и расширение итогового разделяемого объектного файла будет соответствовать той платформе, на которой он собирается.

Следующий файл `CMakeLists.txt` можно найти в каталоге exec (листинг 23.6).

Листинг 23.6. Файл CMakeLists.txt в каталоге exec (exec/CMakeLists.txt)

```
add_executable(ex23_1.out
  main.c
)

target_link_libraries(ex23_1.out
  calc
)
```

Цель, объявленная в этом листинге, представляет собой исполняемый файл, который должен быть скомпонован с целью `calc`, уже объявленной в другом файле `CMakeLists.txt`.

Это позволит создать библиотеку в одной части проекта и затем использовать ее в другом, достаточно написать несколько директив.

Теперь пришло время показать, как из файла `CMakeLists.txt`, размещенного в корневом каталоге, можно сгенерировать Makefile. Обратите внимание: мы это делаем в отдельном каталоге build, чтобы отделить переносимые и итоговые объектные файлы от самих исходников.

Если вы используете *систему управления исходным кодом* (source control management, SCM), такую как *git*, то можете игнорировать каталог build, поскольку он должен генерироваться отдельно на каждой платформе. Нас интересуют только файлы `CMakeLists.txt`, которые всегда хранятся в репозитории исходного кода.

В терминале 23.7 показано, как сгенерировать скрипты сборки (в данном случае Makefile) из файла `CMakeLists.txt`, размещенного в корневом каталоге.

Терминал 23.7. Генерация Makefile из файла CMakeLists.txt, размещенного в корневом каталоге

```
$ cd ex23_1
$ mkdir -p build
$ cd build
$ rm -rfv *
...
$ cmake ..
-- The C compiler identification is GNU 7.4.0
-- The CXX compiler identification is GNU 7.4.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
```

```
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: .../extreme_c/ch23/ex23_1/build
$
```

Судя по этому выводу, команда CMake сумела обнаружить рабочие компиляторы, информацию об их ABI (подробности об ABI ищите в главе 3), их возможности и т. д. В результате в каталоге build был сгенерирован файл Makefile.

В терминале 23.7 предполагается, что каталог build мог существовать ранее, поэтому все его содержимое предварительно удаляется.

В терминале 23.8 показано содержимое каталога build, включая сгенерированный файл Makefile.

Терминал 23.8. Сгенерированный файл Makefile в каталоге build

```
$ ls
CMakeCache.txt  CMakeFiles  Makefile  calc  cmake_install.cmake
exec
$
```

Итак, у вас в каталоге build есть Makefile. Теперь можно воспользоваться командой `make`. Она займется компиляцией и наглядно проинформирует вас о ходе ее выполнения (терминал 23.9).

Отмечу, что перед выполнением команды `make` нужно сначала перейти в каталог build.

Терминал 23.9. Выполнение сгенерированного файла Makefile

```
$ make
Scanning dependencies of target calc
[ 16%] Building C object calc/CMakeFiles/calc.dir/add.c.o
[ 33%] Building C object calc/CMakeFiles/calc.dir/subtract.c.o
[ 50%] Building C object calc/CMakeFiles/calc.dir/multiply.c.o
[ 66%] Linking C static library libcalc.a
[ 66%] Built target calc
```

```
Scanning dependencies of target ex23_1.out
[ 83%] Building C object exec/CMakeFiles/ex23_1.out.dir/main.c.o
[100%] Linking C executable ex23_1.out
[100%] Built target ex23_1.out
$
```

В настоящее время CMake используется во многих крупных проектах; вы можете собирать их исходники с помощью примерно тех же команд, которые были показаны в предыдущих терминалах. Один из таких проектов — *Vim*. Даже сам проект CMake собирается с помощью CMake (но только после того, как Autotools соберет минимальную систему)! У CMake есть множество версий и возможностей, для подробного обсуждения понадобилась бы целая книга.

По следующей ссылке находится официальная документация для последней версии CMake, которая позволит вам получить общее представление о принципе работы этой системы и ее возможностях: https://cmake.org/cmake/help/latest/index.html.

В завершение следует отметить, что CMake может создавать сценарии сборки для Microsoft Visual Studio, Xcode от Apple и других сред разработки.

В следующем разделе мы поговорим о быстрой альтернативе Make, системе сборки Ninja, которая в последнее время набирает популярность. Мы также рассмотрим, как с помощью CMake можно генерировать не только Makefile, но и сценарии сборки Ninja.

Ninja

Ninja — альтернатива системе Make. Я бы не назвал ее заменой. Это, скорее, более быстрая альтернатива. Высокая производительность достигается за счет отказа от некоторых возможностей, предлагаемых Make, таких как операции со строками, циклы и регулярные выражения.

Благодаря отсутствию этих функций Ninja имеет меньше накладных расходов. Но если вы хотите писать скрипты сборки с нуля, то данную систему лучше не использовать.

Написание скриптов Ninja можно сравнить с написанием скриптов командной оболочки, недостатки которых мы рассмотрели в предыдущем разделе. Вот почему Ninja рекомендуется использовать в сочетании со средствами генерации скриптов сборки, такими как CMake.

В данном разделе будет показано, как использовать скрипты сборки Ninja, сгенерированные системой CMake. Таким образом, мы не станем углубляться в синтаксис

Ninja, как делали в случае с файлами Makefile. Дело в том, что мы не будем писать эти сценарии сами; CMake сгенерирует их для нас.

Больше информации о синтаксисе Ninja находится по ссылке https://ninja-build.org/manual.html#_writing_your_own_ninja_files.

Как уже упоминалось ранее, для создания скриптов сборки Ninja лучше всего использовать генераторы. В терминале 23.10 вы можете видеть, как с помощью CMake вместо Makefile можно сгенерировать скрипт сборки Ninja, `build.ninja`.

Терминал 23.10. Генерация build.ninja из файла CMakeLists.txt, размещенного в корневом каталоге

```
$ cd ex23_1
$ mkdir -p build
$ cd build
$ rm -rfv *
...
$ cmake -GNinja ..
-- The C compiler identification is GNU 7.4.0
-- The CXX compiler identification is GNU 7.4.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: .../extreme_c/ch23/ex23_1/build
$
```

Как видите, мы передали параметр `-GNinja`, благодаря чему CMake знает, что вместо файлов Makefile нам нужны скрипты сборки Ninja. CMake генерирует файл `build.ninja`, который можно найти в каталоге build (терминал 23.11).

Терминал 23.11. Генерация build.ninja в каталоге build

```
$ ls
CMakeCache.txt  CMakeFiles  build.ninja  calc  cmake_install.cmake
exec r ules.ninja
$
```

Для компиляции этого проекта достаточно выполнить команду `ninja`, как показано в терминале 23.12. Утилита ninja ищет файл build.ninja в текущем каталоге точно так же, как в случае с make и Makefile.

Терминал 23.12. Выполнение сгенерированного скрипта build.ninja

```
$ ninja
[6/6] Linking C executable exec/ex23_1.out
$
```

В следующем разделе речь пойдет о *Bazel* — еще одной системе, которую можно использовать для сборки проектов на C и C++.

Bazel

Bazel — система сборки, которая была разработана компанией Google. Она изначально задумывалась быстрой и масштабируемой, способной собрать любой проект, независимо от языка программирования. Bazel поддерживает проекты на C, C++, Java, Go и Objective-C. Более того, с ее помощью можно собирать приложения для Android и iOS.

Проект Bazel стал открытым примерно в 2015 году. Это система сборки, и потому ее можно сравнивать с Make и Ninja, но не с CMake. Почти все проекты Google с открытым исходным кодом собираются с помощью Bazel, включая *gRPC*, *Angular*, *Kubernetes*, *TensorFlow* и сам *Bazel*.

Система Bazel написана на Java. Она славится своими параллельными и масштабируемыми сборками, что имеет большое значение в крупных проектах. Параллельная сборка также доступна в Make и Ninja; в обоих случаях для этого предусмотрен параметр `-j` (хотя Ninja работает параллельно по умолчанию).

Официальная документация Bazel находится по адресу https://docs.bazel.build/versions/master/bazel-overview.html.

Порядок использования Bazel напоминает то, как мы работали с Make и Ninja. Bazel требует наличия в проекте двух видов файлов: `WORKSPACE` и `BUILD`. Первый должен находиться в корневом каталоге, а файлы `BUILD` следует размещать внутри модулей, которые должны собираться в рамках одного рабочего пространства (или проекта). Это более или менее похоже на то, как работает система CMake, в которой используется три файла `CMakeLists.txt`, разбросанных по проекту. Однако следует помнить, что Bazel — самостоятельная система сборки, и мы не будем генерировать с ее помощью скрипты для других систем.

Чтобы добавить в проект поддержку Bazel, нужно иметь иерархию файлов, показанную в терминале 23.13.

Терминал 23.13. Иерархия проекта после добавления файлов Bazel

```
$ tree ex23_1
ex23_1/
├── WORKSPACE
├── calc
│   ├── BUILD
│   ├── add.c
│   ├── calc.h
│   ├── multiply.c
│   └── subtract.c
└── exec
    ├── BUILD
    └── main.c

2 directories, 8 files
$
```

В нашем примере файл `WORKSPACE` будет пустым. Он обычно используется для обозначения корня кодовой базы. Если у вас еще более глубокая и вложенная иерархия модулей, то о размещении файлов `WORKSPACE` и `BUILD` можно почитать в документации.

В файле `BUILD` описываются цели сборки в текущем каталоге (или модуле). В листинге 23.7 показан файл `BUILD` для модуля `calc`.

Листинг 23.7. Файл BUILD в каталоге calc (calc/BUILD)

```
c_library(
  name = "calc",
  srcs = ["add.c", "subtract.c", "multiply.c"],
  hdrs = ["calc.h"],
  linkstatic = True,
  visibility = ["//exec:__pkg__"]
)
```

Здесь объявлена новая цель, `calc`. Это статическая библиотека, состоящая из трех исходных файлов, которые находятся в текущем каталоге. Она также доступна для других целей, размещенных в каталоге exec.

Взглянем на файл `BUILD` в каталоге exec (листинг 23.8).

Листинг 23.8. Файл BUILD в каталоге exec (exec/BUILD)

```
cc_binary(
  name = "ex23_1.out",
  srcs = ["main.c"],
  deps = [
    "//calc:calc"
  ],
  copts = ["-Icalc"]
)
```

Подготовив перечисленные выше файлы, мы можем запустить Bazel и собрать наш проект (терминал 23.14). Вам нужно перейти в корневой каталог проекта. Отмечу, что здесь, в отличие от примера с CMake, не нужен каталог build.

Терминал 23.14. Сборка проекта с использованием Bazel

```
$ cd ex23_1
$ bazel build //...
INFO: Analyzed 2 targets (14 packages loaded, 71 targets
configured).
INFO: Found 2 targets...
INFO: Elapsed time: 1.067s, Critical Path: 0.15s
INFO: 6 processes: 6 linux-sandbox.
INFO: Build completed successfully, 11 total actions
$
```

Теперь, заглянув в каталог bazel-bin, который находится в корне проекта, можно увидеть следующие продукты компиляции (терминал 23.15).

Терминал 23.15. Содержимое каталога bazel-bin после выполнения сборки

```
$ tree bazel-bin
bazel-bin
├── calc
│   ├── _objs
│   │   └── calc
│   │       ├── add.pic.d
│   │       ├── add.pic.o
│   │       ├── multiply.pic.d
│   │       ├── multiply.pic.o
│   │       ├── subtract.pic.d
│   │       └── subtract.pic.o
│   ├── libcalc.a
│   └── libcalc.a-2.params
└── exec
    ├── _objs
    │   └── ex23_1.out
    │       ├── main.pic.d
    │       └── main.pic.o
    ├── ex23_1.out
    ├── ex23_1.out-2.params
    ├── ex23_1.out.runfiles
    │   ├── MANIFEST
    │   └── __main__
    │       └── exec
    │           └── ex23_1.out -> .../bin/exec/ex23_1.out
    └── ex23_1.out.runfiles_manifest

9 directories, 15 files
$
```

В представленном списке мы видим продукты компиляции после успешной сборки проекта.

Следующий раздел станет заключительным этапом нашего обсуждения в данной главе. Мы сравним разные системы сборки, которые доступны для проектов на C и C++.

Сравнение систем сборки

В этой главе вы познакомились с тремя известными и широко используемыми системами сборки. Мы также рассмотрели генератор скриптов сборки CMake. Вы должны знать, что проекты на C и C++ можно также собирать с применением других систем.

Следует понимать, что выбор системы сборки — долгосрочное решение; если вы начнете использовать в своем проекте какую-то систему, то поменять ее на другую будет непросто.

Системы сборки можно сравнивать по их различным свойствам. Управление зависимостями, способность справляться со сложными иерархиями вложенных проектов, скорость сборки, масштабируемость, интеграция с существующими сервисами, возможность добавлять новую логику и т. д. — все это можно учитывать для объективного сравнения. Я не стану подробно сравнивать системы сборки в нашей книге, поскольку это утомительно; к тому же на данную тему уже написаны замечательные статьи, которые можно найти в Интернете.

Отмечу, что результаты сравнения зависят от того, для кого проводятся. Ваш выбор системы сборки должен быть основан на требованиях вашего проекта и доступных вам ресурсах. По ссылкам, представленным ниже, можно найти дополнительный материал по этой теме:

- https://www.reddit.com/r/cpp/comments/8zm66h/an_overview_of_build_systems_mostly_for_c_projects/;
- https://github.com/LoopPerfect/buckaroo/wiki/Build-Systems-Comparison;
- https://medium.com/@julienjorge/an-overview-of-build-systems-mostly-for-c-projects-ac9931494444.

Резюме

В этой главе мы обсудили распространенные инструменты для сборки проектов на C и C++. В рамках обсуждения мы:

- выяснили, зачем нужна система сборки;
- познакомились с Make, одной из старейших систем сборки, доступных для C и C++;

- познакомились с Autotools и CMake, двумя известными генераторами скриптов сборки;
- узнали, как сгенерировать нужные файлы Makefile, задействуя CMake;
- поговорили о Ninja и выяснили, как с помощью CMake сгенерировать скрипты сборки для этой системы;
- узнали, как Bazel позволяет собирать проекты на C;
- и наконец, ознакомились с рядом ссылок на онлайн-материалы со сравнениями разных систем сборки.

Послесловие

И напоследок...

Если вы дочитали до этого места, то наше путешествие подошло к концу! В книге мы рассмотрели разные темы и концепции, и я надеюсь, что это помогло вам стать лучшим программистом на языке C. Конечно, читая книгу, не получишь опыта; для этого вы должны поработать над различными проектами. Представленные здесь методы и советы повысят ваш уровень квалификации, что позволит вам участвовать в более серьезных проектах. Теперь вы лучше представляете программные системы в целом и имеете передовые знания об их внутренних механизмах.

Книга получилась тяжелее и толще, чем обычное издание, но и этого недостаточно, чтобы охватить все темы, относящиеся к C, C++ и системному программированию. Поэтому наше путешествие еще не закончилось! Я хочу продолжить работу над другими «экстремальными» темами — возможно, в более специфических областях, таких как асинхронный ввод/вывод, сложные структуры данных, программирование сокетов, распределенные системы, разработка ядра и функциональное программирование. Но всему свое время.

Надеюсь, увидимся в следующем путешествии!

Камран

Камран Амини

Экстремальный Си. Параллелизм, ООП и продвинутые возможности

Перевел с английского А. Павлов

Заведующая редакцией	*Ю. Сергиенко*
Ведущий редактор	*Н. Гринчик*
Литературный редактор	*Н. Хлебина*
Художественный редактор	*В. Мостипан*
Корректоры	*О. Андриевич, Е. Павлович*
Верстка	*Г. Блинов*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 07.2021. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —
Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 22.06.21. Формат 70×100/16. Бумага офсетная. Усл. п. л. 60,630. Тираж 500. Заказ 0000.