O'REILLY®

alist

Rust: атомарности и блокировки

Низкоуровневая конкурентность на практике

Мара Бос

Rust Atomics and Locks

Low-Level Concurrency in Practice

Mara Bos

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY

Мара Бос

Rust: атомарности и блокировки

Астана
«АЛИСТ»
2024

УДК 004.4
ББК 32.973.26-018.2
Б85

Бос М.

Б85 Rust: атомарности и блокировки: Пер. с англ. — Астана: АЛИСТ, 2024. — 240 с.: ил.

ISBN 978-601-09-5050-4

Книга позволяет программистам получить четкое представление о низкоуровневой конкурентности и ее реализации. Даны основы конкурентности в Rust. Раскрыты понятия об атомарности и упорядочении памяти. Рассмотрены практические аспекты создания своих собственных каналов, своего собственного типа Arc и своих собственных блокировок. Дано представление о внутренней "кухне" процессора. Рассказано о примитивах операционной системы. Предложены идеи для самостоятельной разработки решений, связанных с вычислениями в конкурентном режиме.

Для Rust-программистов

УДК 004.4
ББК 32.973.26-018.2

© 2024 ALIST LLP
Authorized Russian translation of the English edition of ***Rust Atomics and Locks***,
ISBN 9781098119447 © 2023 Mara Bos.
This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

Авторизованный перевод с английского языка на русский издания ***Rust Atomics and Locks***,
ISBN 9781098119447 © 2023 Mara Bos.
Перевод опубликован и продается с разрешения компании-правообладателя O'Reilly Media, Inc.

ISBN 978-1-098-11944-7 (англ.)
ISBN 978-601-09-5050-4 (каз.)

© Mara Bos, 2023
© Издание на русском языке. ТОО "АЛИСТ", 2024

Оглавление

Предисловие **11**

Введение **13**
Для кого предназначена эта книга 13
Обзор глав 14
Примеры кода 16
Условные обозначения, используемые в данной книге 16
Благодарности 17

Глава 1. Основы конкурентности в Rust **19**
Потоки в Rust 19
Потоки с областью действия 23
Совместное владение и подсчет ссылок 25
  Статика 25
  Утечка 25
  Подсчет ссылок 26
Заимствования и гонка данных 28
Внутренняя изменяемость 30
  Cell 31
  RefCell 32
  Mutex и *RwLock* 33
  Атомарные типы 33
  UnsafeCell 34
Потокобезопасность: *Send* и *Sync* 34
Блокировка: мьютексы и *RwLock*-блокировки 36
  Мьютекс в языке Rust 37
  Отравление блокировок 39
  Блокировка чтения-записи 41
Ожидание: парковка и условные переменные 42
  Парковка потоков 42
  Условные переменные 45
Резюме 47

Глава 2. Атомарность **48**
Атомарные операции загрузки и сохранения: *load* и *store* 49
  Пример: Флаг остановки 49
  Пример: Отчет о ходе выполнения задачи 50
    Синхронизация 51
  Пример: Отложенная инициализация 52

Операции выборки и изменения 54
Пример: Отчет о ходе выполнения задачи из нескольких потоков 55
Пример: Статистика 56
Пример: Предоставление идентификатора 58
Операции сравнения и обмена 60
Пример: Предоставление идентификатора без переполнения 62
Пример: Отложенная однократная инициализация 63
Резюме 64

Глава 3. Упорядочение памяти 66
Изменение порядка и оптимизация 66
Модель памяти 68
Отношения "происходит до" 68
Порождение и присоединение 70
Расслабленное упорядочение 71
Упорядочение высвобождения и получения памяти 73
Пример: Блокировка 76
Пример: Отложенная инициализация с применением косвенного подхода 78
Упорядочение потребления 81
Последовательно согласованное упорядочение 82
Ограждения 83
Распространенные заблуждения 87
Резюме 89

Глава 4. Создание своей собственной спин-блокировки 91
Минималистичная реализация 91
Небезопасная спин-блокировка 93
Безопасный интерфейс с использованием хранителя блокировки 96
Резюме 99

Глава 5. Создание своих собственных каналов 100
Простой канал на основе мьютекса 100
Небезопасный одноразовый канал 102
Безопасность, достигаемая проверками в ходе выполнения 105
Обеспечение безопасности с использованием системы типов 109
Заимствование во избежание выделения памяти 114
Блокировка 117
Резюме 119

Глава 6. Создание своего собственного типа *Arc* 121
Базовый подсчет ссылок 121
Тестирование 125
Изменение 126
Слабые указатели 127
Тестирование 132
Оптимизация 133
Резюме 140

Глава 7. Представление о внутренней "кухне" процессора 141
Инструкции процессора 142
Загрузка и сохранение 145

Операции чтения-изменения-записи ... 147
Префикс *lock* в *x*86 ... 147
*x*86-инструкция сравнения-обмена ... 149
Инструкции *load-linked* и *store-conditional* ... 150
ARM-инструкции *load-exclusive* и *store-exclusive* ... 151
ARM-инструкции сравнения-обмена ... 152
Кеширование ... 154
Согласованность кеша ... 155
Протокол сквозной записи ... 156
Протокол MESI ... 156
Влияние, оказываемое на производительность ... 157
Переупорядочение ... 162
Упорядочение памяти ... 164
*x*86-64: строго упорядоченная архитектура ... 165
ARM64: слабо упорядоченная архитектура ... 167
Эксперимент ... 169
Ограждения памяти ... 171
Резюме ... 172

Глава 8. Примитивы операционной системы ... 175
Взаимодействие с ядром ... 175
POSIX ... 176
Обертывание в Rust ... 178
Linux ... 180
Фьютекс ... 180
Фьютексные операции ... 183
Фьютексные операции с наследованием приоритета ... 187
macOS ... 188
os_unfair_lock ... 189
Windows ... 189
Тяжеловесные объекты ядра ... 189
Облегченные объекты ... 190
Легкая блокировка чтения-записи ... 190
Адресное ожидание ... 191
Резюме ... 192

Глава 9. Создание своих собственных блокировок ... 194
Мьютекс ... 196
Предотвращение системных вызовов ... 199
Дальнейшая оптимизация ... 201
Сравнительный анализ ... 203
Условная переменная ... 205
Предотвращение системных вызовов ... 210
Предотвращение ложных пробуждений ... 212
Блокировка чтения-записи ... 215
Предотвращение применения ждущего цикла в записывающих потоках ... 218
Предотвращение голода записывающих потоков ... 220
Резюме ... 224

Глава 10. Идеи и творческое воодушевление ... 225
Семафор ... 225
RCU ... 226
Связанный список без блокировки ... 227
Блокировки на основе очереди ... 228
Блокировки на основе парковок ... 229
Последовательная блокировка ... 230
Учебные материалы ... 231

Предметный указатель ... 233

Об авторе ... 237

Об изображении на обложке ... 238

Всем коллегам Rust-разработчикам,
которые ждали моей рецензии на свой код, пока я писала эту книгу

И моим близким тоже, конечно

Светлой памяти
Амелии Ады Луизы (Amélia Ada Louise),
1994–2021 гг.

Предисловие

В этой книге представлен отличный обзор низкоуровневой конкурентности в языке Rust, включая потоки, блокировки, счетчики ссылок, атомарности, каналы и многое другое. В ней исследуется проблематика, связанная с процессорами и операционными системами, причем в отношении последних приводится описание сложностей корректной работы конкурентного кода в Linux, macOS и Windows. Особенно меня обрадовало, что Мара иллюстрирует излагаемые темы примерами рабочего кода Rust. И в заключение она затрагивает тему семафоров, связанных списков без блокировок, блокировок на основе очереди, последовательных блокировок и даже RCU.

И что же эта книга может предложить такому человеку, как я, уже почти 40 лет копающемуся в коде C, а в последнее время — в самых глубинах ядра Linux?

О языке Rust мне впервые стало известно от энтузиастов и на конференциях, посвященных Linux. Но я был всецело поглощен своими делами, пока не увидел свое имя в статье LWN, посвященной Rust, "Using Rust for Kernel Development" ("Использование Rust для разработки ядра"). Это побудило меня разразиться целой серией публикаций в блоге на тему "So You Want to Rust the Linux Kernel?" ("Вам все неймется заразить ядро Linux ржавчиной?"[1]). Эта серия публикаций вызвала целый ряд оживленных дискуссий, которые можно увидеть в комментариях к ним.

В одной из таких дискуссий давний разработчик ядра Linux, написавший также немало кода на языке Rust, сказал мне, что при написании конкурентного кода в Rust нужно придерживаться того, что желает от вас сам язык Rust. И тогда я понял, что полученный мною этот ценный совет оставляет открытым вопрос о том, чего же именно хочет Rust. Эта книга дает превосходные ответы на данный вопрос, представляя немалую ценность как для Rust-разработчиков, желающих освоить конкурентность, так и для разработчиков конкурентного кода на других языках, которые хотели бы узнать, как наилучшим образом все это можно сделать в Rust.

Я, конечно же, отношусь к последней категории. Но должен признаться, что многие из оживленных дискуссий о конкурентной работе в Rust напоминают мне давние жалобы моих родителей, бабушек и дедушек на неудобные защитные приспособления, которые добавлялись к таким электроинструментам, как пилы и дрели. Некоторые из этих приспособлений теперь уже получили повсеместное распространение, но молотки, долота и бензопилы с тех пор не сильно изменились. Было

[1] Название языка Rust можно перевести как "ржавчина". — *Пер.*

совсем непросто определить, какие из механических приспособлений безопасности выдержат испытание временем, поэтому я рекомендую подходить к программным функциям безопасности с позиции глубочайшего смирения. И, пожалуйста, поймите, что это мое обращение адресовано как сторонникам таких функций, так и их противникам.

Это подводит нас к другой группе потенциальных читателей — скептикам Rust. Хотя я верю, что большинство таких скептиков оказывают сообществу ценную услугу, указывая на возможности для улучшений, прочитать эту книгу будет все же полезно всем, кроме, разве что, самых компетентных Rust-скептиков. Во всяком случае это позволило бы им повысить остроту и целенаправленность своих критических высказываний.

Есть ведь еще и закоренелые не-Rust-разработчики, которым предпочтительнее было бы реализовать имеющиеся в Rust механизмы безопасности, связанные с конкурентностью, на своем любимом языке. Эта книга позволит им поглубже разобраться в механизмах Rust, которые им желательно воспроизвести, или же, что еще привлекательнее, усовершенствовать.

И наконец, прогресс, достигнутый Rust на пути к включению в ядро Linux, отмечен многими разработчиками ядра Linux, в качестве примера можно привести статью Джонатана Корбета (Jonathan Corbet) "Next Steps for Rust in the Kernel" ("Следующие шаги по внедрению Rust в ядро"). На октябрь 2022 года все это еще не вышло из стадии эксперимента, но отношение к такому развитию событий становится все серьезнее. И настолько серьезнее, что Линус Торвальдс (Linus Torvalds) включил первые признаки поддержки языка Rust в версию 6.1 ядра Linux.

В чем бы ни заключался ваш интерес к этой книге, в расширении своих навыков работы с Rust за счет включения в них конкурентности, или же в расширении своих навыков конкурентности за счет включения в них программирования на языке Rust, в улучшении уже сложившейся, отличной от Rust среды, или просто в стремлении взглянуть на конкурентность с иной точки зрения, я желаю вам в путешествии по страницам этой книги всего наилучшего!

– Пол Э. МакКенни (Paul E. McKenney)

Команда разработки ядра платформы Meta
Компания Meta
Октябрь 2022 г.

Введение

Rust сыграл и продолжает играть весьма существенную роль в повышении доступности системного программирования. И все же такие темы низкоуровневой конкурентности, как атомарность и упорядочение памяти, по-прежнему зачастую представляются некой мистикой, которую лучше оставить на откуп весьма ограниченной группе специалистов.

Работая в течение последних нескольких лет над системами управления в режиме реального времени на базе Rust и обеспечивая поддержку стандартной библиотеки Rust, я обнаружила, что многие из доступных ресурсов по атомарности и смежным темам охватывают лишь небольшую часть искомой информации. Многие ресурсы целиком и полностью сосредоточены на C и C++, что может усложнить формирование связи с принятой в Rust концепцией безопасности (памяти и потоков) и системой типов. Ресурсы, освещающие подробности такой абстрактной теории, как модель памяти C++, зачастую всего лишь смутно объясняют ее связь с конкретным оборудованием, если вообще чего-то объясняют. Существует множество ресурсов, охватывающих каждый нюанс реального оборудования, например инструкции процессора и согласованность кеша, но для формирования целостного понимания во многих случаях требуется собирать информацию по частям из разных мест.

Эта книга представляет собой попытку собрать всю необходимую информацию в одном месте, сведя ее в единое целое и предоставив все необходимое для создания собственных корректных, безопасных и эргономичных примитивов конкурентных вычислений, обеспечивая при этом возможность в достаточной мере разобраться в базовом оборудовании и роли операционной системы для принятия проектных решений и достижения основных компромиссов в оптимизации программного кода.

Для кого предназначена эта книга

Основная аудитория, на которую рассчитана данная книга, — Rust-разработчики, желающие расширить свои познания в области низкоуровневой конкурентности. Но она может пригодиться и тем, кто еще только поверхностно знаком с Rust, но хотел бы знать, как выглядит низкоуровневая конкурентность с позиции программирования на этом языке.

Предполагается, что читатели уже знакомы с основами языка Rust, имеют установленную последнюю версию компилятора Rust и знают, как компилировать и запус-

кать код Rust с помощью `cargo`. Концепции Rust, играющие важную роль для конкурентных вычислений, кратко объясняются там, где это нужно, поэтому никаких предварительных знаний о конкурентности в Rust не требуется.

Обзор глав

Книга состоит из десяти глав.

Глава 1. Основы конкурентности в Rust. Здесь представлены все инструменты и концепции, необходимые для исследования базовых составляющих конкурентности в Rust: потоки, мьютексы, потокобезопасность, совместно используемые и исключительные ссылки, внутренняя изменяемость и т. д., формирующие основу для изложения всего остального материала книги.

Для опытных Rust-программистов, знакомых с этими концепциями, данная глава может послужить кратким напоминанием. А тем, кто еще нетвердо ориентируется в Rust, но знаком с этими концепциями из работы с другими языками, данная глава позволит быстро усвоить любые относящиеся к Rust знания, которые могут понадобиться для чтения остальной части книги.

Глава 2. Атомарность. Во второй главе изложены сведения об атомарных типах Rust и всех, связанных с ними операциях. Сначала рассматриваются простые операции загрузки и сохранения с последующим переходом к более сложным *циклам сравнения и обмена*, с изучением каждой новой концепции на нескольких полезных примерах, взятых из реальных сценариев ее использования.

Несмотря на актуальность *упорядочения памяти* для каждой атомарной операции, эта тема оставлена для следующей главы. А в данной главе рассматриваются только ситуации, где достаточно применения *расслабленного* упорядочения памяти, встречающиеся гораздо чаще, чем этого можно было бы ожидать.

Глава 3. Упорядочение памяти. После изучения различных атомарных операций и способов их применения в третьей главе происходит знакомство с самой сложной темой книги: упорядочением памяти.

Исследуется работа модели памяти, суть отношений "происходит до" и методы их формирования, выясняется, что собой представляют всевозможные способы упорядочения памяти и почему последовательно согласованное упорядочение не может стать ответом на все вопросы.

Глава 4. Создание своей собственной спин-блокировки. Усвоенные теоретические основы в последующих трех главах найдут свое практическое применение в создании своих собственных версий нескольких, весьма распространенных примитивов конкурентных вычислений. В первой, самой короткой из этих глав рассматривается реализация *спин-блокировки*.

Сначала будет создана минималистичная версия, позволяющая попрактиковаться в *упорядочении высвобождения и получения памяти*, а затем будет исследована концепция *безопасности* Rust, позволяющая превратить эту версию в эргономичный тип данных Rust, которому практически не грозит неправильное применение.

Глава 5. Создание своих собственных каналов. В пятой главе будут с нуля реализованы несколько вариантов *одноразового канала* — примитива, которым можно будет воспользоваться для отправки данных из одного потока в другой.

Начиная с самой минимальной, но совершенно небезопасной (`unsafe`) версии, будут описаны несколько способов разработки безопасного интерфейса с попутным рассмотрением проектных решений и их последствий.

Глава 6. Создание своего собственного типа Arc. В шестой главе будет решаться более сложная головоломка с упорядочением памяти. Будут осуществлены намерения реализовать с нуля свою собственную версию атомарного подсчета ссылок.

После добавления поддержки *слабых указателей* и оптимизации производительности окончательная версия станет практически идентична стандартному Rust-типу `std::sync::Arc`.

Глава 7. Представление о внутренней "кухне" процессора. Седьмая глава представляет собой глубокое погружение во все низкоуровневые подробности. Будет рассмотрено все интересное, что происходит на уровне процессора, на двух самых популярных процессорных архитектурах будет показано, как выглядят инструкции ассемблера, стоящие за атомарными операциями, объяснено, что такое кеширование и как оно влияет на производительность нашего кода, и выяснено, что осталось от модели памяти на аппаратном уровне.

Глава 8. Примитивы операционной системы. В *главе 8* придется признать существование таких подходов, осуществить которые без помощи *ядра* операционной системы просто невозможно, и узнать, какие функции доступны в Linux, macOS и Windows.

Мы рассмотрим примитивы конкурентности, доступные в POSIX-системах посредством расширений `pthreads`, выясним, что можно сделать с помощью Windows API, и узнаем, что можно сделать в Linux с помощью *системного вызова фьютекса.*

Глава 9. Создание своих собственных блокировок. Воспользовавшись всем, что удалось почерпнуть из предыдущих глав, в *главе 9* будут рассмотрены приемы создания с нуля нескольких реализаций *мьютекса*, *условной переменной* и *блокировки чтения-записи.*

Каждая из разработок сначала будет реализована в минимальной, но полноценной версии, а затем будут предприняты попытки их оптимизации различными способами. С помощью целого ряда простых тестов производительности будет выяснено, что предпринимаемые попытки оптимизации не всегда приводят к повышению производительности, и наряду с этим будут рассмотрены различные компромиссы, на которые приходится идти при проектировании.

Глава 10. Идеи и творческое воодушевление. В последней главе будет сделано все, чтобы вы не оказались после прочтения книги в абсолютной пустоте, чтобы в вашем распоряжении остались идеи и вы вдохновились на созидательный и

исследовательский труд, используя вновь приобретенные знания и навыки, возможно, положив начало захватывающему путешествию, еще больше углубляясь в мир низкоуровневой конкурентности.

Примеры кода

Весь программный код в этой книге написан и протестирован с использованием версии Rust 1.66.0, вышедшей 15 декабря 2022 года. В более ранние версии включены не все функции, рассмотренные в данной книге. Но в более поздних версиях все должно работать без проблем.

Для краткости в примерах кода отсутствуют инструкции `use`, за исключением самого первого случая введения нового элемента из стандартной библиотеки. Для удобства можно воспользоваться следующим начальным фрагментом кода для импорта всего необходимого для компиляции любого примера кода, приведенного в данной книге:

```
#[allow(unused)]
use std::{
    cell::{Cell, RefCell, UnsafeCell},
    collections::VecDeque,
    marker::PhantomData,
    mem::{ManuallyDrop, MaybeUninit},
    ops::{Deref, DerefMut},
    ptr::NonNull,
    rc::Rc,
    sync::{*, atomic::{*, Ordering::*}},
    thread::{self, Thread},
};
```

Дополнительные материалы, включая полные версии всех примеров кода, доступны по адресу **https://marabos.nl/atomics/**.

Все примеры кода, представленные в данной книге, вы можете использовать в любых целях.

Указание авторства приветствуется, но не является обязательным.

Условные обозначения, используемые в данной книге

В этой книге приняты следующие типографские соглашения:

- *Курсив* — для новых терминов и акцентов.
- `Моноширинный шрифт` — для листингов программ, а также внутри абзацев для ссылки на такие элементы программы, как имена переменных или функций, типы данных, операторы и ключевые слова.

Данный элемент обозначает подсказку или совет.

Данный элемент обозначает общее примечание.

Данный элемент указывает на предупреждение или предостережение.

Благодарности

Хочу поблагодарить всех, кто принимал участие в создании этой книги. Поддержка и полезный вклад от множества людей стали для меня поистине неоценимой помощью. В частности, считаю нужным высказать свою благодарность Аманье д'Антрас (Amanieu d'Antras), Арию Бейнжеснеру (Aria Beingessner), Полу МакКенни (Paul McKenney), Кэрол Николс (Carol Nichols) и Мигелю Разу Гусману Маседо (Miguel Raz Guzmán Macedo) за их неоценимые и глубокомысленные отзывы о ранних черновых вариантах книги. Я также хотела бы поблагодарить всех в издательстве O'Reilly, и в особенности моих редакторов Ширу Эванс (Shira Evans) и Зана МакКуэйда (Zan McQuade) за их неиссякаемый энтузиазм и поддержку.

ГЛАВА 1

Основы конкурентности в Rust

Операционные системы обеспечивали одновременный запуск множества программ задолго до появления многоядерных процессоров. Делалось это за счет быстрого переключения между процессами, в результате чего они многократно и поочередно выполнялись, постепенно шаг за шагом решая свои задачи. Сегодня многоядерные процессоры, способные в настоящем параллельном режиме выполнять сразу несколько процессов, стоят практически во всех наших компьютерах и даже в телефонах и часах.

Операционные системы добиваются максимальной изолированности процессов друг от друга, позволяя программе выполнять свою работу в полном "неведении" о том, чем занимаются другие процессы. В частности, процесс, как правило, не может получить доступ к памяти другого процесса или каким-либо образом взаимодействовать с ним без предварительного запроса к ядру операционной системы.

Но программа может как часть одного и того же *процесса* создавать дополнительные *потоки выполнения*. Потоки внутри одного процесса не изолированы друг от друга, совместно используют память, посредством чего могут осуществлять взаимодействие.

В данной главе объясняется, как создаются потоки в Rust, а также рассматриваются все основные концепции, связанные с потоками, позволяющие, в частности, добиться безопасного обмена данными между несколькими потоками. Понятия, вводимые в данной главе, являются основополагающими для остальной части книги.

Если все эти особенности Rust вам уже знакомы, можете смело переходить к изучению следующего материала. Но до перехода к последующим главам нелишне будет убедиться в том, что вы уже достаточно неплохо разбираетесь в потоках, внутренней изменяемости, `Send` и `Sync`, а также знаете, что такое мьютекс, условная переменная и парковка потоков.

Потоки в Rust

Работа каждой программы начинается строго с одного, так называемого основного потока. Этот поток будет выполнять функцию `main` и при необходимости может порождать дополнительные потоки.

Новые потоки в Rust создаются с помощью функции стандартной библиотеки `std::thread::spawn`, принимающей один аргумент: функцию, выполняемую новым

потоком. Как только эта функция вернет управление, поток останавливается. Рассмотрим пример:

```
use std::thread;
fn main() {
    thread::spawn(f);
    thread::spawn(f);
    println!("Hello from the main thread.");
}
fn f() {
    println!("Hello from another thread!");
    let id = thread::current().id();
    println!("This is my thread id: {id:?}");
}
```

Здесь порождаются два потока, каждый из которых будет выполнять свою основную функцию `f`. Оба потока выведут сообщение и покажут свой *идентификатор*, а кроме того, свое сообщение выведет еще и основной поток.

Идентификатор потока

Каждому потоку стандартной библиотекой Rust присваивается уникальный идентификатор. Этот идентификатор, имеющий тип `ThreadId`, можно получить посредством вызова функции `Thread::id()`. Идентификатор `ThreadId` можно разве что скопировать и проверить на равенство. У разных потоков будут разные идентификаторы, а вот последовательность их присваивания не гарантируется.

Если запустить данный пример несколько раз, можно будет заметить, что выходные данные от запуска к запуску различаются. Вот как выглядит результат, полученный на моем компьютере в одном из запусков:

```
Hello from the main thread.
Hello from another thread!
This is my thread id:
```

Похоже, часть вывода отсутствует.

Дело вот в чем: основной поток завершил выполнение `main` еще до того, как выполнение своих функций завершили вновь порожденные потоки.

Возврат из `main` приведет к выходу из всей программы, даже при условии, что другие потоки все еще выполняются.

В данном конкретном примере, прежде чем программа была завершена основным потоком, одному из вновь порожденных потоков хватило времени, чтобы вывести половину второго сообщения.

Если нужно обеспечить завершение потоков до возврата управления из `main`, их окончания можно дождаться, *присоединившись* к ним. Для этого следует воспользоваться значением типа `JoinHandle`, возвращаемым функцией `spawn`:

```
fn main() {
    let t1 = thread::spawn(f);
    let t2 = thread::spawn(f);
```

```
    println!("Hello from the main thread.");
    t1.join().unwrap();
    t2.join().unwrap();
}
```

Метод `.join()` ожидает завершения выполнения потока и возвращает значение `std::thread::Result`. Если из-за возникшей паники поток не сможет успешно завершить выполнение своей функции, в этом значении будет содержаться сообщение о панике. Можно попытаться справиться с данной ситуацией или же просто вызвать метод `.unwrap()`, чтобы вызвать панику при присоединении к запаниковавшему потоку.

Запуск данной версии программы больше не будет приводить к усечению вывода:

```
Hello from the main thread.
Hello from another thread!
This is my thread id: ThreadId(3)
Hello from another thread!
This is my thread id: ThreadId(2)
```

Единственное, что будет по-прежнему меняться от запуска к запуску, так это порядок вывода сообщений:

```
Hello from the main thread.
Hello from another thread!
Hello from another thread!
This is my thread id: ThreadId(2)
This is my thread id: ThreadId(3)
```

Блокировка вывода

Чтобы обеспечить непрерывность вывода, в макросе `println` используется метод `std::io::Stdout::lock()`. Прежде чем записывать какой-либо вывод, выражение `println!()` будет ждать, пока не завершится любое одновременно выполняющееся с ним выражение. В противном случае мог бы быть получен общий вывод с произвольным чередованием выводов от каждого потока, например:

```
Hello fromHello from another thread!
 another This is my threthreadHello fromthread id: ThreadId!
( the main thread.
2)This is my thread
id: ThreadId(3)
```

Вместо передачи в `std::thread::spawn` имени функции, как в рассмотренном ранее примере, туда гораздо чаще передается *замыкание*, позволяющее захватывать значения для их перемещения в новый поток:

```
let numbers = vec![1, 2, 3];
thread::spawn(move || {
    for n in &numbers {
        println!("{n}");
    }
}).join().unwrap();
```

Здесь благодаря использованию move-замыкания владение переменной numbers передается вновь порожденному потоку. Без применения ключевого слова move замыкание захватывало бы numbers по ссылке. В таком случае компилятор выдал бы ошибку, поскольку новый поток мог бы просуществовать дольше переменной.

Поскольку поток может выполняться до самого конца выполнения программы, функция spawn определяется с параметром времени жизни 'static, привязанным к типу ее аргумента. Иными словами, она допускает применение только тех функций, которые могут сохраняться вечно. Замыкание, захватывающее локальную переменную по ссылке, вечно сохраняться не может, поскольку эта ссылка станет недействительной в тот самый момент, когда локальная переменная перестанет существовать.

Получение значения из потока осуществляется путем его возврата из замыкания через тип Result, возвращаемый методом join:

```
let numbers = Vec::from_iter(0..=1000);
let t = thread::spawn(move || {
    let len = numbers.len();
    let sum = numbers.iter().sum::<usize>();
    sum / len                                  ❶
});
let average = t.join().unwrap();               ❷
println!("average: {average}");
```

Здесь значение, возвращаемое замыканием потока ❶, передается обратно в основной поток посредством вызова метода join ❷.

Если бы у переменной numbers было пустое значение, то при попытке деления на нуль ❶ поток бы запаниковал, а join вернул бы это паническое сообщение, что из-за использования unwrap ❷ привело бы к панике и в основном потоке.

Средство порождения потоков

Функция std::thread::spawn, по сути, просто удобное сокращение от std::thread::Builder::new().spawn().unwrap().

Структура Std::thread::Builder позволяет настраивать параметры нового потока перед его созданием. Ею можно воспользоваться для настройки размера стека для нового потока и присвоения ему имени. Имя потока можно получить через вызов метода std::thread::current().name(). Это имя будет использоваться в панических сообщениях, и его будет видно в средствах мониторинга и отладки на большинстве платформ.

Кроме того, функция spawn, принадлежащая структуре Builder, возвращает std::io::Result, позволяя тем самым справляться с неудачными попытками порождения нового потока. Такое может случиться, когда операционной системе не хватает памяти или же когда к вашей программе применены ограничения ресурсов. Если функция std::thread::spawn не может породить новый поток, она просто паникует.

Потоки с областью действия

Если точно известно, что порождаемый поток не переживет определенную область видимости, этот поток может безопасно заимствовать то, что не живет вечно, например локальные переменные, главное, чтобы они пережили эту область видимости.

Для порождения *потоков с заданной областью действия* стандартная библиотека Rust предоставляет функцию `std::thread::scope`. Она дает возможность порождать потоки, которые не могут пережить область видимости замыкания, передаваемого этой функции, что позволяет безопасно заимствовать локальные переменные.

Работу данной функции лучше всего показать на примере (листинг 1.1).

Листинг 1.1

```
let numbers = vec![1, 2, 3];
thread::scope(|s| {                          ❶
    s.spawn(|| {                             ❷
        println!("length: {}", numbers.len());
    });
    s.spawn(|| {                             ❷
        for n in &numbers {
            println!("{n}");
        }
    });
});                                          ❸
```

❶ Вызов функции `std::thread::scope` с замыканием. Данное замыкание выполняется напрямую и получает аргумент `s`, представляющий область видимости.

❷ Переменная `s` используется для создания потоков. Замыкания могут заимствоваться такими локальными переменными, как `numbers`.

❸ Как только область видимости завершит свое существование, все еще не объединенные потоки автоматически объединятся.

Такая схема гарантирует, что рассматриваемую область видимости не сможет пережить ни один из порожденных в ней потоков. Поэтому данный метод `spawn` с ограниченной областью видимости не имеет привязки `'static` к типу аргумента, что позволяет ссылаться на любую сущность, пока она не выйдет за пределы области видимости, например на `numbers`.

В приведенном примере к `numbers` одновременно обращаются оба новых потока. Это вполне допустимо, поскольку значение данной переменной не изменяет ни один из потоков (включая также и основной поток). Если, как показано далее, внести в код, выполняемый в первом потоке, изменение значения `numbers`, компилятор не позволит породить другой поток, также использующий `numbers`:

```
let mut numbers = vec![1, 2, 3];
thread::scope(|s| {
    s.spawn(|| {
        numbers.push(1);
    });
    s.spawn(|| {
        numbers.push(2); // Ошибка!
    });
});
```

Конкретное содержание сообщения об ошибке зависит от версии компилятора Rust, поскольку он довольно часто подвергается усовершенствованиям для улучшения проводимой диагностики, и при попытке скомпилировать приведенный код будет получен примерно следующий результат:

```
error[E0499]: cannot borrow 'numbers' as mutable more than once at a time -->
example.rs:7:13                                                          ❶
  |
4 |     s.spawn(|| {
  |     -- first mutable borrow occurs here                                ❷
5 |     numbers.push(1);
  | -- first borrow occurs due to use of 'numbers' in closure              ❸
  |
7 |     s.spawn(|| {
  |         ^^ second mutable borrow occurs here                           ❹
8 |         numbers.push(2);
  | -- second borrow occurs due to use of 'numbers' in closure             ❺
```

❶ Ошибка `[E0499]`: многократное заимствование изменяемой переменной `'numbers'` недопустимо.

❷ -- здесь произошло первое заимствование изменяемой переменной.

❸ -- первое заимствование произошло из-за использования `'numbers'` в замыкании.

❹ ^^ здесь произошло второе заимствование изменяемой переменной.

❺ -- второе заимствование произошло из-за использования `'numbers'` в замыкании.

The Leakpocalypse, или же Утечка сродни апокалипсису

До выпуска Rust 1.0 в стандартной библиотеке была функция `std::thread::scoped`, которая, как и `std::thread::spawn`, занималась непосредственным порождением потока. При этом допускались захваты аргументов, не имеющих `'static`-привязки, поскольку вместо `JoinHandle` она возвращала `JoinGuard`, присоединявшийся к потоку при удалении. И любым заимствованным данным нужно было всего лишь пережить этот `JoinGuard`. Такое решение представлялось безопасным при том условии, что `JoinGuard` когда-либо будет удален.

Но перед самым выпуском Rust 1.0 постепенно пришло понимание того, что гарантировать последующее удаление чего-либо просто невозможно. Есть немало ситуаций, позволяющих о чем-то забыть или же допустить *утечку* какого-либо объекта без его удаления.

Некоторые специалисты предложили название "The Leakpocalypse", и в итоге был сделан вывод, что проектирование безопасного интерфейса не может основываться на предполо-

жении, что объекты непременно будут в конце своего жизненного цикла удаляться. Утечка объекта вполне может привести к утечке еще большего числа объектов (например, утечка `Vec` также приведет к утечке всех его элементов), но может и не привести к неопределенному поведению. Поэтому функция `std::thread::scoped` больше не считалась безопасной и была удалена из стандартной библиотеки. Кроме того, чтобы подчеркнуть постоянно имеющуюся возможность забыть о чем-либо (или допустить утечку), функция `std::mem::forget` была преобразована из небезопасной в безопасную.

И лишь гораздо позже в Rust 1.63 была добавлена новая функция `std::thread::scope` с обновленным содержимым, корректность которой уже не зависела от `Drop`.

Совместное владение и подсчет ссылок

Пока мы рассмотрели передачу владения значением потоку с помощью `move`-замыкания (см. *раздел "Потоки в Rust"*) и заимствование данных из долгоживущих родительских потоков (см. *раздел "Потоки с областью действия"*). Когда при совместном использовании данных двумя потоками нет никакой гарантии, что ни один из них сможет прожить дольше другого, владельцем этих данных не может быть ни один из рассматриваемых потоков. Любые данные, которыми эти потоки обмениваются, должны жить так же долго, как и самый долгоживущий из потоков.

Статика

Для создания чего-то, что не принадлежит ни одному из потоков, существует несколько способов. Самый простой из них заключается в задании `static`-значения, "принадлежащего" всей программе, а не какому-либо из потоков. В следующем примере показана возможность получения доступа к переменной `X` обоими потоками, не являющимися ее владельцами:

```
static X: [i32; 3] = [1, 2, 3];
thread::spawn(|| dbg!(&X));
thread::spawn(|| dbg!(&X));
```

У `static`-элемента имеется постоянный инициализатор, который никогда не удаляется и всегда существует еще до запуска основной функции программы. Позаимствовать этот элемент может каждый поток, поскольку он гарантированно будет существовать всегда.

Утечка

Другой способ совместного владения заключается в организации *утечки* выделенной памяти. Используя `Box::leak`, можно высвободить владение `Box`, обязуясь таким образом никогда не удалять этот объект. С данного момента `Box` будет жить вечно, не являясь ничьим владением и позволяя любому потоку заимствовать его на все время работы программы:

```
let x: &'static [i32; 3] = Box::leak(Box::new([1, 2, 3]));
thread::spawn(move || dbg!(x));
thread::spawn(move || dbg!(x));
```

Может показаться, что `move`-замыкание передает потокам права владения, но если присмотреться к типу `x`, станет понятно, что потоки получают всего лишь *ссылку* на данные.

Типом ссылок является `Copy`, а это означает, что при их "перемещении" (с помощью `move`), оригинал продолжает свое существование, как и в случае с целочисленным или логическим значением.

Следует заметить, что задание `'static`-существования не приводит к существованию значения с момента запуска программы и означает всего лишь то, что оно будет существовать до конца ее работы. Прошлое просто не имеет значения.

Недостаток утечки `Box` — наличие *утечки памяти*. Некий объем памяти выделяется, но при этом выделение никогда не аннулируется и память не высвобождается. Смириться с этим можно только при ограниченном числе подобных случаев. Но при бесконтрольном использовании данного приема программа постепенно исчерпает память.

Подсчет ссылок

Чтобы гарантировать удаление совместно используемых данных и высвобождение выделенной под них памяти, перейти к полному отказу от владения ими не представляется возможным. Вместо этого мы можем применить *совместное владение*. Отслеживая число владельцев, можно гарантировать, что значение будет удалено только в том случае, когда не останется ни одного владельца.

Стандартная библиотека Rust предоставляет эту возможность благодаря наличию типа `std::rc::Rc` (RC — сокращение от "Reference Counted", т. е. "подсчет ссылок"). Этот тип очень похож на `Box`, но при клонировании не происходит нового выделения памяти, а вместо этого просто увеличивается значение счетчика, хранящегося рядом со значением, содержащимся в этом типе. Как исходный, так и клонированный `Rc` будут ссылаться на одно и то же место выделенной памяти; они будут им *совместно владеть*:

```
use std::rc::Rc;
let a = Rc::new([1, 2, 3]);
let b = a.clone();
assert_eq!(a.as_ptr(), b.as_ptr()); // Одно и то же место выделенной памяти!
```

Удаление `Rc` приведет к уменьшению значения счетчика. А к удалению содержащихся в объекте данных и к высвобождению выделенной под них памяти приведет только удаление последнего `Rc`, как только станет заметно, что счетчик был сброшен в нуль.

Но попытка отправки `Rc` в другой поток приведет к следующей ошибке, выдаваемой компилятором:

```
error[E0277]: 'Rc' cannot be sent between threads safely      ❶
  |
8 |        thread::spawn(move || dbg!(b));
  |                      ^^^^^^^^^^^^^^^
```

❶ Ошибка `[E0277]: 'Rc'` не может безопасно передаваться между потоками.

Но, как оказалось, `Rc`-объекты не относятся к *потокобезопасным* (более полно это понятие раскрывается далее в *разделе "Потокобезопасность: Send и Sync"*). Если в одном и том же месте выделенной памяти имеется `Rc`-объект сразу для нескольких потоков, эти потоки могут предпринять попытки одновременного изменения счетчика ссылок, что может привести к непредсказуемым результатам.

Во избежание этого можно воспользоваться методом `std::sync::Arc` (Arc — сокращение от "Atomically reference counted", т. е. "атомарный подсчет ссылок"). Его действие идентично `Rc`, за исключением того, что им гарантируются изменения счетчика в рамках неделимых *атомарных* операций, делая этот метод безопасным для использования сразу в нескольких потоках. Подробнее данная тема рассматривается в *главе 2*.

```
use std::sync::Arc;
let a = Arc::new([1, 2, 3]);          ❶
let b = a.clone();                    ❷
thread::spawn(move || dbg!(a));       ❸
thread::spawn(move || dbg!(b));       ❸
```

❶ Помещение массива в новую выделенную область памяти вместе со счетчиком ссылок, стартующим с единицы.

❷ Клонирование `Arc` приводит к увеличению счетчика ссылок до двух и дает нам второй `Arc`-объект с тем же выделенным местом в памяти.

❸ Оба потока получают свой собственный `Arc`-экземпляр, посредством которого могут обращаться к совместно используемому массиву. Удаление Arc-экземпляра любым из этих потоков приводит к уменьшению значения счетчика ссылок. Последний поток, удаляющий свой `Arc`-экземпляр, заметит, что счетчик обнулился, и будет именно тем потоком, который удалит массив и высвободит занимаемую им память.

Именование клонов

Необходимость давать каждому клону `Arc` другое имя может быстро привести к запутанности и неразборчивости кода. Хотя каждый клон `Arc` является по сути отдельным объектом, все эти клоны представляют собой одно и то же совместно используемое значение, что не находит должного отражения при различном именовании каждого из клонов.

Rust разрешает (и поощряет) применение *затенения* переменных путем определения новой переменной с тем же именем. Если сделать это в той же области видимости, то обращение по имени к исходной переменной станет невозможным. Но при открытии новой области видимости для повторного использования в ней точно такого же имени при условии сохранения доступности исходной переменной за пределами этой области видимости можно воспользоваться такой инструкцией, как `let a = a.clone();`.

Заключив замыкание в новую область видимости (с помощью {}), можно клонировать переменные до их перемещения в замыкание, не испытывая при этом потребности в их переименовании.

```
let a = Arc::new([1, 2, 3]);
let b = a.clone();
thread::spawn(move || {
    dbg!(b);
});
dbg!(a);
```

Клон Arc существует в той же области видимости. Каждый поток получает свой собственный клон с другим именем.

```
let a = Arc::new([1, 2, 3]);
thread::spawn({
    let a = a.clone();
    move || {
        dbg!(a);
    }
});
dbg!(a);
```

Клон Arc существует в другой области видимости. Одно и то же имя может использоваться в каждом потоке.

Поскольку владение является совместным, на указатели подсчета ссылок (`Rc<T>` и `Arc<T>`) распространяются те же ограничения, что и на совместно используемые ссылки (`&T`). Они не предоставляют доступа для внесения изменений к тому значению, которое в них содержится, поскольку в то же самое время это значение может быть заимствовано другой программой.

Например, при попытке сортировки части целых чисел в `Arc<[i32]>` компилятор воспрепятствовал бы этому, сообщив, что изменять данные запрещено:

```
error[E0596]: cannot borrow data in an 'Arc' as mutable        ❶
  |
6 |     a.sort();
  |     ^^^^^^^^
```

❶ Ошибка `[E0596]`: невозможно заимствовать данные в `'Arc'` из-за потенциальной изменяемости.

Заимствования и гонка данных

Значения в Rust можно заимствовать двумя способами:

- *Неизменяемое заимствование* — заимствование чего-либо с помощью `&` приводит к созданию *неизменяемой ссылки*, которую можно скопировать. Доступ к данным, на которые она ссылается, совместно получают все копии этой ссылки. Судя по названию, компилятор, как правило, не позволяет изменять то, на что указывает такая ссылка, поскольку изменение может повлиять на другой код, заимствующий в тот же самый момент те же данные.
- *Изменяемое заимствование* — заимствование чего-либо с помощью `&mut` приводит к созданию *изменяемой ссылки*. Изменяемое заимствование является единственным активным заимствованием тех данных, на которые указывает такая ссылка. Тем самым гарантируется, что изменение данных не приведет к изменению чего-либо, что находится в поле зрения другого программного кода.

Совокупность двух данных концепций полностью исключает возникновение *гонки данных*, характеризующейся ситуацией, в которой один поток изменяет данные, а другой в то же самое время к ним обращается. Гонка данных обычно приводит к *неопределенному поведению*, и компилятору в отсутствие предпосылок к ней уже не нужно брать в расчет ситуации, приводящие к ее возникновению. Он просто будет исходить из предположения, что подобные ситуации никогда не возникнут.

Чтобы уточнить, что именно имеется в виду, рассмотрим пример (листинг 1.2), в котором компилятор на основе правил заимствования способен выдвинуть полезное предположение.

Листинг 1.2

```
fn f(a: &i32, b: &mut i32) {
    let before = *a;
    *b += 1;
    let after = *a;
    if before != after {
        x(); // Никогда не выполняется
    }
}
```

Здесь берется неизменяемая ссылка на целочисленное значение, которое сохраняется как до, так и после увеличения целого числа, на которое ссылается `b`. Компилятор может предполагать, что фундаментальные правила в отношении заимствований и гонок данных неукоснительно соблюдаются, а это означает, что `b` не может ссылаться на то же целое число, что и `a`. Фактически ничто во всей программе не может заимствовать изменяемое целое число, на которое ссылается `a`, пока оно заимствовано `a`. Следовательно, компилятор может легко прийти к заключению, что `*a` не изменится и условие оператора `if` никогда не будет истинным, и может для оптимизации полностью удалить из программы вызов `x`.

Написать программу на Rust, нарушающую предположения компилятора, невозможно, кроме тех случаев, когда для отключения ряда проверок безопасности будет использован `unsafe`-блок.

Неопределенное поведение

Чтобы избежать так называемого *неопределенного поведения*, в таких языках, как C, C++ и Rust имеется набор неукоснительно выполняемых правил. Например, одно из правил Rust гласит, что на любой объект ни в коем случае не может быть более одной изменяемой ссылки.

В Rust любое из этих правил можно нарушить, только применяя небезопасный `unsafe`-код. "Небезопасный" означает скорее не то, что код некорректен или опасен для использования, а то, что компилятор не проводит за вас проверку безопасности кода. Если код нарушает эти правила, его называют *несовершенным*.

Компилятору разрешается предполагать без проверки, что эти правила никогда не нарушаются. Их нарушения приводят к так называемому *неопределенному поведению*, которого следует избегать любой ценой. Разрешение компилятору опираться на предположения, фак-

тически не соответствующие действительности, может легко привести его к еще более неверным выводам о различных частях созданного вами кода, что повлияет на всю программу.

Давайте в качестве конкретного примера рассмотрим небольшой фрагмент кода, в котором для получения среза вызывается метод `get_unchecked`:

```
let a = [123, 456, 789];
let b = unsafe { a.get_unchecked(index) };
```

Метод `get_unchecked` предоставляет элемент среза по индексу, как и `a[index]`, но позволяет компилятору предположить без каких-либо проверок, что индекс никогда не выходит за границы массива.

В данном фрагменте кода, поскольку длина `a` равна 3, компилятор может предположить, что индекс меньше трех. Будет ли это предположение верным, зависит только от нас.

Если нарушить данное предположение, например если запустить код на выполнение с `index`, равным 3, может произойти что угодно. Это может привести к чтению из памяти всего, что было сохранено в байтах сразу после `a`. Может случиться сбой программы. В конечном итоге это может привести к выполнению какой-то совершенно не связанной с задуманной логикой части программы. Может возникнуть любой хаос.

Как ни удивительно, неопределенное поведение может даже инициировать "путешествие во времени", вызывая проблемы в предшествующем коде. Чтобы понять, как это может произойти, давайте представим, что до предыдущего фрагмента имелась следующая `match`-инструкция:

```
match index {
    0 => x(),
    1 => y(),
    _ => z(index),
}
let a = [123, 456, 789];
let b = unsafe { a.get_unchecked(index) };
```

Небезопасность кода позволила компилятору предполагать, что `index` всегда равен 0, 1 или 2. Можно прийти к логическому заключению, что последняя часть нашей инструкции `match` всегда будет соответствовать только 2, и, следовательно, z вызывается когда-либо только как `z(2)`. Данным заключением можно воспользоваться для оптимизации не только `match`, но и самой функции `z`, которая может включать в себя удаление неиспользуемых частей кода.

Если запустить на выполнение код с `index`, равным 3, то программа может попытаться выполнить уже подвергшиеся оптимизации части кода, что повлечет за собой совершенно непредсказуемое поведение задолго до того, как черед выполнения доберется до `unsafe`-блока в последней строке. Точно так же неопределенное поведение может распространяться по всей программе как вперед, так и назад, зачастую весьма неожиданным образом.

При вызове любой `unsafe`-функции следует внимательно изучить ее документацию и убедиться в полном понимании требований безопасности, т. е. тех предположений, которые должны быть неукоснительно соблюдены вызывающей стороной во избежание неопределенного поведения.

Внутренняя изменяемость

Правила заимствования, согласно представлениям, изложенным в предыдущем разделе, достаточно просты, но могут накладывать на код весьма серьезные ограничения, особенно когда в программе задействовано сразу несколько потоков. Сле-

дование этим правилам существенно ограничивает и практически сводит на нет взаимодействие между потоками из-за невозможности внесения изменений в те данные, которые доступны сразу нескольким потокам.

Но выход все же есть, и это *внутренняя изменяемость*. Тип данных с внутренней изменяемостью отчасти нарушает правила заимствования. При определенных условиях соответствующие типы могут допускать внесение изменений посредством "неизменяемой" ссылки.

В предыдущем *разделе "Подсчет ссылок"* уже был показан один наглядный пример, связанный с внутренней изменяемостью. Как `Rc`, так и `Arc` изменяют счетчик ссылок даже при наличии нескольких клонов, использующих один и тот же счетчик ссылок.

Как только речь заходит о внутренних изменяемых типах, возникает путаница и некорректность при употреблении в отношении ссылок таких понятий, как "неизменяемые" или "изменяемые", поскольку есть вещи, которые могут быть изменены с помощью обоих типов. Более точно передают суть предмета обсуждения такие понятия, как "совместно используемые" и "исключительные": *совместно используемая* ссылка (`&T`) может быть скопирована и передана другому коду, в то время как *исключительная ссылка* (`&mut T`) гарантирует, что речь идет именно об *исключительном заимствовании* данного `T`. Для большинства типов совместно используемые ссылки не допускают изменений, но есть и исключения. Поскольку в данной книге в основном будет вестись работа именно с такими исключениями, далее в книге будет принята более точная терминология.

Следует понимать, что внутренняя изменяемость нарушает правила только совместного заимствования и допускает внесение изменений при совместном использовании. А на исключительное заимствование это не распространяется. Исключительное заимствование по-прежнему гарантирует отсутствие других активных заимствований. Небезопасный код, выполнение которого приводит к созданию более чем одной активной исключительной ссылки на что-либо, независимо от внутренней изменяемости всегда приводит к неопределенному поведению.

А теперь давайте рассмотрим несколько типов с внутренней изменяемостью и то, как ими могут допускаться внесения изменений посредством совместно используемых ссылок, не вызывая при этом неопределенного поведения.

Cell

Структура `std::cell::Cell<T>` просто служит оболочкой для `T`, но допускает изменение данных через совместно используемую ссылку. Чтобы избежать неопределенного поведения, она позволяет только скопировать значение (если `T` относится к `Copy`) или заменить его целиком другим значением. Кроме того, эту структуру можно использовать только в пределах одного потока.

Давайте рассмотрим пример (листинг 1.3), аналогичный приведенному в предыдущем разделе (см. листинг 1.2), но на этот раз вместо `i32` воспользуемся `Cell<i32>`.

Листинг 1.3

```
use std::cell::Cell;
fn f(a: &Cell<i32>, b: &Cell<i32>) {
    let before = a.get();
    b.set(b.get() + 1);
    let after = a.get();
    if before != after {
        x(); // Может выполниться
    }
}
```

Теперь, в отличие от прошлого раза, условие `if` может быть истинным. Поскольку `Cell<i32>` обладает внутренней изменяемостью, компилятор больше не может выстроить предположение, что данное значение не изменится, пока на него имеется совместно используемая ссылка. Как `a`, так и `b` могут ссылаться на одно и то же значение, поэтому изменение, проведенное через `b`, может также повлиять и на `a`. Но можно все же предположить, что в это же время к ячейкам памяти не обращаются никакие другие потоки.

Порой работать с ограничениями, накладываемыми на `Cell`, нелегко. Поскольку данная структура не может напрямую позволить позаимствовать хранимое в ней значение, это значение следует переместить (оставив что-либо на его месте), внести в него изменение, а затем, для изменения содержимого ячейки, вернуть его обратно:

```
fn f(v: &Cell<Vec<i32>>) {
    let mut v2 = v.take(); // Замена содержимого Cell пустым Vec
    v2.push(1);
    v.set(v2); // Помещение измененного Vec обратно
}
```

RefCell

В отличие от обычной структуры `Cell`, `std::cell::RefCell` позволяет заимствовать свое содержимое с небольшими издержками времени выполнения. Структура `RefCell<T>` содержит не только `T`, но еще и счетчик, отслеживающий любые непогашенные заимствования. При попытке ее заимствования в тот момент, когда она уже заимствована с возможностью внесения изменений (или же наоборот), возникнет паника, позволяющая избежать неопределенного поведения. `RefCell`, как и `Cell`, можно использовать только в одном потоке.

Заимствование содержимого `RefCell` осуществляется вызовом `borrow` или `borrow_mut`:

```
use std::cell::RefCell;
fn f(v: &RefCell<Vec<i32>>) {
    v.borrow_mut().push(1); // 'Vec' можно изменять напрямую
}
```

Конечно, `Cell` и `RefCell` могут оказаться весьма полезными, но их применение сводится к нулю, когда возникает необходимость работы с несколькими потоками. Поэтому нам стоит перейти к типам, актуальным для конкурентной работы.

Mutex и *RwLock*

Тип `RwLock`, или *блокировка чтения-записи* (reader-writer lock), представляет собой версию `RefCell` для работы в конкурентном режиме. `RwLock<T>` содержит в себе `T` и отслеживает все непогашенные заимствования. Но, в отличие от `RefCell`, эта структура не вызывает панику при конфликтующих заимствованиях. Вместо этого она блокирует текущий поток, переводя его в спящий режим, ожидая прекращения существования конфликтующих заимствований. Придется просто терпеливо ждать своей очереди доступа к данным, наступающей после того, как с ними закончат свою работу другие потоки.

Заимствование содержимого `RwLock` называется *блокировкой*. Путем *блокировки* временно ставится заслон конфликтующим на данный момент заимствованиям, что позволяет выполнять заимствование, не вызывая гонки данных.

Мьютекс (`Mutex`) очень похож на `RWLock`, но по задумке несколько проще. Вместо отслеживания, как в случае с `RWLock`, числа совместно используемых и исключительных заимствований, мьютекс разрешает только исключительные заимствования.

Более подробно эти типы будут рассмотрены далее в *разделе "Блокировка: мьютексы и RwLock-блокировки"*.

Атомарные типы

Атомарные типы представляют собой версию `Cell` для работы в конкурентном режиме и являются основной темой содержимого *глав 2* и *3* данной книги. Как и `Cell`, они позволяют избежать неопределенного поведения, заставляя копировать значения в них и из них целиком, не позволяя заимствовать содержимое напрямую.

Но, в отличие от `Cell`, они не могут иметь произвольный размер. Поэтому для любого `T` обобщенного типа `Atomic<T>` просто не существует, есть только конкретные атомарные типы, такие как `AtomicU32` и `AtomicPtr<T>`. Какие из них доступны, зависит от платформы, поскольку, чтобы избежать гонки данных, для них требуется поддержка со стороны процессора. Подробности будут изложены в *главе 7*.

Из-за ограниченности атомарных типов в размерах зачастую они не содержат напрямую ту информацию, которая должна быть совместно использована потоками. Вместо этого они нередко применяются в качестве инструмента, позволяющего обмениваться другими, чаще всего более объемными данными между потоками. При использовании атомарных данных в качестве какой-либо информации о других данных всё может неожиданно усложниться.

UnsafeCell

Тип `UnsafeCell` является элементарным строительным блоком для внутренней изменяемости.

`UnsafeCell<T>` служит оболочкой для `T`, но не имеет каких-либо условий или ограничений, позволяющих избежать неопределенного поведения. Взамен имеющийся в данной структуре метод `get()` просто дает обычный указатель на содержащееся в этой структуре значение, которое можно целенаправленно задействовать только в `unsafe`-блоках. Пользователь может применить эту структуру так, чтобы не вызвать неопределенного поведения.

Чаще всего `UnsafeCell` используется не напрямую, а заключается в другой тип, обеспечивающий безопасность посредством ограниченного интерфейса, например в `Cell` или в `Mutex`. Все типы с внутренней изменяемостью, включая все рассмотренные типы, построены на основе `UnsafeCell`.

Потокобезопасность: *Send* и *Sync*

В данной главе были рассмотрены несколько типов, не являющихся *потокобезопасными*. Эти типы, в число которых входят также `Rc` и `Cell`, могут использоваться только в одном потоке. Поскольку такое ограничение необходимо во избежание неопределенного поведения, компилятор должен в этом разбираться и проводить за вас проверку, позволяя вам использовать эти типы, не применяя `unsafe`-блоки.

В языке Rust предусмотрены два специальных типажа, позволяющие отслеживать, какие типы можно безопасно использовать во взаимодействующих потоках:

- `Send` — тип относится к типажу `Send`, если его можно отправить в другой поток. Иными словами, если право собственности на значение этого типа может быть передано другому потоку. Например, `Arc<i32>` относится к `Send`, а `Rc<i32>` — нет.
- `Sync` — тип относится к типажу `Sync`, если его можно использовать совместно с другим потоком. Иными словами, тип `T` относится к `Sync` тогда и только тогда, когда совместно используемая ссылка на этот тип `&T` относится к `Send`. Например, `i32` относится к `Sync`, а `Cell<i32>` — нет. (Хотя `Cell<i32>` относится к `Send`.)

Все простые типы, такие как `i32`, `bool` и `str`, относятся одновременно как к `Send`, так и к `Sync`.

Оба этих типажа являются автотипажами, что означает их автоматическую реализацию для ваших типов на основе их полей. Структура с полями, относящимися без исключения к `Send` и к `Sync`, сама также относится к `Send` и `Sync`.

Чтобы отказаться от любого из этих вариантов, нужно добавить в тип поле, не реализующее данный типаж. С этой целью зачастую может пригодиться специальный тип `std::marker::PhantomData<T>`, рассматриваемый компилятором как `T`, за исключением того, что во время выполнения кода он фактически не существует. Это тип нулевого размера, не занимающий места в памяти.

Давайте посмотрим на следующую структуру struct:

```
use std::marker::PhantomData;
struct X {
    handle: i32,
    _not_sync: PhantomData<Cell<()>>,
}
```

В данном примере X, если его единственным полем окажется handle, будет одновременно относиться как к Send, так и к Sync. Но здесь добавлено поле PhantomData<Cell<()>> нулевого размера, обрабатываемое так же, как если бы оно было полем Cell<()>. Поскольку Cell<()> не относится к Sync, X также к нему не относится. Но структура по-прежнему относится к Send, поскольку во всех его полях реализуется Send.

Простые указатели (*const T и *mut T) не относятся ни к Send, ни к Sync, поскольку компилятор мало что знает о том, что они собой представляют.

Способ выбора любого из типажей точно такой же, как и для любого другого типажа, и для реализации типажа вашего типа следует воспользоваться блоком impl:

```
struct X {
    p: *mut i32,
}
unsafe impl Send for X {}
unsafe impl Sync for X {}
```

Обратите внимание, что для реализации этих типажей требуется указать ключевое слово unsafe, поскольку проверить за вас корректность этого кода компилятор не в состоянии. Это обещание, высказанное компилятору, в которое ему придется просто поверить.

При попытке перемещения чего-либо, отличного от Send, в другой поток, компилятор вежливо запретит вам сделать это. Вот небольшой пример, иллюстрирующий данное утверждение:

```
fn main() {
  let a = Rc::new(123);
   thread::spawn(move || { // Ошибка!
      dbg!(a);
   });
}
```

Здесь предпринята попытка отправить Rc<i32> в новый поток, но Rc<i32>, в отличие от Arc<i32>, не имеет реализации типажа Send.

При попытке скомпилировать код этого примера будет выдана ошибка примерно следующего вида:

```
error[E0277]: 'Rc<i32>' cannot be sent between threads safely
 --> src/main.rs:3:5
  |
```

❶

```
3 |     thread::spawn(move || {
  |^^^^^^^^^^^^^ 'Rc<i32>' cannot be sent between threads safely                ❷
  |
  = help: within '[closure]', the trait 'Send' is not implemented for 'Rc<i32>'  ❸
note: required because it's used within this closure                            ❹
   --> src/main.rs:3:19
  |
3 |       thread::spawn(move || {
  |                     ^^^^^^^
note: required by a bound in 'spawn'                                            ❺
```

❶ Ошибка `[E0277]`: безопасная отправка `'Rc<i32>'` между потоками невозможна.

❷ Безопасная отправка `'Rc<i32>'` между потоками невозможна.

❸ Подсказка: внутри замыкания `'[closure]'` типаж `'Send'` для `'Rc<i32>'` не реализован.

❹ Примечание: типаж нужен, поскольку он используется в данном замыкании.

❺ Примечание: требуется для привязки в `'spawn'`.

Функция `thread::spawn` требует, чтобы ее аргумент имел отношение к `Send`, а замыкание относится к `Send` только при условии, что все им захватываемое относится к этому типажу. Если мы попытаемся захватить что-либо не относящееся к `Send`, будет выдана ошибка, защищающая нас от неопределенного поведения.

Блокировка: мьютексы и *RwLock*-блокировки

Чаще всего в качестве инструмента совместного использования потоками изменяемых данных служит мьютекс, название которого происходит из сокращения словосочетания "mutual exclusion" ("взаимное исключение"). Задача мьютекса заключается в выдаче потокам гарантий на монопольный доступ к тем или иным данным путем временной блокировки других потоков, пытающихся в тот же момент времени получить доступ к тем же данным.

Концептуально у мьютекса бывает только два состояния: заблокирован и разблокирован. Когда поток блокирует разблокированный мьютекс, этот мьютекс помечается как заблокированный, и блокирующий поток может немедленно продолжить работу. Когда затем поток попытается заблокировать уже заблокированный мьютекс, эта операция будет заблокирована. Поток приостанавливается в ожидании разблокировки мьютекса. Разблокировка возможна только для заблокированного мьютекса и должна выполняться тем же потоком, который его заблокировал. Если другие потоки ожидают блокировки мьютекса, разблокировка приведет к тому, что один из этих потоков будет разбужен и сможет снова предпринять попытку блокировки мьютекса и продолжения работы.

Защита данных с помощью мьютекса является простым соглашением между всеми потоками о том, что они будут получать доступ к данным только при наличии

заблокированного ими мьютекса. Таким образом, никакие два потока не смогут одновременно получить доступ к одним и тем же данным и вызвать гонку данных.

Мьютекс в языке Rust

Стандартная библиотека Rust предоставляет мьютексы посредством конструкции `std::sync::Mutex<T>`. Это обобщенная структура для типа `T`, являющегося именно тем типом данных, который находится под защитой мьютекса. Сделав этот `T` частью мьютекса, доступ к данным можно получить только через мьютекс, что обеспечивает безопасный интерфейс, способный гарантировать соблюдение соглашения всеми потоками.

Чтобы гарантировать возможность разблокировки мьютекса только тем потоком, который его заблокировал, у мьютекса отсутствует метод `unlock()`. Взамен метод `lock()` возвращает специальный тип, называемый `MutexGuard`, являющийся защитой, предоставляющей гарантию того, что мьютекс заблокирован. Этот тип благодаря типажу `DerefMut` ведет себя как исключительная ссылка, предоставляя исключительный доступ к данным, защищаемым мьютексом. Разблокировка мьютекса осуществляется путем сброса защиты. При сбросе защиты теряется возможность доступа к данным, а принадлежащая защите реализация `Drop` разблокирует мьютекс.

Давайте рассмотрим практический пример использования мьютекса (листинг 1.4).

Листинг 1.4

```
use std::sync::Mutex;
fn main() {
    let n = Mutex::new(0);
    thread::scope(|s| {
        for _ in 0..10 {
            s.spawn(|| {
                let mut guard = n.lock().unwrap();
                for _ in 0..100 {
                    *guard += 1;
                }
            });
        }
    });
    assert_eq!(n.into_inner().unwrap(), 1000);
}
```

Здесь присутствует `Mutex<i32>`, мьютекс, защищающий целочисленное значение, и создаются десять потоков, каждый из которых увеличивает целое число на единицу сто раз. Каждый поток сначала блокирует мьютекс, чтобы получить `MutexGuard`, а затем использует эту защиту для доступа к целому числу и изменения его значения. Защита подразумеваемо сбрасывается сразу после того, как эта переменная выходит за пределы области видимости.

После того как потоки завершены, можно безопасно снять защиту с целого числа с помощью метода `into_inner()`, который берет на себя ответственность за мьютекс, гарантируя, что ничто другое больше не может ссылаться на мьютекс, и снимая необходимость в блокировке.

Несмотря на то что приращение идет с единичным шагом, поток, наблюдающий за целочисленным значением, увидит только числа, кратные 100, поскольку он может просматривать это значение только при разблокированном мьютексе. Фактически благодаря мьютексу теперь сто приращений в совокупности представляют собой единую и неделимую, т. е. атомарную, операцию.

Чтобы наглядно увидеть действие мьютекса, можно заставить каждый поток разблокировать мьютекс с секундной задержкой (листинг 1.5).

Листинг 1.5

```
use std::time::Duration;
fn main() {
    let n = Mutex::new(0);
    thread::scope(|s| {
        for _ in 0..10 {
            s.spawn(|| {
                let mut guard = n.lock().unwrap();
                for _ in 0..100 {
                    *guard += 1;
                }
                thread::sleep(Duration::from_secs(1)); // Новый код!
            });
        }
    });
    assert_eq!(n.into_inner().unwrap(), 1000);
}
```

Теперь при запуске программы будет видно, что на ее завершение уходит около 10 секунд. Каждый поток задерживается только на одну секунду, но мьютекс гарантирует, что это одновременно может быть сделано только одним потоком.

Если перед задержкой на секунду сбросить защиту, разблокировав таким образом мьютекс, вместо этого мы увидим, что задержка выполняется в параллельном режиме (листинг 1.6).

Листинг 1.6

```
fn main() {
    let n = Mutex::new(0);
    thread::scope(|s| {
        for _ in 0..10 {
            s.spawn(|| {
                let mut guard = n.lock().unwrap();
```

```
                for _ in 0..100 {
                    *guard += 1;
                }
                drop(guard); // Нововведение: сброс защиты перед задержкой!
                thread::sleep(Duration::from_secs(1));
            });
        }
    });
    assert_eq!(n.into_inner().unwrap(), 1000);
}
```

Это изменение сокращает время выполнения программы примерно до одной секунды, поскольку теперь 10 потоков могут "засыпать" на одну секунду одновременно. Тем самым демонстрируется важность наименее возможного периода блокировки мьютекса. Удержание мьютекса заблокированным дольше необходимого может полностью свести на нет все преимущества параллелизма, фактически принуждая всю программу к последовательному выполнению.

Отравление блокировок

Вызовы метода `unwrap()`, имеющиеся в приведенных ранее примерах (см. листинги 1.5–1.7), связаны с *отравлением блокировок*.

Объект `Mutex` в Rust помечается как *отравленный*, когда поток паникует, удерживая блокировку. В таком случае `Mutex` больше не будет заблокирован, но вызов его `lock`-метода приведет к выдаче ошибки `Err`, указывающей на то, что он был отравлен.

Тем самым реализуется механизм защиты от оставления данных, защищенных мьютексом, в несогласованном состоянии. В приведенном примере в случае паники со стороны потока после увеличения целого числа менее чем 100 раз мьютекс будет разблокирован, а целое число останется в том неожиданном состоянии, когда оно больше не кратно 100, что, возможно, будет вопреки предположениям, сделанным другими потоками. Тогда автоматическая маркировка мьютекса как отравленного заставит пользователя учитывать подобную возможность.

Вызов `lock()` для отравленного мьютекса по-прежнему заблокирует мьютекс. Значение `Err`, возвращаемое функцией `lock()`, содержит тип `MutexGuard`, позволяющий при необходимости скорректировать несогласованное состояние.

При всей кажущейся эффективности механизма отравления блокировок на практике восстановление из потенциально несогласованного состояния осуществляется нечасто. Большая часть кода либо игнорирует отравление, либо использует `unwrap()`, чтобы вызвать панику, если блокировка была отравлена, эффективно распространяя эту панику на всех пользователей мьютекса.

Время жизни *MutexGuard*

При всем удобстве подразумеваемого сброса защиты для разблокировки мьютекса иногда этот сброс может привести к завуалированным сюрпризам. Если присвоить защите имя

с помощью оператора `let` (как в примерах, приведенных ранее), то не трудно будет заметить, когда защита будет сброшена, поскольку локальные переменные удаляются в конце области своего определения. И все же неявный сброс защиты может привести, как показано в показанных выше примерах, к сохранению блокировки мьютекса дольше необходимого.

Защиту можно также использовать и *без* присвоения ей имени, что порой гораздо удобнее. Поскольку `MutexGuard` ведет себя как исключительная ссылка на защищенные данные, этим типом можно воспользоваться напрямую без предварительного присваивания имени защите. Например, при наличии `Mutex<Vec<i32>>` мьютекс можно заблокировать, добавить элемент в `Vec` и снова разблокировать этот же мьютекс с помощью одной-единственной инструкции:

```
list.lock().unwrap().push(1);
```

Любые временные объекты, созданные в более крупном выражении, например защита, возвращаемая функцией `lock()`, будут удаляться в конце инструкции. При всей своей очевидности и логичности, это становится причиной весьма распространенной ошибки, обычно связанной с использованием инструкций `match`, `if let` или `while let`. Рассмотрим пример, в котором допущена подобная ошибка:

```
if let Some(item) = list.lock().unwrap().pop() {
    process_item(item);
}
```

Если замысел заключался в блокировке списка, извлечении элемента, разблокировке списка *с последующей* обработкой элемента после разблокировки, то здесь допущена едва заметная и все же весьма серьезная ошибка. Временная защита не сбрасывается до конца действия всей инструкции `if let`, следовательно, блокировка на время обработки элемента удерживается безо всяких на то оснований.

Но как ни удивительно, при использовании аналогичной инструкции `if`, показанной в следующем примере, ничего подобного не происходит:

```
if list.lock().unwrap().pop() == Some(1) {
    do_something();
}
```

Здесь временная защита сбрасывается до выполнения тела инструкции `if`. Дело в том, что условие обычной инструкции `if` всегда является простым логическим значением, которое не может ничего заимствовать. И продлевать время жизни временных объектов от условия до конца инструкции нет никакого смысла. Но в случае с инструкцией `if let` все может быть иначе. Если, к примеру, вместо `pop()` использовалась бы функция `front()`, то элемент `item` был бы заимствован из списка, что потребовало бы сохранения защиты. Поскольку средство проверки заимствований является всего лишь проверкой и *не* влияет на то, когда и в каком порядке элементы удаляются, то же самое происходит и при использовании `pop()`, даже если в этом нет необходимости.

Избежать этого можно путем выноса инструкции `pop` в отдельную инструкцию `let` со сбросом защиты в конце данной инструкции перед выполнением инструкции `if let`:

```
let item = list.lock().unwrap().pop();
if let Some(item) = item {
    process_item(item);
}
```

Блокировка чтения-записи

Мьютекс отвечает только за исключительный доступ. `MutexGuard` предоставляет исключительную ссылку (`&mut T`) на защищенные данные, даже если бы требовался только просмотр этих данных, для чего вполне достаточно было бы и совместно используемой ссылки (`&T`).

Блокировка чтения-записи является несколько более сложной версией мьютекса, понимающей разницу между исключительным и совместно используемым доступом к данным, и может предоставлять и то и другое. У нее имеются три состояния: разблокировано, заблокировано одним из *писателей* (для исключительного доступа) и заблокировано любым количеством *читателей* (для совместного доступа). Обычно такая блокировка используется для данных, часто подвергаемых чтению сразу несколькими потоками, но обновляемых лишь время от времени.

Стандартная библиотека Rust предоставляет такую блокировку посредством использования типа `std::sync::RwLock<T>`, который работает аналогично обычному `Mutex`, за исключением того, что его интерфейс весьма практично разбивается на две части. Вместо одного метода `lock()` у него есть методы `read()` и `write()` для блокировки либо читателя, либо писателя. Предусмотрено два типа защиты: один для чтения и один для записи — соответственно `RwLockReadGuard` и `RwLockWriteGuard`. В первом типе защиты `Deref` реализуется только поведение в стиле совместно используемой ссылки на защищенные данные, а во втором реализуется также `DerefMut`, обеспечивая поведение в качестве исключительной ссылки.

По сути, это многопоточная версия `RefCell`, динамически отслеживающая число ссылок для обеспечения соблюдения правил заимствования.

И `Mutex<T>`, и `RwLock<T>` требуют, чтобы в `T` была реализация типажа `Send`, поскольку с их помощью можно отправлять `T` в другой поток. `RwLock<T>` дополнительно требует, чтобы в `T` также была реализация типажа `Sync`, т. к. он позволяет хранить совместно используемую ссылку (`&T`) на защищенные данные сразу нескольким потокам.

Строго говоря, для `T` может быть создана блокировка, не соответствующая данным требованиям, но ее невозможно будет совместно использовать между потоками, поскольку в самой блокировке не будет реализации `Sync`.

Стандартная библиотека Rust предоставляет только один обобщенный тип `RwLock`, но его реализация зависит от конкретной операционной системы. Между реализациями блокировки чтения-записи существует множество едва уловимых различий. В большинстве реализаций при ожидании отработки писателя происходит блокировка новых читателей, даже если уже установлена блокировка для чтения. Это делается для предотвращения *зависания писателя*, т. е. создания ситуации, при которой множество читателей коллективно не позволяют блокировке когда-либо быть снятой, не давая возможности обновлять данные ни одному из писателей.

Мьютексы на других языках

Стандартные типы `Mutex` и `RwLock` выглядят в Rust несколько иначе, чем в других языках, например в C или в C++.

Самая большая разница заключается в том, что `Mutex<T>` в Rust содержит данные, которые он защищает. Например, в C++ `std::mutex` не содержит защищаемых данных и даже не знает, что именно он защищает. Получается, что пользователь обязан помнить, какие данные защищены и каким мьютексом, и обеспечивать блокировку нужного мьютекса всякий раз, когда осуществляется доступ к "защищенным" данным. Это следует учитывать при чтении кода, использующего мьютексы, на других языках или при общении с программистами, незнакомыми с Rust. Программист, работающий на языке Rust, может говорить о "данных внутри мьютекса" или говорить что-то вроде "создать оболочку из мьютекса", что может сбить с толку тех, кто знаком с мьютексами только на других языках.

Если действительно понадобится ничего не содержащий автономный мьютекс, например для защиты какого-либо внешнего оборудования, можно будет воспользоваться записью `Mutex<()>`. Но даже в таком случае вам все же, наверное, для взаимодействия с этим оборудованием лучше будет определить тип (возможно, нулевого размера) и заключить его в мьютекс. Тогда, прежде чем появится возможность взаимодействия с оборудованием, мьютекс все равно придется заблокировать.

Ожидание: парковка и условные переменные

Если данные изменяются несколькими потоками, складывается множество ситуаций, когда этим потокам придется ждать наступления какого-либо события, чтобы в отношении данных стало истинным вполне конкретное условие. Например, если имеется мьютекс, защищающий `Vec`, то можно подождать, пока в векторе не появятся какие-либо данные.

Хотя мьютекс позволяет потокам ждать снятия его блокировки, он не предоставляет возможности ожидать наступление истинности тех или иных условий. Если бы имелся только мьютекс, пришлось бы удерживать его блокировку, чтобы многократно проверять `Vec` на наличие хоть какого-нибудь содержимого.

Парковка потоков

Один из способов ожидания уведомления от другого потока называется *парковкой потока*. Поток может *запарковаться* с переходом в спящий режим, не позволяющий ему потреблять циклы центрального процессора. Затем другой поток может *снять парковку* запаркованного потока, выведя его из состояния сна.

Парковка потоков доступна через функцию `std::thread::park()`. Для снятия парковки в отношении объекта `Thread`, представляющего поток, требующий распарковки, вызывается метод `unpark()`. Такой объект можно получить из дескриптора присоединения, возвращаемого `spawn`, или же из самого потока через вызов `std::thread::current()`.

Давайте разберем пример (листинг 1.7), в котором мьютекс служит для совместного использования очереди двумя потоками. В этом примере вновь созданный поток будет брать элементы из очереди, а основной поток будет посекундно вставлять в очередь новый элемент. Парковка потоков здесь предназначена для того, чтобы поток-потребитель находился в режиме ожидания, когда очередь опустеет.

Листинг 1.7

```
use std::collections::VecDeque;
fn main() {
    let queue = Mutex::new(VecDeque::new());
    thread::scope(|s| {
        // Поток-потребитель
        let t = s.spawn(|| loop {
            let item = queue.lock().unwrap().pop_front();
            if let Some(item) = item {
                dbg!(item);
            } else {
                thread::park();
            }
        });
        // Поток-производитель
        for i in 0.. {
            queue.lock().unwrap().push_back(i);
            t.thread().unpark();
            thread::sleep(Duration::from_secs(1));
        }
    });
}
```

В потоке-потребителе запускается бесконечный цикл, в котором элементы извлекаются из очереди для их отображения с помощью макроса `dbg`. Когда очередь пуста, этот поток останавливается, и с помощью функции `park()` переходит в спящий режим. Если парковка с него снимается, управление из `park()` возвращается, и цикл продолжается, снова извлекая элементы из очереди до тех пор, пока она не опустеет. И так далее.

Поток-производитель ежесекундно создает новое число, помещая его в очередь. При каждом добавлении элемента он вызывает метод `unpark()` объекта `Thread`, ссылающегося на поток-потребитель для его распарковки. И тогда поток-потребитель просыпается для обработки нового элемента.

Здесь важно заметить, что данная программа все равно будет теоретически правильной, хотя и неэффективной, даже если парковку удалить. Дело в том, что вызов `park()` еще не гарантирует того, что возвращение из него состоится только из-за соответствующего вызова `unpark()`. Хотя весьма редко, но все же могут быть и *ложные пробуждения*. Наша программа прекрасно справляется с этой проблемой, поскольку поток-потребитель заблокирует очередь, увидит, что она пуста, и тут же разблокирует ее и снова запаркуется.

Важное свойство парковки потоков состоит в том, что вызов `unpark()` *еще до того*, как сам поток запаркуется, не теряется. Запрос на распарковку все же записывается, и в следующий раз, когда поток попытается запарковаться, этот запрос удаляется и поток продолжает работу, фактически не переходя в спящий режим. Чтобы понять,

почему это так важно для корректной работы, давайте рассмотрим возможный порядок действий, выполняемых обоими потоками:

1. Поток-потребитель (назовем его *C*) блокирует очередь.
2. *C* пытается извлечь элемент из очереди, но она пуста, в результате чего он получает значение `None`.
3. *C* снимает блокировку с очереди.
4. Поток-производитель, назовем его *P*, блокирует очередь.
5. *P* помещает в очередь новый элемент.
6. *P* снова снимает с очереди блокировку.
7. *P* вызывает `unpark()`, чтобы уведомить *C* о наличии новых элементов.
8. *C* вызывает `park()`, чтобы перейти в режим сна и дождаться появления новых элементов.

Хотя между снятием с очереди блокировки в пункте 3 и парковкой в пункте 8 проходит, скорее всего, весьма короткий интервал времени, действия с 4 по 7 потенциально могут произойти в этот момент еще до того, как поток запаркуется. Если `unpark()` ничего не сделает в том случае, если поток не был запаркован, уведомление будет утрачено, и тогда поток-потребитель все равно будет находиться в режиме ожидания, даже если в очереди уже будет новый элемент. Благодаря тому, что запросы на распарковку сохраняются и для будущего вызова `park()`, беспокоиться об этом не придется.

Но все же запросы на распарковку не суммируются. Двойной вызов `unpark()` с последующим двойным вызовом `park()` все равно приводит к переходу потока в спящий режим. При первом вызове `park()` запрос на парковку удаляется с немедленным возвратом управления, а при втором вызове, как обычно, происходит переход в спящий режим.

Иначе говоря, для примера из листинга 1.7 важно, чтобы поток парковался только в том случае, если обнаружилось, что очередь пуста, а не после обработки каждого элемента. Хотя из-за весьма продолжительного (односекундного) периода сна в данном примере это крайне маловероятно, несколько вызовов `unpark()` могут пробудить только один вызов `park()`.

К сожалению, получается, что при вызове `unpark()` сразу же после возврата управления из `park()`, но еще до того, как очередь будет заблокирована и опустошена, в вызове `unpark()` не будет никакой надобности, но он все равно приведет к немедленному возврату управления из следующего вызова `park()`. В результате будет заблокирована и еще раз разблокирована пустая очередь. Хотя на корректность работы программы это никак не повлияет, но все же снизится ее эффективность и производительность.

Рассматриваемый механизм неплохо справляется с простыми ситуациями, вроде тех, что складываются в нашем примере, но тут же становится непригодным при усложнении ситуации. К примеру, при наличии сразу нескольких потоков-потре-

бителей, забирающих элементы из одной и той же очереди, поток-производитель не смог бы узнать, какой из потребителей действительно ожидает наполнения очереди и своего пробуждения. Производитель должен будет точно знать, когда и какого состояния ждет потребитель.

Условные переменные

Более востребованный вариант организации ожидания событий, происходящих с данными, защищенными мьютексом, — использование условных переменных. С ними связаны две основные операции: *ожидания* (wait) и *уведомления* (notify). Потоки могут находиться в ожидании условной переменной для получения возможности своего пробуждения в случае уведомления другим потоком той же самой переменной. Одну и ту же условную переменную могут ожидать сразу несколько потоков, а уведомления могут отправляться либо одному, либо всем ожидающем потокам.

Следовательно, можно создать условную переменную для конкретных интересующих нас событий или условий, например ждать выполнения условия, когда очередь станет непустой.

Любой поток, работа которого становится причиной возникновения данного события или создания условия, уведомляет переменную условия, находясь в неведении о том, какие потоки и в каком количестве заинтересованы в этом уведомлении.

Чтобы не создавать проблему отсутствия уведомлений в короткий интервал времени между разблокировкой мьютекса и ожиданием условной переменной, условными переменными предоставляется способ *атомарной* разблокировки мьютекса и начала ожидания, что полностью исключает возможность утраты уведомления.

Стандартная библиотека Rust предоставляет условную переменную в виде типа `std::sync::Condvar`. В его методе ожидания `wait` задействован `MutexGuard`, подтверждающий блокировку мьютекса. Сначала осуществляется разблокировка мьютекса и переход в режим сна. Затем, когда происходит пробуждение, им повторно осуществляется блокировка мьютекса и возвращается новый `MutexGuard` (подтверждающий новую блокировку мьютекса).

У `std::sync::Condvar` есть две функции уведомления: `notify_one` для пробуждения только одного ожидающего потока (если таковой имеется) и `notify_all` для пробуждения всех ожидающих потоков.

Давайте внесем изменения в листинг 1.7, чтобы вместо парковки потоков воспользоваться условной переменной `Condvar` (листинг 1.8).

Листинг 1.8

```
use std::sync::Condvar;
let queue = Mutex::new(VecDeque::new());
let not_empty = Condvar::new();
thread::scope(|s| {
```

```
    s.spawn(|| {
        loop {
            let mut q = queue.lock().unwrap();
            let item = loop {
                if let Some(item) = q.pop_front() {
                    break item;
                } else {
                    q = not_empty.wait(q).unwrap();
                }
            };
            drop(q);
            dbg!(item);
        }
    });
    for i in 0.. {
        queue.lock().unwrap().push_back(i);
        not_empty.notify_one();
        thread::sleep(Duration::from_secs(1));
    }
});
```

Здесь пришлось внести изменения, в результате которых:

- теперь имеется не только `Mutex`, содержащий очередь, но и `Condvar` для передачи условия "не пусто";
- больше не нужно знать, какой поток пробудить, поэтому возвращаемое от `spawn` значение теперь не сохраняется. Взамен потребитель уведомляется через переменную условия с помощью метода `notify_one`;
- разблокировка, ожидание и повторная блокировка выполняются методом `wait`. Не обошлось и без незначительной реструктуризации потока управления, открывающей возможность передачи защиты методу `wait`, но при этом позволяющей все же удалять ее перед обработкой элемента.

Теперь можно создавать сколько угодно потоков-потребителей и даже порождать их значительно позже, абсолютно ничего не меняя. Условная переменная берет на себя доставку уведомлений в любой, заинтересованный в них поток.

При наличии более сложной системы с потоками, интересующимися разными условиями, можно определить `Condvar` для каждого условия. Например, можно задать одну условную переменную для указания того, что очередь не пуста, а другую — для указания того, что она пуста. Затем каждый поток будет ждать того условия, которое имеет отношение к тому, чем он занимается.

Обычно `Condvar` применяется только вместе с одним `Mutex`. Если два потока попытаются одновременно ожидать условной переменной, используя два разных мьютекса, это может вызвать панику.

Недостаток `Condvar` заключается в том, что этот тип работает только совместно с `Mutex`, но для большинства случаев данное обстоятельство является вполне нормальным, поскольку именно эта связка и требуется нам для защиты данных.

Как у `thread::park()`, так и у `Condvar::wait()` имеются также варианты с ограничением по времени: `thread::park_timeout()` и `Condvar::wait_timeout()`. В качестве дополнительного аргумента они принимают `Duration`, т. е. время, по истечении которого метод должен прекратить ожидание уведомления и безоговорочно пробудить поток.

Резюме

- В одной и той же программе могут выполняться сразу несколько потоков, порождаемых в любое время.
- При завершении работы основного потока завершается выполнение всей программы.
- Под гонкой данных понимается неопределенное поведение, полностью предотвращаемое (в безопасном коде) имеющейся в Rust системой типов.
- Данные с реализацией типажа `Send` могут отправляться другим потокам, а данные с реализацией типажа `Sync` могут совместно использоваться сразу несколькими потоками.
- Обычные потоки могут работать на протяжении жизни всей программы, вследствие чего могут заимствовать только данные с параметром времени жизни `'static`, т. е. те участки памяти, которые были выделены под статические данные или превратившиеся в утечки памяти.
- Подсчетом ссылок (`Arc`) можно воспользоваться для совместного владения, чтобы гарантировать, что данные будут жить до тех пор, пока они используются хотя бы одним потоком.
- Потоки с заданной областью действия пригодятся для ограничения времени их жизни и разрешения им заимствовать данные, не имеющие параметра времени жизни `'static`, например локальные переменные.
- `&T` является *совместно используемой* ссылкой, а `&mut T` — исключительной ссылкой. Обычные типы не допускают внесения изменений через совместно используемую ссылку.
- У некоторых типов благодаря наличию `UnsafeCell` имеется внутренняя изменяемость, что позволяет вносить изменения через совместно используемые ссылки.
- `Cell` и `RefCell` — это стандартные типы для однопоточной внутренней изменяемости. А их многопоточными эквивалентами являются атомарные типы, `Mutex` и `RwLock`.
- `Cell` и атомарные типы позволяют заменять значение только целиком, а `RefCell`, `Mutex` и `RwLock` обеспечивают возможность изменять значение непосредственным образом, динамически применяя правила доступа.
- Парковка потоков может стать удобным способом ожидания конкретного состояния.
- Когда условие касается данных, защищенных мьютексом, удобнее и эффективнее парковки потоков может оказаться применение `Condvar`.

ГЛАВА 2

Атомарность

Слово "атомарный" происходит от греческого слова ατομος, означающего *неделимое*, то, что нельзя разрезать на более мелкие части. В компьютерной сфере это слово употребляют для описания неделимой операции, которая может быть либо полностью завершенной, либо еще не произошедшей.

Как уже упоминалось в *разделе "Заимствования и гонка данных" главы 1*, несколько потоков, одновременно читающих и изменяющих значение одной и той же переменной, обычно становятся причиной возникновения неопределенного поведения. Но безопасное чтение и изменение значения одной и той же переменной различными потоками обеспечивается путем проведения атомарных операций. В силу своей неделимости такая операция проводится целиком либо до, либо после другой операции, позволяя тем самым избежать проявления неопределенного поведения. Далее в *главе 7* будет показано, как все это работает на аппаратном уровне.

Атомарные операции являются основными строительными блоками всего, что связано с работой нескольких потоков. С помощью атомарных операций реализуются все базовые элементы конкурентных вычислений, в числе которых мьютексы и условные переменные.

Атомарные операции доступны в Rust в виде методов стандартных атомарных типов, присутствующих в `std::sync::atomic`. Все их имена начинаются с `Atomic`, например `AtomicI32` или `AtomicUsize`. Доступность конкретных типов зависит от аппаратной архитектуры, а иногда и от операционной системы, но в любом случае все атомарные типы вплоть до размера указателя предоставляются почти всеми платформами.

В отличие от большинства типов, атомарные операции допускают внесение изменений через совместно используемую ссылку (например, `&AtomicU8`). Такая возможность предоставляется благодаря внутренней изменяемости, рассмотренной в *разделе "Внутренняя изменяемость" главы 1*.

В каждом из доступных атомарных типов имеется один и тот же интерфейс с методами хранения и загрузки, методами для атомарных операций "выборки и изменения", а также некоторыми более развитыми методами "сравнения и обмена". Более подробно они будут рассмотрены далее в этой главе.

Прежде чем углубляться в суть различных атомарных операций, нужно вкратце рассмотреть концепции под названием *упорядочение памяти*.

Каждая атомарная операция принимает аргумент типа `std::sync::atomic::Ordering`, определяющий гарантии относительного порядка проведения операций.

Самый простой вариант — `Relaxed`, при котором гарантируется всего лишь непротиворечивость значения одной атомарной переменной, но не дается никаких обещаний насчет относительного порядка проведения операций между различными переменными.

Это означает, что два потока могут видеть, что операции с разными переменными выполняются в разном порядке. Например, если один поток сначала ведет запись в одну переменную, а сразу же после этого во вторую переменную, другой поток может увидеть, что это происходит в обратном порядке.

В данной главе будут рассмотрены только те случаи использования `Relaxed`, где это не создает никаких проблем, и мы просто будем применять `Relaxed` везде, не вдаваясь в подробности. Все особенности упорядочения памяти и другие доступные способы такого упорядочения будут рассмотрены в *главе 3*.

Атомарные операции загрузки и сохранения: *load* и *store*

Сначала рассмотрим две самые простые атомарные операции: `load` и `store`. Сигнатуры их функций (на примере `AtomicI32`) имеют следующий вид:

```
impl AtomicI32 {
    pub fn load(&self, ordering: Ordering) -> i32;
    pub fn store(&self, value: i32, ordering: Ordering);
}
```

Метод `load` атомарно загружает значение, хранящееся в атомарной переменной, а метод `store` атомарно сохраняет в ней новое значение. Заметьте, что метод `store`, несмотря на то, что он предназначен для изменения значения, принимает совместно используемую (`&T`), а не исключительную ссылку (`&mut T`).

Давайте рассмотрим несколько реалистичных вариантов использования этих двух методов.

Пример: Флаг остановки

В первом примере (листинг 2.1) в качестве *флага остановки* используется `AtomicBool`. Такой флаг служит для информирования других потоков о прекращении выполнения кода.

Листинг 2.1

```
use std::sync::atomic::AtomicBool;
use std::sync::atomic::Ordering::Relaxed;
fn main() {
    static STOP: AtomicBool = AtomicBool::new(false);
    // Порождение потока для выполнения работы.
    let background_thread = thread::spawn(|| {
```

```
        while !STOP.load(Relaxed) {
            some_work();
        }
    });
    // Использование основного потока для отслеживания пользовательского ввода.
    for line in std::io::stdin().lines() {
        match line.unwrap().as_str() {
            "help" => println!("commands: help, stop"),
            "stop" => break,
            cmd => println!("unknown command: {cmd:?}"),
        }
    }
    // Информирование фонового потока о необходимости остановки.
    STOP.store(true, Relaxed);
    // Ожидание завершения фонового потока.
    background_thread.join().unwrap();
}
```

В данном примере фоновый поток многократно запускает `some_work()`, а основной поток позволяет пользователю вводить команды для взаимодействия с программой. Единственная полезная команда, останавливающая программу, — `stop`.

Чтобы остановить фоновый поток, задействована атомарная логическая переменная `STOP`, передающая соответствующее условие фоновому потоку. Когда поток переднего плана считывает команду `stop`, он устанавливает флаг в значение `true`, которое проверяется фоновым потоком перед каждой новой итерацией. Основной поток ждет, пока фоновый поток завершит свою текущую итерацию, используя метод `join`.

Данное простое решение великолепно работает при условии, что флаг регулярно проверяется фоновым потоком. Если этот поток застрянет в `some_work()` на долгое время, то может возникнуть неприемлемо длительная задержка между выполнением команды `stop` и завершением работы программы.

Пример: Отчет о ходе выполнения задачи

В следующем примере (листинг 2.2) в фоновом потоке ведется поочередная обработка 100 элементов, в то время как основной поток регулярно предоставляет пользователю информацию о ходе выполнения задачи:

Листинг 2.2

```
use std::sync::atomic::AtomicUsize;
fn main() {
    let num_done = AtomicUsize::new(0);
    thread::scope(|s| {
        // Фоновый поток для обработки всех 100 элементов.
```

```
        s.spawn(|| {
            for i in 0..100 {
                process_item(i); // На это, видимо, уйдет какое-то время.
                num_done.store(i + 1, Relaxed);
            }
        });
        // В основном потоке ежесекундно показываются обновления состояния.
        loop {
            let n = num_done.load(Relaxed);
            if n == 100 { break; }
            println!("Working.. {n}/100 done");
            thread::sleep(Duration::from_secs(1));
        }
    });
    println!("Done!");
}
```

На этот раз используется поток с заданной областью действия (*"Потоки с областью действия"*, *глава 1*), который автоматически присоединяет поток, а также позволяет заимствовать локальные переменные.

При каждом завершении обработки элемента фоновым потоком он сохраняет число обработанных элементов в `AtomicUsize`. В то же самое время, примерно раз в секунду, основной поток показывает это число пользователю, чтобы проинформировать его о ходе работы. Увидев завершение обработки всех 100 элементов, основной поток выходит из области действия, к которой подразумеваемо присоединен фоновый поток, и сообщает пользователю, что задание выполнено.

Синхронизация

После обработки последнего элемента главному потоку, чтобы разобраться в ситуации, может потребоваться чуть ли не целая секунда, создавая абсолютно ненужную задержку в конце работы программы. Проблема может быть решена за счет парковки потоков (*"Парковка потоков"*, *глава 1*), позволяющей вывести основной поток из спящего режима при каждом появлении новой информации, которая может его заинтересовать.

Рассмотрим пример из листинга 2.2, в котором теперь `thread::sleep` заменим на `thread::park_timeout`: (листинг 2.3).

Листинг 2.3

```
fn main() {
    let num_done = AtomicUsize::new(0);
    let main_thread = thread::current();
    thread::scope(|s| {
        // Фоновый поток для обработки всех 100 элементов.
```

```
        s.spawn(|| {
            for i in 0..100 {
                process_item(i); // На это, видимо, уйдет какое-то время.
                num_done.store(i + 1, Relaxed);
                main_thread.unpark(); // Пробуждение основного потока.
            }
        });
        // Основной поток, показывающий обновления состояния.
        loop {
            let n = num_done.load(Relaxed);
            if n == 100 { break; }
            println!("Working.. {n}/100 done");
            thread::park_timeout(Duration::from_secs(1));
        }
    });
    println!("Done!");
}
```

Изменений немного. Благодаря вызову `thread::current()` был получен дескриптор основного потока, который теперь используется фоновым потоком для распарковки основного потока после каждого обновления состояния. Теперь вместо `Sleep` в основном потоке присутствует `park_timeout`, позволяющий прервать парковку.

Теперь пользователь немедленно получает сообщение о любых обновлениях состояния, при этом сообщение о последнем обновлении повторяется каждую секунду, показывая, что программа все еще работает.

Пример: Отложенная инициализация

Прежде чем перейти к более сложным атомарным операциям, давайте рассмотрим еще один пример, касающийся *отложенной (или ленивой) инициализации*.

Представим, что есть некое значение `x`, считываемое из файла, получаемое из операционной системы или вычисляемое каким-либо другим образом, которое, как ожидается, будет постоянным на все время работы программы. Может быть, `x` — это версия операционной системы, или общий объем памяти, или 400-е тау-число. Сейчас это не имеет особого значения.

Поскольку изменение этого значения не ожидается, оно может быть запрошено или вычислено однократно по мере надобности с запоминанием полученного результата. Первый поток, которому оно понадобится, должен будет вычислить значение, но он может сохранить его в атомарном `static`-типе, чтобы сделать его доступным для всех потоков, включая самого себя, если оно снова понадобится ему через некоторое время.

Давайте рассмотрим соответствующий пример (листинг 2.4). Чтобы ничего не усложнять, будем считать, что значение `x` никогда не равно нулю. В этом случае

можно воспользоваться нулем в качестве заполнителя перед вычислением данного значения.

Листинг 2.4

```
use std::sync::atomic::AtomicU64;
fn get_x() -> u64 {
    static X: AtomicU64 = AtomicU64::new(0);
    let mut x = X.load(Relaxed);
    if x == 0 {
        x = calculate_x();
        X.store(x, Relaxed);
    }
    x
}
```

Первый поток, вызвавший `get_x()`, проверит статическое значение `X` и увидит, что оно все еще равно нулю, после чего вычислит его новое значение и сохранит результат опять в виде статического значения, чтобы сделать его доступным для будущего использования. При любом последующем вызове `get_x()` будет видно, что статическое значения не равно нулю, и произойдет его немедленный возврат без повторного вычисления.

Но если вызов `get_x()` состоится во втором потоке, а первый поток все еще будет вычислять `x`, второй поток также увидит нуль и вычислит `x` параллельно первому потоку. В конечном итоге один из потоков перепишет результат другого потока, в зависимости от того, какой из них завершит свою работу первым. Это называется *гонкой*. Но не *гонкой данных*, приводящей к неопределенному поведению и невозможной в Rust без `unsafe`, но все же гонкой с непредсказуемым победителем.

Поскольку ожидается, что `x` будет иметь постоянное значение, неважно, кто именно выиграет гонку, ведь результат в любом случае будет одним и тем же. В зависимости от того, сколько времени, согласно ожиданиям, потребуется методу `calculate_x()`, данная стратегия может быть как весьма удачной, так и неприемлемой.

Если ожидается, что метод `calculate_x()` будет выполняться слишком долго, то во избежание напрасной траты времени центрального процессора лучше сделать так, чтобы потоки подождали завершения инициализации `X` первым потоком. Реализовать соответствующий замысел можно с помощью условной переменной или парковки потоков (*"Ожидание: парковка и условные переменные" глава 1*), но для небольшого примера это превратится в существенное усложнение. Нужная именно в таком случае функциональность предоставляется стандартной библиотекой Rust посредством методов `std::sync::Once` и `std::sync::OnceLock`, поэтому заниматься самостоятельной реализацией подобных функций нет ни малейшей необходимости.

Операции выборки и изменения

Рассмотрев несколько вариантов базовых операций загрузки и сохранения, `load` и `store`, давайте перейдем к более интересным операциям *выборки* и *изменения*. Эти операции изменяют значение атомарной переменной, но в рамках той же самой атомарной операции выполняют загрузку (выборку) исходного значения.

Наиболее распространенными считаются операции `fetch_add` и `fetch_sub`, выполняющие соответственно сложение и вычитание. К другим доступным операциям относятся, в частности, побитовые операции `fetch_or` и `fetch_and`, а также операции `fetch_max` и `fetch_min`, которыми можно воспользоваться для сохранения текущего максимума или минимума.

Сигнатуры их функций (на примере `AtomicI32`) имеют следующий вид:

```
impl AtomicI32 {
    pub fn fetch_add(&self, v: i32, ordering: Ordering) -> i32;
    pub fn fetch_sub(&self, v: i32, ordering: Ordering) -> i32;
    pub fn fetch_or(&self, v: i32, ordering: Ordering) -> i32;
    pub fn fetch_and(&self, v: i32, ordering: Ordering) -> i32;
    pub fn fetch_nand(&self, v: i32, ordering: Ordering) -> i32;
    pub fn fetch_xor(&self, v: i32, ordering: Ordering) -> i32;
    pub fn fetch_max(&self, v: i32, ordering: Ordering) -> i32;
    pub fn fetch_min(&self, v: i32, ordering: Ordering) -> i32;
    pub fn swap(&self, v: i32, ordering: Ordering) -> i32; // "fetch_store"
}
```

Из общего ряда выбивается только операция простого сохранения нового значения независимо от значения, имевшегося до него. Вместо `fetch_store` она называется `swap`.

Вот как выглядит краткая демонстрация возвращения значения перед операцией `fetch_add`:

```
use std::sync::atomic::AtomicI32;
let a = AtomicI32::new(100);
let b = a.fetch_add(23, Relaxed);
let c = a.load(Relaxed);
assert_eq!(b, 100);
assert_eq!(c, 123);
```

Операция `fetch_add` увеличила значение `a` со 100 до 123, но вернула нам старое значение 100. Любая следующая операция увидит значение 123.

Возвращаемое значение этими операциями актуально не всегда. Если нужно применить операцию только к атомарному значению, но само ее значение не интересует, вполне нормально будет просто проигнорировать возвращаемое значение.

Важно помнить, что `fetch_add` и `fetch_sub` реализуют поведение переноса при переполнении. Увеличение значения сверх максимально представляемой величины приведет к получению минимального представляемого значения. В этом будет от-

личие от поведения операторов "плюс" и "минус" для обычных целых чисел, которые в режиме отладки вызывают панику при переполнении.

Как выполняется атомарное сложение с проверкой переполнения, будет показано далее в *разделе "Операции сравнения и обмена"*.

Но сначала давайте посмотрим на некоторые реальные случаи применения ранее рассмотренных методов.

Пример: Отчет о ходе выполнения задачи из нескольких потоков

В предыдущем *разделе "Пример: Отчет о ходе выполнения задачи"* для отчета о ходе выполнения задачи фоновым потоком использовался тип `AtomicUsize` (см. листинг 2.2). Если бы выполнение задачи было разбито, к примеру, на четыре потока, каждый из которых обрабатывал по 25 элементов, потребовалось бы знать о ходе выполнения задачи всеми четырьмя потоками.

Можно было бы воспользоваться отдельными объектами `AtomicUsize` для каждого потока, загрузить их все в основной поток и суммировать их значения, но проще посредством одного `AtomicUsize` отслеживать общее число обработанных элементов во всех потоках.

Метод `store` здесь уже не подойдет, поскольку это приведет к перезаписи показателей хода выполнения задачи от других потоков. Взамен можно воспользоваться операцией атомарного добавления для увеличения значения счетчика после каждого обработанного элемента.

Давайте обновим пример из предыдущего *раздела "Пример: Отчет о ходе выполнения задачи"* (см. листинг 2.2), чтобы разбить работу на четыре потока (листинг 2.5).

Листинг 2.5

```
fn main() {
    let num_done = &AtomicUsize::new(0);
    thread::scope(|s| {
        // Четыре фоновых потока для обработки всех 100 элементов по 25 на поток.
        for t in 0..4 {
            s.spawn(move || {
                for i in 0..25 {
                    process_item(t * 25 + i); // На это, видимо, уйдет какое-то время.
                    num_done.fetch_add(1, Relaxed);
                }
            });
        }
        // В основном потоке ежесекундно показываются обновления состояния.
        loop {
            let n = num_done.load(Relaxed);
```

```
            if n == 100 { break; }
            println!("Working.. {n}/100 done");
            thread::sleep(Duration::from_secs(1));
        }
    });
    println!("Done!");
}
```

Кое-что изменилось. Самое главное, теперь вместо одного порождаются четыре фоновых потока, а вместо `store` для изменения значения атомарной переменной `num_done` используется `fetch_add`.

Из менее заметного: теперь для фоновых потоков применено замыкание `move`, а `num_done` здесь является ссылкой. Это связано не с наличием `fetch_add`, а с тем, что в цикле порождаются четыре потока. Это замыкание захватывает `t`, чтобы узнать, с каким из четырех потоков приходится работать и, следовательно, с какой позиции следует начать, с 0, 25, 50 или 75. Без ключевого слова `move` замыкание попыталось бы захватить `t` по ссылке, но это недопустимо, поскольку `t` существует лишь на время выполнения цикла.

`Move`-замыкание перемещает (или копирует), а не заимствует захваченное, передавая копию `t`. Поскольку оно также захватывает `num_done`, эта переменная была заменена на ссылку, т. к. нам по-прежнему требуется позаимствовать тот же самый тип `AtomicUsize`. Обратите внимание, что в атомарных типах не реализуется свойство `Copy`, поэтому при попытке перемещения одного из них в несколько потоков была бы выдана ошибка.

Если не вдаваться в тонкости захвата, присущего замыканиям, изменение в использовании `fetch_add` здесь очень простое. Нам неизвестно, в каком именно порядке потоки будут увеличивать значение `num_done`, но поскольку сложение является атомарным, беспокоиться абсолютно не о чем и можно быть уверенным, что после завершения работы всех потоков значение будет ровно 100.

Пример: Статистика

В продолжение развития идеи получения с помощью атомарности сообщения о том, что делают другие потоки, давайте расширим наш пример, чтобы собирать статистику о времени, необходимом для обработки элемента, и сообщать о полученных результатах.

Добавим к `num_done` еще две атомарные переменные, `total_time` и `max_time`, позволяющие отслеживать время, затрачиваемое на обработку элементов, и вычислять среднее и пиковое время обработки (листинг 2.6).

Листинг 2.6

```
fn main() {
    let num_done = &AtomicUsize::new(0);
    let total_time = &AtomicU64::new(0);
```

```
    let max_time = &AtomicU64::new(0);
    thread::scope(|s| {
        // Четыре фоновых потока для обработки всех 100 элементов по 25 на поток.
        for t in 0..4 {
            s.spawn(move || {
                for i in 0..25 {
                    let start = Instant::now();
                    process_item(t * 25 + i); // На это, видимо, уйдет какое-то время.
                    let time_taken = start.elapsed().as_micros() as u64;
                    num_done.fetch_add(1, Relaxed);
                    total_time.fetch_add(time_taken, Relaxed);
                    max_time.fetch_max(time_taken, Relaxed);
                }
            });
        }
        // В основном потоке ежесекундно показываются обновления состояния.
        loop {
            let total_time = Duration::from_micros(total_time.load(Relaxed));
            let max_time = Duration::from_micros(max_time.load(Relaxed));
            let n = num_done.load(Relaxed);
            if n == 100 { break; }
            if n == 0 {
                println!("Working.. nothing done yet.");
            } else {
                println!(
                    "Working.. {n}/100 done, {:?} average, {:?} peak",
                    total_time / n as u32,
                    max_time,
                );
            }
            thread::sleep(Duration::from_secs(1));
        }
    });
    println!("Done!");
}
```

Теперь для измерения времени в фоновых потоках, затрачиваемого на `process_item()`, используются `Instant::now()` и `Instant::elapsed()`. Операция атомарного сложения предназначена для добавления числа микросекунд к `total_time`, а операция атомарного максимума — для отслеживания самого высокого показателя в `max_time`.

Для получения среднего времени обработки, которое затем сообщается вместе с пиковым временем из `max_time`, основной поток делит общее время на число обработанных элементов.

Поскольку обновление трех атомарных переменных происходит по-отдельности, основной поток может загрузить значения после того, как поток увеличил `num_done`, но до того, как он обновил `total_time`, что приведет к занижению среднего значения.

Может случиться и менее вероятное событие: поскольку для порядка доступа к памяти указан режим `Relaxed`, никаких гарантий относительно порядка проведения операций, видимых из другого потока, не дается, и этот поток может даже в какой-то момент увидеть новое обновленное значение `total_time`, все еще оперируя старым значением `num_done`, что приведет к переоценке среднего времени обработки.

Но ничто из этих крайних случаев большой проблемы в нашем примере не составляет. Худшее, что может случиться, — это то, что пользователю на короткое время будет сообщено неточное среднее значение.

Если этого требуется избежать, все три статистических показателя следует поместить в `Mutex`. А затем при обновлении трех значений, которые сами по себе больше могут не быть атомарными, можно будет ненадолго заблокировать мьютекс. За счет блокировки и разблокировки мьютекса и, возможно, временной блокировки потоков фактически все три обновления станут одной атомарной операцией.

Пример: Предоставление идентификатора

Давайте перейдем к сценарию, где действительно понадобится значение, возвращаемое из `fetch_add`.

Предположим, нужна некая функция `allocate_new_id()`, которая при каждом вызове выдает новое уникальное число. Этими числами можно воспользоваться для обозначения в программе задач или чего-то еще, требующего однозначной не слишком громоздкой идентификации, которую можно будет легко сохранять и передавать между потоками, например, в виде целого числа.

С `fetch_add` такую функцию легко реализовать:

```
use std::sync::atomic::AtomicU32;
fn allocate_new_id() -> u32 {
    static NEXT_ID: AtomicU32 = AtomicU32::new(0);
    NEXT_ID.fetch_add(1, Relaxed)
}
```

Здесь просто выявляется *следующее* выдаваемое число увеличения значения при каждой загрузке. Первый вызывающий код получит 0, второй — 1 и т. д.

Единственной проблемой остается перенос при переполнении. 4 294 967 296-й вызов приведет к переполнению 32-разрядного целого числа, и при следующем вызове снова будет возвращено число 0.

Будет ли данная проблема критичной, зависит от конкретного сценария, и здесь нужно ответить на вопрос: какова вероятность того, что функция будет вызываться столько раз и каким будет наихудший исход от утраты уникальности? Несмотря на то что показанное здесь число может показаться гигантским, современные компьютеры могут легко выполнить нашу функцию именно столько раз буквально за считанные секунды. Если от уникальности этих чисел зависит безопасность использования памяти, то рассмотренную реализацию следует признать неприемлемой.

Для решения проблемы можно попробовать вызвать панику функции при чрезмерном числе ее вызовов, например:

```
// Это проблемная версия.
fn allocate_new_id() -> u32 {
    static NEXT_ID: AtomicU32 = AtomicU32::new(0);
    let id = NEXT_ID.fetch_add(1, Relaxed);
    assert!(id < 1000, "too many IDs!");
    id
}
```

Теперь инструкция `assert` будет паниковать после тысячи вызовов. Но это произойдет уже после того, как выполнится операция атомарного добавления, следовательно, значение `NEXT_ID` при выдаче паники уже будет увеличено до 1001. Если затем вызов функции состоится в другом потоке, то до выдачи паники значение будет увеличено до 1002, и т. д. Пусть на это уйдет гораздо больше времени, но после выдачи 4 294 966 296 паник придется столкнуться с той же проблемой обнуления значения `NEXT_ID` в результате переполнения.

Есть три самых распространенных решения данной проблемы. Первое — не паниковать, а полностью прервать процесс при переполнении. Прервать весь процесс позволит функция `std::process::abort`, исключив тем самым возможность продолжения вызова нашей функции. Хотя прерывание процесса может занять некоторое крайне незначительное время, в течение которого функция все еще может быть вызвана другими потоками, вероятность того, что это произойдет миллиарды раз, прежде чем программа действительно будет прервана, близка к нулю.

Собственно, именно так и реализована проверка переполнения в `Arc::clone()` из стандартной библиотеки на тот случай, если каким-то образом удастся провести клонирование `isize::MAX` раз. На 64-разрядном компьютере это заняло бы сотни лет, но если `isize` составляет всего 32 бита, то эффект переполнения может быть получен за считанные секунды.

Второе решение, позволяющее справиться с переполнением, заключается в использовании функции `fetch_sub`, позволяющей перед выдачей паники уменьшить значение счетчика:

```
fn allocate_new_id() -> u32 {
    static NEXT_ID: AtomicU32 = AtomicU32::new(0);
    let id = NEXT_ID.fetch_add(1, Relaxed);
    if id >= 1000 {
        NEXT_ID.fetch_sub(1, Relaxed);
        panic!("too many IDs!");
    }
    id
}
```

При одновременном выполнении функции сразу несколькими потоками счетчик и тут может кратковременно быть увеличен до значения, превышающего 1000, но данное увеличение ограничено числом имеющихся активных потоков. Резонно

предположить, что миллиарда активных потоков в одно и то же время никогда не будет, особенно одновременно выполняющих одну и ту же функцию в тот самый короткий момент времени между вызовами `fetch_add` и `fetch_sub`.

Именно так решена обработка переполнения числа запущенных потоков в реализации функции `thread::scope` из стандартной библиотеки.

Третье решение, позволяющее справиться с переполнением, пожалуй, единственное по-настоящему правильное, поскольку оно вообще предотвращает выполнение сложения при угрозе переполнения. Но реализовать его с помощью атомарных операций, показанных до сих пор, невозможно. Для этого понадобятся операции сравнения и обмена, рассматриваемые далее.

Операции сравнения и обмена

Самая совершенная и гибкая операция из разряда атомарных — операция *сравнения и обмена*. Ее суть заключается в проверке равенства атомарного значения заданной величине и в замене его новым значением только в случае равенства, причем все в рамках проведения атомарной операции. Она возвращает предыдущее значение и сообщает, состоялась его замена или нет.

Ее сигнатура несколько сложнее рассмотренных до сих пор. На примере `AtomicI32` она выглядит так:

```
impl AtomicI32 {
    pub fn compare_exchange(
        &self,
        expected: i32,
        new: i32,
        success_order: Ordering,
        failure_order: Ordering
    ) -> Result<i32, i32>;
}
```

Если сейчас проигнорировать упорядочение памяти, то все это, по сути, идентично следующей реализации, за исключением того, что все ранее показанное происходит в рамках единой и неделимой атомарной операции (листинг 2.7).

Листинг 2.7

```
impl AtomicI32 {
    pub fn compare_exchange(&self, expected: i32, new: i32) -> Result<i32, i32> {
        // В реальности и загрузка, и сравнение, и сохранение
        // происходят в рамках единой атомарной операции.
        let v = self.load();
        if v == expected {
            // Значение соответствует ожиданиям.
            // Его следует заменить и отрапортовать об успехе.
```

```
            self.store(new);
            Ok(v)
        } else {
            // Значение не соответствует ожиданиям.
            // Оставляем его как есть и рапортуем о провале.
            Err(v)
        }
    }
}
```

Используя это решение, значение можно загрузить из атомарной переменной, выполнить любые требуемые вычисления, а затем сохранить вновь вычисленное значение только в том случае, если атомарная переменная за это время не изменилась. Если поместить это код в цикл, чтобы повторить попытку в случае изменения первоначального значения, его можно было бы использовать для реализации всех других атомарных операций, придав данной операции универсальный характер.

Давайте в целях демонстрации увеличим `AtomicU32` на единицу, не задействуя `fetch_add`, просто чтобы увидеть пример практического использования `compare_exchange`:

```
fn increment(a: &AtomicU32) {
    let mut current = a.load(Relaxed);                          ❶
    loop {
      let new = current + 1;                                    ❷
      match a.compare_exchange(current, new, Relaxed, Relaxed) { ❸
            Ok(_) => return,                                    ❹
            Err(v) => current = v,                              ❺
        }
    }
}
```

❶ Предварительная загрузка текущего значения `a`.

❷ Вычисление нового значения, которое нужно сохранить в `a`, не беря в расчет возможные одновременно вносимые изменения в `a` со стороны других потоков.

❸ Использование `compare_exchange` для обновления значения `a`, но только при условии, что в данной переменной сохранилось ранее загруженное в нее значение.

❹ Замена прежнего значения `a` новым, вычисленным значением, если `a` действительно осталась такой же, как и раньше, завершая тем самым всю операцию.

❺ Если `a` не была такой же, как и раньше, другой поток, должно быть, изменил ее значение за тот короткий промежуток времени, когда шла загрузка ее прежнего значения. Операция `compare_exchange` выдает измененное значение для попытки повторения операции уже с этим значением. Момент между загрузкой и обновлением настолько краток, что затягивание данного цикла более чем на несколько итераций маловероятно.

Если атомарная переменная изменится с некоторого значения A на значение B, а затем, после операции загрузки, но до операции compare_exchange, обратно на A, операция все равно будет успешной, несмотря на то, что за это время атомарная переменная была изменена (и произошло ее обратное изменение). Во многих случаях, как в нашем примере с приращением, это не создаст никаких проблем. Но существуют определенные алгоритмы, часто использующие атомарные указатели, для которых это может стать проблемой. Данное стечение обстоятельств называется *проблемой ABA*.

Наряду с compare_exchange имеется похожий на него метод compare_exchange_weak. Разница в том, что слабая (weak) версия может иногда оставлять значение нетронутым и возвращать Err, даже если атомарное значение соответствует ожидаемому. На некоторых платформах данный метод можно реализовать более эффективно, и ему следует отдавать предпочтение в тех случаях, когда последствия ложного сбоя сравнения и обмена незначительны, например в показанной ранее функции increment. Выяснение возможной более высокой эффективности слабой версии предстоит в *главе 7*, где низкоуровневые особенности реализаций функций будут рассматриваться более подробно.

Пример: Предоставление идентификатора без переполнения

Теперь вернемся к проблеме переполнения в allocate_new_id() из *раздела "Пример: Предоставление идентификатора"*.

Чтобы остановить увеличение NEXT_ID сверх определенного предела и предотвратить переполнение, можно воспользоваться функцией compare_exchange для реализации атомарного сложения с верхней границей. Давайте воспользуемся этим замыслом и создадим версию allocate_new_id, которая всегда корректно обрабатывает переполнение, даже в практически невероятных ситуациях:

```
fn allocate_new_id() -> u32 {
    static NEXT_ID: AtomicU32 = AtomicU32::new(0);
    let mut id = NEXT_ID.load(Relaxed);
    loop {
        assert!(id < 1000, "too many IDs!");
        match NEXT_ID.compare_exchange_weak(id, id + 1, Relaxed, Relaxed) {
            Ok(_) => return id,
            Err(v) => id = v,
        }
    }
}
```

Теперь проверка и выдача паники происходят до изменения NEXT_ID, гарантируя тем самым, что значение этой переменной не превысит 1000, сделав переполнение невозможным. Теперь, если понадобится, верхний предел можно будет поднять с 1000 до u32::MAX, не беспокоясь о крайних случаях, в которых может быть превышен предел.

Выборка-обновление

Для шаблона цикла сравнения и обмена в атомарных типах имеется удобный метод `fetch_update`. Его работа эквивалентна применению операции `load`, за которой следует цикл, в котором точно так же, как это делалось ранее, повторяется вычисление и выполнение функции `compare_exchange_weak`.

Данный метод позволяет реализовать функцию `allocate_new_id` с помощью всего одной строки кода:

```
NEXT_ID.fetch_update(Relaxed, Relaxed,
    |n| n.checked_add(1)).expect("too many IDs!")
```

Более подробное описание метода можно найти в его документации.

В нашей книге метод `fetch_update` рассматриваться не будет, что позволит сосредоточиться на отдельно взятых атомарных операциях.

Пример: Отложенная однократная инициализация

Ранее, в *разделе "Пример: Отложенная инициализация"*, был рассмотрен пример отложенной инициализации постоянного значения (см. листинг 2.4). Созданная функция выполняла отложенную инициализацию значения при первом вызове, а затем повторно задействовала это значение при всех последующих вызовах. Когда самый первый вызов функции выполняется одновременно сразу несколькими потоками, ими же может быть выполнена и инициализация, и они перезапишут результат друг друга в непредсказуемом порядке.

Для ожидаемо постоянных значений, когда их изменение до инициализации несущественно, такой подход вполне сгодится. Но бывают такие сценарии, при которых значение всякий раз инициализируется другой величиной, и это при потребности возвращения каждым вызовом функции в пределах одного запуска программы одной и той же величины.

Представим, к примеру, функцию `get_key()`, возвращающую произвольно сгенерированный ключ, который создается за запуск программы всего один раз. Он может быть ключом шифрования, предназначенным для связи с программой, который должен быть уникальным при каждом запуске программы, но оставаться постоянным в процессе ее выполнения.

Получается, что просто выполнить операцию сохранения после генерации ключа невозможно, поскольку это может привести к перезаписи ключа, сгенерированного другим потоком всего несколько минут назад, в результате чего два потока будут использовать разные ключи. Во избежание этого можно воспользоваться функцией `compare_exchange`, позволяющей убедиться, что ключ сохраняется только в том случае, если ни один другой поток еще его не сохранил, в противном случае ключ выбрасывается, а вместо него используется сохраненный ключ.

Реализацию данного замысла иллюстрирует листинг 2.8.

Листинг 2.8

```
fn get_key() -> u64 {
    static KEY: AtomicU64 = AtomicU64::new(0);
```

```
    let key = KEY.load(Relaxed);
    if key == 0 {
      let new_key = generate_random_key();                    ❶
      match KEY.compare_exchange(0, new_key, Relaxed, Relaxed) {  ❷
          Ok(_) => new_key,                                   ❸
          Err(k) => k,                                        ❹
      }
  } else {
      key
  }
}
```

❶ Генерация нового ключа только в том случае, если инициализация `KEY` еще не состоялась.

❷ Замена `KEY` вновь сгенерированным ключом, но только если значение `KEY` все еще равно нулю.

❸ Если нуль заменен на новый ключ, возвращение только что сгенерированного ключа. Новые вызовы `get_key()` приведут к возвращению того же нового ключа, который теперь хранится в `KEY`.

❹ Если гонка будет проиграна другому потоку, инициализировавшему `KEY` раньше текущего потока, вновь сгенерированный ключ будет забыт, и вместо него будет взят ключ из `KEY`.

Это наглядный пример ситуации, при которой применение `compare_exchange` представляется более уместным, чем использование слабого варианта данной функции. Здесь операция сравнения и обмена не выполняется в цикле и не нужно возвращать нуль при ее неожиданном неудачном завершении.

Как уже упоминалось в *разделе "Пример: Отложенная инициализация"*, если функция `generate_random_key()` выполняется слишком долго, возможно, имеет смысл заблокировать потоки во время инициализации, исключая тем самым потенциальные затраты времени на генерацию тех ключей, которые не будут задействованы. Такая функциональная возможность предоставляется стандартной библиотекой Rust за счет наличия функций `std::sync::Once` и `std::sync::OnceLock`.

Резюме

- Атомарные операции неделимы; они либо полностью завершаются, либо вообще не выполняются.
- Атомарные операции в Rust выполняются путем использования атомарных типов в `std::sync::atomic`, примером такого типа может послужить `AtomicI32`.
- Не все атомарные типы доступны абсолютно на всех платформах.
- При задействовании нескольких переменных предсказать относительный порядок выполнения атомарных довольно сложно. Более подробно данный вопрос рассматривается в *главе 3*.

- Простые операции загрузки и сохранения хороши для несложного взаимодействия потоков, например для флагов остановки и отчетов о состоянии.
- Отложенную инициализацию можно выполнить в режиме *гонки*, не создавая *гонку данных*.
- Операции выборки и изменения допускают небольшой набор базовых атомарных модификаций, которые особенно пригодятся при изменении одной и той же атомарной переменной сразу несколькими потоками.
- При переполнении атомарные операции сложения и вычитания автоматически переходят к начальным величинам диапазонов чисел.
- Операции сравнения и обмена наиболее гибкие и универсальные, они являются строительными блоками для совершения любой другой атомарной операции.
- Применение слабого варианта операции сравнения и обмена может оказаться более эффективным.

ГЛАВА 3

Упорядочение памяти

В *главе 2* уже состоялось беглое знакомство с понятием *упорядочение памяти*. А здесь нам предстоит углубиться в тему, рассматривая все доступные варианты упорядочения памяти и, самое главное, разбираясь в том, какие из них и когда следует применять.

Изменение порядка и оптимизация

Чтобы программы работали как можно быстрее, процессоры и компиляторы идут на всяческие ухищрения. Процессор может определить, что имеющиеся в программе две отдельные последовательные инструкции не влияют друг на друга, и выполнит их *не по порядку*, если это, к примеру, будет быстрее. А когда одна инструкция ненадолго блокируется при извлечении из основной памяти некоторых данных, то при условии неизменности поведения программы до завершения данной инструкции могут быть выполнены и завершены несколько следующих инструкций. Аналогично этому компилятор может принять решение об изменении порядка следования инструкций или переписать части вашей программы, если у него имеются основания полагать, что данные действия могут привести к ускоренному выполнению кода программы. Но, опять же, это будет сделано только в том случае, если не повлечет за собой изменений в поведении программы.

Рассмотрим в качестве примера следующую функцию:

```
fn f(a: &mut i32, b: &mut i32) {
    *a += 1;
    *b += 1;
    *a += 1;
}
```

Здесь компилятор наверняка поймет, что порядок выполнения операций не играет никакой роли, поскольку между этими тремя операциями сложения не происходит ничего, что зависело бы от значения `*a` или `*b`. (Исходя из предположения, что проверка переполнения отключена.) Поэтому он может переупорядочить вторую и третью операции, а затем объединить первые две в одно сложение:

```
fn f(a: &mut i32, b: &mut i32) {
    *a += 2;
    *b += 1;
}
```

Позже, выполняя данную функцию оптимизированной скомпилированной программы, процессор по разным причинам может выполнить второе сложение раньше первого, возможно, из-за доступности `*b` в кеше и необходимости извлечения `*a` из основной памяти.

Независимо от этих оптимизаций, результат остается тем же: `*a` увеличивается на два, а `*b` увеличивается на единицу. Для остальной части программы порядок, в котором происходит увеличение их значений, абсолютно невидим.

Логика, позволяющая проверить, что определенное изменение порядка или другая оптимизация не повлияет на поведение программы, не учитывает наличие других потоков. В показанном примере она срабатывает, поскольку исключительные ссылки (`&mut i32`) гарантируют, что ничто другое не сможет получить доступ к значениям, делая наличие других потоков абсолютно ненужным. Единственной проблемной ситуацией остается изменение данных, совместно используемых разными потоками. Или, иными словами, работа с атомарностью. Вот почему компилятору и процессору нужно сообщать, что они могут и чего не могут делать с нашими атомарными операциями, поскольку их обычная логика игнорирует моменты взаимодействия потоков и может допускать оптимизацию, меняющую результат выполнения программы.

Интерес представляет вопрос о том, *как именно* им об этом сообщается. Если бы захотелось добиться точного определения, что приемлемо, а что нет, конкурентное программирование стало бы подверженным многословию и просчетам, а возможно, даже зависящим от архитектуры:

```
let x = a.fetch_add(1,
    Дорогой компилятор и процессор,
    можете смело изменять порядок при проведении операций над b,
    но если имеется какой-либо другой поток, одновременно выполняющий f,
    пожалуйста, не изменяйте порядок проведения операций над c!
    И еще, процессор, не забудьте очистить буфер хранилища!
    Но если b равно нулю, это не имеет значения.
    В таком случае смело делайте то, что будет работать быстрее.
    Спасибо~ <3
);
```

Но у нас имеется возможность выбора из весьма скромного набора параметров, представленного перечислением `std::sync::atomic::Ordering`, принимаемого каждой атомарной операцией в качестве аргумента. Вариантов немного, но они тщательно подобраны, чтобы подходить большинству сценариев. Упорядочение носит весьма абстрактный характер и не отражает напрямую реальные механизмы компилятора и процессора, вроде инструкций переупорядочения. Это позволяет создаваемому вами конкурентному коду всегда оставаться актуальным и независимым от архитектуры и оценивать его пригодность без учета особенностей каждой текущей и будущей версии процессора и компилятора.

В Rust доступны следующие варианты упорядочения:

- Расслабленное упорядочение: `Ordering::Relaxed`.
- Упорядочение получения и высвобождения: `Ordering::{Release, Acquire, AcqRel}`.
- Последовательно согласованное упорядочение: `Ordering::SeqCst`.

В C++ также есть нечто, называемое *упорядочением потребления*, которое намеренно опущено в Rust, и все же его тоже будет интересно рассмотреть.

Модель памяти

Различные варианты упорядочения памяти имеют строгое формальное определение, позволяющее программистам четко понимать, что можно предполагать, а авторам компиляторов точно знать, какие гарантии они должны им предоставить. Чтобы отделить это от особенностей конкретных архитектур процессоров, упорядочение памяти определяется в понятиях абстрактной *модели памяти*.

Модель памяти Rust, которая в основном скопирована с C++, не соответствует ни одной существующей архитектуре процессора, представляя собой абстрактную модель со строгим набором правил, которая пытается соответствовать всем текущим и будущим архитектурам, а также открывает компилятору достаточный простор для выдвижения весьма рациональных предположений при анализе и оптимизации программ.

Часть особенностей модели памяти уже раскрывалась в *разделе "Заимствования и гонка данных" главы 1*, где говорилось о том, как гонка данных приводит к неопределенному поведению. Реализованная в Rust модель памяти допускает конкурентные атомарные сохранения, но считает конкурентные неатомарные сохранения одной и той же переменной гонкой данных, приводящей к неопределенному поведению.

Но, как будет показано в *главе 7*, в большинстве процессорных архитектур фактически нет никакой разницы между атомарным и обычным неатомарным сохранением. Кому-то может показаться, что модель памяти несет излишний ограничительный характер, но имеющиеся в ней строгие правила облегчают анализ программы как компилятору, так и программисту и оставляют место для будущих разработок.

Отношения "происходит до"

Модель памяти определяет порядок, в котором происходят операции, с точки зрения *отношений "происходит до"*. По причине абстрактности данной модели в ней не говорится о машинных инструкциях, кешах, буферах, согласованности по времени, переупорядочении инструкций, оптимизации компилятора и т. д., она всего лишь определяет ситуации, когда одно событие гарантированно произойдет раньше другого, и оставляет порядок всего остального неопределенным.

Основное правило отношения "происходит до" заключается в том, что в одном потоке все происходит по порядку. Если поток выполняет `f(); g();`, то `f()` *происходит до* `g()`.

Но между потоками отношения "происходит до" возникают только в нескольких конкретных случаях, например при создании и присоединении потока, разблокировке и блокировке мьютекса, а также при выполнении атомарных операций, использующих нерасслабленное упорядочение памяти. Расслабленное упорядочение

памяти (Relaxed) — основное и наиболее эффективное упорядочение памяти, которое само по себе никогда не приводит к каким-либо пересекающимся в разных потоках отношениям "происходит до".

Чтобы понять, что это значит, давайте рассмотрим следующий пример, в котором предполагается, что a и b одновременно выполняются разными потоками:

```
static X: AtomicI32 = AtomicI32::new(0);
static Y: AtomicI32 = AtomicI32::new(0);
fn a() {
    X.store(10, Relaxed);          ❶
    Y.store(20, Relaxed);          ❷
}
fn b() {
    let y = Y.load(Relaxed);       ❸
    let x = X.load(Relaxed);       ❹
    println!("{x} {y}");
}
```

Как ранее уже упоминалось, основное правило "происходит до" заключается в том, что все происходящее в одном потоке, происходит по порядку. В данном случае, как показано на рис. 3.1, ❶ "происходит до" ❷, а ❸ "происходит до" ❹. Поскольку здесь используется расслабленное упорядочение памяти, других отношений "происходит до" в данном примере нет.

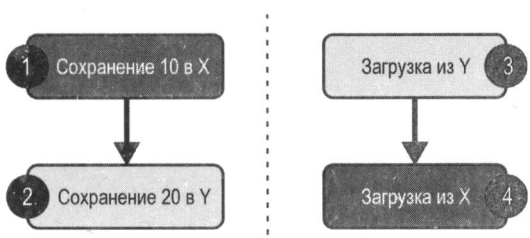

Рис. 3.1. Отношения "происходит до" между атомарными операциями в примере кода

Если какая-либо из функций, a или b, завершится до начала другой, то на выходе будет 0 0 или 10 20. Если a и b выполняются одновременно, нетрудно понять, что на выходе может быть 10 0. Такое может произойти, если операции выполняются в следующем порядке: ❸❶❷❹.

Еще интереснее, что результат также может быть 0 20, хотя возможного глобально согласованного порядка четырех операций, который привел бы к такому результату, попросту не существует. При выполнении ❸ не существует связи "происходит до" с ❷, что означает, что может состояться загрузка либо 0, либо 20. Когда выполняется ❹, нет отношения "происходит до" с ❶, т. е. может состояться загрузка либо 0, либо 10. С учетом этого получение на выходе 0 20 является вполне допустимым результатом.

Здесь следует понять одно важное и неочевидное обстоятельство: операция ❸ загрузки значения 20 *не* приводит к отношению "происходит до" с ❷, даже если это

значение сохранено кодом ❷. Наше интуитивное понимание концепции "до" рассыпается, когда события не обязаны происходить в глобально согласованном порядке, например когда речь заходит о переупорядочении инструкций.

Более практичное и интуитивно понятное, но менее формальное ви́дение происходящего заключается в том, что с позиции потока, выполняющего `b`, операции ❶ и ❷ могут происходить в обратном порядке.

Порождение и присоединение

Порождение потока создает отношение "происходит до" между тем, что произошло до вызова `spawn()`, и тем, что делается в новом потоке. Точно так же присоединение к потоку создает отношение "происходит до" между присоединенным потоком и тем, что происходит после вызова `join()`.

В качестве демонстрации утверждение в следующем примере не может не пройти проверку на достоверность:

```
static X: AtomicI32 = AtomicI32::new(0);
fn main() {
    X.store(1, Relaxed);
    let t = thread::spawn(f);
    X.store(2, Relaxed);
    t.join().unwrap();
    X.store(3, Relaxed);
}
fn f() {
    let x = X.load(Relaxed);
    assert!(x == 1 || x == 2);
}
```

Рис. 3.2. Отношения "происходит до" между операциями порождения, присоединения, сохранения и загрузки в примере кода

Благодаря отношениям "происходит до", формируемым операциями присоединения и порождения, абсолютно точно известно, что, как показано на рис. 3.2, загрузка из X происходит после первого сохранения, но перед последним сохранением. А вот предсказать, когда значение становится видимым, до или после второго сохранения, невозможно. Иными словами, загрузка могла быть либо 1, либо 2, но не 0 или 3.

Расслабленное упорядочение

Хотя атомарные операции, использующие расслабленное упорядочение памяти, не обеспечивают каких-либо отношений "произошло до", они гарантируют *общий порядок модификаций* каждой отдельно взятой атомарной переменной. Это означает, что все модификации *одной и той же атомарной переменной* происходят в порядке, одинаковом с позиции каждого отдельно взятого потока.

Чтобы продемонстрировать значение сказанного, рассмотрим следующий пример, в котором предполагается, что a и b одновременно выполняются разными потоками:

```
static X: AtomicI32 = AtomicI32::new(0);
fn a() {
    X.fetch_add(5, Relaxed);
    X.fetch_add(10, Relaxed);
}
fn b() {
    let a = X.load(Relaxed);
    let b = X.load(Relaxed);
    let c = X.load(Relaxed);
    let d = X.load(Relaxed);
    println!("{a} {b} {c} {d}");
}
```

В данном примере только один поток изменяет значение X, поэтому легко понять, что существует только один возможный порядок модификации X: 0→5→15. Сначала переменная имеет значение нуль, затем значение становится пятью и, наконец, изменяется на пятнадцать. Потокам не могут быть видны какие-либо значения X, несовместимые с этим общим порядком модификаций. Следовательно, в другом потоке возможны, в частности, такие результаты работы инструкции вывода на экран, как "0 0 0 0", "0 0 5 15", "0 15 15 15", но никак не "0 5 0 15" или "0 0 10 15".

Даже если существует более одного возможного порядка модификации атомарной переменной, со всеми потоками будет согласован единый порядок.

Давайте заменим a двумя отдельными функциями, a1 и a2, каждая из которых, как предполагается, выполняется отдельным потоком:

```
fn a1() {
    X.fetch_add(5, Relaxed);
}
```

```
fn a2() {
    X.fetch_add(10, Relaxed);
}
```

Притом что это единственные потоки, модифицирующие X, теперь, в зависимости от того, какая операция fetch_add выполняется первой, существуют два возможных порядка модификации: либо 0→5→15, либо 0→10→15. В любом случае все потоки соблюдают один и тот же порядок. Итак, даже если имеются сотни дополнительных потоков, выполняющих функцию b(), доподлинно известно, что если один из них выведет на экран 10, порядок должен быть 0→10→15, и ни один из них не сможет напечатать 5. И наоборот.

В *главе 2* уже были приведены примеры сценариев, в которых гарантированного общего порядка модификаций отдельных переменных было вполне достаточно, что определяло обоснованность применения расслабленного упорядочения памяти. Но если кроме этих примеров попробовать что-либо посерьезнее, тут же возникнет потребность в чем-то более серьезном, чем расслабленное упорядочение памяти.

Неизвестно откуда взявшиеся значения

Когда операции циклически зависят друг от друга, присущее расслабленному упорядочению памяти отсутствие гарантий упорядочения может теоретически привести к ряду осложнений.

Чтобы продемонстрировать суть вопроса, приведем вымышленный пример, где два потока загружают значение из одного атомарного типа и сохраняют его в другом:

```
static X: AtomicI32 = AtomicI32::new(0);
static Y: AtomicI32 = AtomicI32::new(0);
fn main() {
    let a = thread::spawn(|| {
        let x = X.load(Relaxed);
        Y.store(x, Relaxed);
    });
    let b = thread::spawn(|| {
        let y = Y.load(Relaxed);
        X.store(y, Relaxed);
    });
    a.join().unwrap();
    b.join().unwrap();
    assert_eq!(X.load(Relaxed), 0); // Может ли утверждение быть ложным?
    assert_eq!(Y.load(Relaxed), 0); // Может ли утверждение быть ложным?
}
```

Казалось бы, нетрудно понять, что значения X и Y никогда не будут отличными от нуля, поскольку операции сохранения касаются только тех значений, которые были загружены из этих же атомарных типов, всегда равных нулю.

Но если строго следовать теоретической модели памяти, придется взять за основу наши циклические рассуждения и прийти к пугающему выводу, что здесь можно и ошибиться. Фактически модель памяти чисто технически допускает результат, при котором и X, и Y в конечном итоге равны 37 или любому другому значению, что приводит к ложности утверждений.

Из-за отсутствия гарантий упорядочения операций загрузки в этих двух потоках, оба они могут видеть результат операции сохранения другого потока, допуская в порядке операций

следующий цикл: 37 сохраняется в Y, потому что значение 37 было загружено из X, и оно было сохранено в X, потому что 37 было загружено из Y и являлось значением, которое было сохранено в Y.

К счастью, возможность получения таких *неизвестно откуда взявшихся* значений всеми считается ошибкой, допущенной в теоретической модели, а не чем-то таким, что нужно принимать во внимание на практике. Теоретическая проблема формализации расслабленного упорядочения памяти без учета таких аномалий в модели остается нерешенной. Хотя с формальной точки зрения это можно посчитать досадным недоразумением, лишающим нормального сна многих теоретиков, все, кроме них, могут расслабиться в блаженном неведении, зная, что на практике подобное никогда не встретится.

Упорядочение высвобождения и получения памяти

Для формирования отношений между потоками по типу "происходит до" упорядочение *высвобождения* и *получения* памяти используется в паре. Упорядочение высвобождения памяти (Release) применяется к операциям сохранения, а упорядочение ее получения (Acquire) — к операциям загрузки.

Отношения "происходит до" формируются, когда операция получения-загрузки отслеживает результат операции высвобождения-сохранения. В данном случае операция сохранения и всё, что было до нее, происходит до операции загрузки и всего, что было после нее.

При использовании Acquire для операции выборки и изменения или сравнения и обмена данный порядок применяется только к той части операции, которая загружает значение. Точно так же Release применяется только к той части операции, которая связана с сохранением. Для представления комбинации Acquire и Release, которая заставляет загрузку учитывать порядок получения, а сохранение — порядок высвобождения, служит AcqRel.

Чтобы посмотреть, как это делается на практике, рассмотрим следующий пример (листинг 3.1), в котором 64-разрядное целое число отправляется из порожденного потока в основной поток. Для указания основному потоку, что целое число сохранено и готово к чтению, введено дополнительное атомарное логическое значение:

Листинг 3.1

```
use std::sync::atomic::Ordering::{Acquire, Release};
static DATA: AtomicU64 = AtomicU64::new(0);
static READY: AtomicBool = AtomicBool::new(false);
fn main() {
    thread::spawn(|| {
        DATA.store(123, Relaxed);
        READY.store(true, Release); // Все предшествующее этому сохранению ...
    });
```

```
    while !READY.load(Acquire) { // ... становится видимым после загрузки этим кодом
                                       значения 'true'.
        thread::sleep(Duration::from_millis(100));
        println!("waiting...");
    }
    println!("{}", DATA.load(Relaxed));
}
```

Когда порожденный поток завершает сохранение данных, он использует высвобождение-сохранение, чтобы установить для флага READY значение true. А когда основной поток отслеживает это посредством операции получения-загрузки, то, как показано на рис. 3.3, между этими двумя операциями устанавливается отношение "происходит до". На данном этапе точно известно, что все произошедшее до высвобождения-сохранения в READY видно всему, что происходит после получения-загрузки. В частности, когда основной поток выполняет загрузку из DATA, точно известно, что он загрузит значение, хранящееся в фоновом потоке. Есть только один возможный результат, который данная программа может вывести, выполняя последнюю строку кода: 123.

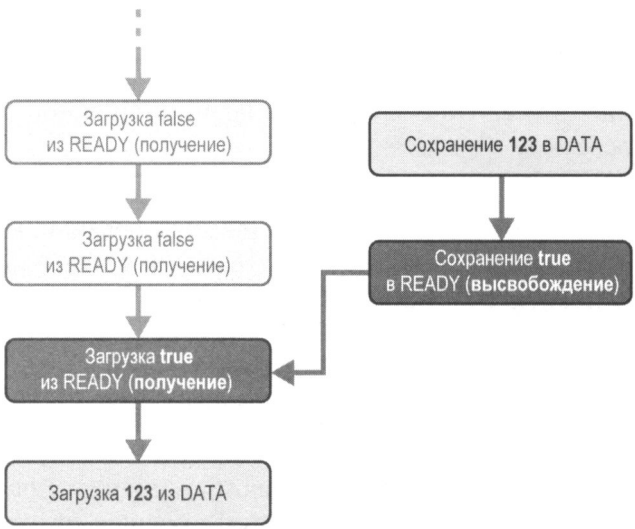

Рис. 3.3. Отношения "происходит до" между атомарными операциями в примере кода, показывающие отношения между потоками, сформированные операциями получения и высвобождения

Если бы для всех операций данного примера было задано расслабленное упорядочение памяти, то основной поток мог бы увидеть, как READY переключился в true, но после этого все равно загружал бы нуль из DATA.

Названия "высвобождение" и "получение" основаны на их основном варианте использования: один поток *высвобождает* данные, атомарно сохраняя некое значение в атомарной переменной, а другой поток *получает* их, атомарно загружая это значение. Именно это и происходит при разблокировке (высвобождении) мьютекса, а затем блокировке (получении) его в другом потоке.

В нашем примере (см. листинг 3.1) отношение "происходит до", возникающее благодаря наличию флага READY, гарантирует, что операции сохранения и загрузки DATA не могут выполняться одновременно. Фактически не нужно, чтобы эти операции были атомарными.

Но если просто попытаться воспользоваться для переменной данных обычным неатомарным типом, компилятор отвергнет программу, поскольку имеющаяся в Rust система типов не позволяет изменять типы из одного потока, когда они также заимствуются другим потоком. Система типов не в состоянии неким волшебным образом разобраться в созданных здесь отношениях "происходит до". Чтобы уверить компилятор в тщательной продуманности кода и в твердом соблюдении всех правил, воспользоваться фрагментом небезопасного unsafe-кода все же придется:

```
static mut DATA: u64 = 0;
static READY: AtomicBool = AtomicBool::new(false);
fn main() {
    thread::spawn(|| {
        // Безопасно: ничто иное не имеет доступа к DATA,
        // поскольку пока флаг READY еще не установлен.
        unsafe { DATA = 123 };
        READY.store(true, Release); // Все предшествующее этому сохранению ..
    });
    while !READY.load(Acquire) { // .. становится видимым после загрузки
                                 // этим кодом значения 'true'.
        thread::sleep(Duration::from_millis(100));
        println!("waiting...");
    }
    // Безопасно: ничто не изменяет значения DATA,
    // поскольку установлен READY.
    println!("{}", unsafe { DATA });
}
```

Более формальное объяснение

Отношение "происходит до" формируется, когда операция получения-загрузки отслеживает результат операции высвобождения-сохранения. Но что именно это означает?

Представим, что два потока сохраняют число семь в одной и той же атомарной переменной, а третий поток загружает из нее это число. Формируется ли при этом отношение "происходит до" между третьим и первым или вторым потоком? Все зависит от того, "какая семерка" загружена третьим потоком: из первого или из второго потока. (Или же, возможно, вообще загружена не связанная с ними семерка.) Получается, что, хотя семерка и равна семи, две семерки от двух разных потоков — вещи разные.

Осмыслить это можно в понятиях *общего порядка модификаций*, о котором говорилось в *разделе "Расслабленное упорядочение"*: упорядоченного списка всех модификаций атомарной переменной. Пусть даже одно и то же значение записывается в одну и ту же переменную несколько раз, каждая из этих операций в общем порядке модификаций этой переменной представляет собой отдельно взятое событие. Значение при загрузке соответствует конкретной точке на "временной шкале" переменной, что показывает, с какой операцией можно синхронизироваться.

Предположим, что общий порядок модификаций атомарной переменной выстраивается в следующую череду событий:

1. Инициализация в 0.
2. Высвобождение-сохранение 7 (из второго потока).
3. Расслабленное сохранение 6.
4. Высвобождение-сохранение 7 (из первого потока).

Тогда получение-загрузка 7 будет синхронизироваться либо с высвобождением-сохранением из второго события, либо с высвобождением-сохранением из последнего события. Но если ранее в поле зрения попадалось (с позиции отношения "происходит до") число 6, то известно, что видна последняя 7, а не первая, т. е. теперь сформировано отношение "происходит до" с потоком 1, а не с потоком 2.

Есть еще одна особенность: значение высвобождения-сохранения может быть модифицировано любым количеством операций выборки-изменения и сравнения-обмена, сохраняя при этом отношение "происходит до" с получением-загрузкой, считывающим конечный результат.

Представим, к примеру, атомарную переменную со следующим общим порядком модификаций:

1. Инициализация в 0.
2. Высвобождение-сохранение 7.
3. Расслабленная выборка и сложение с 1, замена 7 на 8.
4. Высвобождение-выборка и сложение с 1, изменение 8 на 9.
5. Высвобождение-сохранение 7.
6. Расслабленный обмен 10, замена 7 на 10.

Теперь, если из этой переменной выполнить получение-загрузку 9, то будет не только сформировано отношение "происходит до" с четвертой операцией (которая сохранила это значение), но и со второй операцией (которая сохранила 7), даже если третья операция использовала расслабленное упорядочение памяти.

Аналогично этому, если выполняется получение-загрузка из этой переменной значения 10, которое было записано расслабленной операцией, все равно сформируется отношение "происходит до" с пятой операцией (которая сохранила 7). Поскольку это была обычная операция сохранения (а не операция выборки-изменения или сравнения-обмена), цепочка разрывается: отношение "происходит до" не формируется ни с одной из других операций.

Пример: Блокировка

Наиболее распространенным сценарием упорядочения высвобождения и получения являются мьютексы (см. *раздел "Mutex и RwLock" главы 1*). При блокировке для проверки разблокированности используется упорядочение получения с одновременным (атомарным) изменением состояния на "заблокирован". При разблокировке для возвращения состояния "разблокирован" реализуется упорядочение высвобождения. Таким образом, между разблокировкой мьютекса и его последующей блокировкой формируется отношение "происходит до".

Эту схему можно продемонстрировать следующим кодом:

```
static mut DATA: String = String::new();
static LOCKED: AtomicBool = AtomicBool::new(false);
```

```
fn f() {
    if LOCKED.compare_exchange(false, true, Acquire, Relaxed).is_ok() {
        // Безопасно: здесь удерживается исключительная блокировка,
        // и ничто иное не имеет доступа к DATA.
        unsafe { DATA.push('!') };
        LOCKED.store(false, Release);
    }
}
fn main() {
    thread::scope(|s| {
        for _ in 0..100 {
            s.spawn(f);
        }
    });
}
```

В *разделе "Операции сравнения и обмена" главы 2* уже было показано, что данные операции принимают два аргумента упорядочивания памяти: один для случая успешного сравнения и выполненного сохранения, и один для случая неудавшегося сравнения и невыполненного сохранения. В `f` предпринимается попытка изменения значения `LOCKED` с `false` на `true` и получения доступа к `DATA` только в том случае, если все пройдет удачно. Получается, что наша озабоченность касается лишь успешной упорядоченности памяти. Если операция `compare_exchange` не проходит, это должно быть связано с тем, что для `LOCKED` уже установлено значение `true`, и тогда `f` ничего не делает. Это соответствует операции `try_lock` для обычного мьютекса.

Внимательный читатель уже мог заметить, что операция сравнения и обмена могла бы быть и просто операцией обмена, поскольку замена `true` на `true` при уже заблокированном значении не делает код некорректным:

```
// Это также работает.
if LOCKED.swap(true, Acquire) == false {
    ...
}
```

Благодаря упорядочению памяти по получению и высвобождению точно известно, что никакие два потока не могут одновременно получить доступ к `DATA`. На рис. 3.4 показано, что любой предыдущий доступ к `DATA` "происходит до" последующего высвобождения-сохранения `false` в `LOCKED`, что, в свою очередь, "происходит до" следующей операции получения-сравнения-и-обмена (или получения-обмена), которая изменила `false` на `true`, что "происходит до" следующего доступа к `DATA`.

В *главе 4* эта концепция будет превращена в многократно используемый тип: спин-блокировку.

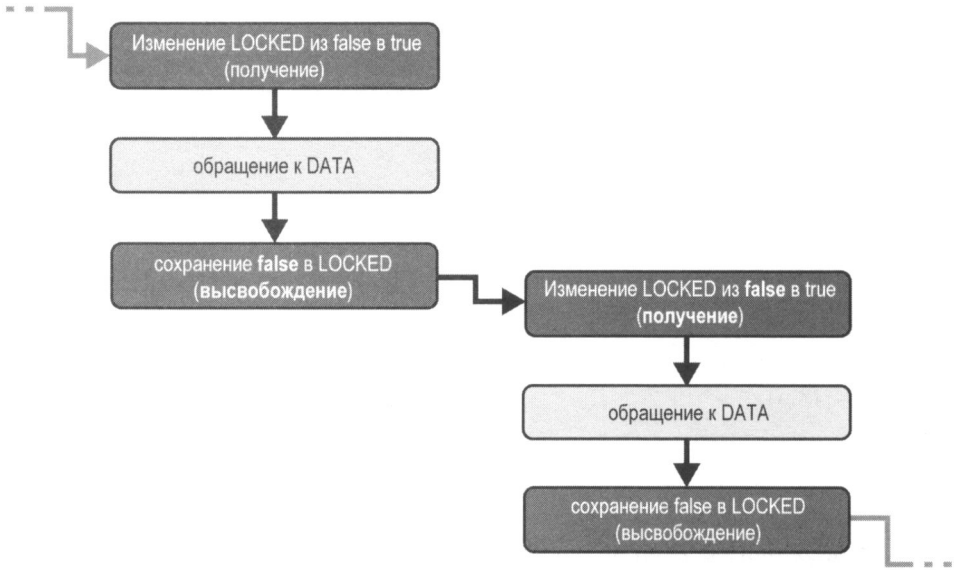

Рис. 3.4. Отношения "происходит до" между атомарными операциями в примере с блокировкой, показывающем в последовательности блокировку и разблокировку двух потоков

Пример: Отложенная инициализация с применением косвенного подхода

В *разделе "Пример: Отложенная однократная инициализация" главы 2* была реализована отложенная инициализация глобальной переменной с использованием операции сравнения и обмена для обработки тех ситуаций, когда в гонку за инициализацию значения одновременно включаются несколько потоков (см. листинг 2.8). Поскольку значение было ненулевым 64-разрядным целым числом, для его сохранения подошел тип `AtomicU64`, а заполнителем перед инициализацией было нулевое значение.

Чтобы сделать то же самое для гораздо большего типа данных, не умещающихся в одной атомарной переменной, нужно найти какую-то альтернативу.

Для рассматриваемого примера предположим, что нужно обеспечить поведение без блокировок, чтобы потокам никогда не приходилось ждать какого-либо другого потока, участвуя вместо этого в гонке и получая значение от победившего в ней потока. Иными словами, по-прежнему сохраняется потребность перехода от "неинициализированного" к "полностью инициализированному" значению за одну атомарную операцию.

Как гласит фундаментальная теорема разработки программных средств, любую проблему в компьютерной сфере можно решить, добавив еще один уровень косвенности, и в этом наша задача ничем не отличается от других задач. Поскольку данные в одной атомарной переменной не поместятся, ее можно использовать для хранения указателя на эти данные.

Объект `AtomicPtr<T>` является атомарной версией `*mut T` и служит указателем на `T`. В качестве заполнителя исходного состояния можно воспользоваться нулевым указателем и задействовать его для атомарной замены на вновь выделенный полностью инициализированный `T`, который затем может быть прочитан другими потоками.

Поскольку совместно используется не только атомарная переменная, содержащая указатель, но и данные, на которые она указывает, применить расслабленное упорядочение памяти, как это было в *главе 2*, уже не получится. Здесь придется убедиться, что выделение и инициализация данных не будут вызывать гонку, связанную с их чтением. Иными словами, для операций сохранения и загрузки следует воспользоваться упорядочением высвобождения и получения, чтобы убедиться, что компилятор и процессор не приведут наш код в негодность, переупорядочив, к примеру, в памяти сохранение указателя и инициализацию самих данных.

Это побуждает к следующей реализации для некоего произвольного типа данных, называемого `Data`:

```
use std::sync::atomic::AtomicPtr;
fn get_data() -> &'static Data {
    static PTR: AtomicPtr<Data> = AtomicPtr::new(std::ptr::null_mut());
    let mut p = PTR.load(Acquire);
    if p.is_null() {
        p = Box::into_raw(Box::new(generate_data()));
        if let Err(e) = PTR.compare_exchange(
            std::ptr::null_mut(), p, Release, Acquire
        ) {
            // Безопасно: p берется из строки Box::into_raw,
            // расположенной чуть выше, и не будет совместно
            // использоваться с каким-либо другим потоком.
            drop(unsafe { Box::from_raw(p) });
            p = e;
        }
    }
    // Безопасно: p не равно нулю и указывает
    // на надлежаще инициализированное значение.
    unsafe { &*p }
}
```

Если указатель, берущийся при выполнении операции получения-загрузки из `PTR`, не нулевой, предполагается, что он указывает на уже инициализированные данные, на основании чего создается ссылка на эти данные.

Но если указатель по-прежнему имеет нулевое значение, генерируются новые данные, сохраняемые в новом выделении с использованием `Box::new`. Затем этот `Box` с помощью `Box::into_raw` превращается в указатель-заготовку для попытки его сохранения `PTR` посредством операции сравнения и обмена. Если гонку инициализации выиграет другой поток, операция `compare_exchange` не пройдет, поскольку `PTR` больше не равен нулю. Если именно так и получится, указатель-заготовка будет

обращен в Box, чтобы высвободить отведенную под него память с помощью drop, избегая тем самым утечки памяти, а работа продолжится с указателем, сохраненным в PTR другим потоком.

В комментарии к последнему небезопасному (unsafe) блоку высказано предположение, что указываемые данные уже инициализированы. Следует заметить, что из этого вытекает также предположение о порядке выполнения действий. Чтобы гарантировать справедливость предположения, реализовано упорядочение высвобождения и получения памяти, позволяющее обеспечить отношение "происходит до" между инициализацией данных и созданием ссылки на них.

Загрузка потенциально ненулевого (т. е. проинициализированного) указателя происходит в двух местах: через операцию load и, когда она не выполняется, через операцию compare_exchange. Поэтому, как уже объяснялось ранее, необходимость использования Acquire обусловлена как упорядочением памяти при выполнении load, так и упорядочением памяти в случае сбоя при выполнении compare_exchange, чтобы получить возможность синхронизации с операцией сохранения указателя. Это сохранение происходит при успешном завершении операции compare_exchange, поэтому в качестве ее успешного упорядочения следует воспользоваться Release.

Наглядная демонстрация операций и отношений "происходит до" для ситуации, когда три потока вызывают функцию get_data(), показана на рис. 3.5. В данной ситуации оба потока, A и B, отслеживают нулевой указатель и оба пытаются инициализировать атомарный указатель. Поток A выигрывает эту гонку, в результате чего

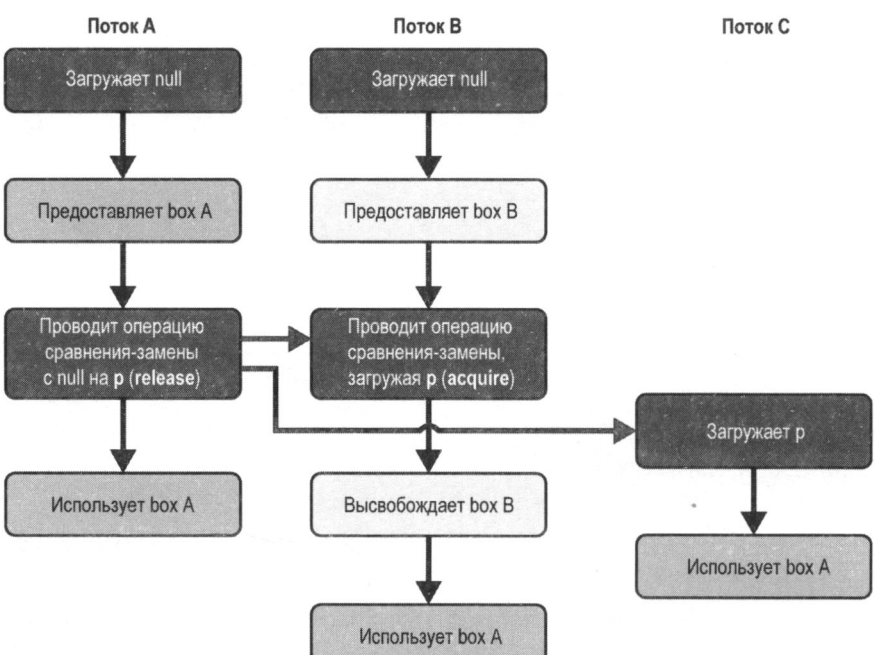

Рис. 3.5. Операции и отношения "происходит до" между тремя потоками, вызывающими функцию get_data()

операция compare_exchange потока B не проходит. Поток C отслеживает атомарный указатель только после того, как он был инициализирован потоком A. В итоге все три потока используют умный указатель (box), предоставленный потоком A.

Упорядочение потребления

Давайте присмотримся к упорядочению памяти в нашем последнем примере. Если изолироваться от строгой модели памяти и посмотреть на все чисто в практическом ключе, то можно сказать, что упорядочение высвобождения предотвращает переупорядочение инициализации данных с операцией сохранения, которая совместно использует указатель с другими потоками. А это весьма важное обстоятельство, поскольку в противном случае другие потоки могли бы увидеть данные еще до их полной инициализации.

Аналогичным образом можно объяснить смысл упорядочения получения как средства предотвращения переупорядочения, которое могло бы привести к тому, что доступ к данным был бы получен еще до загрузки указателя. Но логично задаться вопросом, есть ли во всем этом практический смысл. Как можно получить доступ к данным до того, как будет известен их адрес? Можно прийти к выводу, что достаточно чего-то более слабого, чем упорядочение получения. Так оно и есть, именно таким более слабым и является *упорядочение потребления*.

По сути, упорядочение потребления — это облегченный и более эффективный вариант упорядочения получения, синхронизационное влияние которого распространяется только на то, что *зависит от* загруженного значения.

Если потребляется-загружается прошедшее операцию высвобождения-сохранения значение x из атомарной переменной, то, по сути, это сохранение "происходит до" вычисления таких *зависимых* выражений, как *x, array[x] или table.lookup(x + 1), но *не* обязательно перед независимыми операциями вроде чтения другой переменной, для которого не требуется значение x.

А теперь как хорошие, так и плохие новости.

Хорошая новость заключается в том, что на всех современных процессорных архитектурах упорядочение потребления достигается с помощью точно таких же инструкций, как и расслабленное упорядочение. Иными словами, упорядочивание потребления может быть "без издержек", чего, по крайней мере, на некоторых платформах, нельзя сказать об упорядочении получения.

А плохая новость заключается в том, что упорядочение потребления не реализуется ни одним из компиляторов. Как оказалось, трудно не только определить понятие "зависит от", но и еще труднее сохранить такие зависимости в процессе преобразования и оптимизации программы.

Выяснилось, что понятие "зависимого" вычисления не только трудно определить, но еще труднее сохранить такие зависимости нетронутыми при преобразовании и оптимизации программы. Компилятор, к примеру, может оптимизировать x + 2 - x до 2, эффективно устраняя зависимость от x. Более изощренные варианты этой про-

блемы могут возникать в отношении таких достаточно практичных выражений, как `array[x]`, если компилятор в состоянии прийти к каким-либо логическим выводам о возможных значениях `x` или элементов массива. Проблема сильно усложняется, если принять во внимание поток управления, связанный, например, с наличием инструкций `if` или вызовов функций.

В связи с этим компиляторы ради своей безопасности доводят упорядочение потребления до упорядочения получения. В стандарте C++20 даже имеется явная рекомендация не применять упорядочение потребления, замечая, что реализация, отличная от упорядочения получения, окажется невыполнимой.

Возможно, в будущем будет найдено вполне работоспособное определение и реализация упорядочения потребления. Но до этого никакого `Ordering::Consume` в Rust не предоставляется.

Последовательно согласованное упорядочение

Самое сильное упорядочение — *последовательно согласованное* упорядочение: `Ordering::SeqCst`. Оно включает в себя все гарантии упорядочения получения (для загрузок) и упорядочения высвобождения (для сохранений), а также гарантирует глобально согласованный порядок проведения операций.

Это означает, что каждая отдельно взятая операция, использующая в программе упорядочение `SeqCst`, является частью единого общего порядка, согласованного со всеми потоками. Этот общий порядок согласуется с общим порядком модификации каждой отдельно взятой переменной.

Поскольку данное упорядочение намного сильнее упорядочений получения и высвобождения памяти, последовательно согласованная загрузка или сохранение может занять в паре высвобождение-получение место получения-загрузки или высвобождения-сохранения для формирования отношения "происходит до". Иными словами, получение-загрузка может сформировать отношение "происходит до" не только с высвобождением-сохранением, но также и с последовательно согласованным сохранением, точно так же и наоборот.

Соответствие единому общему порядку операций `SeqCst` гарантируется только в том случае, когда упорядочение `SeqCst` используется обеими сторонами отношения "происходит до".

Хотя может показаться, что рассуждать об упорядочении `SeqCst` проще всего, но на практике это упорядочение почти никогда не бывает востребованным. Практически во всех случаях достаточно обычного упорядочения получения и высвобождения.

Рассмотрим пример, зависящий от последовательно согласованного упорядочения операций:

```
use std::sync::atomic::Ordering::SeqCst;
static A: AtomicBool = AtomicBool::new(false);
static B: AtomicBool = AtomicBool::new(false);
static mut S: String = String::new();
```

```
fn main() {
    let a = thread::spawn(|| {
        A.store(true, SeqCst);
        if !B.load(SeqCst) {
            unsafe { S.push('!') };
        }
    });
    let b = thread::spawn(|| {
        B.store(true, SeqCst);
        if !A.load(SeqCst) {
            unsafe { S.push('!') };
        }
    });
    a.join().unwrap();
    b.join().unwrap();
}
```

Сначала чтобы предупредить другой поток о том, что они собираются получить доступ к `S`, оба потока устанавливают свое атомарное логическое значение в `true`, а затем проверяют атомарное логическое значение другого потока, чтобы убедиться, что они могут безопасно получить доступ к `S`, не вызывая гонки данных.

Если обе операции сохранения происходят до любой из операций загрузки, возможно, что доступ к `S` не получит ни один из потоков. Но получение доступа к `S` и вызов неопределенного поведения невозможны для *обоих* потоков, поскольку последовательно согласованное упорядочение гарантирует, что выиграть гонку сможет только один из них. В каждом возможном общем порядке первой операцией будет операция сохранения, которая предотвращает доступ к `S` со стороны другого потока.

Подобная схема сохранения, которое должно быть глобально видимым перед последующей загрузкой в том же самом потоке, включена практически во все реальные варианты использования `SeqCst`. В подобных ситуациях потенциально более эффективны расслабленные операции в сочетании с рассматриваемым далее *ограждением* `SeqCst`.

Ограждения

Помимо операций над атомарными переменными, существует еще одна вещь, к которой можно применить упорядочение памяти: атомарные ограждения.

Функция `std::sync::atomic::fence` представляет собой атомарное ограждение и является либо *ограждением высвобождения* (`Release`), либо *ограждением получения* (`Acquire`), либо ими обоими (`AcqRel` или `SeqCst`). Ограждение `SeqCst` также принимает участие в последовательно согласованном общем порядке.

Функция `std::sync::atomic::fence` представляет собой атомарное ограждение и является либо *ограждением высвобождения* (`Release`), либо *ограждением получения*

(Acquire), либо и тем и другим (AcqRel или SeqCst). Помимо этого, ограждение SeqCst принимает участие в последовательно согласованном общем порядке.

Атомарное ограждение позволяет отделить упорядочивание памяти от атомарной операции. Это может пригодиться в том случае, если необходимо применить упорядочивание памяти к нескольким операциям или если требуется только его условное применение.

По сути, высвобождение-сохранение можно разбить на ограждение высвобождения, за которым следует расслабленное сохранение, а получение-загрузку можно разбить на расслабленную загрузку, за которой следует ограждение получения:

Сохранение при отношении высвобождения-получения

```
a.store(1, Release);
```

может быть заменено ограждением высвобождения с последующим расслабленным сохранением:

```
fence(Release);
a.store(1, Relaxed);
```

Загрузка при отношении высвобождения-получения

```
a.load(Acquire);
```

может быть заменена расслабленной загрузкой с последующим ограждением получения:

```
a.load(Relaxed);
fence(Acquire);
```

И все же наличие отдельного ограждения может вылиться в дополнительные инструкции процессора, снижая эффективность программы.

Но важнее то, что в отличие от высвобождения-сохранения или получения-загрузки, ограждение не привязано к какой-либо одной атомарной переменной. Значит, одно ограждение может использоваться сразу для нескольких переменных.

Формально ограждение высвобождения может заменить операцию высвобождения в отношении "происходит до", если за ограждением высвобождения следует (в том же потоке) любая атомарная операция, сохраняющая значение, наблюдаемое операцией получения, с которой происходит синхронизация. По аналогии с этим ограждение получения может заменить любую операцию получения, если этому ограждению получения предшествует (в том же потоке) любая атомарная операция, загружающая значение, сохраненное операцией высвобождения.

В совокупности это означает, что между ограждением высвобождения и ограждением получения формируется отношение "происходит до", если *любое* сохранение после ограждения высвобождения наблюдается *любой* загрузкой до ограждения получения.

Предположим, к примеру, что в одном потоке выполняется ограждение высвобождения, за которым следуют три атомарные операции сохранения, относящиеся к разным переменным, а в другом потоке выполняются три операции загрузки из этих же переменных с последующим ограждением получения:

Поток 1:

```
fence(Release);
A.store(1, Relaxed);
B.store(2, Relaxed);
C.store(3, Relaxed);
```

Поток 2:

```
A.load(Relaxed);
B.load(Relaxed);
C.load(Relaxed);
fence(Acquire);
```

В данной ситуации если *любая* из операций загрузки потока 2 загружает значение из соответствующей операции сохранения потока 1, то ограждение высвобождения потока 1 "происходит до" ограждения получения потока 2.

Ограждение не обязательно должно непосредственно предшествовать атомарным операциям или следовать за ними. Между ними может происходить что угодно, включая действия в рамках потока управления. Этим можно воспользоваться, чтобы сделать ограждение условным по аналогии с тем, как операции сравнения и обмена имеют упорядочение как в случае успешного, так и в случае неуспешного завершения.

Например, если указатель загружается из атомарной переменной с использованием упорядочения получения памяти, можно воспользоваться ограждением, чтобы применять упорядочение получения только в том случае, если указатель не содержит нулевого значения.

Использование высвобождения-загрузки:

```
let p = PTR.load(Acquire);
if p.is_null() {
    println!("no data");
} else {
    println!("data = {}", unsafe { *p });
}
```

Использование условного ограждения получения:

```
let p = PTR.load(Relaxed);
if p.is_null() {
    println!("no data");
} else {
    fence(Acquire);
    println!("data = {}", unsafe { *p });
}
```

Это может пригодиться при частом ожидании нулевого значения указателя, чтобы избежать получения памяти, когда в этом нет необходимости.

Рассмотрим более сложный случай использования ограждений высвобождения и получения (листинг 3.2).

Листинг 3.2

```
use std::sync::atomic::fence;
static mut DATA: [u64; 10] = [0; 10];
const ATOMIC_FALSE: AtomicBool = AtomicBool::new(false);
static READY: [AtomicBool; 10] = [ATOMIC_FALSE; 10];
fn main() {
    for i in 0..10 {
        thread::spawn(move || {
            let data = some_calculation(i);
            unsafe { DATA[i] = data };
            READY[i].store(true, Release);
        });
    }
    thread::sleep(Duration::from_millis(500));
    let ready: [bool; 10] = std::array::from_fn(|i| READY[i].load(Relaxed));
    if ready.contains(&true) {
        fence(Acquire);
```

```
        for i in 0..10 {
            if ready[i] {
                println!("data{i} = {}", unsafe { DATA[i] });
            }
        }
    }
}
```

`std::array::from_fn` — простой способ выполнения чего-то определенное число раз и сбора результатов в массив.

В этом примере 10 потоков выполняют вычисления и сохраняют результаты в неатомарной общей переменной. Каждый поток устанавливает атомарное логическое значение, указывающее, что данные готовы к чтению основным потоком с использованием обычной операции высвобождения-сохранения. Основной поток ждет полсекунды, проверяет все 10 логических значений, чтобы определить, какие потоки завершили выполнение, и выводит на экран готовые результаты.

Вместо того чтобы для чтения булевых значений выполнять 10 операций получения-загрузки, основной поток использует расслабленные операции и одно ограждение получения. Ограждение выполняется перед чтением данных, но только при наличии данных для чтения.

Хотя в данном конкретном примере приложение какие-либо усилий для подобной оптимизации может быть совершенно напрасным, эта схема способна сыграть важную роль для экономии накладных расходов на дополнительные операции получения при построении высокоэффективных конкурентных структур данных.

Ограждение `SeqCst` является одновременно и ограждением высвобождения, и ограждением получения (как и `AcqRel`), но при этом также представляет собой часть единого общего порядка последовательно согласованных операций. Но частью общего порядка является только ограждение, а не атомарные операции до или после него. В отличие от операций высвобождения или получения, последовательно согласованную операцию нельзя разбить на расслабленную операцию и ограждение памяти.

Компиляторные ограждения

В дополнение к обычному атомарному ограждению стандартная библиотека Rust также предоставляет компиляторное ограждение: `std::sync::atomic::compiler_fence`. Его сигнатура идентична сигнатуре обычной функции `fence()`, о которой говорилось ранее, но действует это ограждение только в компиляторе. В отличие от обычного атомарного ограждения, оно не мешает процессору, к примеру, переупорядочить инструкции. В подавляющем большинстве случаев использования компиляторных ограждений недостаточно.

Им можно будет воспользоваться при реализации *обработчика сигналов* в Unix или *прерывания* во встроенных системах. Это механизмы, способные внезапно прервать выполнение потока для временного выполнения несвязанной функции на том же самом процессорном ядре.

Поскольку все это происходит на одном и том же процессорном ядре, обычные способы, которыми процессор может повлиять на упорядочение памяти, не применяются. (Более подробно данная тема рассматривается в *главе 7*.) В таком случае может быть достаточно компиляторного ограждения, позволяющего сэкономить одну инструкцию и, возможно, повысить производительность.

Другой вариант использования связан с *барьерами памяти на уровне всего процесса*. Эта технология выходит за рамки модели памяти Rust и поддерживается только в некоторых операционных системах: в Linux через системный вызов `membarrier` и в Windows с использованием функции `FlushProcessWriteBuffers`. Она фактически позволяет потоку принудительно внедрять последовательно согласованное атомарное ограждение во все одновременно выполняющиеся потоки, что дает возможность замены двух одинаковых ограждений облегченным компиляторным ограждением с одной стороны и тяжеловесным барьером, охватывающим весь процесс, с другой стороны. Если код на стороне облегченного ограждения выполняется гораздо чаще, это может повысить общую производительность.

Более подробную информацию и кросс-платформенный способ применения таких барьеров в Rust можно найти в документации по крейту `membarrier` на сайте **crates.io**.

Компиляторное ограждение также может стать интересным инструментом для изучения влияния процессора на упорядочение памяти. В *разделе "Эксперимент" главы 7* программный код будет намеренно поврежден путем замены обычного ограждения на компиляторное. Это позволит прочувствовать едва заметные, но потенциально катастрофические последствия работы процессора при неверном упорядочении памяти.

Распространенные заблуждения

Упорядочение памяти богато на всяческие заблуждения. Прежде чем завершить данную главу, давайте рассмотрим наиболее распространенные из этих заблуждений.

Миф: Чтобы изменения стали "тут же" видны, требуется строгое упорядочение памяти.

Распространенным заблуждением является то, что использование такого расслабленного упорядочения памяти, как `Relaxed`, означает, что изменения атомарной переменной могут так никогда и не попасть в другой поток или же попасть в него только после существенной задержки. Название "расслабленный" может звучать так, будто ничего не происходит, пока что-то не заставит какую-то часть оборудования проснуться и сделать то, что она должна была сделать вместо расслабления.

На самом деле в модели памяти вообще ничего не говорится о синхронизации. В ней всего лишь определяется, в каком порядке происходят определенные события, а не то, как долго их придется ждать. Гипотетический компьютер, в котором для передачи данных из одного потока в другой требуются годы, совершенно непригоден для работы, но может прекрасно соответствовать модели памяти.

В реальности упорядочение памяти связано с такими вещами, как инструкции переупорядочения, которые обычно выполняются в наносекундном масштабе. Более строгое упорядочение памяти не ускоряет перемещение данных и может даже замедлить выполнение программы.

Миф: Отключение оптимизации означает, что при этом уже не нужно заботиться об упорядочении памяти.

И компилятор, и процессор играют немалую роль в том, что все происходит в ином порядке, чем можно было бы ожидать. Отключение оптимизации компилятора не устраняет все возможные преобразования в компиляторе и не отключает функции процессора, приводящие к переупорядочению инструкций и аналогичному потенциально проблемному поведению.

Миф: Использование процессора, не изменяющего порядок выполнения инструкций, означает, что уже не нужно заботиться об упорядочении памяти.

Некоторые простые процессоры, например в небольших микроконтроллерах, имеют только одно ядро и выполняют за один такт только одну инструкцию, и все инструкции выполняются ими по порядку. Но, несмотря на то, что на таких устройствах вероятность неверного упорядочения памяти, приводящего к реальным проблемам, значительно ниже, компилятор по-прежнему может делать неверные предположения на основе некорректного упорядочения памяти, ломая программный код. Кроме того, важно также понимать, что даже если процессор выполняет все инструкции по порядку, у него все равно могут быть другие функции, имеющие отношение к упорядочению памяти.

Миф: Расслабленные операции обходятся без издержек.

Так ли это, зависит от вашего определения слова "без издержек". То, что `Relaxed` — наиболее эффективный способ упорядочения памяти и с ним код может работать значительно быстрее, чем при остальных способах, сущая правда. Верно даже то, что на всех современных платформах расслабленные операции загрузки и сохранения компилируются в те же инструкции процессора, что и неатомарные операции чтения и записи.

Если атомарная переменная используется только одним потоком, любая разница в скорости с неатомарной переменной, скорее всего, будет связана с тем, что компилятор обладает большей свободой и более эффективно оптимизирует неатомарные операции. (Компиляторы склонны избегать большинства типов оптимизации атомарных переменных.)

Но доступ к одной и той же области памяти из нескольких потоков обычно происходит значительно медленнее, чем доступ к ней из одного потока. Поток, постоянно записывающий в атомарную переменную, скорее всего, будет испытывать заметное замедление, когда другие потоки начнут повторно считывать значение переменно, поскольку теперь должно начаться взаимодействие ядер процессора и их кеш-памяти.

Этот эффект будет исследован в *главе 7*.

Миф: Последовательно согласованное упорядочение памяти— отличный исходный и неизменно корректный вариант.

Если абстрагироваться от проблем с производительностью, последовательно согласованное упорядочение памяти из-за его твердых гарантий довольно часто рас-

сматривается в качестве идеального типа исходного упорядочения памяти. Правда в том, что *если* корректен любой другой способ упорядочения памяти, то корректен и `SeqCst`. Это может прозвучать так, будто `SeqCst` корректен всегда. Но вполне возможно, что независимо от способа упорядочения памяти просто некорректен алгоритм конкурентного вычисления.

Более важную роль играет то, что при чтении кода применение `SeqCst`, по сути, сообщает читателю: "данная операция зависит от общего порядка каждой отдельно взятой операции `SeqCst`, имеющейся в программе", что является невероятно далеко идущим утверждением. Тот же код, наверное, было бы легче просматривать и проверять, если `SeqCst` по возможности заменялось на более слабое упорядочение памяти. Например, `Release` фактически сообщает читателю: "это относится к операции получения той же переменной", что требует гораздо меньше внимания при разборе кода.

Целесообразно рассматривать `SeqCst` в качестве предупреждающего знака. Встреча с ним в естественных условиях означает, что либо в коде происходит что-то сложное, либо просто у автора не нашлось времени на анализ своих предположений, связанных с упорядочением памяти, что в обоих случаях является причиной для дополнительного исследования.

Миф: Последовательно согласованное упорядочение памяти может использоваться для "высвобождения-загрузки" или "получения-сохранения".

Хотя упорядочение `SeqCst` может послужить заменой `Acquire` или `Release`, его применение не является способом какого-то создания получения-сохранения или высвобождения-загрузки. Такие варианты так и остаются несуществующими. Высвобождение применяется только к операциям сохранения, а получение — только к операциям загрузки.

Например, `Release`-сохранение не образует никакого отношения высвобождения-получения с `SeqCst`-сохранением. Если нужно, чтобы они стали частью глобально согласованного порядка, обе операции должны будут использовать `SeqCst`.

Резюме

- Возможно, глобального согласованного порядка всех атомарных операций просто не существует, поскольку в разных потоках события могут происходить в различном порядке.
- Но каждая отдельно взятая атомарная переменная имеет свой собственный общий порядок модификации, независимо от упорядочения памяти, с которым согласны все потоки.
- Порядок операций формально определяется посредством отношений "происходит до".
- Внутри одного потока отношение "происходит до" существует между каждой отдельно взятой операцией.

- Порождение потока "происходит до" того, как будут выполнены все действия порожденного потока.
- Все, что делает поток, "происходит до" присоединения к этому потоку.
- Разблокировка мьютекса "происходит до" повторной блокировки этого мьютекса.
- Получение-загрузка значения из высвобождения-сохранения формирует отношение "происходит до". Это значение может быть изменено с помощью любого числа операций выборки-изменения и сравнения-обмена.
- Если бы существовала операция потребления-загрузки, она была бы облегченной версией операции получения-загрузки.
- Последовательно согласованное упорядочение приводит к глобально согласованному порядку выполнения операций, но в нем почти никогда нет потребности, и оно может усложнить проверку кода.
- Ограждения позволяют комбинировать упорядочение памяти для нескольких операций или применять условное упорядочение памяти.

Создание своей собственной спин-блокировки

Блокировка обычного мьютекса (см. *раздел "Блокировка: мьютексы и RwLock-блокировки" главы 1*) переводит поток в спящий режим, когда мьютекс уже заблокирован. Это позволяет не тратить понапрасну ресурсы в ходе ожидания разблокировки. При кратковременном удержании блокировки и возможности выполнения кода блокирующих потоков в параллельном режиме на разных ядрах процессора для потоков, наверное, будет лучше предпринимать многократные попытки заблокировать мьютекс без реального перехода в режим сна.

Спин-блокировка — это мьютекс, который именно так и делает. Попытка заблокировать уже заблокированный мьютекс приведет к *циклу занятости* или *прокрутке* (spinning): настойчивому повторению попыток вплоть до достижения успеха. Это может быть сопряжено с холостой тратой рабочих циклов процессора, но порой дает эффект снижения задержки, связанной с блокировкой.

На некоторых платформах многие фактические реализации мьютексов, включая `std::sync::Mutex`, прежде чем потребовать от операционной системы перевода потока в спящий режим кратковременно ведут себя как спин-блокировки. Насколько практична подобная попытка объединения лучшего из обоих подходов, полностью зависит от конкретного варианта использования.

В данной главе на основе материалов, изложенных в *главах 2* и *3*, будет рассмотрен способ создания своего собственного типа `SpinLock` и разобраны приемы использования системы типов Rust для предоставления безопасного и удобного интерфейса тем разработчикам, которые будут применять создаваемый здесь `SpinLock`.

Минималистичная реализация

Давайте создадим спин-блокировку с чистого листа.

Минималистичная версия исключительно проста и выглядит следующим образом:

```
pub struct SpinLock {
    locked: AtomicBool,
}
```

Понадобилось всего лишь одно логическое значение, указывающее, установлена блокировка или нет. Выбор *атомарного* логического значения обусловлен желанием предоставления возможности одновременного взаимодействия с ним сразу нескольким потокам.

В таком случае можно будет ограничиться функцией-конструктором и методами блокировки и разблокировки `lock` и `unlock`:

```
impl SpinLock {
    pub const fn new() -> Self {
        Self { locked: AtomicBool::new(false) }
    }
    pub fn lock(&self) {
        while self.locked.swap(true, Acquire) {
            std::hint::spin_loop();
        }
    }
    pub fn unlock(&self) {
        self.locked.store(false, Release);
    }
}
```

Изначально логическая переменная `locked` имеет значение `false`, а блокировка заменяет его на `true` и продолжает попытки замены, если у нее уже имелось значение `true`, а метод `unlock` просто возвращает значение обратно в `false`.

Операцию обмена можно также заменить операцией сравнения и обмена, чтобы в атомарном режиме проверять, имеет ли логическая переменная значение `false`, и при положительном исходе проверки устанавливать для нее значение `true`:

```
while self.locked.compare_exchange_weak(
    false, true, Acquire, Relaxed).is_err()
```

Получится немного длиннее, но интереснее для тех, кто предпочитает более понятный код, четко отражающий концепцию операции, способной завершиться либо неудачей, либо успехом. Но в *главе 7* будет показано, что данный вариант может потребовать применения несколько иных инструкций.

В цикле `while` присутствует *подсказка о применении спин-цикла*, сообщающая процессору о прокрутке в ожидании изменений. На большинстве наиболее распространенных платформ эта подсказка приводит к созданию специальной инструкции, заставляющей ядро процессора подстроить свое поведение под такую ситуацию. К примеру, оно может временно замедлить работу или установить приоритет других возможных полезных действий. Но, в отличие от таких операций блокировки, как `thread::sleep` или `thread::park`, подсказка о проведении спин-цикла не приводит к вызову операционной системы для перевода вашего потока в спящий режим в пользу активности другого потока.

Вообще-то включение такой подсказки в спин-цикл — идея неплохая. В зависимости от ситуации, прежде чем повторить попытку получения доступа к атомарной переменной, было бы даже полезно вставлять такую подсказку несколько раз. Если стремиться к сохранению уровня производительности за счет экономии всего нескольких наносекунд и поиска оптимальной стратегии, придется подстраиваться под получающийся конкретный вариант использования. К сожалению, как выяснится в *главе 7*, результаты таких тестов производительности могут сильно зависеть от имеющегося компьютерного оснащения.

Для формирования отношения "происходит до" между каждым вызовом `unlock()` и последующими вызовами `lock()` воспользуемся упорядочением получения и высвобождения памяти. Иными словами, следует твердо увериться в том, что все происходившее со времени последней блокировки уже произошло. Получение и снятие блокировки — самый наглядный вариант использования упорядочения получения и высвобождения.

Ситуация, при которой создаваемый нами объект `SpinLock` служит для защиты доступа к каким-то общим данным в условиях одновременно предпринимаемых попыток получения блокировки двумя потоками, показана на рис. 4.1. Заметьте, что операция разблокировки в первом потоке формирует отношение "происходит до" с операцией блокировки во втором потоке, чем гарантируется невозможность одновременного получения доступа к данным из обоих потоков.

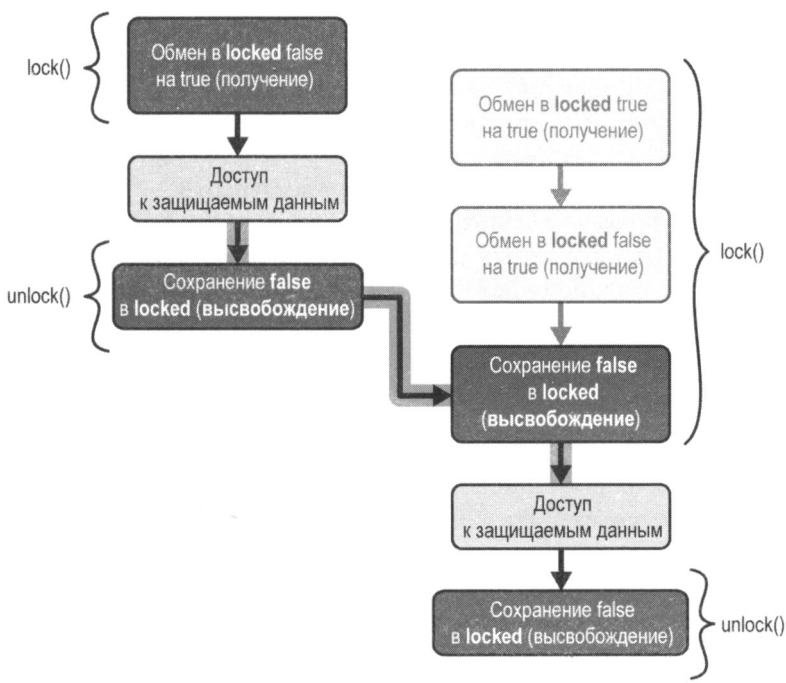

Рис. 4.1. Отношение "произошло до" между двумя потоками, применяющими `SpinLock` для защиты доступа к совместно используемым данным

Небезопасная спин-блокировка

У созданного ранее типа `SpinLock` абсолютно безопасный интерфейс, поскольку сам по себе при неправильном применении он не становится причиной какого-либо неопределенного поведения. Но чаще всего он будет востребован для защиты от изменений совместно используемой переменной, следовательно, пользователю все равно потребуется небезопасный, неконтролируемый код.

Для предоставления более простого интерфейса можно изменить метод `lock`, чтобы он предоставлял на данные, защищенные блокировкой, исключительную ссылку (`&mut T`), поскольку в большинстве случаев возможность безопасно выдвигать предположение об исключительности доступа гарантируется именно операцией блокировки.

Это требует замены защищаемого типа данных на обобщенный и добавления поля для хранения защищаемых данных. Поскольку данные могут быть изменены (или же доступны в исключительном режиме) даже при совместно используемой спин-блокировке, необходимо воспользоваться внутренней изменяемостью (см. *раздел "Внутренняя изменяемость" главы 1*), для чего будет применяться `UnsafeCell`:

```
use std::cell::UnsafeCell;
pub struct SpinLock<T> {
    locked: AtomicBool,
    value: UnsafeCell<T>,
}
```

В качестве меры предосторожности в `UnsafeCell` не реализуется типаж `Sync`, т. е. создаваемый тип больше не может использоваться потоками совместно, что делает его абсолютно бесполезным. Чтобы выправить ситуацию, нужно заверить компилятор в том, что совместное использование потоками данного типа не представляет никакой опасности. Но, поскольку блокировкой можно воспользоваться для отправки значений типа `T` из одного потока в другой, следует свести это заверение только к тем типам, которые можно будет безопасно пересылать между потоками. Небезопасная реализация `Sync` для `SpinLock<T>` для всех `T`, реализующих `Send`, осуществляется следующим образом:

```
unsafe impl<T> Sync for SpinLock<T> where T: Send {}
```

Заметьте, что требовать от `T` реализации `Sync` не нужно, поскольку создаваемый `SpinLock<T>` позволит в любой момент времени получить доступ к защищаемому им `T` только лишь одному потоку. Надобность в `T: Sync` возникла бы (в дополнении ко всему остальному) только при предоставлении одновременного доступа сразу нескольким потокам, как это бывает при блокировке чтения-записи для кода, выполняющего чтение данных.

Далее, для инициализации `UnsafeCell` функция `new` должна получить значение типа `T`:

```
impl<T> SpinLock<T> {
    pub const fn new(value: T) -> Self {
        Self {
            locked: AtomicBool::new(false),
            value: UnsafeCell::new(value),
        }
    }
    ...
}
```

И теперь мы подходим к самому интересному: к `lock` и `unlock`. Ведь все делается ради получения возможности возвращать `&mut T` из `lock()`, чтобы пользователю,

применяющему нашу блокировку для защиты своих данных, не приходилось создавать небезопасный, непроверяемый код. Получается, что теперь небезопасный код придется задействовать на нашей стороне в рамках реализации `lock`. Метод `get()`, принадлежащий `UnsafeCell`, может предоставить на содержимое (`*mut T`) простой указатель, который можно преобразовать в ссылку внутри `unsafe`-блока:

```
pub fn lock(&self) -> &mut T {
    while self.locked.swap(true, Acquire) {
        std::hint::spin_loop();
    }
    unsafe { &mut *self.value.get() }
}
```

Поскольку сигнатура функции `lock` содержит ссылку как на входе, так и на выходе, время жизни `&self` и `&mut T` не указывается и считается идентичным[1]. Время жизни можно самостоятельно указать явным образом:

```
pub fn lock<'a>(&'a self) -> &'a mut T { ... }
```

Здесь четко указано, что время жизни у возвращаемой ссылки такое же, как и у `&self`. Следовательно, мы заявляем, что возвращаемая ссылка действительна до тех пор, пока существует сама блокировка.

Если притвориться, что метода `unlock()` не существует, то интерфейс приобретет абсолютную надежность и безопасность. `SpinLock` быть переведен в режим блокировки, что приведет к возвращению `&mut T`, после чего никогда не сможет быть заблокирован повторно, гарантируя тем самым абсолютную исключительность ссылки.

Но если снова вернуться к применению метода `unlock()`, понадобится способ ограничения времени жизни возвращаемой ссылки до очередного вызова `unlock()`. Если бы компилятор понимал английский язык, то, возможно, сработал бы следующий код:

```
pub fn lock<'a>(&self) -> &'a mut T
where
    'a ends at the next call to unlock() on self, even if that's done by another thread.
    Oh, and it also ends when self is dropped, of course.
    (Thanks!)                                                    ❶
{ ... }
```

❶ `a` продолжает существовать до очередного вызова в отношении `self` метода `unlock()`, даже если это будет сделано другим потоком. И конечно же, жизнь `a` заканчивается при удалении `self`. (Спасибо!)

К сожалению, сделать так в Rust не получится. Объяснять суть ограничения придется не компилятору, а пользователю. Чтобы переложить ответственность на пользователя, функция `unlock` получает отметку `unsafe`, а пользователю рассказывается, что нужно сделать, чтобы все было в порядке:

---

[1] См. раздел "Lifetime Elision" в "Chapter 10: Generic Types, Traits, and Lifetimes" книги "Rust Book".

```
/// Безопасно: ссылка &mut T из lock() должна исчезнуть!
/// (И никакого обмана с хранением где-либо ссылки на поля этого T!)
pub unsafe fn unlock(&self) {
    self.locked.store(false, Release);
}
```

Безопасный интерфейс с использованием хранителя блокировки

Чтобы гарантировать абсолютную безопасность интерфейса, операцию разблокировки нужно привязать к завершению существования `&mut T`. Сделать это можно, обернув ссылку в наш собственный тип с поведением ссылки, но с наличием реализации типажа `Drop` для совершения каких-либо действий при его удалении.

Такой тип часто называют *хранителем*, поскольку он фактически хранит в себе состояние блокировки и несет ответственность за это состояние вплоть до своего удаления.

В создаваемом типе `Guard` будет просто содержаться ссылка на `SpinLock`, позволяющая ему как получать доступ к `UnsafeCell`, так и впоследствии удалить `AtomicBool`:

```
pub struct Guard<T> {
    lock: &SpinLock<T>,
}
```

Но при попытке скомпилировать данный код, компилятор выдаст ошибку:

```
error[E0106]: missing lifetime specifier                    ❶
    --> src/lib.rs
     |
     |         lock: &SpinLock<T>,
     |               ^ expected named lifetime parameter     ❷
     |
help: consider introducing a named lifetime parameter       ❸
     |
     ~    pub struct Guard<'a, T> {
     |                     ^^^
     ~        lock: &'a SpinLock<T>,
     |               ^^
     |
```

❶ Ошибка `[E0106]`: отсутствует спецификатор времени жизни.

❷ Ожидаемый именованный параметр времени жизни.

❸ Справка: изыщите возможность введения именованного параметра времени жизни.

Похоже, проигнорировать указание времени жизни здесь не удастся. Нужно согласиться с предложением компилятора и обозначить ограниченность времени жизни ссылки явным образом:

```
pub struct Guard<'a, T> {
    lock: &'a SpinLock<T>,
}
```

Это даст гарантию того, что `Guard` не сможет пережить `SpinLock`.

Далее следует изменить метод `lock` создаваемого `SpinLock`, чтобы он возвращал `Guard`:

```
pub fn lock(&self) -> Guard<T> {
    while self.locked.swap(true, Acquire) {
        std::hint::spin_loop();
    }
    Guard { lock: self }
}
```

Конструктор у нашего типа `Guard` отсутствует, и его поле является закрытым, поэтому получить `Guard` пользователь сможет только этим способом. Следовательно, можно смело предполагать, что наличие `Guard` означает нахождение `SpinLock` в заблокированном состоянии.

Чтобы сформировать у `Guard<T>` поведение исключительной ссылки, явно предоставляющей доступ к `T`, нужно реализовать специальные свойства `Deref` и `DerefMut`:

```
use std::ops::{Deref, DerefMut};
impl<T> Deref for Guard<'_, T> {
    type Target = T;
    fn deref(&self) -> &T {
        // Безопасно: сам факт существования данного типа Guard
        // гарантирует монопольное хранение блокировки.
        unsafe { &*self.lock.value.get() }
    }
}
impl<T> DerefMut for Guard<'_, T> {
    fn deref_mut(&mut self) -> &mut T {
        // Безопасно: сам факт существования данного типа Guard
        // гарантирует монопольное хранение блокировки.
        unsafe { &mut *self.lock.value.get() }
    }
}
```

И напоследок нужно будет реализовать `Drop` для `Guard`, что позволит полностью отказаться от небезопасного метода `unlock`:

```
impl<T> Drop for Guard<'_, T> {
    fn drop(&mut self) {
        self.lock.locked.store(false, Release);
    }
}
```

Вот так, благодаря магии `Drop` и имеющейся в Rust системы типов, создаваемому типу `SpinLock` предоставлен полностью безопасный (и удобный) интерфейс.

Давайте посмотрим, как все это работает (листинг 4.1).

Листинг 4.1

```
fn main() {
    let x = SpinLock::new(Vec::new());
    thread::scope(|s| {
        s.spawn(|| x.lock().push(1));
        s.spawn(|| {
            let mut g = x.lock();
            g.push(2);
            g.push(2);
        });
    });
    let g = x.lock();
    assert!(g.as_slice() == [1, 2, 2] || g.as_slice() == [2, 2, 1]);
}
```

Эта программа демонстрирует легкость применения созданной спин-блокировки `SpinLock`. Реализация `Deref` и `DerefMut` дает возможность вызвать находящийся под защитой метод `Vec::push` напрямую. А наличие `Drop` позволяет не беспокоиться о разблокировке.

Уничтожив хранителя путем вызова функции `drop(g)`, можно осуществить явную разблокировку. При попытке слишком ранней разблокировки компилятор сразу выдаст ошибку, что позволит убедиться в выполнении хранителем блокировки своей работы. К примеру, если вставить `drop(g);` между двумя строками `push(2)`, вторая инструкция `push` не скомпилируется, поскольку к тому моменту `g` уже будет удален:

```
error[E0382]: borrow of moved value: 'g'                    ❶
    --> src/lib.rs
     |
     |     drop(g);
     |          - value moved here                          ❷
     |     g.push(2);
     |     ^^^^^^^^^ value borrowed here after move          ❸
```

❶ Ошибка `[E0382]`: заимствование перемещенного значения: 'g'.

❷ Здесь значение было перемещено.

❸ А здесь значение было позаимствовано после его перемещения.

Благодаря имеющейся в Rust системе типов можно с уверенностью утверждать, что подобные ошибки будут обнаружены еще до появления возможности запуска создаваемой программы.

Резюме

- *Спин-блокировка* представляет собой мьютекс в цикле занятости или в прокрутке в ходе ожидания.
- Прокрутка способна сократить задержку, но может также стать причиной холостой траты рабочих циклов процессора и снижения производительности.
- Для информирования процессора о спин-цикле, способного повысить эффективность его работы, может использоваться *подсказка о применении спин-цикла* — `std::hint::spin_loop()`.
- `SpinLock<T>` можно реализовать только с помощью типов `AtomicBool` и `UnsafeCell<T>`, последний из которых необходим для обеспечения *внутренней изменяемости* (см. *раздел "Внутренняя изменяемость" главы 1*).
- *Отношение "происходит до"* между операциями разблокировки и блокировки необходимо для предотвращения *гонки данных*, которая может привести к *неопределенному поведению*.
- В рассматриваемые здесь сценарии идеально вписывается *упорядочение получения и высвобождения памяти*.
- При непроверенных предположениях, призванных избежать неопределенного поведения, ответственность, путем объявления функции небезопасной (`unsafe`), может быть переложена на вызывающую сторону.
- Чтобы сформировать у типа поведение ссылки, явно предоставляющей доступ к другому объекту, можно воспользоваться типажами `Deref` и `DerefMut`.
- Для выполнения каких-либо действий при удалении объекта, например когда он выходит за пределы области видимости или передается в `drop()`, можно воспользоваться типажом `Drop`.
- Хранитель блокировки — это весьма полезный шаблон проектирования специального типа, предназначенный для представления безопасного доступа к хранимой блокировке. Благодаря реализации типажа `Deref` такой тип обычно ведет себя аналогично ссылке, а за счет типажа `Drop` в нем реализуется автоматическая разблокировка.

ГЛАВА 5

Создание своих собственных каналов

Каналы предназначены для передачи данных между потоками, и у них есть множество вариантов. Одни каналы могут использоваться только одним отправителем и одним получателем, а другие могут отправлять данные из любого количества потоков или даже допускать наличие нескольких получателей. Бывают блокирующие каналы, т. е. получение (а иногда и отправка данных) является блокирующей операцией, переводящей ваш поток в режим сна вплоть до завершения операции. Существуют каналы, оптимизированные под определенную пропускную способность, и каналы, обеспечивающие низкую задержку.

Варианты просто неисчислимы, а универсальной версии, подходящей для всех ситуаций, просто не существует.

В данной главе будет рассмотрена реализация нескольких относительно простых каналов, чтобы не только дополнительно изучить ряд новых вариантов применения атомарности, но и расширить познания о том, как предъявляемые требования и предположения могут быть отражены в системе типов Rust.

Простой канал на основе мьютекса

Базовая реализация канала не требует каких-либо знаний атомарности. Можно взять тип `VecDeque`, являющийся по сути типом `Vec`, позволяющим оперативно добавлять и удалять элементы на обоих концах, и защищать данные с помощью `Mutex`, открывая доступ к ним сразу нескольким потокам. Затем `VecDeque` будет использоваться в качестве очереди данных, часто называемых отправленными, но еще не полученными *сообщениями*. Любой поток, желающий отправить сообщение, может просто добавить его в конец очереди, а любой поток, желающий получить сообщение, просто должен удалить его из начала очереди.

Остается добавить еще один элемент, предназначенный для блокировки операции получения: `Condvar` (см. *раздел "Условные переменные" главы 1*), служащий для уведомления ожидающих получателей о новом сообщении.

Реализация всего этого может быть весьма лаконичной и относительно простой (листинг 5.1).

Листинг 5.1

```
pub struct Channel<T> {
    queue: Mutex<VecDeque<T>>,
```

```
    item_ready: Condvar,
}
impl<T> Channel<T> {
    pub fn new() -> Self {
        Self {
            queue: Mutex::new(VecDeque::new()),
            item_ready: Condvar::new(),
        }
    }
    pub fn send(&self, message: T) {
        self.queue.lock().unwrap().push_back(message);
        self.item_ready.notify_one();
    }
    pub fn receive(&self) -> T {
        let mut b = self.queue.lock().unwrap();
        loop {
            if let Some(message) = b.pop_front() {
                return message;
            }
            b = self.item_ready.wait(b).unwrap();
        }
    }
}
```

Заметьте, что здесь не пришлось применять какой-либо атомарный или небезопасный код и задумываться о типажах `Send` или `Sync`. Компилятор хорошо разбирается в интерфейсе `Mutex` и в предоставляемых этим типом гарантиях и предполагает, что безопасное совместное использование потоками `Mutex<T>` и `Condvar` можно перенести и на `Channel<T>`.

Создаваемая функция `send` блокирует мьютекс, чтобы поместить новое сообщение в конец очереди, и после разблокировки очереди сразу же уведомляет одного потенциально ожидающего получателя, задействуя условную переменную.

Функция `receive` также блокирует мьютекс, чтобы извлечь очередное сообщение из начала очереди, но будет использовать условную переменную для ожидания поступления доступного сообщения.

Следует помнить, что метод `Condvar::wait` разблокирует `Mutex` в процессе ожидания и снова заблокирует его перед возвращением управления. Следовательно, создаваемая функция `receive` не будет удерживать блокировку мьютекса в ходе ожидания.

Хотя созданный канал весьма гибок в использовании и допускает любое число потоков отправки и получения, его реализация во многих ситуациях может быть далека от оптимальной. Даже при наличии множества готовых к приему сообщений, любая операция отправки или получения на короткое время заблокирует любую другую операцию отправки или получения, поскольку все они должны блокировать

один и тот же мьютекс. Если `VecDeque::push` должен увеличить емкость `VecDeque`, то всем потокам отправки и получения придется ждать, пока этот поток завершит перераспределение, что в целом ряде ситуаций может быть неприемлемо.

Нежелательной может быть и еще одна особенность: неограниченный рост очереди. Отправителям ничто не мешает постоянно помещать новые сообщения со скоростью, превышающей скорость их обработки получателями.

Небезопасный одноразовый канал

Вариантов использования каналов бесчисленное множество. Но далее в главе будет уделено внимание конкретной ситуации: отправке только одного сообщения из одного потока в другой. Такой вариант часто называют *одноразовым каналом*.

Можно было взять рассмотренную в листинге 5.1 реализацию на основе `Mutex<VecDeque>` и заменить `VecDeque` на `Option`, уменьшив емкость очереди до одного сообщения. Тем самым удалось бы избежать выделения больших объемов памяти, но негативные моменты от наличия `Mutex` при этом никуда не делись бы. Избавиться от них можно, создав с нуля свой собственный одноразовый канал с применением атомарности.

Давайте сначала создадим минимальную реализацию одноразового канала без совершенного интерфейса. А улучшением пользовательского восприятия канала за счет совершенствования его интерфейса и взаимодействия с системой типов Rust мы займемся чуть позже.

Исходные инструменты будут практически теми же, что и для `SpinLock<T>` (см. *главу 4*): `UnsafeCell` для хранения и `AtomicBool` для указания состояния. Здесь атомарное логическое значение будет предназначено для указания готовности сообщения к использованию.

Перед отправкой сообщения канал будет "пустым", т. е. не содержащим сообщений типа `T`. Чтобы обозначить отсутствие в ячейке `T`, можно было бы воспользоваться `Option<T>`. Но это может привести к бесполезной трате ценных ресурсов памяти, поскольку атомарное логическое значение уже способно сообщить, есть сообщение или его еще нет. Лучше воспользоваться типом `std::mem::MaybeUninit<T>`, представляющим, по сути, небезопасную версию `Option<T>`. Здесь потребуется, чтобы пользователь самостоятельно отслеживал, была ли выполнена инициализация, и практически весь его интерфейс будет небезопасен, поскольку он не сможет выполнять собственные проверки.

Исходя из всех этих соображений, давайте предпримем первую попытку со следующего определения структуры:

```
use std::mem::MaybeUninit;
pub struct Channel<T> {
   message: UnsafeCell<MaybeUninit<T>>,
   ready: AtomicBool,
}
```

Как и в созданном ранее SpinLock<T>, компилятору следует сообщить, что создаваемый канал безопасен для совместного использования потоками или же, по крайней мере, безопасен, пока в T имеется реализация Send:

```
unsafe impl<T> Sync for Channel<T> where T: Send {}
```

Новый канал пуст, для ready установлено значение false, а инициализация message еще не произошла:

```
impl<T> Channel<T> {
    pub const fn new() -> Self {
        Self {
            message: UnsafeCell::new(MaybeUninit::uninit()),
            ready: AtomicBool::new(false),
        }
    }
    ...
}
```

Пред отправкой сообщение необходимо сохранить в ячейке, после чего его можно будет передать его получателю, установив для флага готовности ready значение true. Повторная попытка этих действий может быть опасной, поскольку после установки флага ready получатель способен прочитать сообщение в любой момент, что может привести к гонке со второй попыткой отправки сообщения.

Пока что ответственность за происходящее возлагается на пользователя, для чего метод объявлен небезопасным, unsafe, и для него оставлено примечание:

```
/// Безопасно: Но только при однократном вызове!
pub unsafe fn send(&self, message: T) {
     (*self.message.get()).write(message);
    self.ready.store(true, Release);
}
```

В приведенном ранее фрагменте для получения указателя на MaybeUninit<T> предусмотрен метод UnsafeCell::get, и с целью инициализации выполняется небезопасное разыменование для вызова MaybeUninit::write. При некорректном использовании может возникнуть неопределенное поведение, но ответственность за правильные действия возложена на вызывающую сторону.

Для упорядочивания памяти нужно воспользоваться упорядочением высвобождения, поскольку атомарное хранилище фактически отправляет сообщение получателю. Тем самым гарантируется завершение инициализации с позиции принимающего потока, если из self.ready с упорядочиванием получения загружается true.

Предоставлять получению интерфейс блокировки не будем. Лучше предоставим два метода: один для проверки доступности сообщения, а другой — для его получения. Возможность воспользоваться чем-то вроде парковки потоков (см. *раздел "Парковка потоков" главы 1*) при желании воспользоваться блокировкой оставим на усмотрение пользователей создаваемого канала.

Эти два метода и завершают данную версию создаваемого канала:

```
pub fn is_ready(&self) -> bool {
    self.ready.load(Acquire)
}
/// Безопасно: Но только при однократном вызове
/// и только после того, как is_ready() возвратит true!
pub unsafe fn receive(&self) -> T {
    (*self.message.get()).assume_init_read()
}
```

Притом что вызов метода `is_ready` может быть безопасен всегда, в методе `receive` присутствует объект `MaybeUninit::assume_init_read()`, выдвигающий небезопасное предположение о том, что инициализация уже состоялась и что он не используется для создания нескольких копий объектов, не имеющих реализации типажа `Copy`. Здесь, как и в случае с `send`, функция просто делается небезопасной, оставляя решение возможных проблем на усмотрение пользователя.

В результате получается технически пригодный канал из разряда трудноуправляемых и ничем особо не впечатляющих. В стандартной ситуации он делает именно то, что должен, но есть множество нюансов, допускающих ненадлежащее использование этого канала.

Вызов `send` более одного раза может привести к гонке данных, поскольку второй отправитель способен перезаписывать данные в тот самый момент, когда получатель предпринимает попытку чтения первого сообщения. Даже при правильной синхронизации получения сообщения вызов метода `send` сразу из нескольких потоков может привести к одновременной попытке записи в ячейку, что опять же повлечет гонку данных. Кроме того, вызов метода `receive` более одного раза приводит к созданию двух копий сообщения, даже если в `T` не имеется реализации типажа `Copy` и, следовательно, копирование не может пройти безопасным образом.

Менее очевидной проблемой является отсутствие для создаваемого канала реализации типажа `Drop`. Тип `MaybeUninit` не отслеживает, была инициализация или нет, поэтому он автоматически не удаляет свое содержимое. Следовательно, если сообщение отправлено, но так и не получено, оно никогда не будет удалено. И пусть в этом нет ничего плохого, но все же следует избегать такого развития событий. Хотя утечка памяти в Rust вообще-то считается безопасной, обычно она допускается при совершенно иных обстоятельствах. Например, при утечке памяти, связанной с типом `Vec`, утрачивается также и его содержимое, но при должном использовании `Vec` никаких утечек памяти не возникает.

Поскольку ответственность за всевозможные неблагоприятные моменты мы возложили на пользователя, то их проявление является всего лишь вопросом времени.

Безопасность, достигаемая проверками в ходе выполнения

Чтобы повысить безопасность интерфейса, можно добавить ряд проверок для вызова паники с четким разъяснением причины в случае недопустимого использования канала, что будет гораздо предпочтительнее неопределенного поведения.

Начнем с вопроса о вызове метода `receive` до готовности сообщения. Справиться с этим довольно легко, поскольку вполне достаточно заставить метод `receive` предварять попытку чтения сообщения проверкой флага `ready`:

```
/// Вызов паники, если сообщение еще не готово.
/// Подсказка: воспользуйтесь сначала для проверки методом 'is_ready'.
///
/// Безопасно: Но только при однократном вызове!
pub unsafe fn receive(&self) -> T {
    if !self.ready.load(Acquire) {
        panic!("no message available!");
    }
    (*self.message.get()).assume_init_read()
}
```

Функция остается небезопасной, поскольку пользователь по-прежнему несет ответственность за то, чтобы она не вызывалась более одного раза, но неопределенное поведение из-за отсутствия предварительной проверки `is_ready()`теперь уже не возникает.

Поскольку теперь обеспечивающая необходимую синхронизацию операция получения-загрузки флага готовности выполняется внутри метода `receive`, степень упорядочения загрузки в память в `is_ready` может быть понижена до `Relaxed`, т. к. это теперь служит только иллюстративным целям:

```
pub fn is_ready(&self) -> bool {
    self.ready.load(Relaxed)
}
```

Следует помнить, что *общий порядок модификации* (см. *раздел "Расслабленное упорядочение" главы 3*), применимый к `ready`, гарантирует, что после того, как `is_ready` загрузит из него `true`, метод `receive` также увидит `true`. Невозможно представить ситуацию, при которой `is_ready`, независимо от реализованного в нем упорядочения памяти, возвращал бы `true`, а метод `receive()` продолжал паниковать.

А теперь нужно разобраться с тем, что происходит при вызове `receive` более одного раза. Все это можно просто обратить в панику, вернув флагу `ready` при выполнении метода `receive` значение `false`, например:

```
/// Вызов паники, если сообщение еще не готово или уже принято.
/// Подсказка: воспользуйтесь сначала для проверки методом 'is_ready'.
pub fn receive(&self) -> T {
    if !self.ready.swap(false, Acquire) {
        panic!("no message available!");
    }
```

```
    // Безопасно: Только что был проверен (и сброшен) флаг ready.
    unsafe { (*self.message.get()).assume_init_read() }
}
```

Для возвращения `false` операция `load` была просто заменена на `swap`, тут же превратив метод `receive` в полностью безопасный для вызова в любых условиях. Функция больше не помечается как `unsafe`. Теперь уже ответственность за небезопасный код полностью возлагается не на пользователя, а на нас самих, что существенно снижает уровень пользовательского дискомфорта.

А вот с функцией `send` не все так просто. Чтобы предотвратить одновременные множественные вызовы `send` в отношении одной и той же ячейки, нужно знать, не начался ли уже другой вызов `send`. Флаг `ready` позволяет лишь узнать, не завершился ли уже другой вызов `send`, но этого нам недостаточно.

Давайте добавим еще один флаг по имени `in_use`, указывающий на использование канала:

```
pub struct Channel<T> {
    message: UnsafeCell<MaybeUninit<T>>,
    in_use: AtomicBool, // Новый!
    ready: AtomicBool,
}
impl<T> Channel<T> {
    pub const fn new() -> Self {
        Self {
            message: UnsafeCell::new(MaybeUninit::uninit()),
            in_use: AtomicBool::new(false), // Новый!
            ready: AtomicBool::new(false),
        }
    }
    ...
}
```

Теперь остается установить для `in_use` значение `true` в методе `send` перед доступом к ячейке и запаниковать, если это значение уже было установлено другим вызовом:

```
/// Вызов паники при попытке отправки более одного сообщения.
pub fn send(&self, message: T) {
    if self.in_use.swap(true, Relaxed) {
        panic!("can't send more than one message!");
    }
    unsafe { (*self.message.get()).write(message) };
    self.ready.store(true, Release);
}
```

Для операции атомарного обмена можно воспользоваться расслабленным упорядочением памяти, поскольку *общий порядок модификации* (см. *раздел "Расслабленное упорядочение" главы 3*) `in_use` гарантирует, что в отношении `in_use` будет проведена только одна операция обмена, возвращающая `false`, и это будет единственный случай, при котором `send` попытается получить доступ к ячейке.

Вот теперь получился полностью безопасный интерфейс, хотя осталась нерешенной еще одна проблема: при отправке сообщения, которое не будет получено, оно так никогда и не будет удалено. Пусть к неопределенному поведению данная ситуация не приводит и она вполне допустима в безопасном коде, но этого все же следует избегать.

Поскольку флаг `ready` сбрасывается в методе `receive` приема, исправить это нетрудно: флаг `ready` указывает на наличие в ячейке еще не полученного сообщения, которое необходимо удалить.

В реализации типажа `Drop` создаваемого нами `Channel` для проверки атомарного флага `ready` атомарная операция совсем не обязательна, поскольку объект может быть удален только в том случае, если он полностью принадлежит тому потоку, который его удаляет, без каких-либо действующих заимствований. Это позволяет воспользоваться методом `AtomicBool::get_mut`, принимающим исключительную ссылку (`&mut self`), подтверждая тем самым бессмысленность атомарного доступа. Те же рассуждения справедливы и для `UnsafeCell` с использованием метода `UnsafeCell::get_mut`.

Исходя из этого, завершающая часть создаваемого нами полностью безопасного и совершенно не склонного к утечкам канала приобретает следующий вид:

```
impl<T> Drop for Channel<T> {
    fn drop(&mut self) {
        if *self.ready.get_mut() {
            unsafe { self.message.get_mut().assume_init_drop() }
        }
    }
}
```

Давайте испытаем его на практике!

Поскольку созданный канал пока что не предоставляет блокирующий интерфейс, парковка потоков в ожидании сообщения будет выполняться самостоятельно. Поток-получатель сам будет вызывать `park()` до готовности сообщения, а поток-отправитель будет применять к получателю `unpark()`, как только что-нибудь отправит.

В листинге 5.2 приведен полный код тестовой программы, отправляющей строковый литерал `"hello world!"` через наш канал из второго потока в основной поток.

Листинг 5.2

```
fn main() {
    let channel = Channel::new();
    let t = thread::current();
    thread::scope(|s| {
        s.spawn(|| {
            channel.send("hello world!");
            t.unpark();
        });
```

```
        while !channel.is_ready() {
            thread::park();
        }
        assert_eq!(channel.receive(), "hello world!");
    });
}
```

Программа компилируется, запускается и завершает работу без ошибок, показывая, что `Channel` работает так, как надо.

Если продублировать строку кода `send`, можно будет увидеть в действии одну из проверок безопасности, выдающую следующее паническое сообщение при запуске программы:

```
thread '<unnamed>' panicked at 'can't send more than one message!', src/main.rs ❶
```

❶ Поток `'<unnamed>'` запаниковал `'не могу отправить более одного сообщения!'`, `src/main.rs`.

Хотя паникующая программа — это не верх совершенства, но любой программе гораздо лучше будет поднять панику, чем вплотную столкнуться со всеми ужасами неопределенного поведения.

Использование одной и той же атомарности для предоставления состояния канала

Если от реализации каналов хочется добиться чего-то большего, есть один хитрый прием, помогающий сэкономить один байт памяти.

Вместо двух отдельных атомарных логических значений, применяемых для представления состояния канала, давайте для представления всех четырех состояний воспользуемся одним значением типа `AtomicU8`. Теперь вместо атомарной замены логических значений придется воспользоваться `Compare_exchange` для проверки в атомарном режиме нахождения канала в ожидаемом состоянии и перевода его в другое состояние:

```
const EMPTY: u8 = 0;
const WRITING: u8 = 1;
const READY: u8 = 2;
const READING: u8 = 3;
pub struct Channel<T> {
    message: UnsafeCell<MaybeUninit<T>>,
    state: AtomicU8,
}
unsafe impl<T: Send> Sync for Channel<T> {}
impl<T> Channel<T> {
    pub const fn new() -> Self {
        Self {
            message: UnsafeCell::new(MaybeUninit::uninit()),
            state: AtomicU8::new(EMPTY),
        }
    }
```

```
    pub fn send(&self, message: T) {
        if self.state.compare_exchange(
            EMPTY, WRITING, Relaxed, Relaxed
        ).is_err() {
            panic!("can't send more than one message!");
        }
        unsafe { (*self.message.get()).write(message) };
        self.state.store(READY, Release);
    }
    pub fn is_ready(&self) -> bool {
        self.state.load(Relaxed) == READY
    }
    pub fn receive(&self) -> T {
        if self.state.compare_exchange(
            READY, READING, Acquire, Relaxed
        ).is_err() {
            panic!("no message available!");
        }
        unsafe { (*self.message.get()).assume_init_read() }
    }
}
impl<T> Drop for Channel<T> {
    fn drop(&mut self) {
        if *self.state.get_mut() == READY {
            unsafe { self.message.get_mut().assume_init_drop() }
        }
    }
}
```

Обеспечение безопасности с использованием системы типов

Успешная защита пользователей `Channel` от неопределенного поведения не исключила риска впадения в панику при случайном некорректном использовании канала. В идеале компилятор должен проверять, насколько правильно задействован канал, и указывать на недопустимые ситуации еще до запуска программы.

Давайте рассмотрим приемы решения проблем многократного вызова `send` или `receive`.

Чтобы предотвратить многократный вызов функции, ей можно разрешить принимать аргумент *по значению*, т. е. позволить использовать объект для тех типов, в которых не реализован типаж `Copy`. После применения или перемещения объект пропадает из вызывающего кода, предотвращая тем самым свое повторное использование.

Предоставляя возможность вызова `send` или `receive` в виде отдельных типов (не имеющих реализации типажа `Copy`) и задействуя объект при выполнении операции,

можно быть уверенным, что каждая из таких операций может быть проведена только один раз.

Этот замысел приводит к следующей конструкции интерфейса, где вместо одного типа `Channel` канал оказывается представленным парой типов `Sender` и `Receiver`, в каждом из которых имеется метод, принимающий `self` по значению:

```
pub fn channel<T>() -> (Sender<T>, Receiver<T>) { ... }
pub struct Sender<T> { ... }
pub struct Receiver<T> { ... }
impl<T> Sender<T> {
    pub fn send(self, message: T) { ... }
}
impl<T> Receiver<T> {
    pub fn is_ready(&self) -> bool { ... }
    pub fn receive(self) -> T { ... }
}
```

Пользователь может создать канал, вызвав функцию `Channel()`, которая предоставит ему один `Sender` и один `Receiver`. Каждый из этих объектов может быть свободно передан, перемещен в другой поток и т. д. Но ни у одного из них не может быть несколько копий, в результате гарантируется, что каждый из методов `send` и `receive` может быть вызван только один раз.

Чтобы реализовать этот замысел, нужно найти место для `UnsafeCell` и `AtomicBool`. Раньше с этими полями была только одна структура, но теперь имеются две отдельные структуры, каждая из которых может пережить другую.

Поскольку отправителю и получателю предстоит совместное владение этими переменными, для получения возможности совместного выделения памяти, в которой будет сохраняться совместно используемый объект `Channel` и подсчета ссылок, необходимо будет воспользоваться `Arc` (см. раздел *главы 1 "Подсчет ссылок"*). Далее показано, что тип `Channel` не обязательно делать открытым, поскольку его существование — всего лишь нюанс реализации, не имеющий отношения к пользователю:

```
pub struct Sender<T> {
    channel: Arc<Channel<T>>,
}
pub struct Receiver<T> {
    channel: Arc<Channel<T>>,
}
struct Channel<T> { // больше не 'pub'
    message: UnsafeCell<MaybeUninit<T>>,
    ready: AtomicBool,
}
unsafe impl<T> Sync for Channel<T> where T: Send {}
```

Как и прежде, для `Channel<T>` реализуется `Sync` при условии, что в `T` реализован типаж `Send`, позволяющий использовать этот тип в разных потоках.

Заметьте, что теперь уже не нужно атомарное логическое значение `in_use`, как это было в предыдущей реализации канала. Оно использовалось операцией `send` только для проверки однократности ее вызова, что теперь статически гарантируется посредством системы типов.

Функция `channel`, предназначенная для создания канала и пары отправитель-получатель, похожа на встречавшуюся ранее функцию `Channel::new`, за исключением того, что она не помещает `Channel` в оболочку `Arc`, а помещает этот объект `Arc` и его клон в типы `Sender` и `Receiver`:

```
pub fn channel<T>() -> (Sender<T>, Receiver<T>) {
    let a = Arc::new(Channel {
        message: UnsafeCell::new(MaybeUninit::uninit()),
        ready: AtomicBool::new(false),
    });
     (Sender { channel: a.clone() }, Receiver { channel: a })
}
```

Методы `send`, `is_ready` и `receive` во многом идентичны ранее реализованным, но имеют ряд отличий:

- Теперь они перемещены в соответствующий тип, допускающий, что сообщение может отправлять только один отправитель, а получать его может только один получатель.
- `send` и `receive` теперь получают `self` по значению, а не по ссылке, чем гарантируется однократность их вызова.
- `send` больше не может поднимать панику, поскольку предусловие этой функции (однократный вызов) теперь статически гарантировано.

Итак, теперь функции приобрели следующий вид:

```
impl<T> Sender<T> {
    /// Теперь это никогда уже не паникует. :)
    pub fn send(self, message: T) {
        unsafe { (*self.channel.message.get()).write(message) };
        self.channel.ready.store(true, Release);
    }
}
impl<T> Receiver<T> {
    pub fn is_ready(&self) -> bool {
        self.channel.ready.load(Relaxed)
    }
    pub fn receive(self) -> T {
        if !self.channel.ready.swap(false, Acquire) {
            panic!("no message available!");
        }
        unsafe { (*self.channel.message.get()).assume_init_read() }
    }
}
```

Функция `receive` по-прежнему склонна к панике в случае ее вызова пользователем ее еще до возвращения функцией `is_ready()` значения `true`. Также в ней для возвращения флагу `ready` значения `false` по-прежнему используется `swap` (а не просто `load`), что позволяет реализации типажа `Drop` в типе `Channel` узнать о наличии непрочитанного сообщения, подлежащего удалению.

Реализация `Drop` полностью совпадает со своей ранее реализованной версией:

```
impl<T> Drop for Channel<T> {
    fn drop(&mut self) {
        if *self.ready.get_mut() {
            unsafe { self.message.get_mut().assume_init_drop() }
        }
    }
}
```

При удалении `Sender<T>` или `Receiver<T>` реализация `Drop` для `Arc<Channel<T>>` будет уменьшать значение счетчика ссылок на выделение памяти. Счетчик при удалении второго из этих двух типов обнуляется, и сам `Channel<T>` удаляется, в результате чего вызывается показанная ранее реализация `Drop`, позволяющая удалить отправленное, но так и не полученное сообщение.

Давайте посмотрим, как это работает (листинг 5.3).

Листинг 5.3

```
fn main() {
    thread::scope(|s| {
        let (sender, receiver) = channel();
        let t = thread::current();
        s.spawn(move || {
            sender.send("hello world!");
            t.unpark();
        });
        while !receiver.is_ready() {
            thread::park();
        }
        assert_eq!(receiver.receive(), "hello world!");
    });
}
```

По-прежнему сохраняется досадная проблема с вынужденным применением парковки потоков при ожидании сообщения, но вскоре и она будет решена.

А пока нужно было исключить при компиляции хотя бы один из вариантов некорректного использования канала. В отличие от прошлых вариантов, попытка двойной отправки сообщения уже приводит не к паникующей, а к полностью негодной программе. Если в показанную в листинге 5.3 рабочую программу добавить еще один вызов `send`, компилятор отловит несоответствие и спокойно сообщит нам о допущенной ошибке:

```
error[E0382]: use of moved value: 'sender'                                  ❶
  --> src/main.rs
   |
   |            sender.send("hello world!");
   |                   --------------------
   |                   'sender' moved due to this method call                ❷
  |
   |            sender.send("second message");
   |            ^^^^^^ value used here after move                            ❸
   |
note: this function takes ownership of the receiver 'self', which moves 'sender'      ❹
  --> src/lib.rs
   |
   |     pub fn send(self, message: T) {
   |                 ^^^^
   = note: move occurs because 'sender' has type 'Sender<&str>',
           which does not implement the 'Copy' trait                                  ❺
```

❶ Ошибка `[E0382]`: использование перемещенного значения: `'sender'`.

❷ `'sender'` перемещен из-за этого вызова метода.

❸ Значение использовано здесь после перемещения.

❹ Примечание: эта функция овладевает получателем `'self'`, переместившим `'sender'`.

❺ Примечание: причиной перемещения является то, что `'sender'` имеет тип `'Sender<&str>'`, в котором не реализован типаж `'Copy'` .

Бывают ситуации, существенно усложняющие разработку интерфейса, позволяющего обнаруживать ошибки в ходе компиляции. Когда сложившиеся обстоятельства благоприятствуют разработке такого интерфейса, он не только приносит немало удобств пользователю, но и позволяет сократить количество проверок в ходе выполнения программы, имеющей статические гарантии. Теперь в наличии флага `in_use` нет никакой необходимости, а из метода `send` удалены функция `swap` и проверка.

К сожалению, здесь можно столкнуться с новыми проблемами, приводящими к издержкам в ходе выполнения программы. В нашем случае проблема связана с совместным владением, для которого пришлось обратиться к использованию `Arc` и смириться с издержками на выделение памяти.

Вынужденный компромисс между безопасностью, удобством, гибкостью, простотой и производительностью огорчает, но порой без него просто не обойтись. Rust обычно стремится позволить преуспеть во всех этих аспектах, но иногда вынуждает все-таки слегка пожертвовать одним, чтобы добиться максимального эффекта в чем-то другом.

Заимствование во избежание выделения памяти

Только что разработанная реализация канала на основе `Arc` весьма удобна в применении, но за все приходится платить потерей производительности из-за необходимости выделения памяти. При желании повысить эффективность канала путем дальнейшей оптимизации можно слегка пожертвовать удобством для пользователя в пользу повышения производительности, возложив ответственность за совместно используемый объект `Channel` на пользователя. Вместо закулисного выделения памяти и владения `Channel` можно заставить пользователя создать `Channel`, который может быть *заимствован* отправителем `Sender` и получателем `Receiver`. Тогда пользователи смогут просто поместить этот канал в локальную переменную, избегая тем самым издержек на выделение памяти.

Придется также поступиться простотой, поскольку теперь нужно будет иметь дело с заимствованиями и временем жизни.

Итак, теперь три типа приобретут следующий вид: `Channel` снова станет открытым, а `Sender` и `Receiver` заимствуют его на конкретный жизненный срок:

```
pub struct Channel<T> {
    message: UnsafeCell<MaybeUninit<T>>,
    ready: AtomicBool,
}
unsafe impl<T> Sync for Channel<T> where T: Send {}
pub struct Sender<'a, T> {
    channel: &'a Channel<T>,
}
pub struct Receiver<'a, T> {
    channel: &'a Channel<T>,
}
```

Вместо функции `channel()` для создания пары `Sender` и `Receiver` пришлось вернуться к уже встречавшемуся ранее в данной главе методу `Channel::new`, позволяющему пользователю создавать такой объект в виде локальной переменной.

Кроме того, здесь понадобится способ, позволяющий пользователю создать заимствующие канал объекты `Sender` и `Receiver`. Это заимствование должно быть исключительным (`&mut Channel`), чтобы гарантировать невозможность наличия для одного и того же канала сразу нескольких отправителей или получателей. Одновременное предоставление `Sender` и `Receiver` позволяет *разбить* исключительное заимствование на два совместно используемых заимствования, благодаря чему и отправитель, и получатель смогут ссылаться на канал, не позволяя касаться канала чему-либо другому.

Данный замысел можно реализовать следующим образом:

```
impl<T> Channel<T> {
    pub const fn new() -> Self {
```

```
        Self {
            message: UnsafeCell::new(MaybeUninit::uninit()),
            ready: AtomicBool::new(false),
        }
    }
    pub fn split<'a>(&'a mut self) -> (Sender<'a, T>, Receiver<'a, T>) {
        *self = Self::new();
         (Sender { channel: self }, Receiver { channel: self })
    }
}
```

Метод `split` с его весьма непростой сигнатурой заслуживает более пристального внимания. Он заимствует `self` посредством исключительной ссылки, но разбивает ее на две совместно используемые ссылки, заключаемые в типы `Sender` и `Receiver`. Время жизни `'a` ясно дает понять, что оба этих объекта заимствуют что-то с ограниченным временем жизни; в данном случае речь идет о самом `Channel`. Поскольку `Channel` заимствован исключительно, вызывающая сторона не сможет заимствовать или перемещать его, пока существует объект `Sender` или объект `Receiver`.

Но как только оба этих объекта перестают существовать, срок жизни изменяемого заимствования истекает и компилятор охотно позволяет снова заимствовать объект `Channel` посредством повторного вызова функции `split()`. Невозможность повторного вызова функции `split()`, пока существуют `Sender` и `Receiver`, предположить нетрудно, а вот воспрепятствовать повторному вызову этой функции после удаления или забвения этих объектов не получится. Нам нужно убедиться, что для канала, у которого уже установлен флаг готовности `ready`, случайно не создан новый объект `Sender` или `Receiver`, поскольку тем самым будет нарушено предположение, предотвращающее неопределенное поведение.

Замена `*self` в функции `split()` новым пустым каналом дает гарантию нахождения в ожидаемом состоянии при создании состояний `Sender` и `Receiver`. Тем самым в старом `*self` вызывается также реализация `Drop`, которая позаботится об удалении ранее отправленного, но так и не полученного сообщения.

Поскольку время жизни в сигнатуре `split` берется из `self`, его указание можно опустить. Сигнатура `split` в приведенном ранее фрагменте идентична следующей более лаконичной версии:

```
pub fn split(&mut self) -> (Sender<T>, Receiver<T>) { ... }
```

Хотя в данной версии ничто явно не указывает на заимствование `self` возвращаемыми объектами, компилятор по-прежнему проверяет корректность учета времени жизни, поступая точно так же, как и в более подробной версии.

Остальные методы, как и реализация `Drop`, ничем не отличаются от прежней реализации на основе `Arc`, за исключением наличия дополнительного аргумента времени жизни `'_` для типов `Sender` и `Receiver`. (Если о них забыть, компилятор выдаст предложение об их добавлении.)

Для полноты складывающейся картины посмотрим на весь оставшийся код (листинг 5.4).

Листинг 5.4

```
impl<T> Sender<'_, T> {
    pub fn send(self, message: T) {
        unsafe { (*self.channel.message.get()).write(message) };
        self.channel.ready.store(true, Release);
    }
}
impl<T> Receiver<'_, T> {
    pub fn is_ready(&self) -> bool {
        self.channel.ready.load(Relaxed)
    }
    pub fn receive(self) -> T {
        if !self.channel.ready.swap(false, Acquire) {
            panic!("no message available!");
        }
        unsafe { (*self.channel.message.get()).assume_init_read() }
    }
}
impl<T> Drop for Channel<T> {
    fn drop(&mut self) {
        if *self.ready.get_mut() {
            unsafe { self.message.get_mut().assume_init_drop() }
        }
    }
}
```

Давайте посмотрим, как это работает!

```
fn main() {
    let mut channel = Channel::new();
    thread::scope(|s| {
        let (sender, receiver) = channel.split();
        let t = thread::current();
        s.spawn(move || {
            sender.send("hello world!");
            t.unpark();
        });
        while !receiver.is_ready() {
            thread::park();
        }
        assert_eq!(receiver.receive(), "hello world!");
    });
}
```

По сравнению с версией на основе `Arc` пользоваться стало неудобнее, но ненамного: для самостоятельного создания объекта `Channel` понадобилась всего лишь одна дополнительная строка. И все же обратите внимание, что канал должен быть создан

до области видимости, чтобы компилятор понял, что его существование переживет как отправителя, так и получателя.

Чтобы увидеть программу проверки заимствований компилятора в действии, попробуйте добавить в разных местах повторный вызов `channel.split()`. Станет понятно, что повторный вызов в пределах области видимости потока приведет к ошибке, а вот вызов этой функции после области видимости будет приемлемым. Если завершить использование возвращаемых `Sender` и `Receiver` до начала области видимости, то можно будет даже вызвать функцию `split()` до начала этой области.

Блокировка

Давайте наконец возьмемся за устранение последнего важного неудобства, присущего `Channel`, заключающегося в отсутствии интерфейса блокировки. Мы уже применяли парковку потоков при каждом тестировании нового варианта канала. Встроить эту же схему в сам канал должно быть не слишком сложно.

Чтобы снять парковку с получателя, отправителю необходимо знать, с какого именно потока нужно снимать парковку. Дескриптором потока является тип `std::thread::Thread`, и он-то нам и нужен для вызова `unpark()`. Давайте сохраним дескриптор потока-получателя внутри объекта `Sender`:

```
use std::thread::Thread;
pub struct Sender<'a, T> {
    channel: &'a Channel<T>,
    receiving_thread: Thread, // Новый код!
}
```

Но если объект `Receiver` передается, то этот дескриптор будет указывать на неправильный поток. Объект `Sender` не будет знать об этом и будет по-прежнему ссылаться на тот поток, который изначально содержал объект `Receiver`.

Эту проблему можно решить, наложив на `Receiver` более жесткие ограничения и запретив его пересылку между потоками. В *главе 1* в *разделе "Потокобезопасность: Send и Sync"* уже рассматривалась возможность добавления в структуру такого ограничения за счет специального маркерного типа `PhantomData`. С этим может справиться `PhantomData<*const()>`, поскольку типаж `Send` в простом указателе `*const()` не реализован:

```
pub struct Receiver<'a, T> {
    channel: &'a Channel<T>,
    _no_send: PhantomData<*const ()>, // New!
}
```

Затем для заполнения новых полей нужно внести изменения в метод `Channel::split`:

```
pub fn split<'a>(&'a mut self) -> (Sender<'a, T>, Receiver<'a, T>) {
    *self = Self::new();
     (
        Sender {
            channel: self,
```

```
            receiving_thread: thread::current(), // Новый код!
        },
        Receiver {
            channel: self,
            _no_send: PhantomData, // Новый код!
        }
    )
}
```

Для поля `receiving_thread` используется дескриптор текущего потока, поскольку возвращаемый объект `Receiver` в нем и останется.

Как показано далее, метод `send` меняется не сильно. В нем только нужно вызвать в отношении `receiving_thread` метод `unpark()`, чтобы разбудить получателя в случае, если он ожидает сообщение:

```
impl<T> Sender<'_, T> {
    pub fn send(self, message: T) {
        unsafe { (*self.channel.message.get()).write(message) };
        self.channel.ready.store(true, Release);
        self.receiving_thread.unpark(); // Новый код!
    }
}
```

Функция `receive` изменяется чуть больше. Если сообщения еще нет, то ее новая версия вместо паники станет терпеливо ждать сообщения, воспользовавшись для этого вызовом функции `thread::park()`, и повторять попытки получения сообщения необходимое число раз:

```
impl<T> Receiver<'_, T> {
    pub fn receive(self) -> T {
        while !self.channel.ready.swap(false, Acquire) {
            thread::park();
        }
        unsafe { (*self.channel.message.get()).assume_init_read() }
    }
}
```

Не забывайте, что функция `thread::park()` может вернуть управление, так и не выполнив свою задачу. (Или же метод `unpark()` может быть вызван не нашим `send`, а каким-либо другим методом.) Не следует предполагать, что при возврате управления из функции `park` флаг `ready` будет обязательно установлен. Значит, для повторной проверки флага после снятия парковки нужно воспользоваться циклом.

Структура `Channel<T>`, ее реализация `Sync`, ее функция `new` и реализация `Drop` остаются без изменений.

Давайте посмотрим, как это работает!

```
fn main() {
    let mut channel = Channel::new();
```

```
    thread::scope(|s| {
        let (sender, receiver) = channel.split();
        s.spawn(move || {
            sender.send("hello world!");
        });
        assert_eq!(receiver.receive(), "hello world!");
    });
}
```

Вполне очевидно, что в этой простой тестовой программе новый вариант `Channel` гораздо удобнее предыдущего. За это удобство нам пришлось расплатиться некоторой потерей гибкости: функция `receive()` может быть вызвана только тем потоком, который вызывает функцию `split()`. Если строки `send` и `receive` поменять местами, то программа перестанет компилироваться. В зависимости от сценария это может быть вполне приемлемо, полезно или же весьма неудобно.

Существует несколько способов решения этой проблемы, для многих из которых придется усложнить код и смириться с потерей производительности. В целом количество вариантов и компромиссов, исследование которых можно продолжить, не поддается исчислению.

Можно убить массу времени на реализацию еще двадцати и более вариантов одноразового канала, каждый из которых имел бы слегка разные свойства для каждого мыслимого сценария использования. При всей его привлекательности впадать в подобное бесконечное занятие, наверное, не стоит, лучше завершить данную главу, пока ситуация не вышла из-под контроля.

Резюме

- *Канал* предназначен для отправки *сообщений* между потоками.
- Простой и гибкий, но потенциально не самый эффективный канал относительно легко реализовать с применением только лишь `Mutex` и `Condvar`.
- *Одноразовый канал* служит для отправки только одного сообщения.
- Тип `MaybeUninit<T>` может использоваться для представления предположительно еще не инициализированного `T`. Его интерфейс, как правило, небезопасен, поэтому пользователь должен следить за тем, была ли инициализация, не дублировать данные `Copy` и при необходимости удалять их содержимое.
- Неудаленные объекты (также называемые *утечками* или *забвениями*) безопасны, но их присутствие без уважительной причины не приветствуется.
- Паника — важный инструмент, предназначенный для создания безопасного интерфейса.
- Для предотвращения повторного выполнения каких-либо действий можно воспользоваться получением по значению объекта, не поддерживающего типаж `Copy`.

- Исключительное заимствование и разбиение заимствования могут стать мощным инструментом принуждения к корректности программного кода.
- Гарантировать присутствие объекта в том же потоке можно, убедившись, что его типом не реализуется типаж `Send`, чего можно добиться с помощью маркерного типа `PhantomData`.
- Каждое проектное и внедренческое решение не обходится без компромиссов, и лучше всего такие решения принимать с учетом особенностей конкретного сценария функционирования программы.
- Разработка чего-либо без учета сценария использования может быть интересным и познавательным, но никогда не заканчивающимся занятием.

ГЛАВА 6

Создание своего собственного типа *Arc*

В *разделе "Подсчет ссылок" главы 1* был показан тип `std::sync::Arc<T>`, допускающий совместное владение посредством подсчета ссылок. Функция `Arc::new`, как и `Box::new`, инициирует новое выделение памяти. Но, в отличие от `Box`, клонированный объект `Arc` будет задействовать исходное выделение памяти, не инициируя новое. Совместно используемая выделенная память высвободится только после удаления `Arc` и всех его клонов.

Вопросы упорядочения памяти, связанные с реализацией этого типа, могут вызвать повышенный интерес. В данной главе теоретические выкладки будут в большей мере иметь практическое воплощение, для чего мы реализуем свой собственный `Arc<T>`. Прототипом станет базовая версия, которую мы расширим для поддержки *слабых указателей* на циклические структуры, и в конечном итоге будет реализована оптимизированная версия, почти что идентичная реализации, имеющейся в стандартной библиотеке.

Базовый подсчет ссылок

В нашей первой версии подсчет числа `Arc`-объектов, совместно использующих выделение памяти, будет реализован с помощью одного `AtomicUsize`. Начнем со структуры, содержащей этот счетчик и объект `T`:

```
struct ArcData<T> {
    ref_count: AtomicUsize,
    data: T,
}
```

Заметьте, что эта структура не является открытой. Это внутренняя особенность нашей реализации `Arc`.

Далее идет сама структура `Arc<T>`, которая фактически просто является указателем на совместно используемый объект `ArcData<T>`.

Можно соблазниться превращением его в оболочку для `Box<ArcData<T>>`, воспользовавшись для обработки выделения памяти под `ArcData<T>` стандартным `Box`. Но `Box` представляет собой исключительную, а не совместно используемую собственность. К тому же невозможно будет воспользоваться ссылкой, поскольку это будет не простое заимствование данных, принадлежащих чему-то другому, и время жизни ("до удаления последнего клона этого `Arc`") не может быть напрямую представлено указателем времени жизни, имеющимся в Rust.

Вместо этого придется воспользоваться указателем и самостоятельно справиться с выделением памяти и концепцией владения. Типы `*mut T` или `*const T` нужно будет заменить типом `std::ptr::NonNull<T>`, представляющим собой указатель на `T`, который никогда не будет равен нулю. Получается, что `Option<Arc<T>>` будет того же размера, что и `Arc<T>`, используя представление нулевого указателя для `None`:

```
use std::ptr::NonNull;
pub struct Arc<T> {
    ptr: NonNull<ArcData<T>>,
}
```

При наличии ссылки или `Box` компилятор автоматически понимает, для какого именно `T` ему следует выполнить структуру `Send` или `Sync`. Но в случае простого указателя или `NonNull`, если не указано иное, он выстраивает скромное предположение о безусловном отсутствии в них типажей `Send` или `Sync`.

Переправка `Arc<T>` между потоками приводит к совместному использованию объекта `T`, для чего требуется наличие у `T` типажа `Sync`. В свою очередь, переправка `Arc<T>` между потоками может привести к тому, что другой поток удалит этот `T`, фактически направив его в другой поток и требуя, чтобы в `T` был реализован `Send`. Иными словами, в `Arc<T>` должна быть реализация `Send` только в том случае, если в `T` одновременно имеется реализация `Send` и `Sync`. То же самое утверждение справедливо и для `Sync`, поскольку совместно используемый `&Arc<T>` можно клонировать в новый `Arc<T>`:

```
unsafe impl<T: Send + Sync> Send for Arc<T> {}
unsafe impl<T: Send + Sync> Sync for Arc<T> {}
```

Для `Arc<T>::new` нам нужно будет создать новое выделение памяти с `ArcData<T>` со счетчиком ссылок, равным единице. Для создания нового выделения памяти будет использоваться `Box::new`, для отказа от исключительного владения этим выделением — `Box::leak`, а для превращения его в указатель — `NonNull::from`:

```
impl<T> Arc<T> {
    pub fn new(data: T) -> Arc<T> {
        Arc {
            ptr: NonNull::from(Box::leak(Box::new(ArcData {
                ref_count: AtomicUsize::new(1),
                data,
            }))),
        }
    }
    ...
}
```

Нам известно, что пока существует объект `Arc`, указатель всегда будет указывать на действующий `ArcData<T>`. Но компилятор не знает об этом и не осуществляет за нас соответствующую проверку, поэтому для доступа к `ArcData` через указатель требуется небезопасный код. Для перехода из `Arc` в `ArcData`, поскольку это придется делать не один раз, мы добавим закрытую вспомогательную функцию:

```
fn data(&self) -> &ArcData<T> {
    unsafe { self.ptr.as_ref() }
}
```

Теперь с ее помощью можно выполнить реализацию типажа `Deref`, заставляющего `Arc<T>` открыто вести себя как ссылка на `T`:

```
impl<T> Deref for Arc<T> {
    type Target = T;
    fn deref(&self) -> &T {
        &self.data().data
    }
}
```

Обратите внимание на отсутствие реализации `DerefMut`. Поскольку `Arc<T>` представляет собой совместное владение, безусловное предоставление `&mut T` невозможно.

Следующим шагом станет реализация `Clone`. В клонированном `Arc` после увеличения значения счетчика ссылок будет задействован тот же указатель:

```
impl<T> Clone for Arc<T> {
    fn clone(&self) -> Self {
        // Предстоящая работа: Обработка переполнения.
        self.data().ref_count.fetch_add(1, Relaxed);
        Arc {
            ptr: self.ptr,
        }
    }
}
```

Для увеличения значения счетчика ссылок можно воспользоваться расслабленным упорядочением памяти, поскольку операции над другими переменными, которые должны выполняться строго до или после этой атомарной операции, у нас отсутствуют. Доступ к содержащемуся `T` до этой операции у нас уже был (через исходный `Arc`), и после нее он также остается (но теперь уже как минимум через два объекта `Arc`).

Прежде чем счетчик сможет переполниться, `Arc` придется клонировать множество раз, но к переполнению может привести запуск в цикле `std::mem::forget(arc.clone())`. Для решения данной проблемы можно воспользоваться любым из методов, рассмотренных в *главе 2*, в *разделах "Пример: Предоставление идентификатора"* и *"Пример: Предоставление идентификатора без переполнения"*.

Для максимальной эффективности в обычном (без переполнения) случае будет сохранен исходный объект `fetch_add`, а весь процесс будет просто прерван при приближении к переполнению:

```
if self.data().ref_count.fetch_add(1, Relaxed) > usize::MAX / 2 {
    std::process::abort();
}
```

Прерывание процесса одномоментно не происходит, оставляя время на то, чтобы метод `Arc::clone`, мог быть вызван другим потоком, еще больше увеличивая показания счетчика ссылок. Поэтому простой проверки с помощью `usize::MAX - 1` будет недостаточно. А вот использование в качестве ограничителя выражения `useize::MAX / 2` работает исправно: если предположить, что каждый поток занимает в памяти хотя бы несколько байтов, то одновременное существование потоков, занимающих `useize::MAX / 2` памяти, просто невозможно.

При клонировании значение счетчика нужно увеличивать, а при удалении `Arc` — уменьшать. Поток, увидевший изменение счетчика от единицы до нуля, знает, что он удалил последний `Arc<T>`, и отвечает за удаление и высвобождение памяти, выделенной под `ArcData<T>`.

Чтобы вернуть себе исключительное право владения выделением памяти, а затем сразу же удалить это выделение с помощью `drop()`, давайте воспользуемся `Box::from_raw`:

```
impl<T> Drop for Arc<T> {
    fn drop(&mut self) {
        // Предстоящая работа: Упорядочение памяти.
        if self.data().ref_count.fetch_sub(1, ...) == 1 {
            unsafe {
                drop(Box::from_raw(self.ptr.as_ptr()));
            }
        }
    }
}
```

Расслабленное упорядочение для этой операции недопустимо, поскольку нужно убедиться в отсутствии откуда-нибудь доступа к удаляемым данным. Иными словами, каждое удаление одного из бывших клонов `Arc` должно было произойти до последнего удаления. Следовательно, последняя операция `fetch_sub` должна установить отношение "происходит до" с каждой предыдущей операцией `fetch_sub`, что можно сделать, воспользовавшись упорядочением высвобождения и получения памяти: уменьшение значения счетчика, например, с двух до одного фактически "высвобождает" данные, а уменьшение их с единицы до нуля "получает" на них право собственности.

Для охвата обоих случаев можно было бы воспользоваться упорядочиванием памяти `AcqRel`, но для завершающего уменьшения до нуля требуется `Acquire`, а для остальных уменьшений требуется только `Release`. Упорядочение `Release` будет рациональнее только для операции `fetch_sub`, а отдельное ограждение `Acquire` потребуется только при необходимости:

```
if self.data().ref_count.fetch_sub(1, Release) == 1 {
    fence(Acquire);
    unsafe {
        drop(Box::from_raw(self.ptr.as_ptr()));
    }
}
```

Тестирование

Чтобы проверить, что создаваемый `Arc` ведет себя подобающе, можно написать модульный тест (листинг 6.1), который создает `Arc`, содержащий специальный объект, сообщающий о его удалении.

Листинг 6.1

```
#[test]
fn test() {
    static NUM_DROPS: AtomicUsize = AtomicUsize::new(0);
    struct DetectDrop;
    impl Drop for DetectDrop {
        fn drop(&mut self) {
            NUM_DROPS.fetch_add(1, Relaxed);
        }
    }
    // Создание двух Arc, совместно использующих объект, содержащий строку
    // и DetectDrop для обнаружения его удаления.
    let x = Arc::new(("hello", DetectDrop));
    let y = x.clone();
    // Отправка x другому потоку, и использование его там.
    let t = std::thread::spawn(move || {
        assert_eq!(x.0, "hello");
    });
    // Параллельно этому y должен быть по-прежнему применим здесь.
    assert_eq!(y.0, "hello");
    // Ожидание завершения потока.
    t.join().unwrap();
    // Один Arc, x, должен быть уже удален.
    // Но у нас все еще есть y, так что объект еще не должен был быть удален.
    assert_eq!(NUM_DROPS.load(Relaxed), 0);
    // Удаление оставшегося 'Arc'.
    drop(y);
    // Теперь, когда также удален и 'y',
    // объект должен был быть удален.
    assert_eq!(NUM_DROPS.load(Relaxed), 1);
}
```

Этот код проходит компиляцию и работает нормально, похоже, что наш `Arc` ведет себя так, как задумано! Хотя это обнадеживает, но еще не доказывает полную корректность реализации. Для большей уверенности рекомендуется воспользоваться длительным стресс-тестом с задействованием множества потоков.

Инструмент Miri

Также может быть весьма эффективен запуск тестов с помощью Miri, экспериментального, но очень полезного и мощного инструмента для проверки небезопасного кода на наличие

различных форм неопределенного поведения. Miri представляет собой принадлежащий компилятору Rust промежуточный интерпретатор среднего уровня. Следовательно, он запускает ваш код, не компилируя его в собственные инструкции процессора, а интерпретируя его в тот самый момент, когда такая информация, как типы и время жизни, все еще доступна. Поэтому интерпретатор Miri запускает программы значительно медленнее, чем в скомпилированном и работоспособном виде, но способен обнаруживать множество ошибок, потенциально приводящих к неопределенному поведению. В него включена экспериментальная поддержка обнаружения гонки данных, позволяющая обнаруживать проблемы с упорядочиванием памяти. Более подробную информацию и руководство по использованию Miri можно найти на странице GitHub.

Изменение

Ранее уже упоминалось, что реализовать `DerefMut` для нашего объекта `Arc` невозможно. Дать безусловное обещание исключительного доступа к данным (`&mut T`) нельзя, поскольку к ним можно получить доступ через другие объекты `Arc`.

Но можно все-таки разрешить такой доступ при определенных условиях. Можно создать метод, который выдает `&mut T` только в том случае, если счетчик ссылок равен единице, доказывая, что нет никакого другого объекта `Arc`, который способен обеспечить доступ к тем же данным.

Создаваемая в этих целях функция под названием `get_mut` должна будет принять `&mut Self`, чтобы гарантировать, что никто не сможет получить доступ к `T` через `Arc`. Знать о существовании только одного объекта `Arc` бессмысленно, если бы этот `Arc` все еще можно было использовать совместно.

Чтобы гарантировать, что потоки, ранее владевшие клоном `Arc`, больше не получают доступ к данным, нужно будет воспользоваться упорядочением получения памяти. С каждым отдельно взятым `drop`, снижающим значение счетчика до единицы, необходимо установить отношение "происходит до".

Все это имеет смысл, только когда значение счетчика ссылок равно единице. Если оно больше, то `&mut T` не будет предоставлен и упорядочение памяти не будет иметь никакого значения. Следовательно, можно воспользоваться расслабленной загрузкой, за которой следует условное ограждение получения:

```
pub fn get_mut(arc: &mut Self) -> Option<&mut T> {
    if arc.data().ref_count.load(Relaxed) == 1 {
        fence(Acquire);
        // Безопасно: Ничто иное не может обращаться к данным, поскольку
        // имеется только один Arc, к которому у нас исключительный доступ.
        unsafe { Some(&mut arc.ptr.as_mut().data) }
    } else {
        None
    }
}
```

Эта функция принимает обычный аргумент (с именем `arc`), а не аргумент `self`. Следовательно, вызов может только иметь вид `Arc::get_mut(&mut a)`, но не `a.get_mut()`. Во

избежание недоразумений, связанных с одноименным методом базового T, данный подход рекомендуется для всех типов, имеющих реализацию типажа Deref.

Возвращаемая изменяемая ссылка подразумеваемо заимствует время жизни из аргумента, следовательно, ничто не может воспользоваться исходным Arc, пока все еще существует возвращенный &mut T, что обеспечивает безопасное внесение изменений.

Когда срок жизни &mut T истечет, Arc можно будет снова задействовать и использовать совместно с другими потоками. И тут может возникнуть вопрос о том, нужно ли после этого беспокоиться об упорядочении памяти для потоков, обращающихся к данным. Во всяком случае, ответственность за это возлагается на любой механизм, осуществляющий совместное использование Arc (или его нового клона) с другим потоком. (Например, на мьютекс, канал или на функцию порождения нового потока.)

Слабые указатели

Подсчет ссылок может пригодиться при представлении в памяти структур, состоящих из нескольких объектов. Например, каждый узел в древовидной структуре может содержать Arc для каждого из своих дочерних узлов. Следовательно, при удалении узла удаляются (в рекурсивном порядке) и вышедшие из употребления все его дочерние узлы.

Но для *циклических структур* такой механизм не работает. Если дочерний узел также содержит Arc для своего родительского узла, ни один такой узел не будет удаляться, поскольку всегда будет иметься хотя бы один Arc, все еще ссылающийся на него.

Объект Arc, имеющийся в стандартной библиотеке, предлагает решение этой проблемы за счет Weak<T>, который также называется *слабым указателем*, ведет себя наподобие Arc<T>, но не препятствует удалению объекта. Тип T может совместно использоваться несколькими объектами Arc<T> и Weak<T>, но когда все объекты Arc<T> исчезают, T удаляется независимо от того, остались еще какие-либо объекты Weak<T> или нет.

Получается, что Weak<T> может существовать и без T и, следовательно, не способен безусловно предоставлять &T, как это может делать Arc<T>. Но для доступа к T при наличии Weak<T> этот объект можно с помощью метода update() *повысить в категории* до Arc<T>. Этот метод возвращает Option<Arc<T>> со значением None, если T уже удален.

В структуре, основанной на Arc, Weak может служить для разрыва циклов. Например, дочерние узлы в древовидной структуре могут использовать для родительского узла Weak, а не Arc. Тогда удалению родительского узла не мешает существование его дочерних узлов.

Давайте реализуем всё это на практике.

Как и прежде, когда количество объектов Arc достигает нуля, содержащийся объект T может быть удален. А вот удалить или высвободить ArcData пока что невозможно, поскольку на него все еще могут иметься слабые указатели. Удалить и высвободить ArcData можно будет только после того, как исчезнет последний указатель Weak.

Итак, воспользуемся двумя счетчиками: одним для "количества объектов, ссылающихся на T", а другим для "количества объектов, ссылающихся на ArcData<T>". Иными словами, первый счетчик будет таким же, как и прежде: он станет подсчитывать количество объектов Arc, а вот второй счетчик станет подсчитывать как объекты Arc, так и объекты Weak.

Нам понадобится также что-то, позволяющее удалить содержащийся объект (T), пока ArcData<T> все еще будет использоваться слабыми указателями. Давайте добавим Option<T>, чтобы можно было при удалении данных воспользоваться значением None и для *внутренней изменяемости* заключить его в UnsafeCell (см. *раздел "Внутренняя изменяемость" главы 1*) для обеспечения желаемого результата, когда ArcData<T > не находится в исключительном владении:

```
struct ArcData<T> {
    /// Количество 'Arc'.
    data_ref_count: AtomicUsize,
    /// Количество вместе взятых 'Arc' и 'Weak'.
    alloc_ref_count: AtomicUsize,
    /// Данные. 'None', если остались только слабые указатели.
    data: UnsafeCell<Option<T>>,
}
```

Если составить представление о Weak<T> как об объекте, ответственном за поддержание активности ArcData<T>, возможно, есть смысл реализовать Arc<T> как структуру, содержащую Weak<T>, поскольку Arc<T> должен сделать то же самое и даже больше.

```
pub struct Arc<T> {
    weak: Weak<T>,
}
pub struct Weak<T> {
    ptr: NonNull<ArcData<T>>,
}
unsafe impl<T: Sync + Send> Send for Weak<T> {}
unsafe impl<T: Sync + Send> Sync for Weak<T> {}
```

Функция new в основном осталась такой же, как и прежде, за исключением того, что теперь в ней имеются два одновременно инициализируемых счетчика:

```
impl<T> Arc<T> {
    pub fn new(data: T) -> Arc<T> {
        Arc {
            weak: Weak {
                ptr: NonNull::from(Box::leak(Box::new(ArcData {
                    alloc_ref_count: AtomicUsize::new(1),
                    data_ref_count: AtomicUsize::new(1),
```

```
                data: UnsafeCell::new(Some(data)),
            })))),
        },
    }
}
...
}
```

Как и прежде, предполагается, что поле `ptr` всегда указывает на действительный объект `ArcData<T>`. Но на этот раз данное предположение будет закодировано в `Weak<T>` в виде закрытого вспомогательного метода `data()`.

```
impl<T> Weak<T> {
    fn data(&self) -> &ArcData<T> {
        unsafe { self.ptr.as_ref() }
    }
    ...
}
```

Теперь, чтобы в реализации `Deref` для `Arc<T>` получить указатель на содержимое ячейки и задействовать небезопасный код в качестве обещания, что здесь он может быть безопасно совместно использован, придется воспользоваться `UnsafeCell::get()`. Также для получения ссылки на `Option<T>` понадобится `as_ref().unwrap()`. Ни о какой панике здесь беспокоиться не стоит, поскольку значение `None` может появиться в `Option` только когда уже не останется ни одного объекта `Arc`.

```
impl<T> Deref for Arc<T> {
    type Target = T;
    fn deref(&self) -> &T {
        let ptr = self.weak.data().data.get();
        // Безопасно: поскольку на данные имеется счетчик Arc,
        // эти данные существуют и могут быть в совместном пользовании.
        unsafe { (*ptr).as_ref().unwrap() }
    }
}
```

Реализация `Clone` для `Weak<T>` весьма проста и во многом идентична предыдущей реализации `Clone` для `Arc<T>`:

```
impl<T> Clone for Weak<T> {
    fn clone(&self) -> Self {
        if self.data().alloc_ref_count.fetch_add(1, Relaxed) > usize::MAX / 2 {
            std::process::abort();
        }
        Weak { ptr: self.ptr }
    }
}
```

В реализации `Clone` для нового `Arc<T>` нужно увеличить значения обоих счетчиков. Для повторного использования показанного кода для первого счетчика просто вос-

пользуемся `self.weak.clone()`, и тогда останется всего лишь самостоятельно увеличить значение второго счетчика:

```
impl<T> Clone for Arc<T> {
    fn clone(&self) -> Self {
        let weak = self.weak.clone();
        if weak.data().data_ref_count.fetch_add(1, Relaxed) > usize::MAX / 2 {
           std::process::abort();
        }
        Arc { weak }
    }
}
```

Удаление `Weak` должно привести к уменьшению значения его счетчика, а также к удалению и высвобождению `ArcData` при переходе значения счетчика от единицы к нулю. Это идентично тому, что делалось в реализации `Drop` в предыдущем `Arc`:

```
impl<T> Drop for Weak<T> {
    fn drop(&mut self) {
        if self.data().alloc_ref_count.fetch_sub(1, Release) == 1 {
            fence(Acquire);
            unsafe {
                drop(Box::from_raw(self.ptr.as_ptr()));
            }
        }
    }
}
```

При удалении `Arc` должны уменьшаться значения обоих счетчиков. Заметьте, что в одном из них это делается автоматически, и поскольку в каждом `Arc` содержится `Weak`, удаление `Arc` также приведет к удалению `Weak`. Остается только заняться другим счетчиком:

```
impl<T> Drop for Arc<T> {
    fn drop(&mut self) {
        if self.weak.data().data_ref_count.fetch_sub(1, Release) == 1 {
            fence(Acquire);
            let ptr = self.weak.data().data.get();
            // Безопасно: счетчик ссылок на данные обнулен,
            // поэтому ничто не будет к ним обращаться.
            unsafe {
                (*ptr) = None;
            }
        }
    }
}
```

В Rust при удалении объекта сначала запускается его функция `Drop::drop` (если в нем реализован `Drop`), а затем рекурсивно одно за другим удаляются все его поля.

Проверка, проводимая в методе get_mut, практически не изменилась, за исключением того, что теперь необходимо брать в расчет и слабые указатели. Может показаться, что при проверке исключительности слабые указатели можно проигнорировать, но категория Weak<T> в любой момент может быть повышена до Arc<T>. Поэтому, прежде чем get_mut сможет выдать &mut T, он должен проверить отсутствие других указателей Arc<T> или Weak<T>:

```
impl<T> Arc<T> {
    ...
    pub fn get_mut(arc: &mut Self) -> Option<&mut T> {
        if arc.weak.data().alloc_ref_count.load(Relaxed) == 1 {
            fence(Acquire);
            // Безопасно: ничто иное не может обратиться к данным,
            // поскольку есть только один Arc, к которому имеется
            // исключительный доступ, и нет Weak-указателей.
            let arcdata = unsafe { arc.weak.ptr.as_mut() };
            let option = arcdata.data.get_mut();
            // Известно, что данные все еще доступны, поскольку
            // к ним имеется Arc, и поэтому не будет никакой паники.
            let data = option.as_mut().unwrap();
            Some(data)
        } else {
            None
        }
    }
    ...
}
```

Следующий шаг: повышение категории слабого указателя. Повысить Weak до категории Arc можно только тогда, когда данные все еще существуют. Если остались только слабые указатели, не останется данных, которые можно было бы совместно использовать посредством Arc. Итак, придется увеличить значение счетчика Arc, но сделать это можно только в том случае, если он еще не равен нулю. Воспользуемся циклом сравнения и обмена (см. *раздел "Операции сравнения и обмена" главы 2*).

Из-за отсутствия каких-либо операций над другими переменными, требующих своего выполнения строго до или после атомарного увеличения значения счетчика ссылок, здесь, как и прежде, вполне подойдет расслабленное упорядочение памяти:

```
impl<T> Weak<T> {
    ...
    pub fn upgrade(&self) -> Option<Arc<T>> {
        let mut n = self.data().data_ref_count.load(Relaxed);
        loop {
            if n == 0 {
                return None;
            }
            assert!(n < usize::MAX);
```

```
            if let Err(e) =
                self.data()
                    .data_ref_count
                    .compare_exchange_weak(n, n + 1, Relaxed, Relaxed)
            {
                n = e;
                continue;
            }
            return Some(Arc { weak: self.clone() });
        }
    }
}
```

Обратите внимание, что на этот раз можно провести проверку `n < usize::MAX`, **поскольку данное утверждение способно вызвать панику** *до* **изменения значения** `data_ref_count`.

А вот получить `Weak<T>` из `Arc<T>` гораздо проще:

```
impl<T> Arc<T> {
    ...
    pub fn downgrade(arc: &Self) -> Weak<T> {
        arc.weak.clone()
    }
}
```

Тестирование

Чтобы быстро протестировать наше творение, давайте внесем изменения в предыдущий модульный тест (см. листинг 6.1), чтобы воспользоваться слабыми указателями, и проверим, что их можно повысить в категории в предполагаемый момент (листинг 6.2).

Листинг 6.2

```
#[test]
fn test() {
    static NUM_DROPS: AtomicUsize = AtomicUsize::new(0);
    struct DetectDrop;
    impl Drop for DetectDrop {
        fn drop(&mut self) {
            NUM_DROPS.fetch_add(1, Relaxed);
        }
    }
    // Создание Arc с двумя слабыми указателями.
    let x = Arc::new(("hello", DetectDrop));
    let y = Arc::downgrade(&x);
    let z = Arc::downgrade(&x);
```

```
    let t = std::thread::spawn(move || {
        // Здесь слабый указатель должен иметь возможность повышения категории.
        let y = y.upgrade().unwrap();
        assert_eq!(y.0, "hello");
    });
    assert_eq!(x.0, "helIo");
    t.join().unwrap();
    // Данные еще не должны быть удалены,
    // а слабый указатель должен иметь возможность повышения категории.
    assert_eq!(NUM_DROPS.load(Relaxed), 0);
    assert!(z.upgrade().is_some());
    drop(x);
    // Теперь данные должны быть удалены,
    // а слабый указатель больше не должен иметь возможности
    // повышения категории.
    assert_eq!(NUM_DROPS.load(Relaxed), 1);
    assert!(z.upgrade().is_none());
}
```

Этот код также компилируется и запускается без проблем, в результате чего мы получаем созданную своими силами весьма удобную реализацию `Arc`.

Оптимизация

Хотя из применения слабых указателей можно извлечь вполне определенную пользу, тип `Arc` зачастую используется без них. Недостаток нашей последней реализации заключается в необходимости проведения двух атомарных операций как для клонирования, так и для удаления `Arc`, поскольку ими должны увеличиваться или уменьшаться значения обоих счетчиков. При этом все пользователи `Arc` обязаны "платить" за слабые указатели, даже если они ими не пользуются.

Казалось бы, решение состоит в том, чтобы вести подсчет указателей `Arc<T>` и `Weak<T>` по отдельности, но тогда будет утрачена возможность атомарной проверки того, что оба счетчика равны нулю. Чтобы разобраться в сути проблемы, давайте представим, что имеется некий поток, выполняющий следующую однообразную функцию:

```
fn annoying(mut arc: Arc<Something>) {
    loop {
        let weak = Arc::downgrade(&arc);
        drop(arc);
        println!("I have no Arc!");          ❶
        arc = weak.upgrade().unwrap();
        drop(weak);
        println!("I have no Weak!");         ❷
    }
}
```

Этот поток постоянно понижает и повышает категорию Arc, многократно выполняя циклический проход через моменты отсутствия Arc ❶ и моменты отсутствия Weak ❷. Если выполнять проверку обоих счетчиков, чтобы выявить наличие каких-либо потоков, все еще использующих выделение памяти, то, если не повезет, данный поток может скрыть свое существование и выполнить проверку счетчика Arc в момент выполнения первой инструкции вывода на экран ❶, а проверку счетчика Weak выполнить в момент выполнения второй инструкции вывода на экран ❷.

В нашей последней реализации эта проблема решалась подсчетом не только каждой Arc, но и Weak. Более утонченный способ решения этой проблемы заключается в подсчете всех указателей Arc объединенными в одно целое с Weak. Тогда, как и в нашей последней реализации, значение счетчика слабых указателей (alloc_ref_count) ни в коем случае не обнулится, пока будет хотя бы один объект Arc, но при клонировании Arc вообще не будет необходимости затрагивать этот счетчик. Уменьшить значение счетчика слабых указателей сможет только удаление самого последнего Arc.

Давайте посмотрим, что получится.

Теперь уже просто Arc<T> не подойдет в качестве оболочки для Weak<T>, поэтому оба они послужат оболочкой для ненулевого указателя на выделение памяти:

```
pub struct Arc<T> {
    ptr: NonNull<ArcData<T>>,
}
unsafe impl<T: Sync + Send> Send for Arc<T> {}
unsafe impl<T: Sync + Send> Sync for Arc<T> {}
pub struct Weak<T> {
    ptr: NonNull<ArcData<T>>,
}
unsafe impl<T: Sync + Send> Send for Weak<T> {}
unsafe impl<T: Sync + Send> Sync for Weak<T> {}
```

Оптимизируя реализацию, можно также слегка сократить ArcData<T>, воспользовавшись вместо Option<T> объектом std::mem::ManuallyDrop<T>. Ведь Option<T> позволял при удалении данных заменить Some(T) на None, но фактически, чтобы сообщить об исчезновении данных в отдельном состоянии None нет никакой необходимости, поскольку об этом уже свидетельствует существование или отсутствие Arc<T>. А ManuallyDrop<T> занимает тот же объем памяти, что и T, но позволяет самостоятельно удалить его в любой момент с помощью небезопасного вызова ManuallyDrop::drop():

```
use std::mem::ManuallyDrop;
struct ArcData<T> {
    /// Количество 'Arc'.
    data_ref_count: AtomicUsize,
    /// Количество 'Weak' плюс единица, если есть какие-либо 'Arc'
    alloc_ref_count: AtomicUsize,
```

```
    /// Данные. Удаляются, если остались только слабые указатели.
    data: UnsafeCell<ManuallyDrop<T>>,
}
```

Функция `Arc::new` осталась практически неизменной: оба счетчика, как и прежде, инициализируются одновременно, но теперь метод `Some()` заменен на `ManuallyDrop::new()`:

```
impl<T> Arc<T> {
    pub fn new(data: T) -> Arc<T> {
        Arc {
            ptr: NonNull::from(Box::leak(Box::new(ArcData {
                alloc_ref_count: AtomicUsize::new(1),
                data_ref_count: AtomicUsize::new(1),
                data: UnsafeCell::new(ManuallyDrop::new(data)),
            }))),
        }
    }
    ...
}
```

В реализации `Deref` в отношении типа `Weak` больше не может использоваться закрытый метод `data`, поэтому такая же закрытая вспомогательная функция будет добавлена в `Arc<T>`:

```
impl<T> Arc<T> {
    ...
    fn data(&self) -> &ArcData<T> {
        unsafe { self.ptr.as_ref() }
    }
    ...
}
impl<T> Deref for Arc<T> {
    type Target = T;
    fn deref(&self) -> &T {
        // Безопасно: Поскольку имеется Arc на данные,
        // эти данные существуют и могут находиться  в совместном пользовании.
        unsafe { &*self.data().data.get() }
    }
}
```

Реализации `Clone` и `Drop` для `Weak<T>` остаются точно такими же, как и в нашей последней версии. Для полноты картины посмотрим, как они выглядят вместе с закрытой вспомогательной функцией `Weak::data`:

```
impl<T> Weak<T> {
    fn data(&self) -> &ArcData<T> {
        unsafe { self.ptr.as_ref() }
    }
    ...
}
```

```
impl<T> Clone for Weak<T> {
    fn clone(&self) -> Self {
        if self.data().alloc_ref_count.fetch_add(1, Relaxed) > usize::MAX / 2 {
            std::process::abort();
        }
        Weak { ptr: self.ptr }
    }
}
impl<T> Drop for Weak<T> {
    fn drop(&mut self) {
        if self.data().alloc_ref_count.fetch_sub(1, Release) == 1 {
            fence(Acquire);
            unsafe {
                drop(Box::from_raw(self.ptr.as_ptr()));
            }
        }
    }
}
```

А теперь перейдем к той части, ради которой и затевалось создание новой оптимизированной реализации, к тому, что теперь клонирование `Arc<T>` должно затрагивать только один счетчик:

```
impl<T> Clone for Arc<T> {
    fn clone(&self) -> Self {
        if self.data().data_ref_count.fetch_add(1, Relaxed) > usize::MAX / 2 {
            std::process::abort();
        }
        Arc { ptr: self.ptr }
    }
}
```

Соответственно и удаление `Arc<T>` теперь требует уменьшения только одного счетчика, за исключением последнего удаления, при котором значение этого счетчика переходит от единицы к нулю. В этом случае счетчик слабых указателей также необходимо уменьшить, чтобы его значение смогло достичь нуля, как только слабых указателей не останется. Это делается простым созданием из ничего `Weak<T>` с его немедленным удалением:

```
impl<T> Drop for Arc<T> {
    fn drop(&mut self) {
        if self.data().data_ref_count.fetch_sub(1, Release) == 1 {
            fence(Acquire);
            // Безопасно: Счетчик ссылок на данные обнулен,
            // поэтому больше к данным ничто обратиться не сможет.
            unsafe {
                ManuallyDrop::drop(&mut *self.data().data.get());
            }
```

```
            // Теперь ни одного 'Arc<T>' не осталось, и можно удалить
            // подразумеваемый слабый указатель, представлявший все 'Arc<T>'.
            drop(Weak { ptr: self.ptr });
        }
    }
}
```

Метод upgrade для Weak<T> остается в основном в прежнем виде, за исключением того, что он больше не клонирует слабый указатель, поскольку ему больше не нужно увеличивать значение его счетчика. Повышение категории завершается успешно только в том случае, если в выделенной памяти уже есть хотя бы один Arc<T>, следовательно, Arc-объекты уже учтены в счетчике слабых указателей:

```
impl<T> Weak<T> {
    ...
    pub fn upgrade(&self) -> Option<Arc<T>> {
        let mut n = self.data().data_ref_count.load(Relaxed);
        loop {
            if n == 0 {
                return None;
            }
            assert!(n < usize::MAX);
            if let Err(e) =
                self.data()
                .data_ref_count
                .compare_exchange_weak(n, n + 1, Relaxed, Relaxed)
            {
                n = e;
                continue;
            }
            return Some(Arc { ptr: self.ptr });
        }
    }
}
```

Пока что различия между этой и нашей предыдущей реализацией крайне невелики. Но как только дело дойдет до еще не реализованных нами двух последних методов downgrade и get_mut, ситуация существенно усложнится.

Теперь, в отличие от прежних задач, методу get_mut необходимо проверить, не установлены ли оба счетчика в единичные значения, что позволит определить, что остался только один Arc<T> и не осталось ни одного Weak<T>, поскольку счетчик слабых указателей с единичным значением теперь может представлять сразу несколько указателей Arc<T>. Считывание значений счетчиков представляет собой две отдельно взятые операции, происходящие в несколько разные моменты времени, поэтому нужно быть крайне осторожным, чтобы не пропустить одновременное понижение категорий как в первом примере *раздела "Оптимизация"* данной главы.

Если сначала выполнить проверку data_ref_count на единичное значение, можно пропустить последующее выполнение upgrade() перед проверкой другого счетчика. Но если сначала проверить alloc_ref_count на равенство единице, можно пропустить последующее выполнение downgrade() перед проверкой другого счетчика.

Выход из этой дилеммы заключается в кратковременной блокировке операции downgrade() путем "блокировки" счетчика слабых указателей. Никакой мьютекс здесь не понадобится. Для представления особого "заблокированного" состояния счетчика слабых указателей можно воспользоваться специальным значением, например usesize::MAX. Его блокировка будет весьма кратковременной, только для загрузки другого счетчика, поэтому метод downgrade до разблокирования может просто находиться в спин-блокировке, а его запуск на выполнение одновременно с get_mut маловероятен.

Итак, в get_mut сначала нужно будет проверить, имеет ли alloc_ref_count единичное значение, и в то же время, если так и есть, заменить его на usesize::MAX. С этой задачей справится операция compare_exchange.

Затем нужно будет проверить, имеет ли единичное значение и другой счетчик, после чего можно будет сразу разблокировать счетчик слабых указателей. Если значение второго счетчика также равно единице, станет понятно, что имеется исключительный доступ к выделенной памяти и к данным, и поэтому можно будет возвратить &mut T:

```
pub fn get_mut(arc: &mut Self) -> Option<&mut T> {
    // Упорядочение получения сочетается с упорядочением высвобождения
    // в Weak::drop, чтобы обеспечить видимость любых повышенных
    // в категории указателей в последующей операции data_ref_count.load.
    if arc.data().alloc_ref_count.compare_exchange(
        1, usize::MAX, Acquire, Relaxed
    ).is_err() {
        return None;
    }
    let is_unique = arc.data().data_ref_count.load(Relaxed) == 1;
    // Упорядочение высвобождения сочетается с повышением в 'downgrade'
    // до упорядочения получения, чтобы гарантировать, что любые
    // изменения в data_ref_count, происходящие после 'downgrade',
    // не изменят показанный выше результат is_unique.
    arc.data().alloc_ref_count.store(1, Release);
    if !is_unique {
        return None;
    }
    // Упорядочение получения используется для сочетания с понижением
    // в Arc::drop до упорядочения высвобождения, чтобы гарантировать
    // отсутствие какого-либо стороннего доступа к данным.
    fence(Acquire);
    unsafe { Some(&mut *arc.data().data.get()) }
}
```

Вполне ожидаемо, операция блокировки (compare_exchange) должна будет использовать упорядочение памяти Acquire, а операция разблокировки (store) — упорядочение памяти Release.

Если вместо этого для compare_exchange осуществлялось бы упорядочение памяти Relaxed, то при последующей операции загрузки load из data_ref_count можно было бы не увидеть новое значение только что обновленного Weak-указателя, даже при подтверждении состоявшегося удаления каждого Weak-указателя операцией compare_exchange.

Если для операции store было задействовано упорядочение памяти Relaxed, то при выполнении предшествующей ей операции load можно было бы наблюдать результат будущей операции Arc::drop для Arc, не утратившей еще возможности понижения своей категории.

Ограждение получения здесь такое же, как и прежде: оно синхронизируется с операцией высвобождения-понижения в Arc::Drop, чтобы гарантировать, что каждый доступ через бывшие клоны Arc происходил до нового исключительного доступа.

Последняя часть головоломки представлена методом downgrade, на который возлагается проверка специального значения usize::MAX, позволяющая увидеть, заблокирован или нет счетчик слабых указателей, и выполнять спин-блокировку до тех пор, пока он не будет разблокирован. Здесь, так же как и в реализации метода upgrade, для проверки специального значения и переполнения перед увеличением счетчика будет предусмотрен цикл сравнения и обмена:

```
pub fn downgrade(arc: &Self) -> Weak<T> {
    let mut n = arc.data().alloc_ref_count.load(Relaxed);
    loop {
        if n == usize::MAX {
            std::hint::spin_loop();
            n = arc.data().alloc_ref_count.load(Relaxed);
            continue;
        }
        assert!(n < usize::MAX - 1);
        // Получение синхронизируется с выполняемой в get_mut
        // операцией высвобождения-сохранения.
        if let Err(e) =
            arc.data()
            .alloc_ref_count
            .compare_exchange_weak(n, n + 1, Acquire, Relaxed)
        {
            n = e;
            continue;
        }
        return Weak { ptr: arc.ptr };
    }
}
```

Для операции compare_exchange_weak используется упорядочение получения памяти, синхронизируемое с высвобождением-сохранением в функции get_mut. Иначе мог

бы возникнуть эффект, при котором последующая операция `Arc::drop` оказалась бы видна потоку, выполняющему операцию `get_mut` еще до того, как он разблокирует счетчик.

Иными словами, здесь операция сравнения и обмена при упорядочении получения памяти эффективно "блокирует" операцию `get_mut`, не позволяя ей успешно завершиться. Ее снова можно будет "разблокировать" последующим выполнением операции `Weak::drop`, уменьшающей значение счетчика, возвращая ему единичное значение, выполняя при этом упорядочение высвобождения памяти.

Только что созданная оптимизированная реализация `Arc<T>` и `Weak<T>`, почти идентична реализации, включенной в стандартную библиотеку Rust.

Запуск предыдущего теста (см. листинг 6.2 из *раздела "Тестирование"*) покажет, что наша оптимизированная реализация также успешно пройдет компиляцию и тестирование.

Не стоит переживать, если принятие правильных решений по упорядочению памяти для представленной здесь оптимизированной реализации показалось вам чересчур сложным. Правильная реализация многих конкурентных структур данных дается намного проще. А выбор для рассмотрения в данной главе именно этой реализации `Arc` обусловлен тем, что упорядочение памяти в ней сопряжено с целым рядом замысловатых тонкостей.

Резюме

- `Arc<T>` обеспечивает совместное владение выделением памяти с подсчетом ссылок.
- Проверка счетчика ссылок на значение, равное единице, позволяет `Arc<T>` условно предоставить исключительный доступ (`&mut T`).
- Увеличить значение атомарного счетчика ссылок можно с помощью расслабленной операции, но последнее уменьшение его значения нужно синхронизировать со всеми предыдущими уменьшениями.
- Во избежание применения циклов можно воспользоваться *слабым указателем* (`Weak<T>`).
- Тип `NonNull<T>` представляет собой указатель на `T`, никогда не имеющий нулевого значения.
- Тип `ManuallyDrop<T>` подходит для самостоятельного решения об использовании небезопасного кода при удалении `T`.
- Как только будет задействовано более одной атомарной переменной, ситуация существенно усложнится.
- Реализация ситуативной спин-блокировки порой может оказаться весьма эффективным подходом к одновременной работе сразу с несколькими атомарными переменными.

ГЛАВА 7

Представление о внутренней "кухне" процессора

Теории, изложенной в *главах 2* и *3*, вполне достаточно для написания работоспособного кода конкурентных вычислений, но этими же положениями можно воспользоваться и для получения примерного представления о том, что происходит на уровне процессора. Данная глава посвящена исследованию тех машинных команд, в которые компилируются атомарные операции. Мы рассмотрим, чем отличаются друг от друга архитектуры различных процессоров, узнаем, почему существует слабая версия `compare_exchange`, уделим внимание упорядочению памяти на самом низком уровне отдельных инструкций и выясним, как со всем этим связано кеширование.

Здесь не ставится цель, понять все ключевые особенности архитектуры каждого отдельно взятого процессора. Для подобного повествования потребовалось бы множество заставленных вплотную книжных полок с трудами, которые, вполне вероятно, еще не написаны или же просто недоступны большинству читателей. Задача главы куда скромнее — дать общее представление о работе атомарности на уровне процессора, позволяющее принимать более обоснованные решения при реализации и оптимизации кода, использующего атомарные вычисления. Или же, проще говоря, удовлетворить любопытство читателей относительно происходящего "за кулисами" и дать отдохнуть от всей рассмотренной ранее абстрактной теории.

Чтобы добиться максимальной конкретики, давайте сфокусируем свое внимание на двух вполне определенных процессорных архитектурах:

- *x*86-64 — 64-разрядной версии архитектуры *x*86, реализованной в процессорах Intel и AMD, используемой в большинстве ноутбуков, настольных компьютеров, серверов и в некоторых игровых консолях. Первоначальная 16-разрядная архитектура *x*86 и ее очень популярное 32-разрядное расширение были разработаны компанией Intel, а вот 64-разрядная версия, которая теперь называется *x*86-64, изначально была расширением, разработанным компанией AMD, и нередко называлась AMD64. Intel также разработала собственную 64-разрядную архитектуру IA-64, но в итоге вместо нее приняла более популярное расширение AMD *x*86 (под названиями IA-32e, EM64T и позже Intel 64).
- APM64 — 64-разрядной версии архитектуры ARM, применяемой практически во всех современных мобильных устройствах, высокопроизводительных встроенных системах, а также все чаще в современных ноутбуках и настольных компьютерах. Она также известна как AArch64 и была представлена как часть ARMv8. Более ранние (32-разрядные), во многом схожие с ней версии ARM используются в еще более широком спектре устройств. На ARMv6 и ARMv7 осно-

ваны многие весьма популярные микроконтроллеры во всевозможных встраиваемых системах, от автомобилей и до электронных тестов на COVID.

Эти две архитектуры сильно различаются. Но самое главное — в них приняты разные подходы к реализации атомарности. Представление о том, как атомарность работает в обоих случаях, даст нам более общую картину, которую можно будет перенести на многие другие архитектуры.

Инструкции процессора

Примерное представление о том, как все работает на уровне процессора, можно получить, внимательно изучив выходные данные компилятора и конкретные инструкции, выполняемые процессором.

Краткое введение в ассемблирование

При компиляции программного кода, написанного на любом компилируемом языке, например на Rust или на C, код преобразуется в *машинные инструкции*, выполняемые процессором, который в конечном итоге и запустит создаваемую программу. Эти инструкции весьма специфичны для архитектуры того процессора, под который компилируется программа.

Такие инструкции, также называемые *машинным кодом*, закодированы в двоичной форме, которая для нас, людей, совершенно нечитаема. *Ассемблер* — это удобочитаемое представление инструкций процессора. Каждая инструкция представлена одной строкой текста, обычно начинающейся с какого-либо слова или аббревиатуры, обозначающей инструкцию, за которой следуют ее аргументы или операнды. *Ассемблер* преобразует текстовое представление в двоичную форму, а *дизассемблер* выполняет обратную операцию.

После компиляции программы на таком языке, как Rust, основная часть структуры исходного кода утрачивается. В зависимости от уровня оптимизации, функции и вызовы функций могут оставаться узнаваемыми. Но такие типы, как структуры или перечисления, сводятся к байтам и адресам, а циклы и условные выражения преобразуются в плоскую структуру с базовыми инструкциями перехода или ветвления.

Рассмотрим в качестве примера фрагмент ассемблера небольшой части программы для некой выдуманной архитектуры:

```
ldr x, 1234 // загрузка из адреса памяти 1234 в x
li y, 0     // присвоение y нулевого значения
inc x       // увеличение значения x на единицу
add y, x    // прибавление x к y
mul x, 3    // умножение x на 3
cmp y, 10   // сравнение y с 10
jne -5      // переход на пять инструкций назад, если не равно
str 1234, x // сохранение x по адресу памяти 1234
```

Здесь `x` и `y` — имена *регистров*. Регистры являются частью процессора, а не основной памяти, и обычно содержат одно целое число или адрес памяти. В 64-разрядных архитектурах их размер обычно составляет 64 бита. Число регистров варьируется в зависимости от архитектуры, но обычно оно очень ограничено. Регистры в основном служат в качестве временной памяти в вычислениях, места для хранения промежуточных результатов перед их обратным сохранением в памяти.

Константы, относящиеся в приведенном примере к конкретным адресам памяти, например 1234 и –5, часто заменяются более удобочитаемыми *метками*. Ассемблер при преобразовании в двоичный машинный код автоматически заменит их фактическим адресом.

Предыдущий пример при использовании меток мог бы выглядеть следующим образом:

```
        ldr x, SOME_VAR
        li y, 0
my_loop: inc x
        add y, x
        mul x, 3
        cmp y, 10
        jne my_loop
        str SOME_VAR, x
```

Поскольку имена меток являются не двоичным машинным кодом, а лишь частью ассемблирования, дизассемблер не будет знать, какие метки были изначально, и, скорее всего, просто сгенерирует произвольные, совершенно бессмысленные имена, например `label1` и `var2`.

Полный курс ассемблирования для всевозможных архитектур выходит за рамки тематики нашей книги и не является обязательным условием для прочтения данной главы. Чтобы разобраться в приводимых в ней примерах вполне достаточно самого общего представления, поскольку все будет сводиться к прочтению ассемблера, а не к написанию на нем программ. Соответствующие инструкции в каждом примере будут сопровождаться весьма подробными объяснениями, позволяющими понять их смысл, не имея предварительного опыта ассемблирования.

Чтобы увидеть конкретный машинный код, создаваемый компилятором Rust, у нас имеется несколько вариантов. Можно скомпилировать код обычным порядком, а затем воспользоваться дизассемблером (например, objdump), чтобы превратить созданный двоичный файл обратно в ассемблер. Используя отладочную информацию, предоставляемую компилятором в процессе компиляции, дизассемблер может создавать метки, соответствующие оригинальным именам функций исходного кода Rust. Недостаток такого подхода — необходимость наличия дизассемблера, поддерживающего конкретную процессорную архитектуру, для которой компилируется код. Хотя компилятор Rust поддерживает множество архитектур, многие дизассемблеры поддерживают только ту архитектуру, для которой они были скомпилированы.

Проще заставить компилятор создать вместо двоичного файла ассемблерный код, воспользовавшись при запуске `rustc` флагом `--emit=asm`. Недостатком этого метода будет наличие в полученном результате множества ненужных строк, содержащих информацию для ассемблера и инструментов отладки, в которых мы совершенно не нуждаемся.

Кроме этого, нам доступны превосходные инструменты, например `cargo-show-asm`, способные интегрироваться с `cargo` и автоматизировать процесс компиляции вашего крейта с нужными флагами, поиска соответствующего ассемблерного кода для интересующей вас функции и выделения соответствующих строк, содержащих фактические инструкции.

Для относительно небольших фрагментов самый простой и настоятельно рекомендуемый способ — веб-сервис, например замечательный во всех отношениях Compiler Explorer от Мэтта Годболта (Matt Godbolt). Данный веб-сайт позволяет создавать код на нескольких языках, включая Rust, и напрямую просматривать

соответствующий скомпилированный ассемблерный код, используя выбранную версию компилятора. В нем даже предусмотрено цветовое выделение для показа, какие именно строки Rust каким строкам ассемблера соответствуют, поскольку после оптимизации такое соответствие все еще существует.

Поскольку нам потребуется просматривать ассемблерный код для разных архитектур, нужно будет указать, для какой именно архитектуры будет компилироваться код на языке Rust. Для *x*86-64 мы выберем вариант `x86_64-unknown-linux-musl`, а для ARM64 — `aarch64-unknown-linux-musl`. Их поддержка уже имеется непосредственно в Compiler Explorer. Если компиляция выполняется на локальной машине, например с помощью `cargo-show-asm` или других, упомянутых ранее приемов, необходимо будет убедиться, что стандартная библиотека Rust установлена именно для этих целевых архитектур, что обычно делается с помощью `rustup target add`.

Во всех случаях целевая архитектура для компиляции выбирается с помощью флага компилятора `--target`. Например, `--target=aarch64-unknown-linux-musl`. Если целевая архитектура не указана, будет автоматически выбрана платформа, на которой вы сейчас находитесь. Или же, в случае применения Compiler Explorer, платформа, на которой размещен данный веб-сайт (в настоящее время это `x86_64-unknown-linux-gnu`).

Кроме того, для включения оптимизации желательно будет установить флаг `-O` (или же, если используется Cargo, флаг `--release`), при этом включится оптимизация и отключится проверка переполнения, что может существенно сократить объем создаваемого ассемблерного кода для небольших, рассматриваемых далее функций.

Чтобы опробовать это на практике, давайте посмотрим на ассемблерный код для *x*86-64 и ARM64, скомпилированный для следующей функции:

```
pub fn add_ten(num: &mut i32) {
    *num += 10;
}
```

Установив флаги компилятора `-O --target=aarch64-unknown-linux-musl` любым из рассмотренных ранее способов, мы получим что-то вроде следующего вывода ассемблерного кода для ARM64:

```
add_ten:
    ldr w8, [x0]
    add w8, w8, #10
    str w8, [x0]
    ret
```

В регистре `x0` содержится аргумент нашей функции `num` — адрес `i32`, который нужно увеличить на десять. Сначала инструкция `ldr` загружает 32-битовое значение из этого адреса памяти в регистр `w8`. Затем инструкция `add` прибавляет десять к `w8` и сохраняет результат обратно в `w8`. После этого инструкция `str` сохраняет регистр `w8` обратно в том же адресе памяти. И наконец, инструкция `ret` отмечает конец функции и заставляет процессор вернуться назад и продолжить выполнение функции, вызвавшей `add_ten`.

Если точно такой же код скомпилировать для `x86_64-unknown-linux-musl`, то будет получено что-то вроде следующего ассемблерного кода:

```
add_ten:
    add dword ptr [rdi], 10
    ret
```

На этот раз для аргумента `num` используется регистр `rdi`. Еще интереснее то, что в *x*86-64 одна инструкция `add` может выполнять то, что в ARM64 занимает три инструкции: загрузку, увеличение значения и его сохранение.

Обычно это происходит в архитектуре компьютера *со сложным набором машинных команд* (CISC), например в *x*86. Инструкции для такой архитектуры зачастую имеют множество вариантов, например для работы с регистром или для прямой работы с памятью определенного размера. (`dword` в ассемблерном коде указывает на 32-битовую операцию.)

А вот архитектура компьютера *с сокращенным набором машинных команд* (RISC), такая как ARM, обычно имеет более простой набор инструкций с весьма скромным числом вариантов. Большинство инструкций могут работать только с регистрами, а для загрузки и сохранения в памяти требуется отдельная инструкция. Это позволяет использовать более простой процессор, что может привести к снижению финансовых затрат, а иногда и к повышению производительности.

Как вскоре будет показано, это различие особенно актуально для атомарных инструкций выборки-изменения.

Несмотря на достаточно разумный подход к своей работе, компиляторы не всегда генерируют наиболее оптимальный ассемблерный код, особенно при наличии атомарных операций. Если при проведении экспериментов обнаружатся смущающие вас случаи, и, казалось бы, ненужная сложность ассемблерного кода, зачастую это просто будет означать, что для будущей версии компилятора открыты более удачные возможности оптимизации.

Загрузка и сохранение

Прежде чем углубиться во что-то более сложное, давайте сначала посмотрим на инструкции загрузки и сохранения, принятые для самых основных атомарных операций.

Обычное неатомарное сохранение с помощью `&mut i32` выполняется и на *x*86-64, и на ARM64 всего за одну инструкцию (см. табл. 7.1).

Таблица 7.1

Исходный код на Rust	Скомпилировано для *x*86-64	Скомпилировано для ARM64

`pub fn a(x: &mut i32) {`
`  *x = 0;`
`}`
	`a:`
	`  mov dword ptr [rdi], 0`
	`  ret`
		`a:`
		`  str wzr, [x0]`
		`  ret`

Имеющаяся в *x*86-64 весьма универсальная инструкция `mov` предназначена для копирования ("перемещения") данных из одного места в другое; в нашем случае из нулевой константы в память. В ARM64 для сохранения 32-разрядного регистра в памяти предусмотрена инструкция `str` (сохранить регистр). В данном случае задействован специальный регистр `wzr`, который всегда содержит нуль.

Если изменить код и вместо этого воспользоваться расслабленным атомарным сохранением с использованием `AtomicI32`, будут получены следующие результаты (см. табл. 7.2).

Таблица 7.2

Исходный код на Rust	Скомпилировано для *x*86-64	Скомпилировано для ARM64

`pub fn a(x: &AtomicI32) {`
`  x.store(0, Relaxed);`
`}`
	`a:`
	`  mov dword ptr [rdi], 0`
	`  ret`
		`a:`
		`  str wzr, [x0]`
		`  ret`

Как ни удивительно, но ассемблерный код идентичен тому, что был скомпилирован для неатомарной версии. Оказывается, инструкции `mov` и `str` уже были атомарными. Они либо выполнялись, либо не выполнялись вообще. По-видимому, любая разница между `&mut i32` и `&AtomicI32` здесь актуальна только для проверок и оптимизации компилятора, но не имеет никакого смысла для процессора — по крайней мере, для расслабленных операций сохранения на этих двух архитектурах.

То же самое происходит, если посмотреть на операции расслабленной загрузки (см. табл. 7.3).

Таблица 7.3

Исходный код на Rust	Скомпилировано для *x*86-64	Скомпилировано для ARM64

`pub fn a(x: &i32) -> i32 {`
`  *x`
`}`
	`a:`
	`  mov eax, dword ptr [rdi]`
	`  ret`
		`a:`
		`  ldr w0, [x0]`
		`  ret`

`pub fn a(x: &AtomicI32) -> i32 {`
`  x.load(Relaxed)`
`}`
	`a:`
	`  mov eax, dword ptr [rdi]`
	`  ret`
		`a:`
		`  ldr w0, [x0]`
		`  ret`

В *x*86-64 снова присутствует инструкция `mov`, только на этот раз для копирования из памяти в 32-разрядный регистр `eax`. В ARM64 загрузку значения из памяти в регистр `w0` выполняет инструкция `ldr` (загрузить в регистр).

32-разрядные регистры `eax` и `w0` служат для обратной передачи 32-битового возвращаемого значения функции. (Для 64-битовых значений используются 64-разрядные регистры `rax` и `x0`.)

И пусть процессор, что вполне очевидно, не различает атомарные и неатомарные операции сохранения и загрузки, спокойно игнорировать разницу между ними

в создаваемом коде Rust мы просто не можем. Если воспользоваться `&mut i32`, компилятор Rust может предположить, что ни один другой поток не может одновременно обращаться к тому же `i32`, и может принять решение о таком преобразовании или оптимизации кода, чтобы операция сохранения больше не приводила к созданию единственной соответствующей инструкции сохранения. Например, было бы совершенно корректно, хотя и несколько необычно, чтобы неатомарная 32-битовая операция загрузки или сохранения происходила с помощью двух отдельных 16-разрядных инструкций.

Операции чтения-изменения-записи

Гораздо интереснее обстоят дела с операциями *чтения-изменения-записи*, например с такими, как сложение. Как уже ранее здесь рассматривалось, неатомарная операция чтения-изменения-записи в RISC-архитектуре, такой как ARM64, обычно компилируется в три отдельные инструкции (чтение, изменение и запись), а в CISC-архитектуре типа *x*86-64 она зачастую может быть выполнена в одной инструкции. Что и демонстрируется в следующем коротком примере (см. табл. 7.4).

Таблица 7.4

Исходный код на Rust	Скомпилировано для *x*86-64	Скомпилировано для ARM64

`pub fn a(x: &mut i32) {`
`  *x += 10;`
`}`
	`a:`
	`  add dword ptr [rdi], 10`
	`  ret`
		`a:`
		`  ldr w8, [x0]`
		`  add w8, w8, #10`
		`  str w8, [x0]`
		`  ret`

Прежде чем взглянуть на соответствующую атомарную операцию, можно вполне резонно предположить, что на этот раз мы увидим разницу между неатомарной и атомарной версиями. Версия ARM64 здесь явно не атомарна, поскольку загрузка и сохранение происходят на отдельных этапах выполнения кода.

Хотя из самого ассемблерного кода это и не очевидно, версия *x*86-64 не является атомарной. Инструкция прибавления будет незаметно разбита процессором на несколько *микроинструкций* с отдельными этапами загрузки значения и сохранения результата. Это не имело бы никакого значения для одноядерного компьютера, поскольку переключение ядра процессора между потоками обычно происходит только между инструкциями. Но когда инструкции выполняются параллельно сразу несколькими ядрами, то без учета нескольких этапов, связанных с выполнением одной инструкции, больше уже нельзя предполагать, что все инструкции выполняются атомарно.

Префикс *lock* в *x*86

Для поддержки многоядерных систем Intel ввела префикс инструкций под названием `lock`. Он служит модификатором таких инструкций, как `add`, чтобы сделать их операцию атомарной.

Префикс lock первоначально заставлял процессор в момент выполнения инструкции временно блокировать доступ к памяти со стороны всех других ядер. Хотя это простой и эффективный способ сделать что-то атомарным для других ядер, останавливать весь мир для каждой атомарной операции крайне неэффективно. Новые процессоры имеют гораздо более совершенную реализацию префикса lock, которая не мешает другим ядрам работать с несвязанной памятью и позволяет ядрам делать что-то полезное в ожидании высвобождения определенного фрагмента памяти.

Префикс lock может применяться только к очень ограниченному числу инструкций, включая add, sub, and, not, or и xor, которые являются весьма полезными операциями, способными выполняться атомарно. Инструкция xchg (обменять), соответствующая операции атомарного обмена, имеет подразумеваемый префикс блокировки: она ведет себя как lock xchg, независимо от наличия или отсутствия префикса lock.

Давайте посмотрим, как действует lock add, изменив последний приведенный пример для работы с AtomicI32 (см. табл. 7.5).

Таблица 7.5

Исходный код на Rust	Скомпилировано для x86-64

`pub fn a(x: &AtomicI32) {`
`  x.fetch_add(10, Relaxed);`
`}`
	`a:`
	`  lock add dword ptr [rdi], 10`
	`  ret`

Как и ожидалось, единственное отличие от неатомарной версии — наличие префикса lock.

В приведенном примере возвращаемое значение из fetch_add, значение x перед этой операцией мы просто игнорируем. Но если им воспользоваться, возможностей инструкции add уже не хватит. Инструкция add может предоставить небольшую полезную информацию для следующей инструкции, например было ли обновляемое значение нулевым или отрицательным, но она не предоставляет полное (исходное или обновленное) значение. Вместо нее можно воспользоваться другой инструкцией: xadd ("обменять и прибавить"), которая помещает первоначально загруженное значение в регистр.

Все это можно увидеть в действии, внеся в наш код небольшое изменение, позволяющее вернуть значение, возвращаемое fetch_add (см. табл. 7.6).

Таблица 7.6

Исходный код на Rust	Скомпилировано для x86-64

`pub fn a(x: &AtomicI32) -> i32 {`
`  x.fetch_add(10, Relaxed)`
`}`
	`a:`
	`  mov eax, 10`
	`  lock xadd dword ptr [rdi], eax`
	`  ret`

Теперь вместо константы 10 задействован регистр, содержащий значение 10. Инструкция xadd повторно использует этот регистр для сохранения старого значения.

К сожалению, кроме xadd и xchg, такого варианта нет больше ни у одной инструкции с префиксом lock, включая sub, and и or. Например, нет такой инструкции, как xsub. Для вычитания это не проблема, поскольку xadd можно использовать и с отрицательным значением. А вот для and и or подобной альтернативы не существует.

Для операций and, or и xor, влияющих только на один бит, таких как fetch_or(1) или fetch_and(!1), можно воспользоваться инструкциями bts (проверить и установить бит), btr (проверить и сбросить бит) и btc (провести побитовую проверку и дополнение). Эти инструкции также допускают наличие префикса lock, но изменяют только один бит и делают предыдущее значение этого бита доступным для следующей инструкции, например для условного перехода.

Когда операции воздействуют на более чем один бит, они не могут быть представлены всего одной инструкцией *x*86-64. Аналогично этому операции fetch_max и fetch_min также не имеют соответствующей им инструкции *x*86-64. Для этих операций нужна другая стратегия, отличная от использования простого префикса lock.

*x*86-инструкция сравнения-обмена

В *разделе "Операции сравнения и обмена" главы 2* было показано, как любая атомарная операция выборки-изменения может быть реализована в виде цикла сравнения-обмена. Именно так компилятор и поступает для тех операций, которые нельзя представить одной инструкцией *x*86-64, поскольку эта архитектура включает в себя инструкцию cmpxchg (сравнить и обменять), допускающую префикс lock.

Как это работает, можно увидеть, заменив в последнем примере fetch_add на fetch_or (см. табл. 7.7).

Таблица 7.7

Исходный код на Rust	Скомпилировано для *x*86-64

`pub fn a(x: &AtomicI32) -> i32 {`
`  x.fetch_or(10, Relaxed)`
`}`
	`a:`
	`  mov eax, dword ptr [rdi]`
	`.L1:`
	`  mov ecx, eax`
	`  or ecx, 10`
	`  lock cmpxchg dword ptr [rdi], ecx`
	`  jne .L1`
	`  ret`

Первая инструкция mov загружает значение атомарной переменной в регистр eax. Следующие инструкции mov и or копируют это значение в ecx и применяют двоичную операцию or, в результате чего eax содержит старое значение, а ecx — новое. Затем инструкция cmpxchg ведет себя точно так же, как метод compare_exchange в Rust. Ее первый аргумент — адрес памяти, с которым нужно работать (атомарная переменная), второй аргумент (ecx) — новое значение, ожидаемое значение подразумеваемо берется из eax, а возвращаемое — сохраняется в eax. Эта инструкция также устанавливает флаг состояния, которым последующая инструкция может восполь-

зоваться для условного перехода в зависимости от того, успешна операция или нет. В данном случае инструкция `jne` (перейти, если не равно) обеспечивает возврат к метке `.L1` и повторную попытку в случае неудачи.

А вот как выглядит эквивалентный цикл сравнения-обмена в Rust, он точно такой же, как и показанный в *разделе "Операции сравнения и обмена" главы 2*:

```
pub fn a(x: &AtomicI32) -> i32 {
    let mut current = x.load(Relaxed);
    loop {
        let new = current | 10;
        match x.compare_exchange(current, new, Relaxed, Relaxed) {
            Ok(v) => return v,
            Err(v) => current = v,
        }
    }
}
```

В результате компиляции этого фрагмента будет получен такой же ассемблерный код, что и в версии `fetch_or`. Тем самым показывается, что по крайней мере на *x*86-64 они действительно эквивалентны во всех отношениях.

На *x*86-64 между `compare_exchange` и `compare_exchange_weak` нет никакой разницы. Обе инструкции компилируются в инструкцию `lock cmpxchg`.

Инструкции *load-linked* и *store-conditional*

В RISC-архитектуре ближе всего к циклу сравнения-обмена находится цикл *load-linked/store-conditional* (LL/SC). Он включает в себя две специальные инструкции, следующие парой: инструкцию связанной загрузки, которая в основном ведет себя как обычная инструкция загрузки, и инструкцию условного сохранения, которая в основном ведет себя как обычная инструкция сохранения. Они используются парой, причем обе инструкции нацелены на один и тот же адрес памяти. Ключевое отличие от обычных инструкций загрузки и сохранения заключается в том, что сохранение является условным: в сохранении в память возникает отказ, если та же самая память после выполнения инструкции связанной загрузки была перезаписана каким-либо другим потоком.

Эти две инструкции позволяют загрузить значение из памяти, изменить его и сохранить новое значение обратно только в том случае, если никто не перезаписал значение с момента его загрузки. В случае неудачи можно просто повторить попытку. Как только все получится, можно смело притвориться, что вся операция была атомарной, поскольку прерванной она не была.

Ключом к тому, чтобы сделать эти инструкции выполнимыми и эффективными в реализации, являются два фактора: 1) одновременно отслеживаться может только один адрес памяти (на каждое ядро) и 2) инструкции условного сохранения разре-

шается иметь ложноотрицательные значения, поэтому значение может не сохраниться, даже если в данном конкретном фрагменте памяти ничего не изменилось.

В результате отслеживание изменений в памяти может быть менее точным, возможно, за счет нескольких дополнительных проходов цикла LL/SC. Доступ к памяти может отслеживаться не по байту, а по блоку в 64 байта, или по килобайту, или даже по всей памяти в целом. Менее точное отслеживание памяти приводит к увеличению числа ненужных проходов в циклах LL/SC, что значительно снижает производительность, но также уменьшает и сложность реализации.

Если довести все до крайности, то базовая гипотетическая одноядерная система может использовать стратегию, при которой она вообще не будет отслеживать записи в память. Вместо этого она может отслеживать прерывания или переключения контекста, т. е. события, которые могут заставить процессор переключиться на другой поток. Если бы в системе без какого-либо параллелизма такого события не произошло, можно было бы с уверенностью предположить, что никакой другой поток не мог что-либо сделать с памятью. Если бы такое событие произошло, инструкция могла бы просто предположить наихудшее, отказаться от сохранения и надеяться на более удачный исход в следующей итерации цикла.

ARM-инструкции *load-exclusive* и *store-exclusive*

В ARM64 или, по крайней мере, в первой версии ARMv8 ни одна атомарная операция выборки-изменения или сравнения-обмена не может быть представлена всего одной инструкцией. В соответствии со свойствами, присущими RISC-архитектуре, этапы загрузки и сохранения отделены от вычисления и сравнения.

Инструкции связанной загрузки и условного сохранения в ARM64 называются `ldxr` (загрузить эксклюзивный регистр) и `stxr` (сохранить эксклюзивный регистр). Кроме этого, в качестве альтернативы `stxr`, позволяющей прекратить отслеживание записей в память без сохранения чего-либо, можно воспользоваться инструкцией `clrex` (очистить эксклюзив).

Чтобы увидеть их в действии, давайте посмотрим, что происходит при выполнении в ARM64 атомарного прибавления (см. табл. 7.8).

Таблица 7.8

Исходный код на Rust	Скомпилировано для ARM64

`pub fn a(x: &AtomicI32) {`
`  x.fetch_add(10, Relaxed);`
`}`
	`a:`
	`.L1:`
	`  ldxr w8, [x0]`
	`  add w9, w8, #10`
	`  stxr w10, w9, [x0]`
	`  cbnz w10, .L1`
	`  ret`

Получается нечто, очень похожее на полученную ранее (в *разделе "Операции чтения-изменения-записи"*) неатомарную версию: инструкцию загрузки, инструкцию

прибавления и инструкцию сохранения. Инструкции загрузки и сохранения были заменены их "эксклюзивной" LL/SC-версией, кроме того, появилась новая инструкция `cbnz` (сравнить и перейти при ненулевом значении). Инструкция `stxr` сохраняет нуль в `w10`, если она завершилась успешно, или единицу, если нет. Инструкция `cbnz` учитывает это для перезапуска всей операции в случае ее неудачного исхода.

Заметьте, что, в отличие от `lock add` в *x*86-64, здесь для получения старого значения не нужно делать ничего особенного. В показанном примере после успешного завершения операции старое значение все еще будет доступно в регистре `w8`, поэтому в такой специальной инструкции, как `xadd`, нет никакой необходимости.

Данный паттерн LL/SC достаточно гибок: он работает практически для любой операции, а не только для ограниченного набора таких операций, как `add` или `or`. Так же легко можно реализовать и атомарные `fetch_divide` или `fetch_shift_left`, поместив соответствующие инструкции между инструкциями `ldxr` и `stxr`. Но если между ними окажется слишком много инструкций, вероятность сбоя, приводящего к дополнительным циклам, возрастет. Как правило, компилятор старается сохранить в паттерне LL/SC как можно меньшее количество инструкций, чтобы избежать циклов LL/SC, которые весьма редко завершаются успешно (или даже вообще никогда не завершаются) и могут, таким образом, выполняться бесконечно.

Атомарные инструкции в ARMv8.1

Более поздняя версия ARM64, являющаяся частью ARMv8.1, также включает ряд новых инструкций в стиле CISC, предназначенных для наиболее распространенных атомарных операций. Например, новая инструкция `ldadd` (загрузить и прибавить) эквивалентна атомарной операции `fetch_add` без необходимости применения цикла LL/SC. В нее даже включены инструкции для таких операций, как `fetch_max`, которых нет в *x*86-64.

Также в нее включена инструкция `cas` (сравнить и обменять), соответствующая и инструкции `compare_exchange`. При использовании этой инструкции, как и в *x*86-64, между `compare_exchange` и `compare_exchange_weak` нет никакой разницы.

При всей гибкости паттерна LL/SC и его удачном соответствии общему RISC-паттерну эти новые инструкции могут быть более производительными, поскольку они легче поддаются оптимизации с помощью специализированного компьютерного оборудования.

ARM-инструкции сравнения-обмена

Операция `compare_exchange` весьма удачно соотносится с паттерном LL/SC путем использования команды условного перехода для пропуска инструкции сохранения, если сравнение не удалось. Давайте посмотрим на сгенерированный ассемблерный код (см. табл. 7.9).

Заметьте, что операция `compare_exchange_weak` обычно используется в цикле, повторяющемся при неудавшемся сравнении. Но в данном примере она вызывается только один раз, а возвращаемое значение игнорируется, что позволяет нам рассмотреть соответствующий ассемблерный код, ни на что не отвлекаясь.

Инструкция `ldxr` загружает значение, которое затем тут же с помощью инструкции `cmp` (сравнить) сравнивается с ожидаемым значением 5. Инструкция `b.ne` (перейти,

Таблица 7.9

Исходный код на Rust	Скомпилировано для ARM64

`pub fn a(x: &AtomicI32) {`
`  x.compare_exchange_weak(5, 6, Relaxed, Relaxed);`
`}`
	`a:`
	`  ldxr w8, [x0]`
	`  cmp w8, #5`
	`  b.ne .L1`
	`  mov w8, #6`
	`  stxr w9, w8, [x0]`
	`  ret`
	`.L1:`
	`  clrex`
	`  ret`

если не равно) вызовет переход к метке `.L1`, если значение было не таким, как ожидалось, и здесь для прерывания паттерна LL/SC присутствует инструкция `clrex`. Если значение было равно пяти, ход программы продолжается инструкциями `mov` и `stxr` для сохранения в памяти нового значения шесть, но только в том случае, если за это время ничем не было перезаписано значение пять.

Не нужно забывать, что выполнение `stxr` может выдавать ложноотрицательный результат, и здесь может произойти сбой, даже если пятерка не была перезаписана. Это вполне нормально, поскольку мы применяем инструкцию `compare_exchange_weak`, которой также позволительно иметь ложноотрицательные результаты. Фактически именно по этой причине и существует слабая версия `compare_exchange`.

Если заменить `compare_exchange_weak` на `compare_exchange`, будет получен почти что идентичный ассемблерный код, за исключением дополнительного ветвления для перезапуска операции в случае ее неудачного завершения (см. табл. 7.10).

Таблица 7.10

Исходный код на Rust	Скомпилировано для ARM64

`pub fn a(x: &AtomicI32) {`
`  x.compare_exchange(5, 6, Relaxed, Relaxed);`
`}`
	`a:`
	`  mov w8, #6`
	`.L1:`
	`  ldxr w9, [x0]`
	`  cmp w9, #5`
	`  b.ne .L2`
	`  stxr w9, w8, [x0]`
	`  cbnz w9, .L1`
	`  ret`
	`.L2:`
	`  clrex`
	`  ret`

Как и ожидалось, теперь для перезапуска цикла LL/SC в случае сбоя в коде есть дополнительная инструкция `cbnz` (сравнить и перейти при ненулевом значении).

Кроме того, чтобы сделать цикл как можно короче, из него была вынесена инструкция `mov`.

Оптимизация циклов сравнения-обмена

Ранее, в *разделе "x86-инструкция сравнения-обмена"* было показано, что операция `fetch_or` и эквивалентный цикл `compare_exchange` компилируются на *x*86-64 в совершенно одинаковые инструкции. Можно было бы ожидать, что такой же результат будет и на ARM, по крайней мере, с `compare_exchange_weak`, поскольку операции загрузки и слабого сравнения-обмена могут быть напрямую сопоставлены с инструкциями LL/SC.

К сожалению, в настоящее время (начиная с версии Rust 1.66.0) такого не происходит.

Но, поскольку компилятор постоянно совершенствуется, вполне вероятно, что в будущем ситуация может измениться, а пока компилятору довольно сложно безопасно превратить написанный от руки цикл сравнения-обмена в соответствующий цикл LL/SC. Одной из причин является действующее ограничение на число и тип инструкций, помещаемых между инструкциями `stxr` и `ldxr`, а это не входит в перечень того, что компилятор должен учитывать при применении других оптимизаций. На тот момент времени, когда такие паттерны, как цикл сравнения-обмена, все еще распознаваемы, точные инструкции, в которые будет скомпилировано выражение, еще не известны, что существенно усложняет реализацию такой оптимизации для общего случая.

Итак, по крайней мере, до получения более "смышленых" компиляторов, желательно по возможности задействовать не цикл сравнения-обмена, а специально предназначенные методы выборки-изменения.

Кеширование

Чтение из памяти и запись в нее происходят довольно медленно и могут легко потребовать столько же времени, сколько выполнение десятков или сотен инструкций. Вот почему во всех высокопроизводительных процессорах реализуется кеширование, позволяющее максимально избегать взаимодействия с относительно медленной памятью. Конкретные подробности реализации кеш-памяти в современных процессорах отличаются особой сложностью, они частично запатентованы и, что наиболее важно, по большей части не имеют для нас при написании программных средств абсолютно никакого значения. Как бы там ни было, само название *кеш* происходит от французского слова *caché*, что означает *скрытый*. И все же понимание основных принципов реализации кеширования большинством процессоров может быть чрезвычайно полезным при оптимизации программных средств для повышения их производительности. И конечно же, чтобы побольше узнать об интересующей нас теме, особый повод не нужен.

Если не брать в расчет весьма скромные микроконтроллеры, кеширование присутствует практически во всех современных процессорах. Именно такой процессор никогда не взаимодействует с основной памятью напрямую, вместо этого каждый отдельно взятый запрос на чтение и запись им направляется в свой кеш. Если инструкции нужно что-то считать из памяти, процессор запросит эти данные в своем кеше. Если запрашиваемая информация уже кеширована, кеш быстро ответит именно такими кешированными данными, избегая тем самым взаимодействия с основной памятью. В противном случае кеш будет вынужден пойти по медлен-

ному пути, где ему, возможно, придется запросить у основной памяти копию соответствующих данных. Как только основная память ответит, кеш не только наконец ответит на исходный запрос на чтение, но и запомнит данные, чтобы иметь возможность в следующий раз, когда эти данные будут запрошены, отреагировать на это гораздо быстрее. Если кеш переполняется, он освобождает место, удаляя некоторые старые данные, которые, по его мнению, наименее востребованы.

Когда инструкция хочет записать что-то в память, кеш может принять решение сохранить измененные данные, не записывая их в основную память. Любые последующие запросы чтения для того же адреса памяти будут получать копию измененных данных, игнорируя устаревшие данные, хранящиеся в основной памяти. Фактически кеш будет возвращать данные обратно в основную память только при необходимости удаления измененных данных из кеша для освобождения в нем места под другие данные.

В большинстве процессорных архитектур кеш считывает данные из памяти и записывает их в нее блоками по 64 байта, даже если был запрошен только один байт. Такие блоки зачастую называют *строками кеша*. Благодаря кешированию всего 64-байтового блока, окружающего запрошенный байт, любым последующим инструкциям, которым потребуется доступ к любому из других байтов в этом блоке, не придется ждать их поступления из оперативной памяти.

Согласованность кеша

В современных процессорах обычно предусмотрен не один уровень кеширования. Самым быстрым и небольшим по объему является первый кеш, или кеш первого уровня (L1). Вместо обращения к основной памяти, он обращается к кешу второго уровня (L2), который намного больше, но медленнее. Кеш L2 может взаимодействовать с основной памятью, или же с возможно имеющимся еще одним более крупным и медленным кешем L3, а может быть, даже и с кешем L4.

Добавление дополнительных уровней кеширования не сильно меняет их работу; каждый уровень может работать независимо от других. Но все становится гораздо интереснее, когда у процессора есть несколько ядер, каждое из которых обладает своим собственным кешем. В многоядерной системе каждое ядро процессора обычно имеет свой собственный кеш L1, тогда как кеши L2 или L3 часто используются совместно с некоторыми или со всеми другими ядрами.

В таких условиях упрощенная реализация кеширования быстро выйдет из строя, поскольку кеш больше не сможет предполагать, что контролирует все взаимодействия со следующим уровнем. Если один кеш примет запись и пометит некоторую строку кеша как измененную, не сообщая об этом другим кешам, состояние кешей может стать рассогласованным. Мало того, что измененные данные не будут доступны другим ядрам до тех пор, пока кеш не запишет данные на следующий уровень (или уровни), в конечном итоге они могут вступить в конфликт с различными изменениями, кешированными в других кешах.

Для решения данной проблемы предусмотрен *протокол поддержки согласованности кеша*, в котором определяется, как именно кеши работают и взаимодействуют

друг с другом, чтобы поддерживать все в согласованном состоянии. Конкретный протокол зависит от архитектуры, модели процессора и даже от уровня кеша.

Здесь будут рассмотрены два основных протокола поддержки согласованности кеша, но в современных процессорах существует множество их вариаций.

Протокол сквозной записи

В кешах, реализующих *протокол согласованности кеша со сквозной записью*, записи не кешируются, а тут же передаются на следующий уровень. Остальные кеши подключены к следующему уровню через тот же общий канал связи, что означает, что они могут наблюдать за связью других кешей со следующим уровнем. Когда кеш обнаруживает запись по адресу, который он в данный момент кеширует, он тут же либо удаляет, либо обновляет свою собственную строку кеша, чтобы сохранить общую согласованность.

Согласно этому протоколу кеши никогда не содержат строк кеша в измененном состоянии. Хотя это существенно упрощает задачу, но все же сводит на нет все преимущества кеширования операций записи. При оптимизации, рассчитанной только на чтение, данный протокол может стать отличным выбором.

Протокол MESI

Протокол согласованности кеша MESI назван в честь четырех возможных статусов, которые он определяет для строк кеша: modified (измененные), exclusive (эксклюзивные), shared (совместно используемые) и invalid (недействительные). Статус измененных (M) используется для строк кеша, содержащих данные, которые были изменены, но еще не записаны в память (или в кеш следующего уровня). Статус эксклюзивных (E) назначается для строк кеша, содержащих неизмененные данные, которые не кешируются ни в каком другом кеше (того же уровня). Статус совместно используемых (S) применяется для неизмененных строк кеша, которые также могут появиться в одном или нескольких других кешах (того же уровня). Статус недействительных (I) предназначен для неиспользуемых (пустых или удаленных) строк кеша, которые не содержат никаких полезных данных.

Кеши, реализующие этот протокол, взаимодействуют со всеми остальными кешами на том же уровне. Они отправляют друг другу обновления и запросы, чтобы сохранять согласованность друг с другом.

Когда кеш получает запрос на адрес, который им еще не был кеширован (что также называется *кеш-промахом*), он не запрашивает его сразу со следующего уровня, а сначала запрашивает другие кеши (того же уровня) на предмет наличия в каком-либо из них доступной строки кеша. Если ни у одного из них ее нет, кеш продолжит запрашивать адрес у следующего (более медленного) уровня и пометит полученную новую строку кеша как эксклюзивную (E). Когда впоследствии эта строка кеша изменится в результате выполнения операции записи, кеш может поменять ее статус на измененную (M), не сообщая об этом другим кешам, поскольку ему известно, что ни у одного другого кеша такой же кешированной строки кеша просто нет.

При запросе строки кеша, доступной в любом из других кешей, результатом станет строка со статусом совместно используемой (S), полученной непосредственно из другого кеша (или кешей). Если строка кеша находилась в статусе измененной (M), то прежде чем ее изменить, она сначала будет записана (или сброшена) на следующий уровень. Если строка находилась в статусе эксклюзивной (E), то ее статус будет тут же изменен на совместно используемую (S).

Если кешу потребуется эксклюзивный, а не совместно используемый доступ (например, потому что сразу после этого он собирается изменить данные), другой кеш не будет сохранять строку кеша в статусе совместно используемой (S), а вместо этого полностью ее удалит, изменив ее статус на недействительную (I). В таком случае в результате появится строка кеша со статусом эксклюзивной (E).

Если кешу потребуется эксклюзивный доступ к строке кеша, которая уже доступна в статусе совместного использования (S), он просто прикажет всем остальным удалить строку кеша, прежде чем обновлять ее статус до эксклюзивного (E).

Существует несколько вариантов этого протокола. Например, протокол *MOESI* добавляет дополнительный статус, позволяющий распространять измененные данные без их немедленной записи на следующий уровень, а протокол *MESIF* использует дополнительное состояние, чтобы решить, какой именно кеш будет отвечать на запрос строки в статусе совместно используемой и доступной сразу в нескольких кешах. В современных процессорах зачастую применяются более сложные и собственные протоколы поддержки согласованности кеша.

Влияние, оказываемое на производительность

Хотя кеширование по большей части скрыто от нас, его поведение может существенно повлиять на производительность наших атомарных операций. Давайте попробуем оценить влияние некоторых из имеющихся факторов.

Измерить скорость выполнения одной атомарной операции очень сложно, поскольку все они выполняются очень быстро. Чтобы получить какие-либо полезные численные показатели, придется повторить операцию, скажем, миллиард раз и измерить, сколько времени это займет в целом. Например, можно было бы попытаться измерить время, необходимое для миллиарда операций загрузки:

```
static A: AtomicU64 = AtomicU64::new(0);
fn main() {
    let start = Instant::now();
    for _ in 0..1_000_000_000 {
        A.load(Relaxed);
    }
    println!("{:?}", start.elapsed());
}
```

К сожалению, работать ожидаемым образом этот код не станет.

При запуске с включенной оптимизацией (например, с `cargo run --release` или `rustc -O`) мы увидим неоправданно низкий показатель затраченного времени. А произошло

вот что: компилятор оказался слишком умен, чтобы понять, что мы не используем загруженные значения, поэтому он решил полностью оптимизировать "ненужный" цикл.

Во избежание этого можно воспользоваться специальной функцией `std::hint::black_box`, которая принимает аргумент любого типа и просто возвращает его, ничего при этом не делая. Ее особенность заключается в том, что компилятор изо всех сил старается ничего не предполагать о том, что она делает; он обращается с ней как с "черным ящиком", в котором может делаться все что угодно.

Этим можно воспользоваться, чтобы избежать конкретных оптимизаций, превращающих тест в бесполезный. В данном случае, чтобы остановить любые оптимизации, предполагающие фактическую незаинтересованность в загруженных значениях, в `black_box()` можно будет передать результат загрузки. Но этого будет недостаточно, поскольку компилятор все равно может предположить, что `A` всегда равно нулю, что делает операцию загрузки ненужной. Во избежание этого в начале в функцию `black_box()` можно будет передать ссылку на `A`, чтобы компилятор больше не мог предположить, что есть только один поток, обращающийся к `A`. В конечном счете он должен предположить, что в `black_box(&A)` мог быть порожден дополнительный поток, взаимодействующий с `A`.

Давайте посмотрим, что из этого выйдет:

```
use std::hint::black_box;
static A: AtomicU64 = AtomicU64::new(0);
fn main() {
    black_box(&A); // New!
    let start = Instant::now();
    for _ in 0..1_000_000_000 {
        black_box(A.load(Relaxed)); // New!
    }
    println!("{:?}", start.elapsed());
}
```

Выходные данные при многократном запуске немного колеблются, но на не очень новом компьютере *x*86-64 этот тест, похоже, дает результат около 300 мс.

Чтобы оценить влияние факторов кеширования, давайте создадим фоновый поток, взаимодействующий с атомарной переменной. Тогда можно будет понять, повлияет ли это на операции загрузки основного потока или нет.

Во-первых, давайте попробуем, как показано далее, применить задуманное, просто запустив операции загрузки в фоновом потоке:

```
static A: AtomicU64 = AtomicU64::new(0);
fn main() {
    black_box(&A);
    thread::spawn(|| { // Новый код!
        loop {
            black_box(A.load(Relaxed));
        }
    });
```

```
    let start = Instant::now();
    for _ in 0..1_000_000_000 {
        black_box(A.load(Relaxed));
    }
    println!("{:?}", start.elapsed());
}
```

Заметьте: производительность операций в фоновом потоке не измеряется. Измеряется время, необходимое основному потоку для выполнения миллиарда операций загрузки.

Запуск этой программы дает такие же результаты, как и раньше: при тестировании на том же компьютере *x*86-64 оно немного колеблется в районе 300 мс. Фоновый поток не оказывает существенного влияния на основной поток. Предположительно, каждый из них работает на отдельном ядре процессора, но кеши обоих ядер содержат копию `A`, что обеспечивает очень быстрый доступ.

Теперь давайте изменим фоновый поток, чтобы вместо загрузки в нем выполнялись операции сохранения:

```
static A: AtomicU64 = AtomicU64::new(0);
fn main() {
    black_box(&A);
    thread::spawn(|| {
        loop {
            A.store(0, Relaxed); // Новый код!
        }
    });
    let start = Instant::now();
    for _ in 0..1_000_000_000 {
        black_box(A.load(Relaxed));
    }
    println!("{:?}", start.elapsed());
}
```

На этот раз видна существенная разница. Теперь при запуске программы на той же машине *x*86-64 получаем выходные данные в районе трех целых секунд, что почти в десять раз больше, чем раньше. Более поздние компьютеры покажут менее значительную, но все же вполне измеримую разницу. Например, на последнем процессоре Apple M1 время увеличилось с 350 до 500 мс, а на самом последнем процессоре AMD *x*86-64 с 250 до 650 мс.

Такое поведение соответствует нашему представлению о протоколах согласованности кеша: операция сохранения требует монопольного доступа к строке кеша, что замедляет последующие операции загрузки на других ядрах, которые больше не используют эту строку кеша совместно.

Неудачные операции сравнения-обмена

Интересно, что на большинстве процессорных архитектур тот же эффект, который мы наблюдали при операциях сохранения, проявляется и при выполнении фоновым потоком только лишь операций сравнения-обмена, даже если все они завершаются неудачно.

Чтобы посмотреть, как это происходит на практике, можно заменить операцию сохранения (в фоновом потоке) заведомо провальным вызовом compare_exchange:

```
...
    loop {
        // Провал гарантирован, поскольку A никогда
        // не будет равно 10.
        black_box(A.compare_exchange(10, 20, Relaxed, Relaxed).is_ok());
    }
...
```

Поскольку A всегда равно нулю, операция compare_exchange всегда будет провальной. Она загрузит текущее значение A, но никогда не заменит его новым значением.

Вполне обоснованно можно ожидать, что данная операция будет вести себя точно так же, как и операция загрузки, поскольку она не изменяет атомарную переменную. Но в большинстве процессорных архитектур инструкция compare_exchange будет требовать монопольный доступ к соответствующей строке кеша, независимо от того, удалось сравнение или нет.

Получается, что для проверки снятия блокировки разумнее вместо использования в спин-цикле операции compare_exchange (или swap), как это делалось для SpinLock в *главе 4*, воспользоваться операцией load. Это позволит избежать ненужного требования исключительного доступа к соответствующей строке кеша.

Поскольку кеширование происходит в кеше не по отдельно взятым байтам или переменным, а построчно, точно такой же эффект будет наблюдаться и при задействовании не одной и той же переменной, а какой-нибудь ее соседки. Чтобы убедиться в этом на практике, давайте воспользуемся не одной, а тремя атомарными переменными, и сделаем так, чтобы основной поток задействовал только среднюю переменную, а фоновый поток — только две другие переменные (листинг 7.1).

Листинг 7.1

```
static A: [AtomicU64; 3] = [
    AtomicU64::new(0),
    AtomicU64::new(0),
    AtomicU64::new(0),
];
fn main() {
    black_box(&A);
    thread::spawn(|| {
        loop {
            A[0].store(0, Relaxed);
            A[2].store(0, Relaxed);
        }
    });
    let start = Instant::now();
    for _ in 0..1_000_000_000 {
        black_box(A[1].load(Relaxed));
    }
    println!("{:?}", start.elapsed());
}
```

Запуск этого кода даст такие же результаты, что и раньше: на том же компьютере *x*86-64 его выполнение займет несколько секунд. Несмотря на то что каждая из переменных `A[0]`, `A[1]` и `A[2]` используются только одним потоком, все равно наблюдается все тот же эффект, как если бы одна и та же переменная была задействована в обоих потоках. Дело в том, что переменная `A[1]` совместно использует одну и ту же строку кеша с одной или с обеими другими переменными. Ядро процессора, выполняющее фоновый поток, многократно запрашивает исключительный доступ к строке (или строкам) кеша, содержащим `A[0]` и `A[2]`, где также содержится `A[1]`, замедляя тем самым выполнение "несвязанных" операций над `A[1]`. Этот эффект называется *ложным разделением*.

Избежать этого можно, расположив атомарные переменные подальше друг от друга, чтобы каждая из них получила свою собственную строку кеша. Как уже упоминалось ранее, 64 байта — вполне разумное предположение о размере строки кеша, поэтому давайте попробуем заключить наши атомарности в 64-байтовую выровненную структуру (листинг 7.2).

Листинг 7.2

```
#[repr(align(64))] // Эта структура должна быть выровнена по 64-байтовой границе.
struct Aligned(AtomicU64);
static A: [Aligned; 3] = [
    Aligned(AtomicU64::new(0)),
    Aligned(AtomicU64::new(0)),
    Aligned(AtomicU64::new(0)),
];
fn main() {
    black_box(&A);
    thread::spawn(|| {
        loop {
            A[0].0.store(1, Relaxed);
            A[2].0.store(1, Relaxed);
        }
    });
    let start = Instant::now();
    for _ in 0..1_000_000_000 {
        black_box(A[1].0.load(Relaxed));
    }
    println!("{:?}", start.elapsed());
}
```

Атрибут `#[repr(align)]` позволяет сообщить компилятору о минимальном выравнивании нашего типа в байтах. Поскольку размер `AtomicU64` составляет всего 8 байт, к нашей структуре `Aligned` будет добавлено 56 байтов заполнения.

Запуск этой программы больше не выдает медленных результатов. Вместо них получаются те же результаты, что и при отсутствии фонового потока: около 300 мс при запуске на прежнем компьютере *x*86-64.

В зависимости от типа процессора, на котором будет запускаться данная программа, для наблюдения точно такого же эффекта может потребоваться выравнивание по 128 байтам.

Приведенный эксперимент показывает, что желательно не располагать несвязанные атомарные переменные близко друг к другу. Например, плотный массив небольших мьютексов не всегда может работать так же хорошо, как альтернативная структура, в которой мьютексы располагаются на более значительном расстоянии друг от друга.

А вот когда несколько атомарных переменных связаны друг с другом и зачастую доступны в быстрой последовательности, будет полезнее, наверное, разместить их недалеко друг от друга. Например, `SpinLock<T>` из *главы 4* хранит `T` рядом с `AtomicBool`, следовательно, вполне вероятно, что строка кеша, содержащая `AtomicBool`, также будет содержать и `T`, так что требование исключительного доступа к одной из этих переменных будет также распространяться и на другую переменную. Насколько это окажется полезным, полностью зависит от конкретно складывающейся ситуации.

Переупорядочение

Согласованное кеширование, например с применением рассмотренного ранее в данной главе протокола MESI, обычно не влияет на корректность программы, даже если задействовано несколько потоков. Единственные различия, вызванные согласованным кешированием, наблюдаются только во времени ее выполнения. Но в современных процессорах реализуется гораздо больше всевозможных оптимизаций, способных существенно повлиять на корректность программы, по крайней мере, при наличии сразу нескольких потоков.

Переупорядочение инструкций, т. е. возможность изменения порядка их следования со стороны компилятора и процессора, уже рассматривалось в краткой форме в начале *главы 3*. Если принять во внимание один только процессор, то можно привести целый ряд примеров различных путей, заставляющих инструкции или вызываемые ими эффекты следовать *не по порядку*:

- *Буферы хранения* — поскольку запись, даже при наличии кеширования, может быть медленной, ядра процессора часто имеют собственный *буфер хранения*. Операции записи в память могут храниться в этом буфере хранения, что делается очень быстро, и это позволяет процессору немедленно продолжить выполнение следующих инструкций. Затем в фоновом режиме операция записи завершается записью в кеш (L1), что может занять гораздо больше времени. Таким образом, процессору для получения исключительного доступа к соответствующей строке кеша не придется ждать, пока вступит в действие протокол согласованности кеша.

  Поскольку при обработке последующих операций чтения из одного и того же адреса памяти уделяется особое внимание, это совершенно незаметно для инст-

рукций, выполняемых в рамках одного и того же потока на одном и том же ядре процессора. Но на некоторый короткий промежуток времени операция записи еще не видна другим ядрам, что приводит к несогласованному представлению о том, как выглядит память из разных потоков, работающих на разных ядрах.

- *Очереди объявления недействительности* — независимо от конкретного протокола согласованности, кеши, работающие параллельно, должны обрабатывать запросы на недействительность: инструкции по удалению определенной строки кеша, поскольку она вот-вот будет изменена и станет недействительной. Для оптимизации производительности такие запросы обычно обрабатываются не в срочном порядке, а ставятся в очередь для несколько более поздней обработки. При использовании таких вот очередей объявления недействительности кеши перестают быть согласованными, поскольку строки кеша, прежде чем удалиться, могут на короткое время устареть. На однопоточную программу, кроме ее ускорения, это абсолютно не влияет. Единственным последствием будет видимость операций записи из других ядер, которые теперь могут выглядеть (весьма незначительно) отсроченными.
- *Конвейеризация* — еще одна весьма распространенная особенность процессора, значительно повышающая производительность: по возможности параллельное выполнение последовательных инструкций. Процессор может начать выполнение следующей инструкции еще до завершения выполнения ее предшественницы. Современные процессоры нередко могут начать выполнение сразу нескольких инструкций подряд, пока первая из них еще выполняется.

  Если каждая инструкция оперирует результатом предыдущей инструкции, пользы от всего этого будет немного: каждой из них все еще нужно будет дожидаться результата выполнения своей предшественницы. Но когда инструкция может быть выполнена независимо от предыдущей инструкции, она может даже завершиться раньше своей предшественницы. Например, инструкция, просто увеличивающая значение регистра, может завершиться очень быстро, в то время как ранее запущенная инструкция может все еще ожидать чтения каких-либо данных из памяти или завершения какой-либо другой медленной операции.

  На однопоточную программу (кроме скорости) это абсолютно никак не влияет, а вот когда инструкция, работающая с памятью, завершает свое выполнение еще до того, как это сделает предыдущая инструкция, взаимодействие с другими ядрами может происходить не по порядку.

Есть множество путей, по которым современный процессор может выполнять инструкции в совершенно неожиданном порядке. При этом применяется целый ряд запатентованных методов, некоторые из которых становятся достоянием общественности только тогда, когда обнаруживается лазейка, позволяющая использовать вредоносный код. Но, когда эти методы работают безупречно, у них всех есть одна общая черта: они не влияют на однопоточные программы, не считая времени выполнения, но могут привести к тому, что взаимодействие с другими ядрами будет происходить в хаотичном порядке.

Архитектуры процессоров, позволяющие переупорядочивать операции с памятью, наряду с этим предоставляют возможность предотвратить переупорядочение с по-

мощью специальных инструкций. Эти инструкции могут, к примеру, заставить процессор очистить буфер хранения или завершить перед продолжением выполнения программы выполнение любых, попавших в конвейер инструкций. Иногда такие инструкции предотвращают только определенный тип переупорядочения. Например, может иметься инструкция, предотвращающая переупорядочение относительно друг друга операций сохранения и в то же время позволяющая переупорядочивать операции загрузки. Какие именно типы переупорядочения могут произойти и как их можно предотвратить, зависит от архитектуры процессора.

Упорядочение памяти

Чтобы при выполнении любой атомарной операции в таких языках, как Rust или C, сообщить компилятору о требованиях к упорядочению, мы указываем тип упорядочения памяти. Чтобы предотвратить переупорядочение инструкций теми путями, которые могут нарушить правила, компилятор сгенерирует для процессора нужные инструкции.

Какие типы переупорядочения команд разрешены, зависит от типа операции с памятью. Для неатомарных и расслабленных атомарных операций допускается любой тип переупорядочения. С другой стороны, последовательно согласованные атомарные операции вообще не допускают какого-либо изменения порядка.

Операцию получения нельзя переупорядочить с любыми последующими операциями с памятью, а операцию высвобождения нельзя переупорядочить с какими-либо предшествующими ей операциями с памятью. В противном случае доступ к данным, защищенным мьютексом, может быть получен до получения или после высвобождения мьютекса, что приведет к гонке данных.

Дополнительная атомарность с множественным копированием

Влияние порядка следования операций с памятью на некоторые процессорные архитектуры, например, на применяемые в видеокартах, не всегда объясняется переупорядочением инструкций. Эффект двух последовательных операций сохранения на одном ядре может быть виден в том же самом порядке на втором ядре, но в совершенно обратном порядке на третьем ядре. Такое может произойти, к примеру, из-за несогласованного кеширования или из-за совместно используемых буферов хранения. Этот эффект нельзя объяснить тем, что были переупорядочены инструкции первого ядра, поскольку этим не объясняется несогласованность между тем, что видится второму и третьему ядрам.

Теоретическая модель памяти, рассматриваемая в *главе 3*, оставляет место для таких процессорных архитектур, поскольку не требует глобально согласованного порядка для чего-либо, кроме последовательно согласованных атомарных операций.

Архитектуры, на которых сосредоточено наше внимание в данной главе, *x*86-64 и ARM64, обладают *дополнительной атомарностью с множественным копированием* (Other-multi-copy atomic), следовательно, как только операции записи становятся видимыми для любого ядра, они одновременно становятся видимыми для всех ядер. Для архитектур, обладающих дополнительной атомарностью с множественным копированием, упорядочение памяти сводится только к переупорядочению команд.

Некоторые архитектуры, например ARM64, называются *слабо упорядоченными*, поскольку они позволяют процессору свободно переупорядочивать любые опера-

ции с памятью. С другой стороны, *строго упорядоченные* архитектуры, например *x*86-64, весьма ограничены в отношении допустимости переупорядочения операций с памятью.

*x*86-64: строго упорядоченная архитектура

На процессоре *x*86-64 операция загрузки никогда не происходит после последующей операции с памятью. Точно так же данная архитектура не позволяет появляться операции сохранения там, где она может быть выполнена до предыдущей операции с памятью. Единственное переупорядочение, которое можно увидеть на *x*86-64, — это отсрочка операции сохранения до завершения более поздней операции загрузки.

Из-за ограничений на переупорядочение, накладываемых архитектурой *x*86-64, ее часто называют *строго упорядоченной* архитектурой, хотя некоторые специалисты предпочитают зарезервировать этот термин для тех архитектур, которые сохраняют порядок *всех* операций с памятью.

Эти ограничения удовлетворяют всем потребностям получения-загрузки (поскольку загрузка никогда не переупорядочивается с более поздней операцией) и высвобождения-сохранения (поскольку сохранение никогда не переупорядочивается с более ранней операцией). Это означает, что на *x*86-64 мы получаем семантику высвобождения и получения "без каких-либо затрат": операции высвобождения и получения идентичны расслабленным операциям.

В этом можно убедиться, посмотрев, что происходит с некоторыми фрагментами кода из предыдущих *разделов "Загрузка и сохранение"* и *"Префикс lock в x86"*, в которых `Relaxed` заменяется на `Release`, `Acquire` или `AcqRel` (см. табл. 7.11).

Таблица 7.11

Исходный код на Rust	**Скомпилировано для *x*86-64**

`pub fn a(x: &AtomicI32) {`
`  x.store(0, Release);`
`}`
	`a:`
	`  mov dword ptr [rdi], 0`
	`  ret`

`pub fn a(x: &AtomicI32) -> i32 {`
`  x.load(Acquire)`
`}`
	`a:`
	`  mov eax, dword ptr [rdi]`
	`  ret`

`pub fn a(x: &AtomicI32) {`
`  x.fetch_add(10, AcqRel);`
`}`
	`a:`
	`  lock add dword ptr [rdi], 10`
	`  ret`

Как и ожидалось, ассемблерный код идентичен предыдущему, хотя было указано более строгое упорядочение памяти.

Можно прийти к выводу, что на *x*86-64, если проигнорировать возможную оптимизацию со стороны компилятора, операции получения и высвобождения обходятся

ничуть не дороже расслабленных операций. Или же, если выразиться точнее, расслабленные операции обходятся ничуть не дешевле операций получения и высвобождения.

Давайте проверим, что получится при указании упорядочения `SeqCst` (см. табл. 7.12).

Таблица 7.12

Исходный код на Rust	**Скомпилировано для *x*86-64**

`pub fn a(x: &AtomicI32) {`
`  x.store(0, SeqCst);`
`}`
	`a:`
	`  xor eax, eax`
	`  xchg dword ptr [rdi], eax`
	`  ret`

`pub fn a(x: &AtomicI32) -> i32 {`
`  x.load(SeqCst)`
`}`
	`a:`
	`  mov eax, dword ptr [rdi]`
	`  ret`

`pub fn a(x: &AtomicI32) {`
`  x.fetch_add(10, SeqCst);`
`}`
	`a:`
	`  lock add dword ptr [rdi], 10`
	`  ret`

Операции `load` и `fetch_add` по-прежнему приводят к созданию точно такого же ассемблерного кода, что и раньше, а вот ассемблерный код для операции `store` полностью изменился. Инструкция `xor` выглядит странно, но это всего лишь распространенный способ обнулить регистр `eax` путем выполнения операции `xor` в отношении его самого, что всегда дает в результате нуль. Для этого подошла бы и инструкция `mov eax, 0`, но она занимает немного больше места.

Интерес здесь вызывает инструкция `xchg`, которая обычно предназначена для операции обмена: операции сохранения, которая также извлекает старое значение.

Прежде использованная инструкция `mov` для сохранения с упорядочением `SeqCst` не подойдет, поскольку она позволит переупорядочить себя с более поздней операцией загрузки, нарушая глобально согласованный порядок. Заменив ее операцией, также выполняющей загрузку, даже не заботясь о загружаемом ею значении, мы получаем дополнительную гарантию того, что наша инструкция не будет переупорядочена с последующими операциями с памятью, что и станет решением проблемы.

Операция загрузки с упорядочением `SeqCst` по-прежнему может быть выполнена с помощью обычной инструкции `mov` именно потому, что операции сохранения с упорядочением `SeqCst` переделаны в `xchg`. Операции с упорядочением `SeqCst` гарантируют глобально согласованный порядок только с другими операциями с упорядочением `SeqCst`. Инструкция `mov` из операции загрузки с упорядочением `SeqCst` может по-прежнему быть переупорядочена с `mov` из более ранней операции сохранения, не имеющей упорядочения `SeqCst`, но это вполне нормально.

Операция сохранения является в *x*86-64 единственной атомарной операцией, для которой существует разница между `SeqCst` и более слабым упорядочением памяти.

Иными словами, SeqCst-операции в *x*86-64, за исключением операций сохранения, обходятся ничуть не дороже Release-, Acquire-, AcqRel- и даже Relaxed-операций. Или же, если предпочитаете иное изложение, то в *x*86-64 Relaxed-операции, отличные от сохранения, обходятся ничуть не дешевле SeqCst-операций.

ARM64: слабо упорядоченная архитектура

В *слабо упорядоченной* архитектуре, например в ARM64, все операции с памятью потенциально могут быть переупорядочены друг с другом. Это означает, что, в отличие от архитектуры *x*86-64, операции получения и высвобождения не будут идентичны расслабленным операциям.

Давайте посмотрим, что происходит в ARM64 при упорядочении памяти Release, Acquire и AcqRel (см. табл. 7.13).

Таблица 7.13

Исходный код на Rust	Скомпилировано для ARM64

`pub fn a(x: &AtomicI32) {`
`  x.store(0, Release);`
`}`
	`a:`
	`  stlr wzr, [x0]` ❶
	`  ret`

`pub fn a(x: &AtomicI32) -> i32 {`
`  x.load(Acquire)`
`}`
	`a:`
	`  ldar w0, [x0]` ❷
	`  ret`

`pub fn a(x: &AtomicI32) {`
`  x.fetch_add(10, AcqRel);`
`}`
	`a:`
	`.L1:`
	`  ldaxr w8, [x0]` ❸
	`  add w9, w8, #10`
	`  stlxr w10, w9, [x0]` ❶
	`  cbnz w10, .L1`
	`  ret`

Изменения по сравнению с ранее рассмотренными версиями Relaxed весьма незначительны:

❶ str (сохранить регистр) теперь называется stlr (сохранить-высвободить регистр).

❷ ldr (загрузить регистр) теперь называется ldar (загрузить-получить регистр).

❸ ldxr (загрузить эксклюзивный регистр) теперь называется ldaxr (загрузить-получить эксклюзивный регистр).

❹ stxr (сохранить эксклюзивный регистр) теперь называется stlxr (сохранить-высвободить эксклюзивный регистр).

Как здесь показано, в ARM64 для упорядочения получения и высвобождения имеются специальные версии инструкций. В отличие от инструкций ldr или ldxr, инструкции ldar или ldxar никогда не будут переупорядочены при последующих операциях с памятью. Аналогично этому, в отличие от инструкций str или stxr, инструк-

ции stlr или stxlr никогда не будут переупорядочены при выполнении какой-либо предыдущей операции с памятью.

Операция выборки-изменения, применяющая вместо упорядочения AcqRel только Release или Acquire, использует только одну из инструкций stlxr и ldxar соответственно в сочетании с обычной инструкцией ldxr или stxr.

В дополнение к ограничениям, необходимым для семантики высвобождения и получения, ни одна из специальных инструкций получения и высвобождения никогда не будет переупорядочена с какой-либо другой из этих специальных инструкций, поэтому они также подходят для применения упорядочения SeqCst.

Из следующего примера видно, что замена на SeqCst приводит к получению точно такого же ассемблерного кода, что и раньше (см. табл. 7.14).

Таблица 7.14

Исходный код на Rust	Скомпилировано для ARM64

`pub fn a(x: &AtomicI32) {`
`  x.store(0, SeqCst);`
`}`
	`a:`
	`  stlr wzr, [x0]`
	`  ret`

`pub fn a(x: &AtomicI32) -> i32 {`
`  x.load(SeqCst)`
`}`
	`a:`
	`  ldar w0, [x0]`
	`  ret`

`pub fn a(x: &AtomicI32) {`
`  x.fetch_add(10, SeqCst);`
`}`
	`a:`
	`.L1:`
	`  ldaxr w8, [x0]`
	`  add w9, w8, #10`
	`  stlxr w10, w9, [x0]`
	`  cbnz w10, .L1`
	`  ret`

Это означает, что в ARM64 последовательно согласованные операции обходятся так же дешево, как и операции получения и высвобождения. Или же, точнее выражаясь, Acquire-, Release- и AcqRel-операции в ARM64 обходятся ничуть не дешевле SeqCst-операции. Но, в отличие от *x*86-64, Relaxed-операции относительно дешевы, поскольку они не влекут за собой применение гарантий упорядочения строже реально необходимых.

Инструкции атомарного высвобождения и получения в ARMv8.1

Как уже упоминалось ранее, в версию ARMv8.1 архитектуры ARM64 в качестве альтернативы циклу ldxr/stxr для атомарных операций, таких как ldadd (загрузить и прибавить), включены инструкции в стиле CISC.

Аналогично тому, что у операций загрузки и сохранения имеются специальные версии с семантикой получения и высвобождения, у этих инструкций также имеются варианты для более строгого упорядочения памяти. Поскольку эти инструкции включают в себя как загрузку, так и сохранение, каждая из них имеет три дополнительных варианта: один для высвобож-

дения (-l), один для получения (-a) и еще один для объединенной семантики высвобождения и получения (-al).

Например, для операции ldadd существуют также варианты ldaddl, ldadda и ldaddal. Точно так же и инструкция cas поставляется с вариантами casl, casa и casal.

По аналогии с инструкциями загрузки и сохранения для SeqCst-операций также достаточно будет объединенных вариантов высвобождения и получения (-al).

Эксперимент

К сожалению, из-за высокой популярности строго упорядоченных архитектур некоторые классы ошибок в упорядочении памяти могут легко оставаться незамеченными. Использовать упорядочение Relaxed там, где необходимо применять Acquire или Release, некорректно, но на практике, если компилятор не изменил порядок следования атомарных операций, на *x*86-64 может получиться так, что все будет работать должным образом.

Не забывайте, что сбои в работе может вызвать не только процессор. Компилятору также разрешено переупорядочивать производимые им инструкции при условии, что он учитывает ограничения, накладываемые на упорядочение памяти.

Фактически компиляторы в отношении оптимизации, включающей атомарные операции, как правило, весьма консервативны, но вполне вероятно, что в будущем ситуация может измениться.

Следовательно, можно без особого труда написать некорректный код для конкурентных вычислений, который (совершенно случайно) весьма успешно будет работать на *x*86-64, но может отказать при компиляции и запуске на процессоре ARM64.

Давайте попробуем именно так и сделать.

Изменим в создаваемом счетчике, защищенном спин-блокировкой, все упорядочение памяти на Relaxed, но при этом не станем утруждаться созданием собственного типа или небезопасного кода, а просто воспользуемся для блокировки AtomicBool, а для счетчика AtomicUsize.

Чтобы увериться в том, что компилятор не станет переупорядочивать наши операции, воспользуемся функцией std::sync::compiler_fence(), чтобы сообщить компилятору об операциях, которые должны были быть с упорядочением Acquire или Release, не сообщая об этом процессору.

А теперь заставим четыре потока многократно блокировать счетчик, увеличивать его значение и разблокировать его — по миллиону раз каждый. Сложив все вместе, мы получим код, приведенный в листинге 7.3.

Листинг 7.3

```
fn main() {
    let locked = AtomicBool::new(false);
    let counter = AtomicUsize::new(0);
```

```
    thread::scope(|s| {
        // Порождение четырех потоков, с проходом в каждом из них по миллиону раз.
        for _ in 0..4 {
            s.spawn(|| for _ in 0..1_000_000 {
                // Получение блокировки с неверным
                // указателем упорядочения памяти.
                while locked.swap(true, Relaxed) {}
                compiler_fence(Acquire);
                // Неатомарное приращение счетчика
                // при удержании блокировки.
                let old = counter.load(Relaxed);
                let new = old + 1;
                counter.store(new, Relaxed);
                // Высвобождение блокировки с неверным
                // указателем упорядочения памяти.
                compiler_fence(Release);
                locked.store(false, Relaxed);
            });
        }
    });
    println!("{}", counter.into_inner());
}
```

При правильной работе блокировки ожидается, что финальным значением счетчика будет ровно четыре миллиона. Заметьте, что приращение счетчика происходит неатомарным способом, с отдельной загрузкой и сохранением, а не с одной операцией `fetch_add`, чтобы гарантировать, что любые проблемы со спин-блокировкой смогут привести к пропущенным приращениям и, следовательно, к снижению общего значения счетчика.

Несколько запусков этой программы на компьютере с процессором *x*86-64 дают следующий результат:

```
4000000
4000000
4000000
```

Как и ожидалось, семантика высвобождения и получения досталась нам совершенно "бесплатно", а наша ошибка не привела к проблемам.

Если попробовать запустить эту программу на телефоне с Android 2021 года выпуска и на Raspberry Pi 3 модели B, в которых установлены процессоры ARM64, то в результате будет получен точно такой же вывод на экран:

```
4000000
4000000
4000000
```

Это говорит о том, что не все процессоры ARM64 используют все формы переупорядочения команд, хотя на основе этого эксперимента нельзя выстраивать далеко идущие предположения.

Пробуя запуск программы на Apple iMac 2021 года, содержащем процессор Apple M1 на базе архитектуры ARM64, мы получаем совершенно иной результат:

```
3988255
3982153
3984205
```

Наша ранее совершенно незаметная ошибка внезапно превратилась в реальную проблему, проявляющуюся только в слабо упорядоченной системе. Погрешность счетчика составляет всего лишь 0,4%, что свидетельствует о том, насколько незаметной может быть такая проблема. В реальности подобная проблема может оставаться незамеченной в течение очень долгого времени.

При попытке воспроизведения приведенных здесь результатов не забудьте включить оптимизацию (применив команду `cargo run --release` или `rustc -O`). Без оптимизации один и тот же код зачастую приводит к появлению огромного количества инструкций, что может скрыть едва уловимые нюансы переупорядочения инструкций.

Ограждения памяти

Остался еще один пока не рассмотренный нами тип инструкций, связанный с упорядочиванием памяти и касающийся ограждения памяти. Инструкция *ограждения памяти*, или *барьера памяти*, используется для представления функции `std::sync::atomic::fence`, которая рассматривалась в *разделе "Ограждения" главы 3*.

Как было показано ранее, упорядочение памяти в *x*86-64 и ARM64 заключается в переупорядочении инструкций. Инструкция ограждения не дает переупорядочить определенные типы инструкций после нее.

Ограждение получения должно препятствовать переупорядочению предыдущих операций загрузки с любыми последующими операциями с памятью. Подобным образом, ограничение высвобождения должно препятствовать переупорядочению последующих операций сохранения с любыми предыдущими операциями с памятью. Последовательно согласованное ограждение должно препятствовать переупорядочению всех операций с памятью, предшествующих ему, с операциями с памятью после этого ограждения.

В *x*86-64 базовая семантика упорядочения памяти уже удовлетворяет потребностям ограждений получения и высвобождения. В любом случае данная архитектура не допускает типов переупорядочения, которые препятствуют этим ограждениям.

Давайте перейдем к делу и посмотрим, в какие инструкции компилируются четыре разных ограждения как на *x*86-64, так и на ARM64 (см. табл. 7.15).

Неудивительно, что ограждения высвобождения и получения на *x*86-64 не приводят к появлению каких-либо инструкций. Семантику высвобождения и получения мы приобретаем на этой архитектуре "совершенно бесплатно". И лишь ограждение `SeqCst` приводит к созданию инструкции `mfence` (оградить память). Эта инструкция перед продолжением гарантирует, что все операции с памятью, предшествующие ей, были завершены.

Таблица 7.15

Исходный код на Rust	Скомпилировано для x86-64	Скомпилировано для ARM64

`pub fn a() {`
`  fence(Acquire);`
`}`
	`a:`
	`  ret`
		`a:`
		`  dmb ishld`
		`  ret`

`pub fn a() {`
`  fence(Release);`
`}`
	`a:`
	`  ret`
		`a:`
		`  dmb ish`
		`  ret`

`pub fn a() {`
`  fence(AcqRel);`
`}`
	`a:`
	`  ret`
		`a:`
		`  dmb ish`
		`  ret`

`pub fn a() {`
`  fence(SeqCst);`
`}`
	`a:`
	`  mfence`
	`  ret`
		`a:`
		`  dmb ish`
		`  ret`

В ARM64 эквивалентной инструкцией является `dmb ish` (data memory barrier, inner shared domain — барьер памяти данных, внутренний общий домен). В отличие от x86-64, она также используется для упорядочений `Release` и `AcqRel`, поскольку эта архитектура не обеспечивает подразумеваемую семантику получения и высвобождения. Для `Acquire` действует менее эффективный вариант: инструкция `dmb ishld`, которая всего лишь ожидает завершения операций загрузки, но позволяет свободно переупорядочивать предыдущие операции сохранения после нее.

Складывается точно такая же картина, что и ранее увиденная нами с атомарными операциями: x86-64 дает нам ограждения высвобождения и получения "бесплатно", а в ARM64 последовательно согласованные ограждения предоставляются по той же цене, что и ограждения высвобождения.

Резюме

- В x86-64 и ARM64 операции расслабленной загрузки и сохранения идентичны их неатомарным эквивалентам.
- Наиболее распространенные атомарные операции выборки-изменения, а также сравнения-обмена в x86-64 (и в ARM64, начиная с ARMv8.1) имеют свои собственные инструкции.
- В x86-64 атомарная операция, для которой нет эквивалентной инструкции, компилируется в цикл сравнения-обмена.
- В ARM64 любую атомарную операцию можно представить в виде цикла инструкции связанной загрузки и инструкции условного сохранения (load-linked/store-conditional loop), который автоматически перезапускается, если предпринятая операция с памятью была прервана.
- Кеши работают со строками кеша, размер которых зачастую составляет 64 байта.

load

	non-atomic	Relaxed	Acquire	SeqCst
ARMv8	ldr		ldar	
x86-64	mov			

store

	non-atomic	Relaxed	Release	SeqCst
ARMv8	str		stlr	
x86-64	mov			xchg

swap

	non-atomic	Relaxed	Acquire	Release	AcqRel	SeqCst
ARMv8	ldr str	ldxr stxr cbnz	ldaxr stxr cbnz	ldxr stlxr cbnz	ldaxr stlxr cbnz	
ARMv8.1		swp	swpa	swpl	swpal	
x86-64	mov mov	xchg				

fetch_op

	non-atomic	Relaxed	Acquire	Release	AcqRel	SeqCst
ARMv8	ldr op str	ldxr op stxr cbnz	ldaxr op stxr cbnz	ldxr op stlxr cbnz	ldaxr op stlxr cbnz	
ARMv8.1		ldop	ldopa	ldopl	ldopal	
x86-64 op	op	lock op				
x86-64 fetch_and_op	mov op	lock { xadd / bts / btr / btc } or mov op lock cmpxchg jne				

compare_exchange

	weak	strong
ARMv8	ldaxr[1] cmp b.ne stlxr[2] clrex	ldaxr[1] cmp b.ne stlxr[2] clrex cbnz
ARMv8.1	casal[3]	
x86-64	lock cmpxchg	

[1] ldxr or ldaxr
[2] stxr or stlxr
[3] cas, casa, casl, or casal

fence

	Acquire	Release	AcqRel	SeqCst
ARMv8	dmb ishld	dmb ish		
x86-64	-			mfence

Рис. 7.1. Сводка инструкций, в которые компилируются различные атомарные операции на ARM64 и x86-64 для каждого упорядочения памяти

- Кеши поддерживаются в соответствии с протоколом согласованности кеша, таким как протокол сквозной записи или протокол MESI.
- Заполнение, например, с помощью атрибута `#[repr(align(64))]` может позволить поднять производительность за счет предотвращения *ложного разделения*.
- Операция загрузки может обходиться намного дешевле, чем неудачная операция сравнения-обмена, отчасти потому, что последняя зачастую требует исключительного доступа к строке кеша.
- В однопоточной программе переупорядочение инструкций ничем себя не проявляет.
- В большинстве архитектур, включая *x*86-64 и ARM64, упорядочение памяти заключается в предотвращении переупорядочения определенных типов инструкций.
- В *x*86-64 каждая операция с памятью имеет семантику получения и высвобождения, что делает ее ничуть не дешевле и не дороже расслабленной операции. Все, кроме сохранений и ограждений, также имеет последовательно согласованную семантику без каких-либо дополнительных затрат.
- В ARM64 семантика получения и высвобождения не обходится так же дешево, как расслабленные операции, но включает в себя последовательно согласованную семантику без дополнительных затрат.

Краткая сводка ассемблерных инструкций, ранее встречавшихся в данной главе, приведена на рис. 7.1.

ГЛАВА 8

Примитивы операционной системы

Пока что наше внимание было сосредоточено в основном на неблокирующих операциях. Если же нужно реализовать что-то вроде мьютекса или условной переменной, что-то, что может ждать, пока другой поток разблокирует или выдаст уведомление, потребуется способ эффективной блокировки текущего потока.

В *главе 4* рассматривались способы самостоятельного решения этой задачи, без задействования средств операционной системы, путем *прокрутки* с повторяющимися снова и снова попытками какого-либо действия, чем легко можно было впустую потратить уйму процессорного времени. Но при желании получить в свое распоряжение по-настоящему эффективную блокировку, потребуется помощь ядра операционной системы.

Ядро, а точнее его часть — *диспетчер*, отвечает за решение о запуске того или иного процесса или потока, когда, насколько долго и на каком конкретном ядре процессора он будет работать. Ядро может прекратить выделение процессорного времени тому потоку, который ожидает наступления какого-то события, отдавая приоритет другим потокам, которые смогут более рационально воспользоваться этим дефицитным ресурсом.

Для этого понадобится способ оповещения ядра об ожидании с просьбой перевода нашего потока в спящий режим, пока не наступит нечто важное.

Взаимодействие с ядром

Способ взаимодействия с ядром во многом зависит от конкретной операционной системы, а зачастую даже от ее версии. Обычно подробности соответствующих механизмов скрыты в недрах одной или нескольких библиотек, делающих всю нужную работу за нас. Используя, к примеру, стандартную библиотеку Rust, можно просто вызвать `File::open()`, чтобы открыть файл, не вникая в подробности интерфейса ядра операционной системы. Точно так же можно открыть файл, задействуя стандартную библиотеку C, `libc` и вызвав стандартную функцию `fopen()`. Вызов подобной функции в конечном итоге приведет к вызову ядра операционной системы, поэтому он также носит название *системного вызова* (*syscall*) и обычно выполняется с помощью специализированной инструкции процессора. (В некоторых архитектурах эта инструкция именно так и называется `syscall`.)

От программ обычно ожидается, а иногда даже требуется, что они будут выполнять какие-либо системные вызовы не напрямую, за счет поставляемых с операционной системой библиотек более высокого уровня. В системах Unix, например в тех, что

основаны на Linux, особую роль по обеспечению стандартного интерфейса с ядром берет на себя `libc`.

Стандарт "Интерфейс переносимой операционной системы", более известный как POSIX, включает дополнительные требования к библиотеке `libc`, имеющейся в системах Unix. Например, наряду с функцией `fopen()` из стандарта C POSIX дополнительно требует для открытия файлов наличия функций нижнего уровня `open()` и `openat()`, которые зачастую напрямую соответствуют системному вызову. Из-за особого статуса `libc` в системах Unix программы, написанные на языках, отличных от C, продолжают использовать `libc` для всех своих взаимодействий с ядром.

Программные средства Rust, включая стандартную библиотеку, часто используют `libc` через одноименный крейт `libc`.

Что касается операционной системы Linux, то интерфейс системных вызовов в ней гарантированно стабилен, что позволяет выполнять системные вызовы напрямую, без участия `libc`. И пусть это не самый распространенный и рекомендуемый образ действий, но он постепенно становится все более и более популярным.

А вот в macOS, которая также является операционной системой семейства Unix, соответствующей стандарту POSIX, интерфейс системных вызовов ядра нестабилен, и им не следует пользоваться напрямую. Единственные стабильные интерфейсы, которые разрешено использовать программам, предоставляются через библиотеки, поставляемые вместе с системой, в числе которых `libc`, `libc++` и различные другие библиотеки для C, C++, Objective-C и Swift, любимых языков программирования компании Apple.

Windows не соответствует стандарту POSIX. Она не поставляется с расширенной версией `libc`, которая служит в качестве основного интерфейса с ядром, вместо этого с ней поставляется отдельный набор таких библиотек, как `kernel32.dll`, предоставляющих специфичные для Windows функции, в числе которых имеется и функция `CreateFileW`, предназначенная для открытия файлов. Как и в случае с работой под управлением macOS, мы не должны применять недокументированные функции нижнего уровня или же выполнять системные вызовы напрямую.

Операционные системы через свои библиотеки предоставляют нам такие примитивы синхронизации, как мьютексы и условные переменные, которые должны взаимодействовать с ядром. Какая часть их реализации является частью такой библиотеки или частью ядра, во многом зависит от конкретной операционной системы. Например, иногда операции блокировки и разблокировки мьютекса напрямую соответствуют системному вызову ядра, а в других системах библиотека обрабатывает основную часть операций и выполняет системный вызов только в случае необходимости блокировки или пробуждения потока. Последний вариант более эффективен, поскольку системный вызов может выполняться довольно долго.

POSIX

В составе расширений POSIX Threads, более известных как pthreads, POSIX определяет типы данных и функции, предназначенные для вычислений в конкурентном режиме. Хотя в техническом плане эти функциональные возможности определяют-

ся как часть отдельной системной библиотеки libpthread, в настоящее время они зачастую включаются непосредственно в libc.

Кроме функций порождения и соединения потоков (pthread_create и pthread_join), pthread предоставляет самые распространенные примитивы синхронизации: мьютексы (pthread_mutex_t), блокировки чтения-записи (pthread_rwlock_t) и условные переменные (pthread_cond_t).

- pthread_mutex_t — pthread-мьютекс должен быть инициализирован вызовом pthread_mutex_init() и уничтожен с помощью pthread_mutex_destroy(). Функция инициализации принимает аргумент типа pthread_mutexattr_t, которым можно воспользоваться для настройки ряда свойств мьютекса.

  Одно из таких свойств — его поведение при рекурсивной блокировке, которая происходит, когда один и тот же поток, уже удерживающий блокировку, пытается снова установить блокировку. При настройке по умолчанию (PTHREAD_MUTEX_DEFAULT) это приводит к неопределенному поведению, но мьютекс можно настроить так, чтобы это приводило к выдаче ошибки (PTHREAD_MUTEX_ERRORCHECK), к взаимоблокировке (PTHREAD_MUTEX_NORMAL) или к успешной второй блокировке (PTHREAD_MUTEX_RECURSIVE).

  Эти мьютексы блокируются посредством вызова функции pthread_mutex_lock() или pthread_mutex_trylock() и разблокируются с помощью вызова функции pthread_mutex_unlock(). Кроме этого, в отличие от стандартного мьютекса Rust, они также, посредством вызова функции pthread_mutex_timedlock() поддерживают блокировку с ограничением по времени.

  pthread_mutex_t можно статически инициализировать без вызова функции pthread_mutex_init(), присвоив ему значение PTHREAD_MUTEX_INITIALIZER. Но сделать это можно только для мьютекса с настройками по умолчанию.

- pthread_rwlock_t — предоставляемая pthread блокировка чтения-записи инициализируется и уничтожается с помощью функций pthread_rwlock_init() и pthread_rwlock_destroy(). Как и мьютекс, блокировка pthread_rwlock_t с настройками по умолчанию также может быть инициализирована статически с помощью присвоения значения PTHREAD_RWLOCK_INITIALIZER.

  Pthread-блокировка чтения-записи имеет, по сравнению с pthread-мьютексом, гораздо меньше свойств, настраиваемых с помощью функции инициализации. В частности, попытка рекурсивной блокировки записи всегда приводит к взаимоблокировке.

  А вот попытки рекурсивно получить дополнительные блокировки чтения гарантированно увенчаются успехом даже в ожидании записей. Тем самым, по сути, исключается любая эффективная реализация, отдающая приоритет функциям записи над функциями чтения, поэтому в большинстве реализаций pthread приоритет отдается функциям чтения.

  Интерфейс блокировки чтения-записи почти идентичен интерфейсу pthread_mutex_t, включая поддержку ограничений по времени, за исключением того, что каждая функция блокировки существует в двух вариантах, один из которых для

чтения (pthread_rwlock_rdlock), а второй для записи (pthread_rwlock_wrlock). Как ни удивительно, но есть только одна функция разблокировки (pthread_rwlock_unlock), используемая для снятия блокировки любого типа.

- pthread_cond_t — предоставляемая pthread условная переменная используется вместе с pthread-мьютексом. Она инициализируется и уничтожается с помощью функций pthread_cond_init и pthread_cond_destroy и имеет несколько настраиваемых атрибутов. В частности, ее можно настроить на то, какие именно часы будут использоваться для ограничений по времени, монотонные (например, как Instant в Rust) или часы реального времени (например, как SystemTime в Rust, иногда называемые временем настенных часов). Условная переменная с настройками по умолчанию, например статически инициализированная присваиванием PTHREAD_COND_INITIALIZER, задействует часы реального времени.

  Ожидание такой условной переменной можно ограничить по времени, воспользовавшись функцией pthread_cond_timedwait(). Пробуждение ожидающего потока осуществляется вызовом функции pthread_cond_signal() или же, чтобы одновременно разбудить все ожидающие потоки, вызовом функции pthread_cond_broadcast().

К остальным примитивам синхронизации, предоставляемым pthread и не рассматриваемым в данной главе, относятся барьеры (pthread_barrier_t), спин-блокировки (pthread_spinlock_t) и однократная инициализация (pthread_once_t).

Обертывание в Rust

Казалось бы, примитивы синхронизации из pthread можно легко и просто предоставлять Rust-программе, обернув соответствующий им C-тип (полученный через крейт libc) в такую вот Rust-структуру:

```
pub struct Mutex {
    m: libc::pthread_mutex_t,
}
```

Но вот незадача, данный тип pthread-потока был разработан для C, а не для Rust.

Прежде всего, в Rust существуют правила изменяемости и заимствований, которые вообще-то не допускают изменений при совместном использовании. Поскольку такие функции, как pthread_mutex_lock, скорее всего, внесут в мьютекс изменения, то для подтверждения их приемлемости придется воспользоваться внутренней изменяемостью. Следовательно, функцию потребуется обернуть в UnsafeCell:

```
pub struct Mutex {
    m: UnsafeCell<libc::pthread_mutex_t>,
}
```

Но куда более серьезная проблема связана с *перемещением*.

В Rust объекты постоянно перемещаются. Например, когда возвращаются из функции, передаются ей в качестве аргумента или просто присваиваются новому размещению. Все, находящееся в нашем владении (и что не заимствовано ничем другим), можно свободно переместить в новое место.

Но в C все немного не так. Довольно часто тип в C зависит от постоянства своего адреса в памяти. Например, он может содержать указатель на самого себя или хранить указатель на себя в какой-либо глобальной структуре данных. И тогда перемещение его в новое место может привести к неопределенному поведению.

Перемещаемость рассмотренных нами типов pthread не гарантируется, что становится для Rust серьезной проблемой. Даже простая, весьма характерная для Rust функция `Mutex::new()` создает проблему, поскольку она возвращает объект-мьютекс, перемещаемый на новое место в памяти.

Поскольку пользователь всегда может в произвольном порядке перемещать любой принадлежащий ему объект-мьютекс, нужно либо заручиться у него обещанием, что он не станет этого делать, создав `unsafe`-интерфейс, либо забрать у него право собственности и спрятать все под оболочкой (под чем-то, для чего можно будет воспользоваться `std::pin::Pin`). Но удачным ни одно из этих решений признать невозможно, поскольку они влияют на интерфейс нашего типа мьютекса, превращая его в легкую мишень для ошибок и (или) в продукт, крайне неудобный в применении.

Решение этой проблемы — обернуть мьютекс в `Box`. Когда pthread-мьютекс помещен в свою собственную, выделенную именно для него память, он остается в том же месте в памяти, даже если его владелец перемещается:

```
pub struct Mutex {
    m: Box<UnsafeCell<libc::pthread_mutex_t>>,
}
```

Именно так `std::sync::Mutex` был реализован на всех платформах Unix до выхода версии Rust 1.62.

Минус такого подхода — высокие издержки: каждый мьютекс теперь получает свою собственную выделенную память, что существенно увеличивает накладные расходы на его создание, уничтожение и использование. Еще одним недостатком является то, что функция `new` не может быть с объявлением `const`, что мешает применению `static`-мьютекса.

Даже если бы `pthread_mutex_t` был перемещаемым, объект `const fn new` мог бы инициализировать его только с настройками по умолчанию, что при рекурсивной блокировке привело бы к неопределенному поведению. Разработать безопасный интерфейс, предотвращающий рекурсивную блокировку, невозможно, поэтому функцию `lock` нужно сделать небезопасной (`unsafe`) и заручиться обещанием пользователей, что они так делать не будут.

Нерешенной в нашем подходе с применением `Box`, остается проблема удаления заблокированного мьютекса. Может показаться, что при условии выполнения всех правил проектирования удалить заблокированный `Mutex` будет просто невозможно, поскольку удалить его, пока он все еще заимствован `MutexGuard`, нельзя. Чтобы разблокировать `Mutex`, сначала нужно удалить `MutexGuard`. Но в Rust совершенно безо-

пасно можно, не удалив объект, просто забыть о нем (или же допустить утечку занимаемой им памяти). Получается, что можно создать примерно следующий код:

```
fn main() {
    let m = Mutex::new(..);
    let guard = m.lock();     // Его блокировка ..
    std::mem::forget(guard); // .. но без ее снятия.
}
```

В приведенном примере `m` будет удален в конце области видимости, пока она будет все еще заблокирована. Компилятор Rust считает это в порядке вещей, поскольку защита превратилась в утечку памяти и ее больше нельзя использовать.

Но для pthread оговаривается, что вызов `pthread_mutex_destroy()` для заблокированного мьютекса может и не сработать, что чревато неопределенным поведением. Один из способов обхода данного обстоятельства — попытаться при удалении нашего `Mutex` сначала заблокировать (и разблокировать) pthread-мьютекс и вызвать панику (или смириться с утечкой памяти, выделенной под `Box`), когда он уже заблокирован, но это добавит программе еще больше накладных расходов.

Подобные проблемы относятся не только к `pthread_mutex_t`, но и к другим, рассмотренным ранее типам. Получается так, что конструкция примитивов синхронизации pthread подходит для C, но не очень подходит для Rust.

Linux

В системах Linux все примитивы синхронизации pthread реализуются с использованием системного вызова *фьютекса*. Его название происходит от выражения "fast user-space mutex" (быстрый мьютекс в пользовательском пространстве), поскольку первоначальная мотивация добавления данного системного вызова заключалась в том, чтобы позволить библиотекам (например, реализациям pthread) включать быструю и эффективную реализацию мьютекса. Но фьютекс гораздо более гибок и подходит для создания множества различных инструментов синхронизации.

Фьютекс как системный вызов был добавлен в ядро Linux в 2003 году и с тех пор претерпел целый ряд улучшений и расширений. Впоследствии аналогичные функции были добавлены и в некоторые другие операционные системы, в первую очередь Windows 8 в 2012 году с дополнением в виде функции `WaitOnAddress` (которая будет рассмотрена чуть позже в *разделе "Windows"*). В 2020 году поддержка базовых операций, подобных фьютексам, с дополнением в виде функций `atomic_wait` и `atomic_notify`, была даже добавлена в стандартную библиотеку языка C++.

Фьютекс

В Linux `SYS_futex` является системным вызовом, реализующим различные операции, работающие с 32-разрядным атомарным целым числом. К основным двум операциям относятся `FUTEX_WAIT` и `FUTEX_WAKE`.

Операция ожидания переводит поток в спящий режим, а операция пробуждения в отношении той же атомарной переменной снова пробуждает поток. В атомарном целом числе эти операции ничего не сохраняют. Ядро просто запоминает, какие потоки и по какому адресу памяти находятся в режиме ожидания, чтобы позволить операции пробуждения разбудить нужные потоки.

В *разделе "Ожидание: парковка и условные переменные" главы 1* было показано, что другим механизмам блокировки и пробуждения потоков нужен способ выдачи гарантий, что операции пробуждения не потеряются в гонке. При парковке потоков эта проблема решается путем применения операции `unpark()` также и к будущим операциям `park()`. Что же касается условных переменных, то об этом позаботится мьютекс, используемый вместе с условной переменной.

Для фьютексных операций ожидания и пробуждения действует другой механизм. Операция ожидания принимает аргумент, определяющий значение, которое, как мы ожидаем, будет иметь атомарная переменная, и эта операция отклонит блокировку, если значение не будет соответствовать ожидаемому. Операция ожидания ведет себя атомарно по отношению к операции пробуждения, т. е. ни один сигнал пробуждения не может быть потерян между моментом проверки ожидаемого значения и моментом фактического перехода в спящий режим.

Если убедиться, что значение атомарной переменной изменилось прямо перед операцией пробуждения, можно быть уверенным, что поток, собирающийся перейти в режим ожидания, не впадет в спячку, так что возможный пропуск фьютексной операции пробуждения уже не имеет значения

Давайте рассмотрим небольшой пример, позволяющий увидеть это на практике.

Прежде всего, нужно получить возможность делать эти системные вызовы. Для этого можно воспользоваться функцией `syscall` из крейта `libc` и обернуть каждый из вызовов в подходящую функцию Rust (листинг 8.1).

Листинг 8.1

```
#[cfg(not(target_os = "linux"))]
compile_error!("Linux only. Sorry!");
pub fn wait(a: &AtomicU32, expected: u32) {
    // Сигнатуру системного вызова можно найти
    // на странице руководства futex (2).
    unsafe {
        libc::syscall(
            libc::SYS_futex, // Системный вызов фьютекса.
            a as *const AtomicU32, // Рабочая атомарная переменная.
            libc::FUTEX_WAIT, // Фьютексная операция.
            expected, // Ожидаемое значение.
            std::ptr::null::<libc::timespec>(), // Без указания времени ожидания.
        );
    }
}
```

```
pub fn wake_one(a: &AtomicU32) {
    // Сигнатуру системного вызова можно найти
    // на странице руководства futex (2).
    unsafe {
        libc::syscall(
            libc::SYS_futex, // Системный вызов фьютекса.
            a as *const AtomicU32, // Рабочая атомарная переменная.
            libc::FUTEX_WAKE, // Фьютексная операция.
            1, // Число пробуждаемых потоков.
        );
    }
}
```

Теперь в качестве примера применения давайте воспользуемся всем этим, чтобы заставить один поток ожидать другого потока. Воспользуемся атомарной переменной, проинициализированной нулем, на значении которой будет строиться фьютексная операция ожидания основного потока. Второй поток изменит значение переменной на единицу, а затем на нем будет запущена фьютексная операция пробуждения, чтобы разбудить основной поток.

По аналогии с парковкой потока и ожиданием на основе условной переменной фьютексная операция ожидания может просыпаться *без видимых на то причин*, даже если ничего не произошло. Поэтому ее довольно часто используют в цикле, повторяя вызов, если ожидаемое условие еще не было выполнено.

Давайте рассмотрим пример (листинг 8.2).

Листинг 8.2

```
fn main() {
    let a = AtomicU32::new(0);
    thread::scope(|s| {
        s.spawn(|| {
            thread::sleep(Duration::from_secs(3));
            a.store(1, Relaxed);                    ❶
            wake_one(&a);                           ❷
        });
        println!("Waiting...");
        while a.load(Relaxed) == 0 {                ❸
            wait(&a, 0);                            ❹
        }
        println!("Done!");
    });
}
```

❶ Установка через несколько секунд порожденным потоком атомарной переменной в единицу.

❷ Выполнение этим же потоком фьютексной операции пробуждения, чтобы разбудить основной поток, если он был в режиме сна, чтобы он смог увидеть, что переменная изменилась.

❸ Прежде чем продолжить свою работу и вывести на экран финальное сообщение, основной поток находится в режиме ожидания, пока переменная равна нулю.

❹ Фьютексная операция ожидания служит для перевода потока в спящий режим. Здесь очень важно, что эта операция перед переходом в режим сна проверит, является ли значение `a` по-прежнему нулевым, и именно поэтому сигнал из порожденного потока не может потеряться между ❸ и ❹. Либо ❶ (а значит, и ❷) еще не происходило и поток засыпает, либо ❶ (и может быть, и ❷) уже произошло и поток сразу же продолжит работу.

Здесь важно заметить следующее: вызов `wait` полностью исключается, если перед циклом `while` значение `a` уже было установлено в единицу. Таким же образом, если бы основной поток выполнял операцию сохранения в атомарной переменной независимо от ожидания сигнала (устанавливая для него значение, отличное от нуля или единицы), сигнальный поток мог бы пропустить фьютексную операцию пробуждения в том случае, если основной поток еще не перешел в режим ожидания. Именно это делает примитивы синхронизации на основе фьютексов такими быстрыми: поскольку мы сами управляем состоянием, нам не нужно полагаться на ядро, за исключением тех случаев, когда мы действительно нуждаемся в блокировке.

Начиная с Rust 1.48, в стандартной библиотеке под Linux подобным же образом реализованы функции парковки потоков. Они используют одну атомарную переменную для каждого потока с тремя возможными состояниями: нуль для простоя и начального состояния, единицу для "распаркован, но еще не запаркован" и минус единицу для состояния "запаркован, но еще не распаркован".

В *главе 9* все эти операции будут использоваться для реализации мьютексов, условных переменных и блокировки чтения-записи.

Фьютексные операции

Помимо операций ожидания и пробуждения, системный вызов фьютекса поддерживает ряд других операций. В данном разделе будет дан краткий обзор каждой операции, поддерживаемой этим системным вызовом.

Первый аргумент фьютекса всегда является указателем на рабочую 32-битовую атомарную переменную. Второй аргумент — константа, представляющая операцию, например `FUTEX_WAIT`, к которой можно добавить до двух флагов: `FUTEX_PRIVATE_FLAG` и (или) `FUTEX_CLOCK_REALTIME`, которые будут рассмотрены чуть позже. Остальные аргументы зависят от конкретной операции, и далее приводится их описание для каждой из операций.

- `FUTEX_WAIT` — эта операция принимает два дополнительных аргумента: ожидаемое значение атомарной переменной и указатель на спецификацию времени `timespec`, представляющую максимальное время ожидания.

Если значение атомарной переменной соответствует ожидаемому значению, операция ожидания блокируется до тех пор, пока не будет разбужена одной из операций пробуждения или пока не истечет срок, указанный в timespec. Если указатель на timespec имеет значение null, ограничения по времени не существует. Кроме того, операция ожидания может активироваться безо всяких на то причин и выполнить возврат управления без соответствующей операции пробуждения еще до того, как истечет срок ожидания.

Операции проверки и блокировки выполняются по отношению к другим фьютексным операциям в виде единой атомарной операции, следовательно, сигналы пробуждения потеряться между ними не могут.

Продолжительность, указанная в спецификации timespec, по умолчанию представляет собой отрезок времени по показаниям монотонных часов (например, такие как Instant в Rust). При добавлении флага FUTEX_CLOCK_REALTIME вместо монотонных используются часы реального времени (например, такие как SystemTime в Rust).

Возвращаемое значение указывает на то, совпало ли значение с ожидаемым и вышел срок ожидания или нет.

- FUTEX_WAKE — эта операция принимает один дополнительный аргумент: число потоков для пробуждения в виде типа i32.

  Операция пробуждает столько потоков, заблокированных операцией ожидания для одной и той же атомарной переменной, сколько ей указано. (Или же меньше, если ожидающих потоков не так много.) Чаще всего этот аргумент используется либо для пробуждения только одного потока, либо при указании i32::MAX для пробуждения всех потоков.

  Возвращается число пробудившихся потоков.

- FUTEX_WAIT_BITSET — эта операция принимает четыре дополнительных аргумента: ожидаемое значение атомарной переменной, указатель на переменную timespec, представляющую максимальное время ожидания, указатель, который игнорируется, и 32-разрядный "набор битов" (u32).

  Операция ведет себя так же, как FUTEX_WAIT, но с двумя отличиями.

  Первое отличие — она принимает аргумент набора битов, которым можно воспользоваться для ожидания только вполне определенных операций пробуждения, а не всех операций пробуждения для одной и той же атомарной переменной. Операция FUTEX_WAKE никогда не игнорируется, но сигнал операции FUTEX_WAKE_BITSET игнорируется, если набор битов ожидания и набор битов пробуждения не имеют хотя бы одного общего бита.

  Например, операция FUTEX_WAKE_BITSET с набором битов 0b0101 пробуждает операцию FUTEX_WAIT_BITSET с набором битов 0b1100, но не операцию с набором битов 0b0010.

  Это может пригодиться при реализации чего-либо вроде блокировки чтения-записи, чтобы разбудить записывающий поток, не пробуждая читающие потоки.

И все же следует заметить, что две отдельные атомарные переменные могут оказаться эффективнее одной для двух разных типов ожидающих потоков, поскольку ядро будет хранить для каждой атомарной переменной единый список ожидающих потоков.

Другое отличие от FUTEX_WAIT заключается в том, что спецификация timespec используется не как продолжительность, а как абсолютная метка времени. Поэтому FUTEX_WAIT_BITSET довольно часто применяется с набором битов u32::MAX (со всеми установленными битами), что фактически превращает ее в обычную операцию FUTEX_WAIT, но с абсолютной меткой времени для ограничения срока ожидания.

- FUTEX_WAKE_BITSET — эта операция принимает четыре дополнительных аргумента: число пробуждаемых потоков, два указателя, которые игнорируются, и 32-разрядный "набор битов" (u32).

  Операция идентична FUTEX_WAKE, за исключением того, что она не пробуждает операции FUTEX_WAIT_BITSET с набором неперекрывающихся битов. (См. описание операции FUTEX_WAIT_BITSET.)

  С битовым набором u32::MAX (со всеми установленными битами) она идентична операции FUTEX_WAKE.

- FUTEX_REQUEUE — эта операция принимает три дополнительных аргумента: число пробуждаемых потоков (i32), число потоков, отправляемых в очередь (i32), и адрес вторичной атомарной переменной.

  Операция пробуждает заданное число ожидающих потоков, а затем отправляет в очередь заданное число остальных ожидающих потоков, использующих для ожидания другую атомарную переменную.

  Отправленные в очередь ожидающие потоки продолжают находиться в режиме ожидания, но на них больше не влияют операции пробуждения основной атомарной переменной. Теперь они пробуждаются операциями пробуждения, завязанными на вторичную атомарную переменную.

  Это может пригодиться для реализации чего-либо вроде операции "уведомить всех", завязанной на условную переменную. Вместо пробуждения всех потоков, которые впоследствии попытаются заблокировать мьютекс, с большой долей вероятности сразу же после этого заставляя все потоки, кроме одного, дожидаться разблокировки этого мьютекса, можно было бы разбудить только один поток, а все остальные отправить в очередь на непосредственное ожидание разблокировки мьютекса без их опережающего это событие пробуждения.

  По аналогии с операцией FUTEX_WAKE значением i32::MAX можно воспользоваться для отправки в очередь всех ожидающих потоков. (Указание значения i32::MAX для количества пробуждаемых потоков вряд ли имеет смысл, поскольку тогда операция станет эквивалентной операции FUTEX_WAKE.)

  Возвращается число пробудившихся потоков.

- FUTEX_CMP_REQUEUE — эта операция принимает четыре дополнительных аргумента: число пробуждаемых потоков (i32), число потоков, отправляемых в очередь

(i32), адрес вторичной атомарной переменной и ожидаемое значение первичной атомарной переменной.

Операция практически идентична FUTEX_REQUEUE, за исключением того, что она отказывается работать, если значение первичной атомарной переменной не соответствует ожидаемому. По отношению к другим операциям фьютекса проверка значения и операция отправки в очередь происходят атомарно.

В отличие от FUTEX_REQUEUE, операция возвращает сумму числа пробужденных и отправленных в очередь потоков.

- FUTEX_WAKE_OP — эта операция принимает четыре дополнительных аргумента: число потоков, пробуждаемых по первичной атомарной переменной (i32), число потоков, которые потенциально могут проснуться по вторичной атомарной переменной (i32), адрес вторичной атомарной переменной, и 32-битовое значение, кодирующее как операцию, так и сравнение, которое необходимо будет выполнить.

Весьма специфичная операция, которая изменяет вторичную атомарную переменную, пробуждает несколько потоков, ожидание которых завязано на первичную атомарную переменную, проверяет, соответствует ли предыдущее значение атомарной переменной заданному условию, и если да, то пробуждает те несколько потоков, ожидание которых завязано на вторичную атомарную переменную.

Иными словами, она идентична следующему коду, за исключением того, что вся она выполняется атомарно по отношению к другим операциям фьютекса:

```
let old = atomic2.fetch_update(Relaxed, Relaxed, some_operation);
wake(atomic1, N);
if some_condition(old) {
    wake(atomic2, M);
}
```

Выполняемая операция изменения и проверяемое условие определяются последним аргументом системного вызова, закодированным в его 32 битах. Операция может быть одной из следующих: присваивание, сложение, двоичное or, двоичное and-not и двоичное xor с 12-разрядным аргументом или 32-разрядным аргументом, представляющим степень двойки. Для сравнения можно выбрать любую из операции ==, !=, <, <=, > и >= с 12-разрядным аргументом.

Подробную информацию о кодировке данного аргумента можно найти на странице futex(2) справочного руководства по Linux или же в крейте linux-futex, находящемся по адресу **crates.io**, который к тому же включает в себя еще и удобный способ создания рассматриваемого аргумента.

Эта операция возвращает общее число пробудившихся потоков.

На первый взгляд операция может показаться гибкой со многими вариантами применения. Но разработана она была только для одного конкретного сценария в GNU libc, когда два потока нужно было разбудить из двух отдельно взятых

атомарных переменных. Теперь данный сценарий заменен другой реализацией, в которой операция FUTEX_WAKE_OP больше не используется.

Чтобы обеспечить возможную оптимизацию, если все соответствующие операции фьютекса с одной и той же атомарной переменной (или переменными) происходят из потоков одного и того же процесса, как это обычно и бывает, к любой из этих операций можно добавить флаг FUTEX_PRIVATE_FLAG. Чтобы им можно было воспользоваться, этот же флаг должна включать каждая соответствующая операция фьютекса. Если позволить ядру выстроить предположение об отсутствии взаимодействия с другими процессами, оно может при выполнении операций фьютекса пропустить некоторые потенциально затратные шаги, повышая тем самым производительность.

Помимо Linux, все описанные операции с фьютексами также поддерживает и NetBSD. В OpenBSD также имеется системный вызов фьютекса, но поддерживаются только операции FUTEX_WAIT, FUTEX_WAKE и FUTEX_REQUEUE. FreeBSD собственного системного вызова фьютекса не имеет, но в нее включен системный вызов _umtx_op с функциональными возможностями, практически идентичными FUTEX_WAIT и FUTEX_WAKE: UMTX_OP_WAIT (для 64-разрядных атомарных систем), UMTX_OP_WAIT_UINT (для 32-разрядных атомарных систем) и UMTX_OP_WAKE. В Windows также включены функции, знакомство с которыми нам еще предстоит, поведение которых весьма похоже на операции ожидания и пробуждения фьютекса.

Новые фьютексные операции

В Linux, начиная с версии 5.16, вышедшей в 2022 году, появился дополнительный системный вызов фьютекса: futex_waitv. Этот новый системный вызов позволяет одновременно выстраивать ожидание сразу на нескольких фьютексах, для чего ему нужно предоставить список атомарных переменных (и их ожидаемых значений). Поток, заблокированный в futex_waitv, может быть разбужен операцией пробуждения для любой из указанных переменных.

В этом новом системном вызове также оставлено место для будущих расширений. Например, можно указать размер атомарной переменной, на которой основано ожидание. Хотя первоначальная реализация, как и исходный системный вызов futex, поддерживает только 32-разрядные атомарные переменные, в будущем их линейка может быть расширена за счет включения 8-разрядных, 16-разрядных и 64-разрядных атомарных переменных.

Фьютексные операции с наследованием приоритета

Проблема инверсии приоритетов возникает, когда поток с высоким приоритетом становится на блокировку, удерживаемую потоком с низким приоритетом. Поток с высоким приоритетом фактически подвергается "инверсии" приоритета, поскольку теперь, чтобы продолжить свою работу, он вынужден ждать, пока поток с низким приоритетом не снимет с него блокировку.

Решением этой проблемы является *наследование приоритета*, при котором блокирующий поток наследует приоритет того потока, который ожидает от него снятия блокировки, при этом поток с низким приоритетом на время удерживания блокировки увеличивает свой приоритет.

В дополнение к уже рассмотренным семи фьютексным операциям имеется еще шесть фьютексных операций, специально предназначенных для реализации блокировок с *наследованием приоритета*.

В рассмотренных ранее основных фьютексных операциях к конкретному содержимому атомарной переменной не предъявляется никаких особых требований. Мы должны сами выбрать, что именно будут представлять собой эти 32 бита. Но для мьютекса, наследующего приоритет, ядро должно иметь возможность понимать, заблокирован ли мьютекс, и если да, то какой именно поток его заблокировал.

Во избежание необходимости выполнения системного вызова при каждом изменении состояния и чтобы ядро могло разобраться во фьютексных операциях с наследованием приоритета, для них определяется конкретное содержимое 32-разрядной атомарной переменной: самый старший бит показывает, есть ли какие-либо потоки, находящиеся в режиме ожидания блокировки мьютекса, а младшие 30 битов содержат идентификатор потока (Linux-идентификатор `tid`, а не Rust-идентификатор `ThreadId`), который удерживает блокировку, или нуль, если он разблокирован.

В качестве дополнительной функции ядро устанавливает второй по старшинству бит, если поток, удерживающий блокировку, завершается, так и не сняв ее, но только при наличии каких-либо ожидающих потоков. Это позволяет сделать мьютекс *надежным*: данный термин применяется для описания мьютекса, способного изящно справиться с ситуацией, когда "владеющий" им поток неожиданно завершается.

Между фьютексными операциями с наследованием приоритета и стандартными операциями имеется взаимно однозначное соответствие: для блокировки используется `FUTEX_LOCK_PI`, для разблокировки — `FUTEX_UNLOCK_PI`, а для синхронизации без блокировки — `FUTEX_TRYLOCK_PI`. Кроме того, для реализации условной переменной, работающей на пару с мьютексом, наследующим приоритет, можно воспользоваться операциями `FUTEX_CMP_REQUEUE_PI` и `FUTEX_WAIT_REQUEUE_PI`.

Здесь подробно рассматривать эти операции мы не будем. Дополнительные сведения о них можно найти на странице `futex(2)` справочного руководства по Linux или же в крейте `linux-futex` по адресу **crates.io**.

macOS

Ядро, являющееся частью macOS, поддерживает разнообразные полезные системные вызовы, связанные с низкоуровневыми вычислениями в конкурентном режиме. Но, как и в большинстве других операционных систем, интерфейс ядра считается нестабильным, и его непосредственное использование не рекомендуется.

Единственный способ взаимодействия программных средств с ядром macOS — использование библиотек, поставляемых вместе с системой. В них включены реализации стандартных библиотек для C (libc), C++ (libc++), Objective-C и Swift.

Поскольку macOS относится к POSIX-совместимому семейству Unix, ее библиотека языка C включает полную реализацию pthread. Стандартными блокировками в других языках обычно закулисно пользуются примитивы pthread.

Pthread-блокировки в macOS работают относительно медленнее своих эквивалентов в других операционных системах. Одна из причин состоит в том, что блокировки в macOS ведут себя по умолчанию как *справедливые блокировки*. Когда один и тот же мьютекс пытаются заблокировать сразу несколько потоков, они обслуживаются как в идеальной очереди в порядке поступления. Хотя справедливость и может порой оказаться желательным свойством, она способна существенно снизить производительность, особенно в условиях высокой конкуренции.

os_unfair_lock

Наряду с примитивами pthread в macOS 10.12 появился новый привязанный к этой платформе облегченный мьютекс, не страдающий справедливостью: `os_unfair_lock`. Его размер составляет всего 32 бита, инициализация проходит статически с применением константы `OS_UNFAIR_LOCK_INIT`, и он не требует уничтожения. Его можно заблокировать с помощью `os_unfair_lock_lock()` (блокировка) или `os_unfair_lock_trylock()` (синхронизация без блокировки) и разблокировать с помощью `os_unfair_lock_unlock()`.

К сожалению, в нем нет ни условной переменной, ни варианта чтения-записи.

Windows

Операционная система Windows поставляется с набором библиотек, в совокупности образующих *Windows API*, часто называемый Win32 API (даже в 64-разрядных системах). Этот набор является надстройкой поверх "Native API", практически незадокументированным интерфейсом ядра, которым не следует пользоваться напрямую.

Windows API доступен для программ Rust через официальные крейты `windows` и `windows-sys` от Microsoft, доступные на сайте **crates.io**.

Тяжеловесные объекты ядра

Многие из прежних примитивов синхронизации, доступных в Windows, полностью управляются ядром, что придает им особую тяжеловесность и наделяет свойствами, аналогичными свойствам других объектов, управляемых ядром, к числу которых можно отнести файлы. Они могут использоваться несколькими процессами, им можно присваивать имена и размещать по именам, и они, подобно файлам, поддерживают узкоспециализированные допущения. Например, процессу можно разрешить находиться в режиме ожидания какого-либо объекта, не позволяя ему переправлять через себя сигналы для пробуждения других процессов.

К числу тяжеловесных объектов синхронизации, управляемых ядром, относятся `Mutex` (который можно заблокировать и разблокировать), `Event` (в отношении которого можно послать сигнал и построить на нем ожидание) и `WaitableTimer` (о котором можно сигнализировать автоматически через выбранное время или в периодиче-

ском режиме). Создание подобного объекта приводит, так же как и при открытии файла, к появлению дескриптора HANDLE, который легко можно передавать куда угодно и использовать с обычными HANDLE-функциями, и в первую очередь с семейством функций ожидания. Эти функции могут выстраивать ожидания на основе одного или нескольких объектов различных типов, включая тяжеловесные примитивы синхронизации, процессы, потоки и различные формы ввода-вывода.

Облегченные объекты

Облегченный примитив синхронизации, включенный в Windows API, называется критическим разделом.

Термин "критический раздел" относится к части программы, "разделу" кода, который не может быть введен одновременно более чем одним потоком. Механизм защиты критического раздела довольно часто называют мьютексом. Но в данном случае компания Microsoft назвала этот механизм критическим разделом, скорее всего, потому, что имя "мьютекс" уже было занято упомянутым тяжеловесным объектом Mutex.

Имеющийся в Windows CRITICAL_SECTION по сути является *рекурсивным мьютексом*, за исключением того, что в нем употребляются термины "вход" и "выход", а не "блокировка" и "разблокировка". Будучи рекурсивным мьютексом, он предназначен только для защиты от других потоков. Это позволяет одному и тому же потоку блокировать (или "входить") в критический раздел более одного раза с тем требованием, чтобы он также разблокировал (покинул) этот раздел такое же количество раз.

Данное обстоятельство следует учитывать при обертывании этого типа в Rust. Успешная блокировка (вход в раздел) CRITICAL_SECTION не должна приводить к исключительной ссылке (&mut T) на защищаемые им данные. Иначе поток может воспользоваться этим для создания двух исключительных ссылок на одни и те же данные, что тут же приведет к неопределенному поведению.

Раздел CRITICAL_SECTION инициализируется с помощью функции InitializeCriticalSection(), уничтожается с помощью функции DeleteCriticalSection() и не может быть перемещен. Он блокируется с помощью EnterCriticalSection() или TryEnterCriticalSection() и разблокируется с помощью LeaveCriticalSection().

До выхода версии Rust 1.51 std::sync::Mutex в Windows XP основывался на выделяемой под Box памяти для размещения объекта CRITICAL_SECTION. (В Rust 1.51 поддержка Windows XP прекращена.)

Легкая блокировка чтения-записи

Начиная с Windows Vista (и Windows Server 2008), в Windows API включается гораздо более удобный и существенно облегченный примитив блокировки: легкая блокировка чтения-записи, или, для краткости, SRW-блокировка.

Тип SRWLOCK по размеру равен указателю, может быть статически инициализирован с помощью SRWLOCK_INIT и не требует уничтожения. Пока он не используется (будучи заимствованным), его даже разрешено перемещать, что превращает его во вполне подходящего кандидата для обертывания в тип Rust.

С помощью AcquireSRWLockExclusive(), TryAcquireSRWLockExclusive() и ReleaseSRWLockExclusive() обеспечивается исключительная блокировка и разблокировка записывающего потока, а с помощью AcquireSRW1LockShared(), TryAcquireSRWLockShared() и ReleaseSRWLockShared() — совместно используемая блокировка и разблокировка читающего потока. SRW-блокировку часто применяют как обычный мьютекс, просто игнорируя совместно используемые функции блокировки (для читающего потока).

SRW-блокировка не устанавливает приоритеты ни для записывающих, ни для читающих потоков. Пусть и не давая никаких гарантий, она пытается, насколько это возможно, обработать все запросы на блокировку по порядку без снижения производительности. Не следует пытаться получить вторую совместно используемую блокировку для читающего потока, который ее уже удерживает. Если операция будет поставлена в очередь после операции исключительной блокировки другого записывающего потока, который будет заблокирован из-за первой совместно используемой блокировки читающего потока, которую уже удерживает первый поток, это может привести к бесконечной взаимоблокировке.

SRW-блокировка была введена в Windows API вместе с условной переменной. Как и SRW-блокировка, CONDITION_VARIABLE по размеру равна указателю, может быть статически инициализирована с помощью CONDITION_VARIABLE_INIT и не требует уничтожения. Ее также разрешено перемещать, пока она не используется (будучи заимствованной).

Эту условную переменную можно задействовать не только вместе с SRW-блокировкой, посредством SleepConditionVariableSRW, но и с критическим разделом через SleepConditionVariableCS.

Пробуждение ожидающих потоков осуществляется либо с помощью WakeConditionVariable для пробуждения одного потока, либо с помощью WakeAllConditionVariable для пробуждения всех ожидающих потоков.

Изначально SRW-блокировки и условные переменные Windows, используемые в стандартной библиотеке, во избежание перемещения объектов были заключены в Box. Компания Microsoft не документировала гарантии перемещаемости до тех пор, пока мы в 2020 году не отправили соответствующий запрос. С тех пор, начиная с Rust 1.49, std::sync::Mutex, std::sync::RwLock и std::sync::Condvar при работе под управлением Windows Vista и более поздних версий могут напрямую служить оберткой для SRWLOCK или CONDITION_VARIABLE без каких-либо выделений памяти.

Адресное ожидание

В Windows 8 (и Windows Server 2012) появился новый, более гибкий тип функций синхронизации, очень похожий на ранее рассмотренные в данной главе Linux-операции FUTEX_WAIT и FUTEX_WAKE.

Функция `WaitOnAddress` может работать с 8-, 16-, 32- или 64-разрядной атомарной переменной. Она принимает четыре аргумента: адрес атомарной переменной, адрес переменной, содержащей ожидаемое значение, размер атомарной переменной (в байтах) и максимальное число миллисекунд ожидания перед отказом (или `u32::MAX` для бесконечного ожидания).

Точно так же, как операция `FUTEX_WAIT`, она сравнивает значение атомарной переменной с ожидаемым значением и, если они совпадают, переходит в спящий режим, ожидая соответствующую операцию пробуждения. По отношению к операциям пробуждения операции проверки и перехода в спящий режим выполняются атомарно, т. е. между ними не может быть потерян ни один сигнал пробуждения.

Пробуждение потока, находящегося в ожидании на основе применения `WaitOnAddress`, осуществляется либо с помощью `WakeByAddressSingle` для пробуждения одного потока, либо с помощью `WakeByAddressAll` для пробуждения всех ожидающих потоков. Эти две функции принимают только один аргумент: адрес атомарной переменной, который также был передан в `WaitOnAddress`.

С использованием этих функций реализованы некоторые, но не все, примитивы синхронизации Windows API. Важнее всего то, что они являются отличным строительным материалом для создания наших собственных примитивов, чему и будет посвящена *глава 9*.

Резюме

- *Системный вызов* является обращением к ядру операционной системы, выполняемым гораздо медленнее относительно обычного вызова функции.
- Обычно программы не выполняют системные вызовы напрямую, используя для взаимодействия с ядром библиотеки операционной системы (например, `libc`). Во многих операционных системах этот способ — единственное поддерживаемое средство взаимодействия с ядром.
- Доступ к `libc` предоставляется коду Rust посредством крейта `libc`.
- В POSIX-системах `libc` включает в себя больше функциональных возможностей, чем того требует стандарт C для соблюдения стандарта POSIX.
- Стандарт POSIX включает в себя *pthreads* — библиотеку с такими примитивами конкурентных вычислений, как, например, `pthread_mutex_t`.
- Типы Pthread предназначены для C, а не для Rust. В частности, они неперемещаемы, что может стать серьезной проблемой.
- В Linux имеется системный вызов *futex*, поддерживающий несколько операций ожидания и пробуждения на основе применения `AtomicU32`. Операция ожидания проверяет наличие ожидаемого значения в атомарной переменной, применяемой для того, чтобы избежать пропуска уведомлений.
- В добавление к pthread в macOS в качестве облегченного примитива также предоставлена блокировка `os_unfair_lock`.

- Имеющиеся в Windows тяжеловесные примитивы конкурентных вычислений неизменно требуют взаимодействия с ядром, но могут передаваться между процессами и использоваться со стандартными функциями ожидания Windows.
- Имеющиеся в Windows облегченные примитивы конкурентных вычислений включают "легкую" блокировку чтения-записи (SRW-блокировку) и условную переменную. В силу своей перемещаемости они без особого труда обертываются в элементы программного кода Rust.
- Посредством функций `WaitOnAddress` и `WakeByAddress` в Windows также предоставлены базовые функциональные возможности, подобные фьютексу.

ГЛАВА 9

Создание своих собственных блокировок

В этой главе будет рассмотрено создание своего собственного мьютекса, условной переменной и блокировки чтения-записи. Сначала будут рассмотрены самые простые версии каждого из элементов, которые затем будут расширяться с повышением их эффективности.

Из-за отказа от использования типов блокировок из стандартной библиотеки (что было бы жульничеством), чтобы заставить потоки ждать без применения ждущего цикла, придется воспользоваться инструментами из *главы 8*. Но, как там было показано, все зависит от рабочей платформы, и доступные инструменты, предоставляемые операционной системой на разных платформах, имеют существенные различия, затрудняя создание кросс-платформенных программных средств.

На нашу удачу, в большинстве современных операционных систем поддерживаются функции, подобные фьютексу, или, по крайней мере, операции пробуждения и ожидания. В *главе 8* уже упоминалось, что в Linux с помощью системного вызова `futex` они поддерживаются с 2003 года, в Windows с помощью семейства функций `WaitOnAddress` — с 2012 года, в FreeBSD как часть системного вызова `_umtx_op` — с 2016 года, и т. д.

Наиболее заметное исключение — macOS. Хотя ядро данной операционной системы поддерживает все эти операции, они не предоставляются через какие-либо стабильные, общедоступные функции C, которыми нам можно было бы воспользоваться. Но macOS поставляется с последней версией `libc++`, реализацией стандартной библиотеки C++. Эта библиотека включает поддержку C++20, версии C++ со встроенной поддержкой очень простых атомарных операций ожидания и пробуждения (например, `std::atomic<T>::wait()`). Хотя по ряду причин воспользоваться этим в Rust довольно сложно, такая возможность, конечно же, имеется, давая нам доступ к базовым функциям ожидания и пробуждения, подобным фьютексу, и в macOS.

Давайте не будем углубляться в ненужные подробности, а лучше воспользуемся крейтом `atomic-wait` с сайта **crates.io**, который и предоставит нам строительные материалы для создаваемых наших блокирующих примитивов. Этот крейт предоставляет всего три функции: `wait()`, `Wake_one()` и `Wake_all()`. В нем они реализованы для всех основных платформ с применением различных реализаций для той или иной рассмотренной ранее платформы. И если придерживаться этих трех функций, то тревожиться за тонкости, специфичные для любой из платформ, будет совершенно ни к чему.

Эти функции ведут себя точно так же, как и одноименные реализованные нами для Linux в *разделе "Фьютекс" главы 8*, но давайте вспомним, как они работают:

- `wait(&AtomicU32, u32)` — функция используется для ожидания того, что атомарная переменная больше уже не будет содержать заданное значение. Она выставляет блокировку, если значение, хранящееся в атомарной переменной, равно заданному значению. Когда другой поток изменяет значение атомарной переменной, этому потоку, чтобы вывести ожидающий поток из спящего режима, необходимо вызвать для той же атомарной переменной одну из приведенных далее функций пробуждения.

  Эта функция может без видимых на то причин вернуть результат без соответствующей операции пробуждения. Поэтому следует обязательно проверить значение атомарной переменной после возврата функцией управления и при необходимости повторить вызов `wait()`.

- `Wake_one(&AtomicU32)` — функция пробуждает один поток, который на текущий момент заблокирован функцией `wait()` для той же атомарной переменной. Чтобы сообщить одному ожидающему потоку об изменении, эту функцию следует вызвать сразу же после изменения значения атомарной переменной.

- `Wake_all(&AtomicU32)` — функция пробуждает все потоки, которые на текущий момент заблокированы функцией `wait()` для одной и той же атомарной переменной. Чтобы сообщить всем потокам об изменении, эту функцию следует вызвать сразу же после изменения значения атомарной переменной.

Допустима только 32-разрядная атомарность, поскольку это единственный размер, поддерживаемый на всех основных платформах.

Простейший сценарий применения этих функций рассматривался в *разделе "Фьютекс" главы 8* (см. листинги 8.1 и 8.2). Если пример уже стерся из памяти, для продолжения чтения главы его нужно просмотреть еще раз.

Чтобы воспользоваться крейтом `atomic-wait`, добавьте к разделу `[dependencies]` вашего файла `Cargo.toml` строку `atomic-wait = "1"` или же запустите на выполнение команду `cargo add atomic-wait@1`, которая сделает то же самое за вас. Наши три функции определены в корне крейта и могут быть импортированы командой `use atomic_wait::{wait, wake_one, wake_all};`.

К моменту чтения этих строк может появиться доступ к более поздним версиям данного крейта, но при написании главы была готова только основная, первая версия. А в более поздних версиях может не быть совместимого интерфейса.

Теперь, располагая строительными материалами, можно приступать к основным действиям.

Мьютекс

Давайте в качестве образца при создании нашего `Mutex<T>` возьмем тип `SpinLock<T>` из *главы 4*. Фрагменты кода, незадействованные в блокировке, в частности конструкции типа, называемого хранителем, останутся неизменными.

Начнем с определения типа. По сравнению со спин-блокировкой придется внести одно изменение: вместо `AtomicBool`, для которого устанавливается значение `false` или `true`, будет использоваться `AtomicU32` со значениями нуль или единица, что позволит воспользоваться этим типом с атомарными функциями ожидания и пробуждения:

```
pub struct Mutex<T> {
    /// 0: разблокирован
    /// 1: заблокирован
    state: AtomicU32,
    value: UnsafeCell<T>,
}
```

Точно так же, как и со спин-блокировкой, нужно пообещать, что `Mutex<T>` может использоваться несколькими потоками, даже если он содержит тревожащий тип `UnsafeCell`:

```
unsafe impl<T> Sync for Mutex<T> where T: Send {}
```

Добавим также тип `MutexGuard`, реализующий типаж `Deref`, чтобы гарантировать полностью безопасный интерфейс блокировки, как мы это уже делали в *разделе "Безопасный интерфейс с использованием хранителя блокировки" главы 4*:

```
pub struct MutexGuard<'a, T> {
    mutex: &'a Mutex<T>,
}
impl<T> Deref for MutexGuard<'_, T> {
    type Target = T;
    fn deref(&self) -> &T {
        unsafe { &*self.mutex.value.get() }
    }
}
impl<T> DerefMut for MutexGuard<'_, T> {
    fn deref_mut(&mut self) -> &mut T {
        unsafe { &mut *self.mutex.value.get() }
    }
}
```

Сведения о конструкции и работе типа хранителя блокировки можно найти в *разделе "Безопасный интерфейс с использованием хранителя блокировки" главы 4*.

И прежде чем перейти к самому интересному, давайте также уберем функцию `Mutex::new`:

```
impl<T> Mutex<T> {
    pub const fn new(value: T) -> Self {
        Self {
            state: AtomicU32::new(0), // разблокированное состояние
            value: UnsafeCell::new(value),
        }
    }
    ...
}
```

Теперь, после избавления от всего лишнего, остались две части: блокировка (`Mutex::lock()`) и разблокировка (`Drop for MutexGuard<T>`).

Функция, реализованная для нашей спин-блокировки, использует при попытке получения блокировки операцию атомарного обмена, возвращая управление при успешном изменении состояния с "разблокирован" на "заблокирован". В случае неудачи тут же предпринимается новая попытка.

Чтобы заблокировать мьютекс, мы сделаем почти то же самое, но для ожидания перед повторной попыткой воспользуемся `wait()`:

```
pub fn lock(&self) -> MutexGuard<T> {
    // Установка состояния в 1: заблокирован.
    while self.state.swap(1, Acquire) == 1 {
        // Если уже был заблокирован...
        // ... ждать, пока состояние уже не будет равно 1.
        wait(&self.state, 1);
    }
    MutexGuard { mutex: self }
}
```

Для упорядочивания памяти применима та же логика, что и для нашей спин-блокировки. За более подробной информацией обратитесь к материалу *главы 4*.

Заметьте, что функция `wait()` будет ставить блокировку, только если в момент ее вызова состояние все еще установлено в 1 (заблокирован), что позволяет не беспокоиться о возможном пропуске пробуждающего вызова между вызовами обмена и ожидания.

За разблокировку мьютекса отвечает реализация `Drop` для типа хранителя. Разблокировать спин-блокировку было легко, достаточно просто вернуть для состояния значение `false` (разблокирована). Но для нашего мьютекса этого будет недостаточно. Если есть поток, ожидающий блокировки мьютекса, он не узнает, что мьютекс разблокирован, пока не будет уведомлен с помощью операции пробуждения. Если он не будет разбужен, то, скорее всего, так и будет пребывать в вечном сне. Возможно, ему повезет случайно проснуться в нужный момент, но не будем на это рассчитывать.

Итак, состоянию не только будет возвращено значение 0 (разблокирован), но и тут же вызвана функция `wake_one()`:

```
impl<T> Drop for MutexGuard<'_, T> {
    fn drop(&mut self) {
        // Возвращение состоянию значения 0: разблокирован.
        self.mutex.state.store(0, Release);
        // Пробуждение одного из ожидающих потоков, если таковой имеется.
        wake_one(&self.mutex.state);
    }
}
```

Пробуждения одного потока достаточно, потому что даже при нескольких ожидающих потоках на блокировку сможет претендовать только один из них. Следующий, блокирующий поток по завершении блокировки разбудит другой поток, и т. д. Пробуждение более одного потока одновременно просто ничего этим потокам не даст, и когда все потоки, кроме одного, прежде чем снова впасть в спячку, осознают что их шанс на блокировку был отнят у них другим, более удачливым потоком, драгоценное процессорное время уже будет потрачено впустую.

Никакой гарантии, что пробуждаемый поток сможет захватить блокировку, нет. Прежде чем у него появится такая возможность, блокировку все еще может захватить другой поток.

Здесь важно понять, что данная реализация мьютекса по-прежнему будет технически корректной (т. е. безопасной для памяти) и без функций ожидания и пробуждения. Поскольку пробуждение в операции `wait()` может произойти спонтанно, выстраивать какие-либо предположения о том, когда она вернет управление, не представляется возможным. Управлять состоянием наших блокирующих примитивов все равно придется нам самим. Если бы были удалены вызовы функций ожидания и пробуждения, наш мьютекс оказался бы практически идентичен нашей же спин-блокировке.

В общем, атомарные функции ожидания и пробуждения на корректность с точки зрения безопасности памяти вообще никак не влияют. Их применение является всего лишь довольно серьезной оптимизацией, позволяющей избежать ждущего цикла. Но это не значит, что непригодная, неэффективная блокировка будет считаться "правильной" по каким-либо практическим стандартам, но осознание этого может пригодиться в рассуждениях о небезопасном (`unsafe`) коде Rust.

API блокировки

Если вы увлечетесь разработкой Rust-блокировок, то шаблонный код, задействованный в предоставлении безопасного интерфейса, может быстро надоесть. Имеется в виду `UnsafeCell`, реализация типажа `Sync`, тип хранителя, реализация типажа `Deref` и т. д.

Обо всем этом автоматически может позаботиться крейт `lock_api` на **crates.io**. Вам останется только создать тип, представляющий состояние блокировки, и предоставить небезопасные функции блокировки и разблокировки посредством использования небезопасного типажа `lock_api::RawMutex`. А вот тип `lock_api::Mutex` предоставит полностью безопасный и удобный тип мьютекса, включая его хранителя на основе вашей реализации блокировки.

Предотвращение системных вызовов

Несомненно, самая медленная часть нашего мьютекса — ожидание и пробуждение, поскольку они могут повлечь за собой *системный вызов*, т. е. обращение к ядру операционной системы. Общение с ядром — весьма непростой процесс, обычно не отличающийся быстротой, особенно по сравнению с атомарными операциями. Поэтому для эффективной реализации мьютекса нужно постараться максимально избегать вызовов ожидания и пробуждения.

И, к счастью, мы уже на полпути именно к такому режиму работы. Поскольку в нашей функции блокировки проверка состояния в цикле `while` происходит перед вызовом `wait()`, то в той ситуации, когда не нужно ждать, поскольку мьютекс не заблокирован, операция ожидания полностью игнорируется. А вот при снятии блокировки все же неизменно вызывается функция `wake_one()`.

Если известно, что других потоков нет, вызов `wake_one()` можно пропустить. Чтобы быть в курсе наличия ожидающих потоков, нам нужно самим отслеживать соответствующую информацию.

Это можно сделать, разделив состояние "заблокирован" на два отдельных состояния: "заблокирован без ожидающих потоков" и "заблокирован с ожидающими потоками". Давайте воспользуемся для этого значениями 1 и 2 и обновим в определении структуры комментарий к полю состояния:

```
pub struct Mutex<T> {
    /// 0: разблокирован
    /// 1: заблокирован без других ожидающих потоков
    /// 2: заблокирован при наличии других ожидающих потоков
    state: AtomicU32,
    value: UnsafeCell<T>,
}
```

Теперь, чтобы заблокировать разблокированный мьютекс, нашей функции блокировки по-прежнему необходимо установить его состояние в 1. Но, если он уже заблокирован, теперь нашей функции блокировки необходимо установить состояние 2 перед переходом в режим сна, чтобы функция разблокировки могла определить наличие ожидающего потока.

Для этого мы сначала воспользуемся функцией сравнения-обмена, чтобы попытаться сменить состояние с 0 на 1. При удачном исходе мы заблокируем мьютекс и будем знать, что других ожидающих потоков нет, поскольку мьютекс раньше не был заблокирован. Неудачный исход должен быть связан с тем, что мьютекс на текущий момент заблокирован (находится в состоянии 1 или 2). Тогда, чтобы установить для его состояния значение 2, мы воспользуемся операцией атомарного обмена. Если эта операция обмена возвращает старое значение 1 или 2, значит, мьютекс действительно все еще заблокирован, и только тогда, пока состояние не изменится, мы воспользуемся для блокировки функцией `wait()`. Если операция обмена возвращает 0, значит, мы успешно заблокировали мьютекс, сменив его состояние с 0 на 2:

```
pub fn lock(&self) -> MutexGuard<T> {
    if self.state.compare_exchange(0, 1, Acquire, Relaxed).is_err() {
        while self.state.swap(2, Acquire) != 0 {
            wait(&self.state, 2);
        }
    }
    MutexGuard { mutex: self }
}
```

Теперь наша функция разблокировки может воспользоваться новой информацией, пропуская вызов `wake_one()`, когда в этом нет необходимости. Вместо сохранения 0 для разблокировки мьютекса мы, чтобы проверить его предыдущее значение, будем использовать операцию обмена. Пробуждение потока мы продолжим только в том случае, если значение состояния мьютекса равно 2:

```
impl<T> Drop for MutexGuard<'_, T> {
    fn drop(&mut self) {
        if self.mutex.state.swap(0, Release) == 2 {
            wake_one(&self.mutex.state);
        }
    }
}
```

Заметьте, что после возвращения значения состояния в нуль оно больше не указывает на наличие или отсутствие ожидающих потоков. Пробужденный поток отвечает за возврат значения состояния в 2, чтобы гарантировать, что не будут забыты другие ожидающие потоки. Именно поэтому в нашей функции блокировки операция сравнения-обмена не является частью цикла `while`.

Следовательно, всякий раз, когда во время блокировки потоку приходилось ждать, т. е. вызывать `wait()`, при разблокировке, даже если в этом нет необходимости, он также вызывает `wake_one()`. Но тут важнее, что при *отсутствии конкурентности* (в идеальной ситуации, когда потоки не пытаются получить блокировку одновременно) вызовы `wait()` и `wake_one()` полностью исключаются.

На рис. 9.1 показаны операции и отношения "происходит до" в ситуации, когда наш `Mutex` одновременно пытаются заблокировать два потока. Первый поток блокирует мьютекс, меняя состояние с 0 на 1. В этот момент второй поток не сможет получить блокировку и поэтому после смены состояния с 1 на 2 переходит в спящий режим. Когда первый поток разблокирует `Mutex`, он меняет состояние обратно на 0. Поскольку оно было равно 2, что указывало на наличие ожидающего потока, он, чтобы разбудить второй поток, вызывает `wake_one()`. Обратите внимание на отсутствие между операциями пробуждения и ожидания зависимости от каких-либо отношений "происходит до". Хотя вполне вероятно, что именно операция пробуждения отвечает за пробуждение ожидающего потока, связь "происходит до" устанавливается посредством операции получения-обмена, отслеживающей значение, сохраненное операцией высвобождения-обмена.

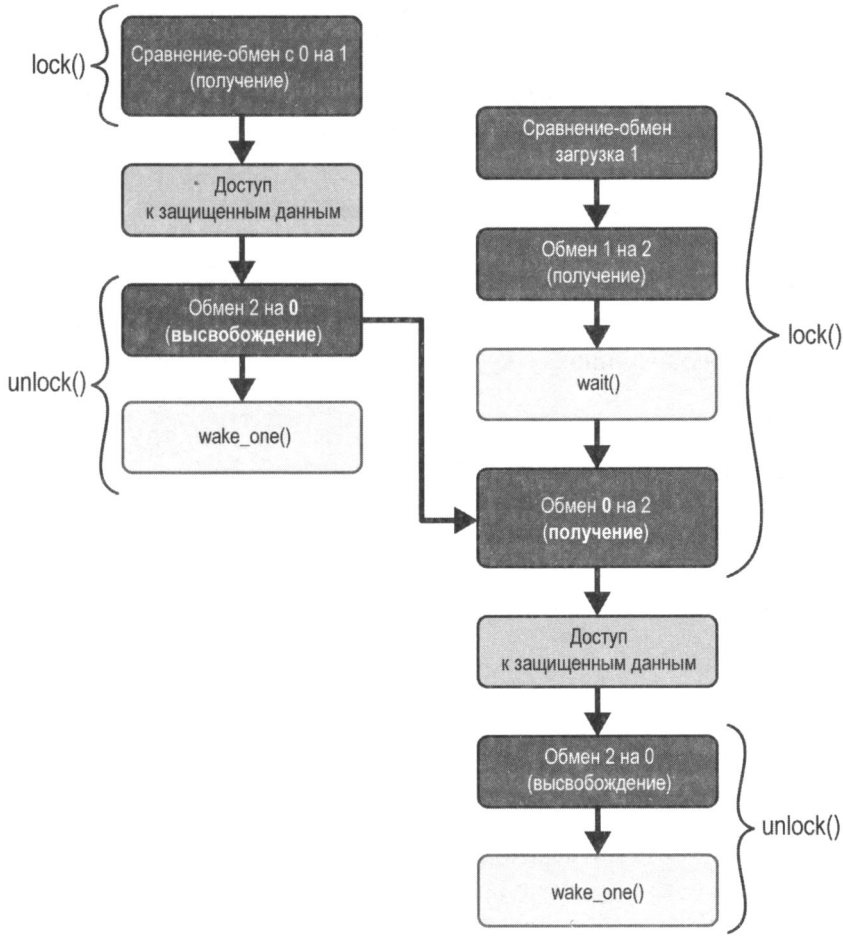

Рис. 9.1. Отношения "происходит до" между двумя потоками, одновременно пытающимися заблокировать наш мьютекс

Дальнейшая оптимизация

Теперь уже может показаться, что оптимизировать больше нечего. При отсутствии конкурентности системные вызовы не выполняются, и остаются всего лишь две весьма простые атомарные операции.

Единственное, что позволит избежать применения операций ожидания и пробуждения, — вернуться к нашей реализации спин-блокировки. Хотя прокрутка обычно крайне неэффективна, она, по крайней мере, позволяет избежать потенциальных издержек, связанных с системным вызовом. Более эффективной прокрутка может оказаться только при крайне малом времени ожидания.

При блокировке мьютекса так бывает, только если поток, удерживающий в данный момент блокировку, работает параллельно на другом ядре процессора и будет удерживать блокировку совсем недолго. Но такое случается довольно часто.

Можно попробовать объединить лучшее из обоих подходов, предваряя вызов `wait()` кратковременной прокруткой. Тогда, если блокировка снимается очень быстро, вызов `wait()` вообще не потребуется, но мы же по-прежнему стремимся избежать пустой траты процессорного времени, которым более эффективно могли бы воспользоваться другие потоки.

Для реализации данного замысла нужно будет изменить только нашу функцию блокировки.

Чтобы обеспечить максимальную производительность при отсутствии конкурентности, исходная операция сравнения-обмена в начале функции блокировки будет сохранена. А ожидающую прокрутку оставим для отдельной функции:

```
impl<T> Mutex<T> {
    ...
    pub fn lock(&self) -> MutexGuard<T> {
        if self.state.compare_exchange(0, 1, Acquire, Relaxed).is_err() {
            // Блокировка уже установлена. :(
            lock_contended(&self.state);
        }
        MutexGuard { mutex: self }
   }
}
fn lock_contended(state: &AtomicU32) {
    ...
}
```

В функции `lock_contended` до перехода к циклу ожидания можно было бы просто повторить одну и ту же операцию сравнения-обмена несколько сотен раз. Но операция сравнения-обмена обычно пытается получить исключительный доступ к соответствующей строке кеша (см. *раздел "Протокол MESI" главы 7*), что может оказаться более затратным действием по сравнению с простой операцией загрузки при ее многократном выполнении.

Учитывая это, можно прийти к следующей реализации `lock_contended`:

```
fn lock_contended(state: &AtomicU32) {
    let mut spin_count = 0;
    while state.load(Relaxed) == 1 && spin_count < 100 {
        spin_count += 1;
        std::hint::spin_loop();
    }
    if state.compare_exchange(0, 1, Acquire, Relaxed).is_ok() {
        return;
    }
    while state.swap(2, Acquire) != 0 {
        wait(state, 2);
    }
}
```

Сначала выполняется прокрутка до 100 раз, для чего, как и в *главе 4*, используется подсказка о применении спин-цикла. Прокрутка выполняется только до тех пор, пока мьютекс заблокирован и у него нет ожидающих потоков. Если другой поток уже находится в режиме ожидания, значит, прокрутка прекратилась, поскольку заняла слишком много времени, что может указывать на сомнительную для данного потока пользу от прокрутки.

Чаще всего продолжительность прокрутки в сто циклов выбирается чисто произвольно. Время, затрачиваемое на итерацию, и продолжительность системного вызова (которого мы стараемся избежать) сильно зависят от платформы. Помощь в выборе правильного числа циклов может оказать длительный сравнительный анализ, но, к сожалению, ни одного заведомо правильного ответа не существует.

Реализация `std::sync::Mutex` в стандартной библиотеке Rust для работы под Linux, по крайней мере, в Rust версии 1.66.0, использует счетчик прокрутки, равный 100.

После изменения состояния блокировки, прежде чем сдаться и приступить к ожиданию, предпринимается еще одна попытка заблокировать мьютекс, установив для его состояния значение 1. Как уже ранее говорилось, после вызова `wait()` заблокировать мьютекс, установив его состояние в 1, невозможно, поскольку это может привести к забвению всех других ожидающих потоков.

Атрибуты *cold* и *inline*

Чтобы компилятор понял, что функция `lock_contended` в обычном (неконкурентном) случае не вызывается, к ее определению можно добавить аргумент `#[cold]`, помогая ему тем самым с оптимизацией метода блокировки.

Кроме того, чтобы сообщить компилятору, что было бы неплохо сделать методы `Mutex` и `MutexGuard` *встраиваемыми*, т. е. разместить результирующие инструкции непосредственно в месте вызова метода, к их определению можно будет добавить атрибут `#[inline]`. Насколько в целом это поднимет производительность, сказать трудно, но для подобных весьма небольших функций, такой эффект вполне возможен.

Сравнительный анализ

Тестирование производительности реализации мьютекса дается нелегко. Написать тест производительности и получить какие-то цифры нетрудно, но извлечь из этого какой-либо по-настоящему значимый результат весьма непросто.

Подогнать оптимизацию кода мьютекса для получения высоких показателей в конкретном тесте производительности относительно легко, но не слишком полезно. Ведь конечная цель заключается в создании чего-то, что будет хорошо работать в реальных, а не только в тестовых программах.

Давайте попробуем написать два простых эталонных теста, свидетельствующих как минимум о положительном влиянии предпринятой оптимизации на некоторые сценарии использования, не забывая о том, что любые выводы не обязательно будут справедливы для разных сценариев.

Для первого теста будет создан `Mutex`, подвергаемый блокировке и разблокировке несколько миллионов раз, и все это в одном и том же потоке с замером общего за-

трачиваемого времени. Это проверка тривиального неконкурентного сценария, в котором никогда не будет потоков, требующих пробуждения.

Будем надеяться, что он продемонстрирует существенную разницу между нашими версиями с двумя и тремя состояниями:

```
fn main() {
    let m = Mutex::new(0);
    std::hint::black_box(&m);
    let start = Instant::now();
    for _ in 0..5_000_000 {
        *m.lock() += 1;
    }
    let duration = start.elapsed();
    println!("locked {} times in {:?}", *m.lock(), duration);
}
```

Здесь, чтобы заставить компилятор предположить наличие дополнительного кода, обращающегося к мьютексу, и не дать ему оптимизировать цикл или операции блокировки, используется подсказка `std::hint::black_box` (как это уже делалось в *разделе "Влияние, оказываемое на производительность" главы 7*).

В зависимости от имеющегося оборудования и операционной системы результаты будут сильно различаться. Если запустить тест на одном конкретном компьютере с новейшим процессором AMD и под управлением Linux, общее время для нашего неоптимизированного мьютекса с двумя состояниями составит около 400 мс, а на тестирование оптимизированного мьютекса с тремя состояниями уйдет около 40 мс. Улучшение в десять раз! На другом компьютере с более старым процессором Intel под управлением Linux разница будет еще существеннее: около 1800 мс против 60 мс. Тем самым подтверждается, что добавление третьего состояния действительно может стать весьма существенной оптимизацией.

А вот запуск этого же процесса на компьютере под управлением macOS дает совершенно иные результаты: около 50 мс для обеих версий, что показывает серьезную зависимость от платформы.

Оказывается, реализация `std::atomic<T>::wake()` в `libc++`, используемая в macOS, уже выполняет собственный, независимый от ядра учет системных ресурсов, позволяющий избежать ненужных системных вызовов. То же самое относится и к `WakeByAddressSingle()` в Windows.

Отказ от вызова рассматриваемых функций все равно может привести к небольшому повышению производительности, поскольку их реализация далеко не так проста, в частности потому, что они не могут хранить какую-либо информацию в самой атомарной переменной. Но если ориентироваться только на эти операционные системы, следует задаться вопросом, действительно ли добавление в наш мьютекс третьего состояния стоило затраченных на это усилий.

Чтобы увидеть, принесла ли наша оптимизация прокрутки какие-либо положительные результаты, нужен другой тест производительности: с большим количеством

конфликтов и с несколькими потоками, неоднократно пытающимися заблокировать уже заблокированный мьютекс.

Давайте возьмем сценарий, в котором четыре потока одновременно пытаются заблокировать и разблокировать мьютекс несколько миллионов раз:

```
fn main() {
    let m = Mutex::new(0);
    std::hint::black_box(&m);
    let start = Instant::now();
    thread::scope(|s| {
        for _ in 0..4 {
            s.spawn(|| {
                for _ in 0..5_000_000 {
                    *m.lock() += 1;
                }
            });
        }
    });
    let duration = start.elapsed();
    println!("locked {} times in {:?}", *m.lock(), duration);
}
```

Следует учесть, что это вымышленный, абсолютно нереальный сценарий. Мьютекс удерживается весьма кратковременно (только для увеличения значения целого числа), и потоки сразу же станут пытаться снова заблокировать мьютекс после его разблокировки. Другой сценарий приведет, скорее всего, к совершенно иным результатам.

Давайте запустим этот тест на тех же двух компьютерах под управлением Linux, как и раньше. В случае со старым процессором Intel для версии мьютекса без прокрутки получим около 900 мс, а для версии мьютекса с прокруткой — около 750 мс. Явное улучшение! Но вот на компьютере с новым процессором AMD будут получены совершенно противоположные результаты: около 650 мс без прокрутки и около 800 мс с прокруткой.

Таким образом, ответ на вопрос, действительно ли прокрутка поднимает производительность, даже если рассматривать всего лишь один сценарий, к сожалению, будет таким: "Все зависит от конкретной ситуации".

Условная переменная

А теперь давайте перейдем к чему-то более интересному: к реализации условной переменной.

Из *раздела "Условные переменные" главы 1* известно, что условная переменная используется вместе с мьютексом для ожидания, пока данные, защищенные мьютексом, не будут соответствовать некоторому условию. Имеется метод ожидания, который снимает блокировку с мьютекса, ждет сигнала и снова блокирует тот же

мьютекс. Другие потоки посылают сигналы обычно сразу же после изменения данных, защищенных мьютексом, либо в один ожидающий поток (что часто называют "известить одного" или "послать сигнал"), либо во все ожидающие потоки (что часто называют "известить всех" или "сделать рассылку").

Притом что условная переменная пытается удержать ожидающий поток в спящем режиме, пока не будет получен сигнал, ожидающий поток может быть случайно разбужен без соответствующего сигнала. Но перед возвратом управления операция ожидания условной переменной все равно повторно заблокирует мьютекс.

Заметьте, что этот интерфейс практически идентичен нашим фьютексным функциям `wait()`, `Wake_one()` и `Wake_all()`. Основное отличие заключается в механизме, предотвращающем потерю сигналов. Условная переменная начнет "прослушивать" сигналы перед разблокировкой мьютекса, чтобы сразу после этого не пропустить ни одного сигнала, а вот наша фьютексная функция `wait()`, чтобы убедиться, что ожидание по-прежнему актуально, полагается на проверку состояния атомарной переменной.

Это наводит на следующий минималистичный замысел реализации условной переменной: если есть уверенность, что каждое уведомление изменяет значение атомарной переменной (например, счетчика), то нашему методу `Condvar::wait()` перед разблокировкой мьютекса и передачей его в фьютексную функцию `wait()` после его разблокировки остается всего лишь проверить значение этой переменной. Следовательно, если после разблокировки мьютекса поступил какой-либо сигнал уведомления, переход в спящий режим не состоится.

Давайте испытаем все это на практике!

Начнем со структуры `Condvar`, в которой содержится только один объект типа `AtomicU32`, инициализируемый нулевым значением:

```
pub struct Condvar {
    counter: AtomicU32,
}
impl Condvar {
    pub const fn new() -> Self {
        Self { counter: AtomicU32::new(0) }
    }
    ...
}
```

Методы уведомления весьма просты. Им нужно всего лишь изменить значение счетчика и воспользоваться соответствующей операцией пробуждения, чтобы уведомить любой ожидающий поток (или потоки):

```
pub fn notify_one(&self) {
    self.counter.fetch_add(1, Relaxed);
    wake_one(&self.counter);
}
```

```
pub fn notify_all(&self) {
    self.counter.fetch_add(1, Relaxed);
    wake_all(&self.counter);
}
```

(Упорядочение памяти будет рассмотрено чуть позже.)

В методе ожидания будет использоваться хранитель `MutexGuard`, поскольку он представляет собой доказательство заблокированности мьютекса. Этот метод также будет возвращать `MutexGuard`, поскольку перед возвращением управления он убедится, что мьютекс снова заблокирован.

Как уже ранее было отмечено, прежде чем разблокировать мьютекс, метод проверит текущее значение счетчика. Чтобы гарантировать, что ни один сигнал не будет пропущен, после разблокировки мьютекса он должен находиться в режиме ожидания только в том случае, если значение счетчика не изменилось. В виде кода это выглядит следующим образом:

```
pub fn wait<'a, T>(&self, guard: MutexGuard<'a, T>) -> MutexGuard<'a, T> {
    let counter_value = self.counter.load(Relaxed);
    // Разблокировка мьютекса путем удаления хранителя,
    // но с запоминанием мьютекса, чтобы впоследствии
    // его можно было бы заблокировать.
    let mutex = guard.mutex;
    drop(guard);
    // Ожидание, но только если с момента разблокировки
    // значение счетчика не изменилось.
    wait(&self.counter, counter_value);
    mutex.lock()
}
```

Здесь используется закрытое поле `mutex` типа `MutexGuard`. Конфиденциальность в Rust основана на модулях, поэтому если определение будет в модуле, отличном от `MutexGuard`, то, чтобы сделать поле `mutex` типа `MutexGuard` доступным для других модулей крейта, его нужно будет пометить, например, как `pub(crate)`.

Прежде чем отпраздновать успешное завершение создания нашей условной переменной, давайте немного подумаем об упорядочении памяти.

Пока мьютекс заблокирован, изменить защищенные им данные не сможет никакой другой поток. Следовательно, до разблокировки мьютекса беспокоиться об уведомлениях не нужно, поскольку, пока мьютекс удерживается в заблокированном состоянии, с данными не может случиться ничего, что заставило бы изменить наше мнение о желании заснуть и ждать.

Единственное, что может нас заинтересовать, — появление после высвобождения нами мьютекса другого потока, блокирующего мьютекс, изменяющего защищенные данные и посылающего нам сигнал (надеемся, что уже после разблокировки мьютекса).

В данной ситуации между разблокировкой мьютекса в `Condvar::wait()` и блокировкой мьютекса в уведомляющем потоке существует отношение "происходит до", гарантирующее, что расслабленная загрузка, происходящая до разблокировки, будет наблюдать значение *до* уведомляющей операции расслабленного увеличения значения, происходящей после блокировки.

Неизвестно, когда операция `wait()` увидит значение, до или после его увеличения, поскольку пока нет ничего, что могло бы гарантировать какой-либо порядок. Но это не важно, поскольку функция `wait()` ведет себя атомарно по отношению к соответствующим операциям пробуждения. Либо она видит новое значение и в этом случае вообще не переходит в режим сна, либо видит старое значение и в этом случае переходит в режим сна, и поток будет разбужен соответствующим вызовом из уведомления `wake_one()` или `wake_all()`.

Операции и отношения "происходит до" для ситуации, в которой один поток использует `Condvar::wait()` для ожидания изменения неких данных, защищенных мьютексом, и пробуждается вторым потоком, который изменяет данные и вызывает `Condvar::wake_one()`, показаны на рис. 9.2. Заметьте: благодаря операциям разблокировки и блокировки первая операция загрузки гарантированно отслеживает значение до его увеличения.

Нужно также подумать о том, что произойдет, если счетчик переполнится.

Само по себе значение счетчика, если оно меняется после каждого уведомления, не играет никакой роли. К сожалению, после чуть более четырех миллиардов уведомлений счетчик переполнится и обнулится, вернувшись к ранее использовавшимся значениям. Чисто технически наша реализация `Condvar::wait()` может перейти в спящий режим, когда этого не следует делать: если она пропустит ровно 4 294 967 296 уведомлений (или любое кратное этому число), счетчик будет переполнен до имевшегося ранее значения.

Вполне разумно полагать, что вероятность подобного развития событий крайне незначительна. В отличие от того, что делалось в нашем методе блокировки мьютекса, здесь состояние повторно не проверяется и вызов `wait()` после пробуждения не повторяется, поэтому беспокойство нужно проявлять только о том, что переполнение произойдет в момент между расслабленной загрузкой значения счетчика и вызовом `wait()`. Если прерывание потока может быть настолько продолжительным, что это позволит получить именно столько уведомлений, то, вероятно, что-то уже пошло не так и программа уже перестала отвечать на запросы. И здесь с позиции здравого смысла можно заявить, что микроскопический дополнительный риск того, что поток заснет, больше не имеет никакого значения.

На платформах, поддерживающих ожидание в стиле фьютекса с ограничением по времени, риск переполнения можно снизить, используя срок действия операции ожидания не более нескольких секунд. Отправка четырех миллиардов уведомлений займет значительно больше времени, и в этот момент несколько дополнительных секунд будут иметь абсолютно ничтожное влияние. Это полностью устраняет любой риск блокировки программы из-за того, что ожидающий поток ошибочно провалится в вечный сон.

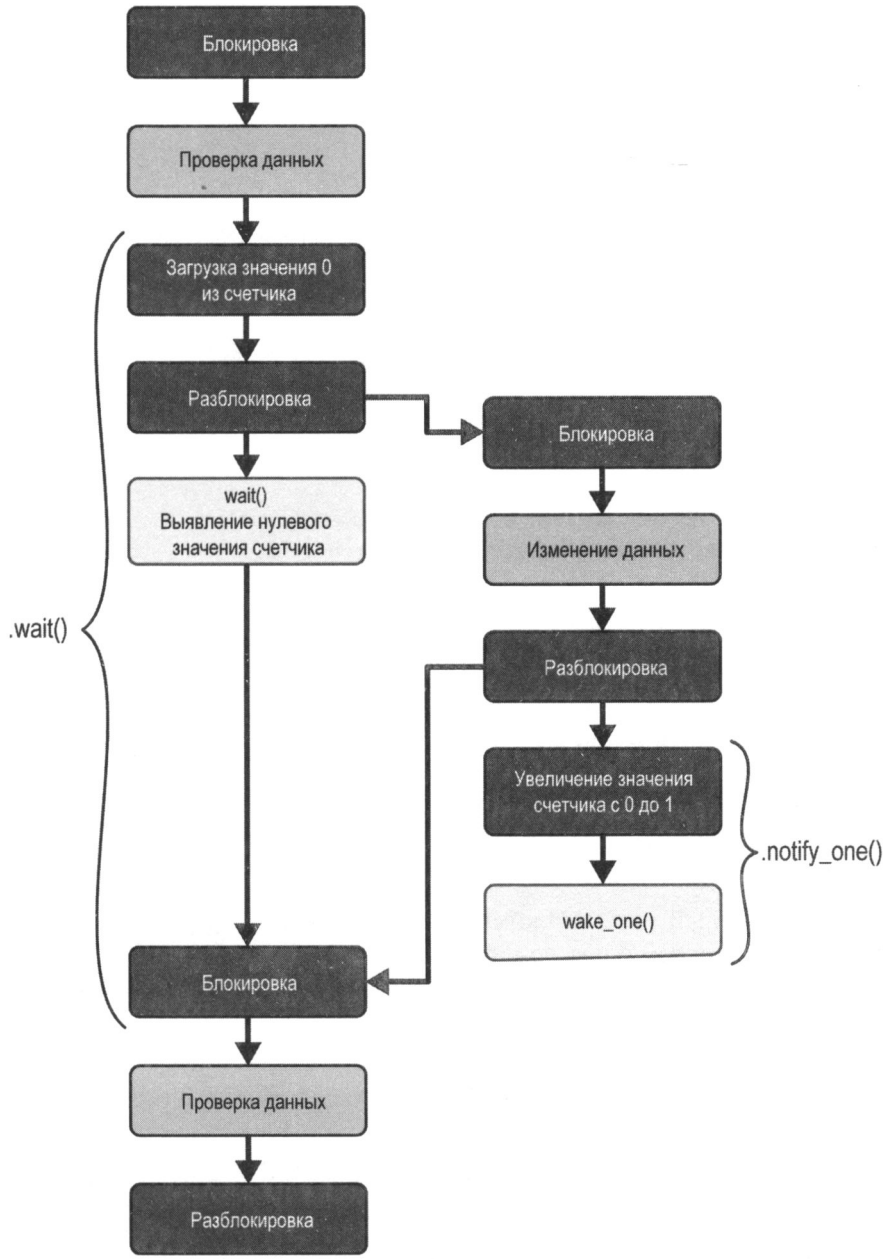

Рис. 9.2. Операции и отношения "происходит до" одного потока, использующего `Condvar::wait()`, **пробуждаемого другим потоком, использующим** `Condvar::notify_one()`

Посмотрим, сработает ли это!

```
#[test]
fn test_condvar() {
    let mutex = Mutex::new(0);
    let condvar = Condvar::new();
```

```
    let mut wakeups = 0;
    thread::scope(|s| {
        s.spawn(|| {
            thread::sleep(Duration::from_secs(1));
            *mutex.lock() = 123;
            condvar.notify_one();
        });
        let mut m = mutex.lock();
        while *m < 100 {
            m = condvar.wait(m);
            wakeups += 1;
        }
        assert_eq!(*m, 123);
    });

    // Проверка того, что основной поток действительно находится
    // в ожидании (а не в прокрутке), допуская при этом несколько ложных пробуждений
    assert!(wakeups < 10);
}
```

Чтобы убедиться, что поток действительно переходит в спящий режим, мы ведем подсчет того, сколько раз для условной переменной происходит возвращение управления из ее функции ожидания. Если данное число было бы очень большим, это означало бы, что вместо нормальной работы мы случайно вошли в режим зацикливания. Важность такой проверки обусловливается тем, что условная переменная, никогда не впадающая в спячку, неизменно приведет к "корректному" поведению, но фактически превратит цикл ожидания в цикл прокрутки.

Если запустить данный тест, он скомпилируется и будет пройден, подтверждая то, что наша условная переменная действительно перевела основной поток в спящий режим. Конечно, это не доказывает правильность его реализации. Если нужно, то для большей уверенности можно воспользоваться более продолжительным стресс-тестом, включающим множество потоков, запускаемым в идеале на компьютере со слабоупорядоченной архитектурой процессора.

Предотвращение системных вызовов

В рассмотренном ранее *разделе "Предотвращение системных вызовов"* выяснилось, что оптимизация примитива блокировки направлена главным образом на предотвращение ненужных операций ожидания и пробуждения.

Но для условной переменной в попытке предотвращения вызова `wait()` в нашей реализации `Condvar::wait()` нет никакого смысла. К тому моменту, когда поток решает дождаться нужного значения условной переменной, он уже проверил, что данное ожидаемое событие еще не произошло, и понимает, что ему нужно перейти в режим сна. Если бы в ожидании не было необходимости, то функция `Condvar::wait()` вообще не вызывалась бы.

Но, если нет ожидающих потоков, то по аналогии с тем, что было сделано для нашего `Mutex`, можно будет уйти от вызовов `wake_one()` и `wake_all()`.

Для этого нужно просто отследить число ожидающих потоков. Наш метод ожидания должен будет увеличить это число перед ожиданием и уменьшить его, когда ожидание завершится. Тогда, если это число будет равно нулю, наши методы уведомления могут пропустить отправку сигнала.

Давайте исходя из этого добавим в нашу структуру `Condvar` новое поле для отслеживания текущего количества ожидающих потоков:

```
pub struct Condvar {
    counter: AtomicU32,
    num_waiters: AtomicUsize, // Новый код!
}
impl Condvar {
    pub const fn new() -> Self {
        Self {
            counter: AtomicU32::new(0),
            num_waiters: AtomicUsize::new(0), // Новый код!
        }
    }
    ...
}
```

Указание для переменной `num_waiters` типа `AtomicUsize` позволяет не беспокоиться о переполнении ее значения. Размер `usize` достаточно велик, чтобы посчитать каждый имеющийся в памяти байт, поэтому, если предположить, что каждый активный поток занимает хотя бы один байт памяти, значение определенно достаточно велико для подсчета любого числа одновременно существующих потоков.

А теперь давайте обновим наши функции уведомлений, чтобы они бездействовали в случае отсутствия ожидающих потоков:

```
pub fn notify_one(&self) {
    if self.num_waiters.load(Relaxed) > 0 { // Новый код!
        self.counter.fetch_add(1, Relaxed);
        wake_one(&self.counter);
    }
}
pub fn notify_all(&self) {
    if self.num_waiters.load(Relaxed) > 0 { // Новый код!
        self.counter.fetch_add(1, Relaxed);
        wake_all(&self.counter);
    }
}
```

(Упорядочение памяти будет рассмотрено чуть позже.)

И наконец, зададим самые важные действия: увеличение числа в начале нашего метода ожидания и его уменьшение сразу же, как наступит пробуждение:

```
pub fn wait<'a, T>(&self, guard: MutexGuard<'a, T>) -> MutexGuard<'a, T> {
    self.num_waiters.fetch_add(1, Relaxed); // Новый код!
    let counter_value = self.counter.load(Relaxed);
    let mutex = guard.mutex;
    drop(guard);
    wait(&self.counter, counter_value);
    self.num_waiters.fetch_sub(1, Relaxed); // Новый код!
    mutex.lock()
}
```

И тут еще раз нужно серьезно задаться вопросом: достаточно ли будет расслабленного упорядочения памяти для всех этих атомарных операций.

Представленный ранее новый потенциальный риск заключается в том, что один из методов уведомления может наблюдать в `num_waiters` нулевое значение и пропустить операцию пробуждения при фактическом наличии потока, нуждающегося в пробуждении. Такая ситуация может сложиться, когда метод уведомления наблюдает значение либо до операции увеличения, либо после операции уменьшения.

Так же как и при расслабленной загрузке значения из счетчика, тот факт, что во время увеличения значения `num_waiters` ожидающий поток по-прежнему удерживает мьютекс заблокированным, гарантирует нам, что любая загрузка `num_waiters`, происходящая после разблокировки мьютекса, не увидит значение, существовавшее до его увеличения.

Не стоит также беспокоиться о том, что уведомляющий поток увидит уменьшенное значение "преждевременно", поскольку как только операция уменьшения будет выполнена, возможно, после ложного пробуждения, ожидающий поток все равно больше не нужно будет пробуждать.

Иными словами, отношение "происходит до", определяющее мьютекс, по-прежнему обеспечивает все необходимые нам гарантии.

Предотвращение ложных пробуждений

Еще один способ оптимизации нашей условной переменной заключается в предотвращении ложного пробуждения. При каждом пробуждении поток пытается заблокировать мьютекс, потенциально конкурируя с другими потоками, что может существенно повлиять на производительность.

Ложное пробуждение базовой операции `wait()` происходит довольно редко, но наша реализация условной переменной может запросто позволить функции `notify_one()` стать причиной прекращения ожидания сразу несколькими потоками. Если поток находится в процессе перехода в спящий режим, только что загрузил значение счетчика, но еще не заснул, вызов `notify_one()` помешает ему перейти в спящий режим из-за обновленного счетчика, но наличие следующей операции `wake_one()` вызовет еще и пробуждение второго потока. Затем оба этих потока будут конкурировать за блокировку мьютекса, тратя впустую драгоценное процессорное время.

Какой бы редкой ни казалась такая ситуация, но на самом деле из-за того, как мьютекс завершает синхронизацию потоков, она может сложиться довольно легко. По-

ток, который вызовет `notify_one()` на основе значения условной переменной, скорее всего, непосредственно перед этим, чтобы изменить что-то в данных, изменения в которых ждет ожидающий поток, заблокирует и разблокирует мьютекс. Это означает, что как только метод `Condvar::wait()` разблокирует мьютекс, тут же может быть снята блокировка и с ожидавшего мьютекс уведомляющего потока. И тогда в гонке будут участвовать два потока: ожидающий поток, чтобы перейти в спячку, и уведомляющий поток, чтобы заблокировать и разблокировать мьютекс и отправить уведомление, изменив значение условной переменной. Если уведомляющий поток выиграет эту гонку, ожидающий поток не перейдет в спящий режим из-за увеличенного значения счетчика, но уведомляющий поток все равно вызовет `wake_one()`. Сложится именно та, рассмотренная ранее проблемная ситуация, в которой безо всякой необходимости будет разбужен еще один ожидающий поток.

Относительно простое решение — отслеживать число потоков, которым разрешено пробуждаться (т. е. возвращать управление из `Condvar::wait()`). Метод `notify_one` увеличит их число на единицу, а метод `wait` попытается уменьшить это число на единицу, если оно не равно нулю. Если счетчик будет равен нулю, то поток вместо попытки повторной блокировки мьютекса и возврата управления из функции может вернуться в спящий режим. Уведомление всех потоков можно выполнить, добавив специально для `notify_all` еще один счетчик, значение которого никогда не уменьшается.

Такой подход сработает, и все же он не лишен новой, более хитрой проблемы: уведомление может разбудить поток, который еще даже не вызвал `Condvar::wait()`, включая и сам уведомляющий поток. Вызов `Condvar::notify_one()` увеличит число подлежащих пробуждению потоков и воспользуется для пробуждения одного из ожидающих потоков вызовом функции `wake_one()`. Затем, если после всего этого другой (или даже тот же самый) поток вызывает `Condvar::wait()` прежде чем поток, уже находящийся в ожидании получает шанс на пробуждение, новый ожидающий поток может увидеть наличие еще одного нерассмотренного уведомления и воспользоваться им, уменьшив значение счетчика до нуля и тут же вернув управление из функции. Затем первый ожидающий поток снова перейдет в спящий режим, поскольку уведомлением уже воспользовался другой поток.

В зависимости от сценария применения одни потоки могут выполняться без проблем, а вот другие могут столкнуться с серьезной проблемой и станут пробуксовывать.

Раньше от этой проблемы страдала реализация `pthread_cond_t` в GNU libc. После долгих дискуссий о том, разрешено ли это спецификацией POSIX, проблема была в конечном итоге решена с выпуском в 2017 году библиотеки GNU libc 2.25, которая включала совершенно новую реализацию условной переменной.

Перехват предыдущего уведомления ожидающим потоком будет для многих сценариев применения условной переменной вполне нормальным явлением. Но при реализации не узкоспециализированной, а универсальной условной переменной такое развитие событий может оказаться абсолютно неприемлемым.

И опять же нам приходится констатировать, что ответ на вопрос о том, стоит ли осуществлять оптимизацию, как ни удивительно, будет таким: "Все зависит от конкретной ситуации".

Способы устранения данной проблемы, избегая при этом ложных пробуждений, имеются, но они гораздо сложнее других подходов.

Решение, используемое новой переменной условия библиотеки GNU libc, включает в себя разделение ожидающих потоков на две группы с разрешением получать уведомления только первой группе и с заменой первой группы на вторую, когда в первой уже не осталось ожидающих потоков.

Недостаток такого подхода не только в сложности алгоритма, но и в том, что он существенно увеличивает размер типа условной переменной, поскольку теперь уже приходится отслеживать гораздо больше информации.

Проблема "громоподобного стада"

Еще одна проблема снижения производительности, с которой можно столкнуться при использовании условной переменной, возникает при вызове `notify_all()` для пробуждения множества потоков с одинаковым предметом ожидания.

Суть проблемы заключается в том, что после пробуждения все эти потоки сразу же пытаются заблокировать один и тот же мьютекс. Успех будет сопутствовать только одному из них, а всем остальным придется снова засыпать. Такое напрасное поглощение вычислительных ресурсов, сопряженное со спешной востребованностью одного и того же ресурса множеством потоков, называется *проблемой громоподобного стада*.

Нетрудно согласиться с тем, что `Condvar::notify_all()` является по своей сути антипаттерном, не заслуживающим оптимизации. Цель применения условной переменной — снятие блокировки с мьютекса и его повторная блокировка при получении уведомления, поэтому, наверное, одновременное уведомление нескольких потоков никогда не приведет ни к чему хорошему.

Даже в этом случае, при желании сгладить данную ситуацию путем оптимизации, сделать это можно только в тех операционных системах, которые поддерживают фьютексную операцию отправки в очередь, например `FUTEX_REQUEUE` в Linux. (См. *раздел "Фьютексные операции" главы 8*.)

Вместо пробуждения множества потоков, из которых все, кроме одного, тут же переходят в спящий режим, как только поймут, что блокировка уже захвачена, можно *отправить в очередь* все потоки, кроме одного, так, чтобы их фьютексные операции ожидания больше не ждали счетчика условной переменной, а начинали вместо этого ждать состояния мьютекса.

Отправка в очередь ожидающего потока не сопряжена с его пробуждением. Фактически поток даже не узнает, что он был отправлен в очередь. Но, к сожалению, это может привести к ряду весьма коварных неприятностей.

Помните, к примеру, как мьютекс с тремя состояниями после пробуждения должен быть всегда заблокирован в надлежащем состоянии ("заблокирован с наличием ожидающих потоков"), чтобы гарантировать, что не уйдут в забвение другие ожидающие потоки? Это означает, что обычный метод блокировки мьютекса в нашей реализации `Condvar::wait()`, который может установить мьютекс в ненадлежащее состояние, нам больше не подойдет.

Реализация условной переменной, отправляющей в очередь, должна будет хранить указатель на мьютекс, используемый ожидающими потоками. Иначе уведомляющие методы не будут знать, к какой атомарной переменной (состоянию мьютекса) следует отправлять в очередь ожидающие потоки. Вот почему условная переменная обычно не позволяет двум потокам выстраивать ожидания на разных мьютексах. И даже притом, что во многих реализациях условных переменных отправка в очередь не применяется, сохранение этого приема в арсенале будущих версий может принести вполне ощутимую пользу.

Блокировка чтения-записи

А теперь давайте займемся реализацией блокировки чтения-записи!

Помнится, в отличие от мьютекса, в блокировке чтения-записи поддерживаются два типа блокировки: блокировка для чтения и блокировка для записи, которые иногда называются совместно используемой блокировкой и исключительной блокировкой. Блокировка для записи ведет себя идентично блокировке мьютекса, допуская одномоментно только одну блокировку, а блокировка для чтения позволяет одновременно удерживать блокировку сразу нескольким читающим потокам. Иными словами, все в точном соответствии тому, как в Rust работают исключительные ссылки (`&mut T`) и совместно используемые ссылки (`&T`), допуская в отдельно взятый момент времени активность только одной исключительной ссылки или любого числа совместно используемых ссылок.

Для мьютекса нужно было только отслеживать, в заблокированном он состоянии или нет. Но для создаваемой блокировки чтения-записи нужно будет знать, сколько блокировок для чтения удерживается на данный момент, гарантируя тем самым установку блокировки для записи только после снятия всех блокировок, имевшихся у читающих потоков.

Для начала займемся структурой `RwLock`, использующей в качестве состояния один объект типа `AtomicU32`. Эта структура будет представлять число захваченных на данный момент блокировок чтения, а нулевое значение будет означать, что блокировка снята. Для представления состояния блокировки записи давайте воспользуемся специальным значением `u32::MAX`:

```
pub struct RwLock<T> {
    /// Число читающих потоков, или же значение u32::MAX,
   // если установлена блокировка записи.
    state: AtomicU32,
    value: UnsafeCell<T>,
}
```

Чтобы гарантировать, что наш `Mutex<T>` нельзя будет использовать для отправки, к примеру, `Rc` в другой поток, пришлось ограничить его реализацию типажа `Sync` типами `T`, имеющими реализацию типажа `Send`. Создаваемому нами новому типу `RwLock<T>` нужно выставить дополнительное требование о реализации в `T` еще и типажа `Sync`, поскольку возможность одновременного доступа к данным должна быть сразу у нескольких читающих потоков:

```
unsafe impl<T> Sync for RwLock<T> where T: Send + Sync {}
```

Поскольку наш `RwLock` можно заблокировать двумя разными способами, у нас будут две отдельные функции блокировки, каждая со своим типом хранителя:

```
impl<T> RwLock<T> {
    pub const fn new(value: T) -> Self {
        Self {
            state: AtomicU32::new(0), // Разблокирован.
```

```
            value: UnsafeCell::new(value),
        }
    }
    pub fn read(&self) -> ReadGuard<T> {
        ...
    }
    pub fn write(&self) -> WriteGuard<T> {
        ...
    }
}
pub struct ReadGuard<'a, T> {
    rwlock: &'a RwLock<T>,
}
pub struct WriteGuard<'a, T> {
    rwlock: &'a RwLock<T>,
}
```

Хранитель для записи должен вести себя как исключительная ссылка (`&mut T`), что мы и делаем путем реализации для него как `Deref`, так и `DerefMut`:

```
impl<T> Deref for WriteGuard<'_, T> {
    type Target = T;
    fn deref(&self) -> &T {
        unsafe { &*self.rwlock.value.get() }
    }
}
impl<T> DerefMut for WriteGuard<'_, T> {
    fn deref_mut(&mut self) -> &mut T {
        unsafe { &mut *self.rwlock.value.get() }
    }
}
```

А в хранителе для чтения должен быть реализован только `Deref`, но не `DerefMut`, поскольку исключительный доступ к данным отсутствует, что заставляет его вести себя наподобие совместно используемой ссылки (`&T`):

```
impl<T> Deref for ReadGuard<'_, T> {
    type Target = T;
    fn deref(&self) -> &T {
        unsafe { &*self.rwlock.value.get() }
    }
}
```

Теперь, разобравшись со всем этим шаблонным кодом, давайте перейдем к более интересным частям: блокировке и разблокировке.

Чтобы заблокировать наш `RwLock` для чтения, нужно увеличить состояние на единицу, но только если оно еще не было заблокировано для записи. Давайте воспользуемся циклом сравнения-обмена (см. *раздел "Операции сравнения и обмена" главы 2*). Если состояние имеет значение `u32::MAX`, означающее, что `RwLock` заблокирован

для записи, воспользуемся операцией `wait()` для перехода в спящий режим и повторим попытку чуть позже:

```
pub fn read(&self) -> ReadGuard<T> {
    let mut s = self.state.load(Relaxed);
    loop {
        if s < u32::MAX {
            assert!(s != u32::MAX - 1, "too many readers");
            match self.state.compare_exchange_weak(
                s, s + 1, Acquire, Relaxed
            ) {
                Ok(_) => return ReadGuard { rwlock: self },
                Err(e) => s = e,
            }
        }
        if s == u32::MAX {
            wait(&self.state, u32::MAX);
            s = self.state.load(Relaxed);
        }
    }
}
```

Блокировка для записи выполняется проще: нужно просто изменить состояние с нуля на `u32::MAX` или воспользоваться операцией `wait()`, если блокировка уже установлена:

```
pub fn write(&self) -> WriteGuard<T> {
    while let Err(s) = self.state.compare_exchange(
        0, u32::MAX, Acquire, Relaxed
    ) {
        // Ожидание, если блокировка уже установлена.
        wait(&self.state, s);
    }
    WriteGuard { rwlock: self }
}
```

Обратите внимание на то, как варьируется точное значение состояния заблокированного `RwLock`, но операция `wait()` для сравнения состояния ждет предоставления ей строго соответствующего значения. Именно поэтому для операции `wait()` используется возвращаемое значение из операции сравнения-обмена.

Разблокировка читающего потока предполагает уменьшение значения состояния на единицу. Читающий поток, который в конечном итоге разблокирует `RwLock`, меняет состояние с единицы на нуль и отвечает за пробуждение ожидающего записывающего потока, если таковой имеется.

Здесь достаточно пробуждения всего лишь одного потока, поскольку точно известно, что на данный момент никаких ожидающих читающих потоков быть не может. Читающему потоку просто незачем было ожидать заблокированного для чтения `RwLock`:

```
impl<T> Drop for ReadGuard<'_, T> {
    fn drop(&mut self) {
        if self.rwlock.state.fetch_sub(1, Release) == 1 {
            // Пробуждение записывающего потока, если таковой имеется.
            wake_one(&self.rwlock.state);
        }
    }
}
```

Для разблокировки записывающий поток должен сбросить значение состояния в нуль, после чего разбудить либо один ожидающий записывающий поток, либо все ожидающие читающие потоки.

Нам неизвестно, какие потоки находятся в режиме ожидания: читающие или записывающие, и у нас нет способа разбудить только записывающий или же только читающие потоки. Поэтому давайте разбудим все потоки:

```
impl<T> Drop for WriteGuard<'_, T> {
    fn drop(&mut self) {
        self.rwlock.state.store(0, Release);
        // Пробуждение сразу всех читающих и записывающих потоков.
       wake_all(&self.rwlock.state);
   }
}
```

Вот и все! Мы создали очень простую, но вполне удобную блокировку чтения-записи.

Пришло время исправить некоторые проблемы.

Предотвращение применения ждущего цикла в записывающих потоках

Одна из проблем нашей реализации заключается в том, что блокировка записи может привести к случайному циклу занятости.

Если для `RwLock` предполагается, что блокировкой и разблокировкой неоднократно будет заниматься множество читающих потоков, значение состояния блокировки может постоянно меняться, быстро возрастая и падая. Для нашего метода `write` это приводит к большой вероятности того, что изменение значения состояния блокировки может произойти между операцией сравнения-обмена и последующей операцией `wait()`, особенно если операция `wait()` непосредственно реализована как (относительно медленный) системный вызов. Это может привести к тому, что операция `wait()` весьма часто будет сразу же и завершаться, даже если блокировка так и не была снята, просто из-за того, что число читающих потоков было не таким, как ожидалось.

Решением может стать использование другой атомарной переменной типа `AtomicU32`, предназначенного для ожидания записывающих потоков, и изменение значения

этой атомарной переменной только при реальных намерениях разбудить записывающий поток.

Давайте попробуем это, добавив новое поле `Writer_wake_counter` в наш `RwLock`:

```
pub struct RwLock<T> {
    // Число читающих потоков, или же u32::MAX,
    // если установлена блокировка для записи.
    state: AtomicU32,
    // Увеличивается в значении для пробуждения записывающих потоков.
    writer_wake_counter: AtomicU32, // Новый код!
    value: UnsafeCell<T>,
}
impl<T> RwLock<T> {
    pub const fn new(value: T) -> Self {
        Self {
            state: AtomicU32::new(0),
            writer_wake_counter: AtomicU32::new(0), // Новый код!
            value: UnsafeCell::new(value),
        }
    }
    ...
}
```

Метод `read` остается без изменений, а вот методу `write` теперь придется ждать соответствующего значения новой атомарной переменной. Чтобы гарантировать полное отсутствие пропусков уведомлений между блокировкой `RwLock` для чтения и фактическим переходом в спящий режим, давайте воспользуемся таким же паттерном, как и для реализации создаваемой условной переменной: контролируя `write_wake_counter` до проверки, не прошло ли "желание" по-прежнему пребывать в спящем режиме:

```
pub fn write(&self) -> WriteGuard<T> {
    while self.state.compare_exchange(
        0, u32::MAX, Acquire, Relaxed
    ).is_err() {
        let w = self.writer_wake_counter.load(Acquire);
        if self.state.load(Relaxed) != 0 {
            // Ожидание, если блокировка RwLock все еще не снята,
            // но только если не было сигналов на пробуждение
            // с момента проверки.
            wait(&self.writer_wake_counter, w);
        }
    }
    WriteGuard { rwlock: self }
}
```

Операция получения-загрузки `write_wake_counter` сформирует отношение "происходит до" с операцией высвобождения-приращения, выполняемой сразу же после разблокировки состояния, но до пробуждения ожидающего записывающего потока:

```
impl<T> Drop for ReadGuard<'_, T> {
    fn drop(&mut self) {
        if self.rwlock.state.fetch_sub(1, Release) == 1 {
            self.rwlock.writer_wake_counter.fetch_add(1, Release);
            // Новый код!
            wake_one(&self.rwlock.writer_wake_counter); // Код изменился!
        }
    }
}
```

Отношение "происходит до" гарантирует, что метод `write` не сможет наблюдать увеличенное значение `write_wake_counter`, но при этом впоследствии видеть еще не уменьшенное значение `state`. В противном случае поток блокировки для записи может прийти к выводу, что блокировка с `RwLock` все еще не снята, хотя сигнал пробуждения пропущен.

Как и прежде, разблокировка для записи должна разбудить либо один ожидающий записывающий поток, либо все ожидающие читающие потоки. Поскольку нам попрежнему неизвестно, какие именно потоки находятся в режиме ожидания, записывающие или читающие, придется разбудить и один ожидающий записывающий поток (посредством `wake_one`), и все ожидающие читающие потоки (посредством `wake_all`):

```
impl<T> Drop for WriteGuard<'_, T> {
    fn drop(&mut self) {
        self.rwlock.state.store(0, Release);
        self.rwlock.writer_wake_counter.fetch_add(1, Release); // Новый код!
        wake_one(&self.rwlock.writer_wake_counter); // Новый код!
        wake_all(&self.rwlock.state);
    }
}
```

В некоторых операционных системах операция, положенная в основу операций пробуждения, возвращает число разбуженных потоков. При этом указываться может меньшее число, чем фактическое число пробудившихся потоков (из-за ложного пробуждения потоков), но возвращаемым значением все равно можно будет воспользоваться для оптимизации кода.

Например, в приведенной ранее реализации удаления можно было бы просто пропустить вызов `wake_all()`, если операция `wake_one()` укажет, что она действительно разбудила поток.

Предотвращение голода записывающих потоков

Обычный вариант использования `RwLock` приводит к ситуации, в которой много часто читающих потоков, но очень мало, зачастую только один, нечасто записывающий поток. Например, один поток может отвечать за считывание входных данных датчика или периодическую загрузку новых данных, востребованных многими другими потоками.

В таком случае можно довольно быстро столкнуться с проблемой, называемой *голодом записывающих потоков*: ситуацией, когда записывающему потоку (или потокам) так и не будет предоставлена возможность заблокировать `RwLock`, поскольку всегда найдутся читающие потоки, удерживающие блокировку `RwLock` для чтения.

Один из вариантов решения данной проблемы — предотвращение получения блокировки любыми новыми читающими потоками при наличии ожидающего записывающего потока, даже если с `RwLock` все еще не снята блокировка для чтения. Таким образом, всем новым читающим потокам придется ждать, пока придет очередь записывающего потока, гарантируя тем самым, что читающие потоки получат доступ к последним данным, которыми записывающий поток хотел поделиться.

Давайте реализуем этот замысел.

Для этого потребуется отслеживать наличие ожидающих записывающих потоков. Чтобы освободить место для этой информации в переменной состояния, можно умножить количество читающих потоков на 2 и прибавить 1 при наличии ожидающего записывающего потока. Тогда состояние 6 или 7 представляет собой ситуацию с тремя активными блокировками для чтения: 6 — без ожидающего записывающего потока и 7 с таким ожидающим потоком.

Если в качестве состояния блокировки для записи сохранить значение `u32::MAX`, являющееся нечетным числом, то читающим потокам придется ждать, т. к. значение состояния нечетное. Но если значение состояния представлено четным числом, то они могут запросто получить блокировку для чтения путем увеличения значения на два:

```
pub struct RwLock<T> {
    /// Удвоенное количество читающих потоков плюс единица, если
    /// имеется ожидающий записывающий поток.
    /// u32::MAX если установлена блокировка для записи.
    ///
    /// Это означает, что читающие потоки могут получить блокировку,
    /// когда у состояния четное значение, но при нечетном значении
    /// их нужно блокировать.
    state: AtomicU32,
    /// Для пробуждения записывающих потоков значение увеличивается.
    writer_wake_counter: AtomicU32,
    value: UnsafeCell<T>,
}
```

В нашем методе `read` придется изменить две инструкции `if`, чтобы они больше не сравнивали состояние с `u32::MAX`, а проверяли, выражено ли значение состояния четным или нечетным числом. Нужно также изменить верхнюю границу в инструкции `assert` и проследить, что блокировка устанавливается путем увеличения значения на два, а не на единицу:

```
pub fn read(&self) -> ReadGuard<T> {
    let mut s = self.state.load(Relaxed);
```

```
    loop {
        if s % 2 == 0 { // Четное.
            assert!(s != u32::MAX - 2, "too many readers");
            match self.state.compare_exchange_weak(
                s, s + 2, Acquire, Relaxed
            ) {
                Ok(_) => return ReadGuard { rwlock: self },
                Err(e) => s = e,
            }
        }
        if s % 2 == 1 { // Нечетное.
            wait(&self.state, s);
            s = self.state.load(Relaxed);
        }
    }
}
```

В наш метод `write` нужно будет внести весьма существенные изменения. Как и в показанном ранее методе `read`, воспользуемся циклом сравнения-обмена. Если значение состояния равно 0 или 1, значит, блокировка с `RwLock` снята, и мы попытаемся изменить состояние на `u32::MAX`, чтобы установить блокировку для записи. В противном случае придется подождать. Но, прежде чем сделать это, нужно убедиться, что значение состояния является нечетным числом, чтобы не дать получить блокировку новым читателям. Убедившись, что значение состояния нечетное, нужно перейти в ожидание нужного значения переменной `write_wake_counter`, одновременно проверяя, что блокировка за это время не была снята.

В коде все это выглядит следующим образом:

```
pub fn write(&self) -> WriteGuard<T> {
    let mut s = self.state.load(Relaxed);
    loop {
        // Попытка заблокировать, если блокировка снята.
        if s <= 1 {
            match self.state.compare_exchange(
                s, u32::MAX, Acquire, Relaxed
            ) {
                Ok(_) => return WriteGuard { rwlock: self },
                Err(e) => { s = e; continue; }
            }
        }
        // Блокировка новых читающих потоков,
        // сделав значение состояния нечетным.
        if s % 2 == 0 {
            match self.state.compare_exchange(
                s, s + 1, Relaxed, Relaxed
            ) {
                Ok(_) => {}
```

```
                Err(e) => { s = e; continue; }
            }
        }
        // Ожидание, если блокировка еще не снята
        let w = self.writer_wake_counter.load(Acquire);
        s = self.state.load(Relaxed);
        if s >= 2 {
            wait(&self.writer_wake_counter, w);
            s = self.state.load(Relaxed);
        }
    }
}
```

Поскольку теперь отслеживается наличие ожидающих записывающих потоков, снятие блокировки для чтения может пропускать вызов `wake_one()`, когда он не нужен:

```
impl<T> Drop for ReadGuard<'_, T> {
    fn drop(&mut self) {
        // Уменьшение значения состояния на 2 для удаления
        // одной блокировки для чтения.
        if self.rwlock.state.fetch_sub(2, Release) == 3 {
            // Если значение уменьшилось с 3 до 1, значит,
            // сейчас блокировка с RwLock снята и имеется
            // пробуждаемый нами ожидающий записывающий поток.
            self.rwlock.writer_wake_counter.fetch_add(1, Release);
            wake_one(&self.rwlock.writer_wake_counter);
        }
    }
}
```

Пока установлена блокировка для записи (значение состояния равно `u32::MAX`), никакая информация об ожидающих потоках не отслеживается. Следовательно, мы не располагаем никакой новой информацией, которой можно было бы воспользоваться для снятия блокировки для записи, которая останется в том же состоянии:

```
impl<T> Drop for WriteGuard<'_, T> {
    fn drop(&mut self) {
        self.rwlock.state.store(0, Release);
        self.rwlock.writer_wake_counter.fetch_add(1, Release);
        wake_one(&self.rwlock.writer_wake_counter);
        wake_all(&self.rwlock.state);
    }
}
```

Для блокировки чтения-записи, оптимизированной под сценарий "частое чтение и нечастая запись", это было бы вполне приемлемо, поскольку блокировка для записи (а следовательно, и снятие этой блокировки) происходит нечасто.

Но для блокировки чтения-записи более универсального плана определенно стоит провести дальнейшую оптимизацию, чтобы приблизить производительность уста-

новки и снятия блокировки для записи к производительности более рационального с этой позиции мьютекса с тремя состояниями. Пусть это станет увлекательным упражнением для читателя данной книги.

Резюме

- Крейт `atomic-wait` предоставляет базовые фьютексные функциональные средства, работающие (в последних версиях) во всех основных операционных системах.
- Для минимальной реализации мьютекса требуется только два состояния, как, к примеру, и для нашего `SpinLock` из *главы 4*.
- В более рациональной реализации мьютекса отслеживается наличие ожидающих потоков, что позволяет избежать ненужной операции пробуждения.
- Прокрутка перед переходом в режим сна в некоторых случаях может принести определенную пользу, но все сильно зависит от складывающейся ситуации, операционной системы и оборудования.
- Минимальной реализации условной переменной нужен только счетчик уведомлений, значение которого функция `Condvar::wait` должна будет проверять как до, так и после разблокировки мьютекса.
- Чтобы избежать ненужных операций пробуждения, условная переменная может отслеживать число ожидающих потоков.
- Предотвращение ложных пробуждений от `Condvar::wait` может даваться непросто, требуя введения дополнительного учета.
- Минимальная реализация блокировки чтения-записи требует, чтобы ее состояние находилось только в атомарном счетчике.
- Для пробуждения записывающих потоков независимо от их читающих собратьев можно воспользоваться дополнительной атомарной переменной.
- Предотвратить голод записывающих потоков позволяет дополнительное состояние для установки приоритета ожидающих записывающих потоков над новыми читающими потоками.

ГЛАВА 10

Идеи и творческое воодушевление

Существует бессчетное количество связанных с вычислениями в конкурентном режиме тем, алгоритмов, структур данных, историй и другой информации для потенциальных глав, которые могли бы войти в данную книгу. Но осталась всего одна, последняя глава, после которой нам предстоит расстаться, надеясь, что у вас сохранится предчувствие открывшихся перед вами новых возможностей и стремление применить новые знания и навыки на практике.

Цель последней главы — вдохновить читателей на собственные творения и будущую работу, раскрыв ряд идей, которые можно усвоить, исследовать и реализовать самостоятельно.

Семафор

Семафор по сути — простой счетчик с двумя операциями: *сигнал* (также называемый *вверх* или *V*) и *ожидание* (*вниз* или *P*). Сигнальная операция увеличивает значение счетчика до определенного максимума, а операция ожидания уменьшает его значение. Если значение счетчика равно нулю, операция ожидания блокируется и ждет соответствующей сигнальной операции, не позволяя счетчику когда-либо принимать отрицательное значение (рис. 10.1). Получается весьма гибкий инструмент, которым можно воспользоваться для реализации других примитивов синхронизации.

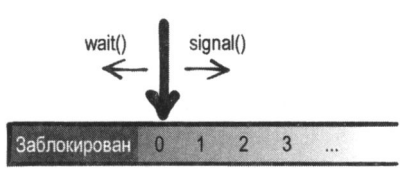

Рис. 10.1. Семафор

Семафор можно реализовать в виде сочетания `Mutex<u32>` для счетчика и `Condvar` для операций ожидания. Но есть ряд более эффективных способов создания семафора. В частности, на платформах, поддерживающих фьютексные операции (см. *раздел "Фьютекс" главы 8*), его можно реализовать более рационально, задействовав всего лишь одно значение типа `AtomicU32` (или даже `AtomicU8`).

Семафор с максимальным значением, равным единице, иногда называют *двоичным семафором*, которым можно воспользоваться для создания других примитивов. На-

пример, его можно применить в качестве мьютекса, инициализируя счетчик единицей, используя операцию ожидания для блокировки и операцию сигнала для разблокировки. А при инициализации нулем семафор подойдет для сигнализации в качестве условной переменной. Например, стандартные функции `park()` и `unpark()` в `std::thread` можно реализовать как операции ожидания и сигнала для двоичного семафора, связанного с потоком.

Важно отметить, что мьютекс можно создать с помощью семафора, а семафор можно реализовать с помощью мьютекса (и условной переменной). Но лучше не использовать семафор на основе мьютекса для реализации мьютекса на основе семафора и наоборот.

Дополнительные источники информации:

- Статья в Википедии о семафорах (**https://oreil.ly/_rSRZ**).
- Конспекты курса Стэнфордского университета по семафорам (**https://oreil.ly/ZVaei**).

RCU

Если требуется разрешить нескольким потокам в основном читать и иногда изменять некоторые данные, можно воспользоваться `RwLock`. А если эти данные являются одним целым числом, то, чтобы избежать блокировки, более рациональным решением будет применение атомарной переменной (например, `AtomicU32`).

Но для более крупных фрагментов данных, например структур со многими полями, доступного атомарного типа, позволяющего выполнять атомарные операции без блокировок, распространяемых на весь объект, просто нет.

Решение этой, как и любой другой проблемы в компьютерной науке, заключается в добавлении уровня абстракции. Вместо самой структуры можно для хранения указателя на нее воспользоваться атомарной переменной. Это все равно не позволяет атомарно изменять структуру в целом, но дает возможность атомарно заменять всю структуру, что тоже неплохо.

Такой паттерн часто называют *RCU*, что означает "Read, Copy, Update" (чтение, копирование, обновление) по тем действиям, которые необходимы для замены данных (рис. 10.2). После прочтения указателя структуру можно скопировать в новый выделенный участок памяти, куда потом внести изменения, не беспокоясь о других потоках. Когда все будет готово, атомарный указатель можно обновить с помощью операции сравнения-обмена (см. *раздел "Операции сравнения и обмена" главы 2*), которая пройдет успешно, только если за это время данные не были заменены никаким другим потоком.

Самое интересное в паттерне RCU — последнее действие, буквы которого нет в аббревиатуре: высвобождение памяти со старыми данными. После успешного обновления другие потоки, если они считали указатель до обновления, могут продолжать читать старую копию. Поэтому до высвобождения памяти, которая была выделена под старую копию, придется дождаться завершения работы всех этих потоков.

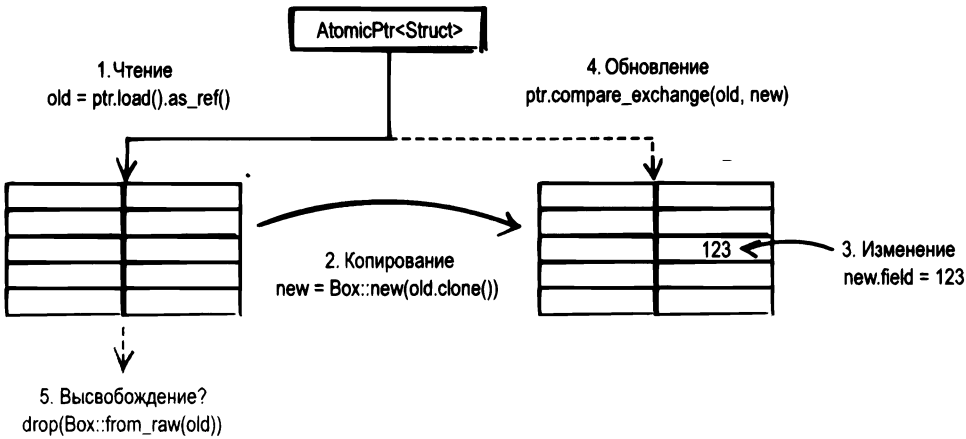

Рис. 10.2. Паттерн RCU

Эту задачу можно решить множеством способов, включая применение подсчета ссылок (например, `Arc`), согласие с утечкой памяти (с игнорированием проблемы), осуществление сбора мусора, использование указателей опасности (способ, позволяющий потокам сообщить другим потокам, какие указатели требуются им в текущий момент) и отслеживание состояния покоя (ожидание момента, когда каждый поток достигнет того места, в котором он точно не задействует указатели). Последний способ в определенных условиях может быть весьма эффективен.

На RCU основаны многие структуры данных, имеющиеся в ядре Linux, и существует немало интересных докладов и статей о подробностях их реализации, которые могут послужить источником вдохновения.

Дополнительные источники информации:

- Статья в Википедии о паттерне Read-copy-update (**https://oreil.ly/egIIi**).
- Статья в LWN "What is RCU, Fundamentally?" (**https://oreil.ly/GQZ6r**).

Связанный список без блокировки

Расширяя базовый RCU-паттерн, к структуре можно добавить атомарный указатель на следующую структуру, превращая ее тем самым в *связанный список*. Тогда потоки смогут атомарно добавлять или удалять в этом списке элементы, обходясь при каждом обновлении без копирования всего списка.

Для вставки в начало списка нового элемента нужно всего лишь выделить под него память и нацелить его указатель на первый элемент списка, а затем атомарно обновить исходный указатель, чтобы он нацеливал на элемент, для которого только что была выделена память (рис. 10.3).

По аналогии с этим удалить элемент можно путем атомарного обновления предшествующего ему элемента, чтобы он показывал на элемент, находящийся после удаляемого элемента. Но, когда задействовано сразу несколько записывающих потоков, следует озаботиться конкурентно выполняемыми операциями вставки или

удаления в соседних элементах. Иначе можно случайно удалить заодно и только что вставленный элемент или же попутно отменить удаление удаляемого элемента.

Чтобы не усложнять решение и исключить возможность сопутствующих изменений, можно воспользоваться обычным мьютексом. В результате чтение по-прежнему останется операцией без блокировок, но теперь уже не придется справляться с сопутствующими изменениями.

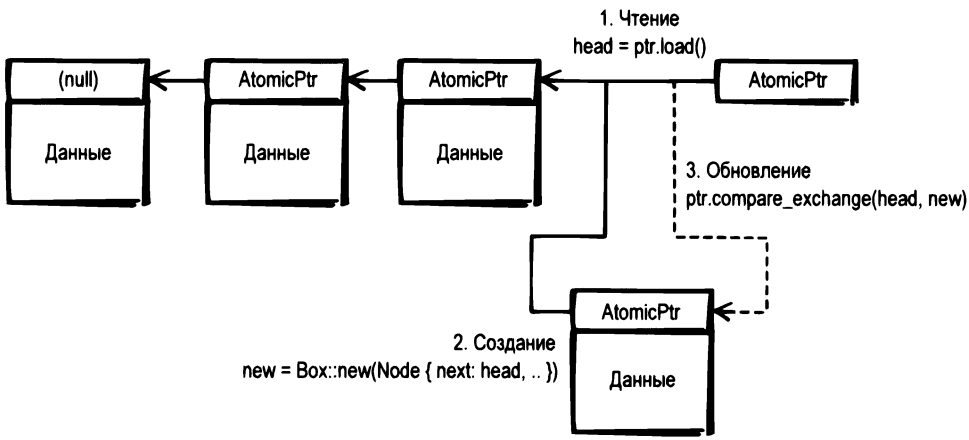

Рис. 10.3. Вариант реализации связанного списка без блокировки

После открепления элемента от связанного списка возникнет уже знакомая нам проблема: ожидание момента, подходящего для высвобождения выделенной ему памяти (или иного заявления прав на владение этой памятью). В данном случае могут сработать такие же решения, как и для базового RCU-паттерна.

В принципе, можно создавать самые разнообразные сложные структуры данных без блокировок, основанные на выполнении операций сравнения-обмена над атомарными указателями, но для высвобождения памяти или иного варианта завладения ею всегда понадобится хорошая стратегия.

Дополнительные источники информации:

- Статья в Википедии о связанных списках без блокировки (**https://oreil.ly/kVQ1O**).
- Статья в LWN "Using RCU for linked lists — a case study" (**https://oreil.ly/H0lt4**).

Блокировки на основе очереди

Для большинства стандартных примитивов блокировки ядро операционной системы отслеживает заблокированные в нем потоки и отвечает за выбор одного из них для пробуждения по запросу. В качестве альтернативы можно создать мьютекс (или другой блокирующий примитив), самостоятельно отслеживая очередь ожидающих потоков.

Такой мьютекс можно реализовать в виде одного `AtomicPtr`, способного указывать на список ожидающих потоков (рис. 10.4).

В каждом элементе такого списка должно содержаться что-то, чем можно будет воспользоваться для пробуждения соответствующего потока, например объект `std::thread::Thread`. Частью неиспользуемых битов атомарного указателя можно воспользоваться для хранения состояния самого мьютекса и всего того, что будет необходимо для управления состоянием очереди.

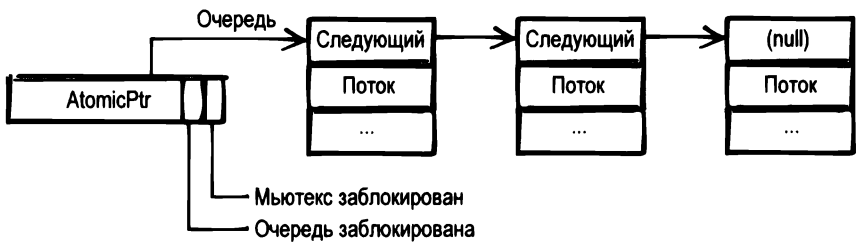

Рис. 10.4. Вариант блокировки на основе очереди

Возможных вариантов великое множество. Очередь может быть защищена собственным битом блокировки или же может быть реализована как частично свободная от блокировок структура. Элементы не обязательно должны размещаться в куче, они могут быть представлены локальными переменными ожидающих потоков. Очередь может представлять собой двусвязный список с указателями не только на следующий, но и на предыдущий элемент. Для удобства добавления элемента в конец очереди первый элемент также может включать указатель на последний элемент.

Данный паттерн позволяет реализовать весьма практичные примитивы блокировки, используя только средства блокировки и пробуждения одного потока, например парковку потоков.

На основе данного паттерна в Windows реализованы SRW-блокировки (см. *раздел "Легкая блокировка чтения-записи" главы 8*).

Дополнительные источники информации:

- Заметки по реализации имеющихся в Windows SRW-блокировок (**https://oreil.ly/El8GA**).
- Реализация на Rust блокировок на основе очередей (**https://oreil.ly/aFyg1**).

Блокировки на основе парковок

Чтобы получить высокоэффективный мьютекс наименьшего размера, можно воспользоваться идеей блокировки на основе очередей, переместив очередь в глобальную структуру данных и оставив внутри самого мьютекса только один или два бита. Тогда мьютекс должен состоять только из одного байта. Его даже можно будет поместить в неиспользуемые части указателя, что позволит выполнять утонченную блокировку практически без дополнительных затрат.

Глобальная структура данных может быть представлена типом `HashMap`, в котором адреса памяти сопоставляются с очередью потоков, ожидающих мьютекс по этому адресу. Такую глобальную структуру данных довольно часто называют парковкой, поскольку она представляет собой набор припаркованных потоков (рис. 10.5).

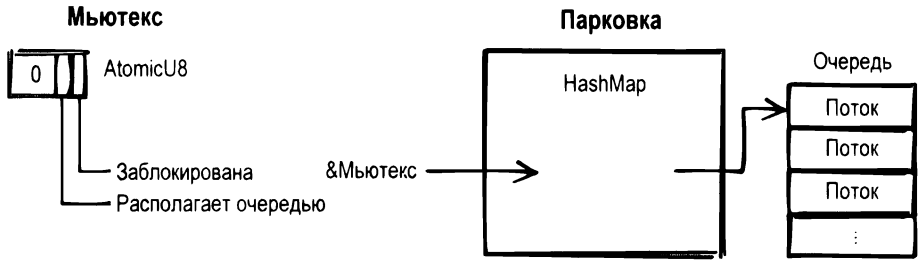

Рис. 10.5. Вариант блокировки на основе парковок

Этот паттерн можно распространить не только на отслеживание очереди на мьютексы, но также и на условные переменные и другие примитивы. Отслеживание в нем очереди для любой атомарной переменной предоставляет весьма действенные пути реализации функциональных средств наподобие фьютексов на тех платформах, которые изначально не поддерживают фьютексы.

Данный паттерн наиболее известен по его реализации в WebKit в 2015 году, где он использовался для блокировки объектов JavaScript. Этот успех вдохновил на создание других реализаций, в числе которых весьма популярный Rust-крейт `parking_lot`.

Дополнительные источники информации:

- Публикация в блоге WebKit "Locking in WebKit" (**https://oreil.ly/6dPim**).
- Документация по крейту `parking_lot` (**https://oreil.ly/UPcXu**).

Последовательная блокировка

Последовательная блокировка (Sequence lock) — еще одно решение проблемы атомарного обновления больших данных без использования традиционных (выставляемых) блокировок (рис. 10.6). В ней задействован атомарный счетчик, значением которого во время обновления данных и даже при готовности данных к чтению является нечетное число.

Записывающий поток перед изменением данных должен будет увеличить значение счетчика с четного до нечетного числа, после чего он должен снова увеличить счетчик, чтобы его значением стало другое четное число.

Любой читающий поток может в любой момент и без блокировки прочитать данные, считав предыдущее и последующее значения счетчика. Если два значения счетчика равны и четны, значит, изменений в конкурентном режиме внесено не было, т. е. была прочитана достоверная копия данных. В противном случае прочитанные данные конкурентно изменялись, и тогда следует просто повторить попытку.

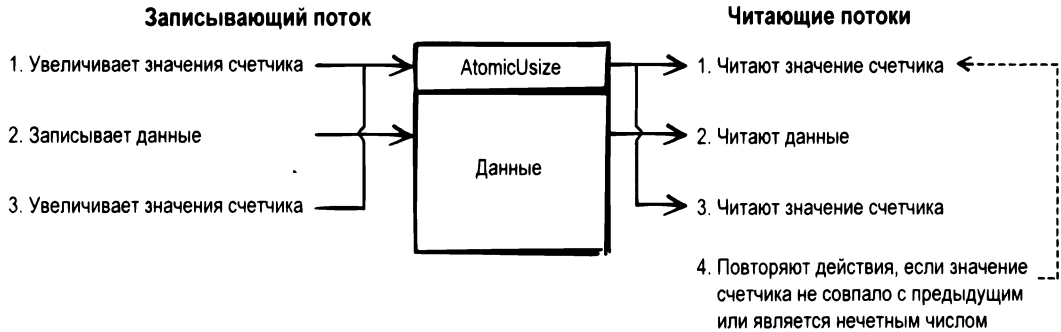

Рис. 10.6. Иллюстрация последовательной блокировки

Это великолепный паттерн для предоставления доступа к данным другим потокам, не допускающий блокировки читающих потоков записывающим потоком. Он довольно часто встречается в ядрах операционных систем и многих встроенных системах. Поскольку читающим потокам для чтения нужен только доступ к памяти, а указатели при этом не задействованы, структура данных может отлично подойти для безопасного применения в совместно используемой процессами памяти в отсутствие каких-либо обязательств перед читающими потоками. В ядре Linux этот паттерн, к примеру, служит для весьма практичного предоставления процессам меток времени, открывая им допуск только для чтения к совместно используемой памяти.

Возникает вопрос: как все это вписывается в модель памяти. Конкурентное неатомарное чтение и запись одних и тех же данных приводят к неопределенному поведению, даже если считанные данные игнорируются. С чисто технической точки зрения, и чтение, и запись данных должны выполняться только с помощью атомарных операций, даже если все чтение или запись не обязательно должны быть одной атомарной операцией.

Дополнительные источники информации:

- Статья в Википедии о Linux Seqlock (**https://oreil.ly/T28bW**).
- Rust RFC 3301, `AtomicPerByte` (**https://oreil.ly/Qavc7**).
- Документация по крейту `seqlock` (**https://oreil.ly/yHd_7**).

Учебные материалы

Придумывание новых структур данных с конкурентным доступом и разработка их эргономичных реализаций на языке Rust может стать весьма увлекательным занятием на протяжении многих часов (или лет). В данном увлечении может оказаться весьма полезным поиск дополнительных информационных ресурсов, применимых к вашим знаниям о Rust, атомарности, блокировках, структурах данных с конкурентным доступом и в конкурентном режиме работы программ в целом, а также создание новых учебных материалов для обмена своими знаниями с другими энтузиастами этого направления.

Информационное поле по данным направлениям страдает от недостатка материалов для новичков. Внедрение в практику программирования языка Rust сыграло существенную роль в повышении доступности системного программирования для всех разработчиков ПО, но многие из них по-прежнему не желают связываться с низкоуровневой конкурентностью. Атомарность нередко считают сродни мистике, которую лучше оставить на откуп весьма небольшой группе экспертов, что не делает чести сторонникам подобной позиции.

Я надеюсь, что именно эта книга исправит ситуацию, но остается еще так много неосвоенного информационного пространства для других книг, публикаций в блогах, статей, видеокурсов, выступлений на конференциях и других материалов, посвященных разработке программных средств на языке Rust для работы в конкурентном режиме.

* * *

И мне не терпится увидеть плоды вашего творчества.

Удачи. ♥

Предметный указатель

A

Acquire 73
Atomic 48
Atomically reference counted 27

C

CISC 145
Compiler Explorer 143
Condvar 45, 100

D

Drop 96, 104

F

Fast user-space mutex 180

M

Miri 125
Mutex 33
mutual exclusion 36

O

Other-multi-copy atomic 164

P

POSIX 176
pthreads 176

R

RCU, Read, Copy, Update 226
Reference Counted, RC 26
Relaxed 49, 69
Release 73
RISC 145

S

SeqCst 82
Sequence lock 230
SpinLock 93
spinning 91
SRW-блокировка 190
static-элемент 25
syscall 175

U

unsafe-блок 34

А

Архитектура
- ◊ процессора 141
  - ▫ слабо упорядоченная 167
  - ▫ строго упорядоченная 165

Ассемблер 142

Б

Блокировка
- ◊ SRW 190
- ◊ на основе очереди 228
- ◊ на основе парковок 230
- ◊ последовательная 230

Блокировка *(прод.)*
◊ с наследованием приоритета 188
◊ чтения-записи 33, 41, 215
Буфер хранения 162

Г

Гонка 53
◊ данных 29, 104

Д

Дизассемблер 142
Диспетчер 175

З

Зависание писателя 41
Заимствование 28
Замыкание 21, 56

К

Канал 100
◊ одноразовый 102
Кеш
◊ протокол MESI 156
◊ протокол MOESI 157
Кеширование 141, 154
Ключевое слово unsafe 35
Конвейеризация 163

Л

Ложное пробуждение 212
Ложное разделение 161

М

Метод
◊ compare_exchange 62
◊ Condvar 100
◊ fetch_update 63
◊ join 50
◊ load 49
◊ split 115
◊ store 49
◊ unlock() 95
◊ unpark() 42
Модель памяти 68
Мьютекс 33, 36, 76, 91
◊ облегченный 189
◊ рекурсивный 190
◊ тестирование производительности 203

Н

Наследование приоритета 187

О

Объект
◊ Arc 122
◊ AtomicPtr<T> 79
◊ MutexGuard 207
◊ Receiver 115
◊ Sender 115
◊ отравленный 39
Ограждение 83
◊ компиляторное 86
◊ памяти 171
Операция
◊ compare_exchange 152
◊ fetch_sub 124
◊ атомарная 48
◊ выборки 54
◊ изменения 54
◊ ожидания 45
◊ сравнения и обмена 60
◊ уведомления 45
◊ фьютексная 181, 183
◊ чтения-изменения-записи 147
Отношение "происходит до" 68

П

Паттерн RCU 226
Переменная
◊ Condvar 45
◊ условная 45, 205
Поведение неопределенное 29
Поток
◊ выполнения 19
◊ парковка 42, 51
◊ с заданной областью действия 23
Примитивы синхронизации 177
Проблема
◊ голода записывающих потоков 221
◊ громоподобного стада 214
Прокрутка 91

С

Связанный список 227
Семафор 225
Системный вызов 175
Слабый указатель 127
Совместное владение 26
Спин-блокировка 91
Спин-цикл 92
Ссылка
◊ исключительная 31
◊ совместно используемая 31
Стандарт POSIX 176

Т

Тип
◊ Condvar 45
◊ PhantomData 117
◊ Rc 26
◊ RwLock 33, 41
◊ RwLock<T> 41
◊ SpinLock 93
◊ UnsafeCell 34
◊ атомарный 33, 48

У

Упорядочение
◊ Relaxed 169
◊ SeqCst 166
◊ в Rust 67
◊ высвобождения и получения памяти 73
◊ памяти расслабленное 69, 131
◊ получения и высвобождения 93
◊ последовательно согласованное 82
◊ потребления 81
◊ расслабленное 71
Утечка
◊ Box 26
◊ объекта 24
◊ памяти 25

Ф

Функция
◊ abort 59
◊ Channel() 110
◊ compare_exchange 63
◊ drop() 98
◊ fence 83
◊ fopen() 175
◊ get_data() 80
◊ lock 95
◊ notify_one() 212
◊ park() 42
◊ spawn 22
◊ wait(&AtomicU32, u32) 195
◊ WaitOnAddress 192
◊ Wake_all(&AtomicU32) 195
◊ Wake_one(&AtomicU32) 195
Фьютекс 180

Х

Хранитель 96, 207

Э

Элемент static 25

Об авторе

Мара Бос (Mara Bos) занимается поддержкой стандартной библиотеки Rust и создает на Rust системы управления в режиме реального времени. Как руководителю группы поддержки библиотеки Rust, ей известны все тонкости языка и стандартной библиотеки. Кроме того, в качестве основателя и технического директора компании Fusion Engineering она уже много лет работает с конкурентными системами реального времени. Поддержка наиболее востребованной в экосистеме Rust библиотеки и повседневная работа над критически важными для безопасности системами наделили ее практическим опытом, позволяющим досконально разобраться в теории и применить ее на практике.

Об изображении на обложке

Животное на обложке книги "Rust: атомарности и блокировки" — медведь кадьяк (Ursus arctos Middendorffi). Этот вид бурого медведя является эндемиком архипелага Кадьяк на Аляске. Кадьяки были изолированы от других медведей примерно 12 000 лет.

Кадьяки — одни из крупнейших медведей в мире. Рост самца, стоящего на задних лапах, может достигать трех метров, а на всех четырех лапах — около полутора метров. Самцы могут весить до 680 кг, а самки — на 20–30 % меньше. Кадьяки крупнее черных медведей, имеют более заметный плечевой горб, менее выступающие уши и более длинные и прямые когти. Хотя кадьяки считаются разновидностью бурого медведя, их трудно идентифицировать по цвету меха, который может варьироваться от темно-коричневого до светло-русого.

Архипелаг Кадьяк представляет собой нетронутую среду обитания медведей. Его леса умеренного пояса полны пышной зелени, цветущей благодаря обильным дождям. Зима на архипелаге долгая и холодная, за ней следует мягкое лето. Медведи приспосабливаются к климату, оптимизируя свой рацион в зависимости от сезона. Весной и в начале лета они питаются быстрорастущими травами. Ягоды употребляют в пищу в конце лета — начале осени. Лосось приходит на нерест с мая по сентябрь, и медведи лакомятся тихоокеанским лососем, нерестящимся в близлежащих лиманах и ручьях. Кадьяки легко адаптируются к разнообразному питанию, и их может привлечь беспорядочно разбросанный мусор и пищевые отходы в кемпингах и возле жилых домов.

На кадьяков когда-то активно охотились, защищая от них домашний скот, но теперь, чтобы поддерживать оптимальную численность популяции, охота регулируется. В результате медведям Кадьяка присвоен охранный статус Least Concern, вызывающий наименьшее беспокойство за их выживание. На обложках книг издательства O'Reilly можно увидеть изображения многих животных, находящихся под угрозой исчезновения, и все они важны для всего человечества.

Иллюстрация на обложке выполнена Карен Монтгомери (Karen Montgomery) на основе черно-белой гравюры из журнала Zoology.

Мара Бос

Rust: атомарности и блокировки

Перевод с английского Н. Вильчинского

ТОО "АЛИСТ"
010000, Республика Казахстан,
г. Астана, пр. Сарыарка, д. 17, ВП 30

Подписано в печать 07.05.24.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 19,35.
Тираж 1200 экз. Заказ № 9438.

Отпечатано с готового оригинал-макета
ООО "Принт-М", 142300, РФ, М.О., г. Чехов, ул. Полиграфистов, д. 1

O'REILLY®

Rust: атомарности и блокировки

Язык программирования Rust великолепно справляется с конкурентными вычислениями, а в его экосистеме имеется множество библиотек, включающих широкий набор структур данных, блокировок и многого другого, предназначенного для работы в среде конкурирующих потоков. Но корректная реализация этих структур может оказаться весьма непростой задачей. Ошибки, допущенные в вопросах упорядочения памяти, нередко встречаются даже в самых востребованных библиотеках.

В этой книге, имеющей сугубо практическую направленность, Мара Бос, руководитель команды разработчиков библиотеки Rust, помогает Rust-программистам всех уровней получить четкое представление о низкоуровневой конкурентности. Вы узнаете об атомарности и упорядочении памяти, а также о том, как они сочетаются с базовыми API-интерфейсами операционной системы для создания таких наиболее востребованных примитивов, как мьютексы и условные переменные. Прочитав книгу, вы получите четкое представление о том, как связаны друг с другом используемая в Rust модель памяти, процессор и особенности той или иной операционной системы.

Вы узнаете:

- **Об исключительной роли имеющейся в Rust системы типов в корректном программировании конкурентности**
- **Все о мьютексах, условных переменных, атомарности и упорядочении памяти**
- **Что на самом деле происходит с атомарными операциями в процессорах Intel и ARM**
- **Как реализуются блокировки при поддержке операционной системы**
- **Как создается корректный программный код, включающий конкурентность, атомарность и блокировки**
- **Какими приемами можно воспользоваться для правильного создания своих собственных примитивов блокировки и синхронизации**

«Это удивительная книга! В ней именно то, что мне самой хотелось рассказать о конкурентности, но у Мары получилось гораздо лучше, о чем я даже не смела мечтать. Досконально и все на своих местах».

— **Ария Бейнжеснер**, автор книги *«The Rustonomicon»*

Мара Бос занимается поддержкой стандартной библиотеки Rust и создает на Rust системы управления в реальном времени. Как руководителю группы поддержки библиотеки Rust, ей известны все тонкости языка и стандартной библиотеки. Кроме того, в качестве основателя и технического директора компании Fusion Engineering она уже много лет работает с конкурентными системами реального времени. Поддержка наиболее востребованной в экосистеме Rust библиотеки и повседневная работа над критически важными для безопасности системами наделили ее практическим опытом, позволяющим досконально разобраться в теории и применить ее на практике.

ISBN 978-601-09-5050-4

9 786010 950504

alst